

Alberto Griggio / Neha Rungta (Eds.)

**PROCEEDINGS OF THE 22ND
CONFERENCE ON FORMAL
METHODS IN COMPUTER-AIDED
DESIGN – FMCAD 2022**



Academic Press



fmcad.²²

Alberto Griggio / Neha Rungta (Eds.)
PROCEEDINGS OF THE 22ND CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED
DESIGN – FMCAD 2022

Conference Series: Formal Methods in Computer-Aided Design

Volume 3

Conference Series: Formal Methods in Computer-Aided Design

Series edited by:

Warren A. Hunt, Jr., The University of Texas at Austin
Austin, TX 78705 | hunt@cs.utexas.edu

Georg Weissenbacher, TU Wien
Karlsplatz 13, 1040 Wien, Austria | georg.weissenbacher@tuwien.ac.at

The Conference on Formal Methods in Computer-Aided Design (FMCAD) is an annual conference on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing.

Information on this publication series and the volumes published therein is available at www.tuwien.ac.at/academicpress.

Volume 3 edited by:

Alberto Griggio, Fondazione Bruno Kessler
Via Sommarive 18, 38122 Trento, Italy | griggio@fbk.eu

Neha Rungta, Amazon Web Services, Inc.
Seattle, WA, USA | rungta@amazon.com

Alberto Griggio / Neha Rungta (Eds.)

PROCEEDINGS OF THE 22ND CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN – FMCAD 2022

Cite as:

Griggio, A. & Rungta, N. (Eds.). (2022). *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022*. TU Wien Academic Press. <https://doi.org/10.34727/2022/isbn.978-3-85448-053-2>

TU Wien Academic Press, 2022

c/o TU Wien Bibliothek
TU Wien
Resselgasse 4, 1040 Wien
academicpress@tuwien.ac.at
www.tuwien.at/academicpress



This work is licensed under a Creative Commons attribution 4.0 international license (CC BY 4.0).
<https://creativecommons.org/licenses/by/4.0/>

ISBN (online): 978-3-85448-053-2
ISSN (online): 2708-7824

Available online: <https://doi.org/10.34727/2022/isbn.978-3-85448-053-2>

Media proprietor: TU Wien, Karlsplatz 13, 1040 Wien
Publisher: TU Wien Academic Press
Publication series editor: Warren A. Hunt, Jr. and Georg Weissenbacher
Editors (responsible for the content): Alberto Griggio and Neha Rungta

Preface

These are the proceedings of the twenty-second International Conference on Formal Methods in Computer-Aided Design (FMCAD), which was held in Trento, Italy from October 18 – October 21, 2022. FMCAD was first held in 1996, and was a bi-annual conference until 2006, when the FMCAD and CHARME conferences merged into a single FMCAD conference, and since then has been held annually. FMCAD 2022 is the twenty-second edition in the series, covering formal aspects of computer-aided system design including verification, specification, synthesis, and testing. It provides a leading forum to researchers in academia and industry to present and discuss groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems.

The program of FMCAD 2022 consists of two tutorials, two invited talks, a student forum, and the main program consisting of presentations of 40 accepted peer-reviewed papers.

The tutorial day featured two presentations:

- *On Applying Model Checking in Formal Verification* by Håkan Hjort
- *Verification of Distributed Protocols: Decidable Modeling and Invariant Inference* by Oded Padon

and the main conference featured two invited talks:

- *The seL4 Verification Journey: How Have the Challenges and Opportunities Evolved* by June Andronick
- *Why Do Things Go Wrong (or Right)? Applications of Causal Reasoning to Verification* by Hana Chockler

FMCAD 2022 received 88 submissions out of which the committee decided to accept 40 for publication. Each submission received at least four reviews. The topics of the accepted papers include hardware and software verification, SAT, SMT, learning, synthesis, neural network verification, and others. Among the accepted papers, there are 31 regular papers (28 long and 3 short) and 9 tool/case study papers (6 long and 3 short).

FMCAD 2022 hosted the tenth edition of the Student Forum, which has been held annually since 2013 and provides a platform for graduate students at any career stage to introduce their research to the FMCAD community. The FMCAD Student Forum 2022 was organized by Mathias Preiner and featured short presentations of 21 accepted contributions. The proceedings provide a detailed description of the Student Forum and lists all accepted contributions.

Organizing this event was made possible by the support of a large number of people and our sponsors. The program committee members and additional reviewers, listed on the following pages, did an excellent job providing detailed and insightful reviews. The reviews helped us build a strong program and helped the authors improve their submissions. We thank each and everyone of them for dedicating their time and providing their expertise. We thank Martin Jonáš for acting both as the web master and as the Sponsorship Chair, and Mathias Preiner for organizing this year's FMCAD Student Forum. We thank Georg Weissenbacher both for his exceptional assistance in organizing the event, communicating to us the decisions of the steering committee, as well as being the publication chair.

Holding a conference like FMCAD would not be feasible without the financial support of our sponsors. We would like to express our gratitude to our sponsors (in alphabetical order): Amazon Web Services, Cadence, Intel, Meta, and Synopsys.

The conference proceedings are available as Open Access Proceedings published by TU Wien Academic Press, and through the IEEE Xplore Digital Library. Last but not least, we thank all authors who submitted their papers to FMCAD 2022 (accepted or not), and whose contributions and presentations form the core of the conference. We are grateful to everyone who presented their paper, gave a keynote or gave a tutorial. We thank all attendees of FMCAD for supporting the conference and making FMCAD an engaging and enjoyable event.

October 2022

Alberto Griggio, Fondazione Bruno Kessler
Neha Rungta, Amazon Web Services, Inc.

Organizing Committee

Program Co-Chairs

Alberto Griggio
Neha Rungta

Fondazione Bruno Kessler, Italy
Amazon Web Services, Inc., CA, USA

Student Forum Chair

Mathias Preiner

Stanford University, CA, USA

Sponsorship and Web Chair

Martin Jonáš

Fondazione Bruno Kessler, Italy

Local Organization

Isabella Masè
Annalisa Armani

Fondazione Bruno Kessler, Italy
Fondazione Bruno Kessler, Italy

Publication Chair

Georg Weissenbacher

TU Wien, Austria

FMCAD Steering Committee

Clark Barrett
Armin Biere
Ruzica Piskac
Anna Slobodova
Georg Weissenbacher

Stanford University, CA, USA
University of Freiburg, Germany
Yale University, CT, USA
Intel Corporation, TX, USA
TU Wien, Austria

Program Committees

FMCAD 2022 Program Committee

Erika Abraham	RWTH Aachen University
Josh Berdine	Meta
Per Bjesse	Synopsys, Inc.
Nikolaj Bjørner	Microsoft
Roderick Bloem	Graz University of Technology
Supratik Chakraborty	IIT Bombay
Sylvain Conchon	Universite Paris-Sud
Vijay D'Silva	Google
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Rohit Dureja	IBM Corporation
Grigory Fedyukovich	Florida State University
Arie Gurfinkel	University of Waterloo
Fei He	Tsinghua University
Ahmed Irfan	Amazon Web Services
Alexander Ivrii	IBM
Barbara Jobstmann	EPFL and Cadence Design Systems
Tim King	Google
Kuldeep S. Meel	National University of Singapore
Sergio Mover	Ecole Polytechnique
Alexander Nadel	Intel
Aina Niemetz	Stanford University
Elizabeth Polgreen	University of California, Berkeley
Rahul Purandare	Indraprastha Institute of Information Technology Delhi
Andrew Reynolds	University of Iowa
Marco Roveri	University of Trento
Kristin Yvonne Rozier	Iowa State University
Philipp Ruemmer	University of Regensburg
Christoph Scholl	University of Freiburg
Natasha Sharygina	Università della Svizzera Italiana
Elena Sherman	Boise State University
Sharon Shoham	Tel Aviv University
Anna Slobodova	Intel
Christoph Sticksel	The MathWorks
Michael Tautschnig	Queen Mary University of London
Nestan Tsiskaridze	Stanford University
Yakir Vizel	The Technion
Georg Weissenbacher	TU Wien
Michael Whalen	Amazon Web Services

FMCAD 2022 Student Forum Committee

Armin Biere	University of Freiburg
Martin Blicha	University of Lugano
Rayna Dimitrova	CISPA Helmholtz Center for Information Security
Rohit Dureja	IBM Corporation
Mathias Fleury	University of Freiburg
Aman Goel	Amazon Web Services
Stéphane Graham-Lengrand	SRI International
Antti Hyvärinen	Università della Svizzera Italiana
Ahmed Irfan	Amazon Web Services
Martin Jonáš	Fondazione Bruno Kessler, Italy
Daniela Kaufmann	Software Competence Center Hagenberg
Daniel Larraz	University of Iowa
Makai Mann	MIT Lincoln Laboratory
Alexander Nadel	Intel
Nina Narodytska	VMware Research
Andres Noetzli	Stanford University
Mark Santolucito	Barnard College
Nestan Tsiskaridze	Stanford University
Tom van Dijk	University of Twente
Florian Zuleger	TU Wien

Additional Reviewers

Andraus, Zaher
Asadi, Sepideh

Barrett, Clark
Becchi, Anna
Biere, Armin
Blich, Martin
Bourgeat, Thomas
Britikov, Konstantin

Cano, Filip

De Masellis, Riccardo
Debrestian, Darin

Eiers, William
Esen, Zafer

Fan, Hongyu
Fazekas, Katalin
Feldman, Yotam M. Y.
Fleury, Mathias

Gamboa Guzman, Laura P.
Garcia-Contreras, Isabel
Geatti, Luca
Gidon, Ernst
Goel, Aman
Golia, Priyanka

Hadarean, Liana
Hadzic, Vedad
Hamza, Ameer
Hamza, Jad
Hyvärinen, Antti

Itzhaky, Shachar

Jain, Himanshu
Jain, Mitesh
Johannsen, Chris
Jovanović, Dejan
Junges, Sebastian

Kaivola, Roope
Kapoor, Ashish
Kaufmann, Daniela
Khasidashvili, Zurab
Koenig, Jason
Könighofer, Bettina
Korneva, Alexandrina
Kroening, Daniel
Kuncak, Viktor

Larrauri, Alberto
Larraz, Daniel
Leslie-Hurd, Joe
Liang, Chencheng
Lonsing, Florian
Luppen, Zachary

Maderbacher, Benedikt
Magnago, Enrico
Martins, Ruben
Mohajerani, Sahar
Mony, Hari
Mora, Federico

O’Leary, John
Otoni, Rodrigo

Parsert, Julian
Peled, Doron
Prabhu, Sumanth
Preiner, Mathias
Priya, Siddharth

Rao, Vikas
Rappaport, Omer
Riley, Daniel
Rosner, Nicolás

Soos, Mate
Sosnovich, Adi
Strichman, Ofer
Su, Yusen
Sumners, Rob
Swords, Sol

Torfah, Hazem

Vediramana Krishnan,
Hari Govind

Weiss, Gail

Yu, Qianshan

Zohar, Yoni
Zuleger, Florian

Table of Contents

Invited Talks

- The seL4 Verification Journey: How Have the Challenges and Opportunities Evolved 1
June Andronick
- Why Do Things Go Wrong (or Right)? Applications of Causal Reasoning to Verification 2
Hana Chockler

Tutorials

- On Applying Model Checking in Formal Verification 3
Håkan Hjort
- Verification of Distributed Protocols: Decidable Modeling and Invariant Inference 4
Oded Padon

Student Forum

- The FMCAD 2022 Student Forum 5
Matthias Preiner

Verification in Machine Learning

- Proving Robustness of KNN Against Adversarial Data Poisoning 7
Yannan Li, Jingbo Wang and Chao Wang
- On Optimizing Back-Substitution Methods for Neural Network Verification 17
Tom Zelazny, Haoze Wu, Clark Barrett and Guy Katz
- Verification-Aided Deep Ensemble Selection 27
Guy Amir, Tom Zelazny, Guy Katz and Michael Schapira
- Neural Network Verification with Proof Production 38
Omri Isac, Clark Barrett, Min Zhang and Guy Katz

Proofs

- TBUDDY: A Proof-Generating BDD Package 49
Randal Bryant
- Stratified Certification for k-Induction 59
Emily Yu, Nils Frolyeks, Armin Biere and Keijo Heljanko
- Reconstructing Fine-Grained Proofs of Complex Rewrites Using a Domain-Specific Language 65
Andres Noetzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Cesare Tinelli and Clark Barrett
- Small Proofs from Congruence Closure 75
Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock and Pavel Panchekha

Proof-Stitch: Proof Combination for Divide-and-Conquer SAT Solvers	84
<i>Abhishek Nair, Saranyu Chattopadhyay, Haoze Wu, Alex Ozdemir and Clark Barrett</i>	
Hardware and RTL	
Reconciling Verified-Circuit Development and Verilog Development	89
<i>Andreas Lööw</i>	
Timed Causal Fanin Analysis for Symbolic Circuit Simulation	99
<i>Roope Kaivola and Neta Bar Kama</i>	
Divider Verification Using Symbolic Computer Algebra and Delayed Don't Care Optimization	108
<i>Alexander Konrad, Christoph Scholl, Alireza Mahzoon, Daniel Große and Rolf Drechsler</i>	
Formally Verified Isolation of DMA	118
<i>Jonas Haglund and Roberto Guanciale</i>	
Foundations and Tools in HOL4 for Analysis of Microarchitectural Out-of-Order Execution	129
<i>Karl Palmskog, Xiaomo Yao, Ning Dong, Roberto Guanciale and Mads Dam</i>	
Synthesizing Instruction Selection Rewrite Rules from RTL using SMT	139
<i>Ross Daly, Caleb Donovan, Jack Melchert, Raj Setaluri, Nestan Tsiskaridze, Priyanka Raina, Clark Barrett and Pat Hanrahan</i>	
Error Correction Code Algorithm and Implementation Verification using Symbolic Representations .	151
<i>Aarti Gupta, Roope Kaivola, Mihir Parang Mehta and Vaibhav Singh</i>	
SAT and SMT	
First-Order Subsumption via SAT Solving	160
<i>Jakob Rath, Armin Biere and Laura Kovacs</i>	
BaxMC: a CEGAR approach to MAX#SAT	170
<i>Thomas Vigouroux, Cristian Ene, David Monniaux, Laurent Mounier and Marie-Laure Potet</i>	
Compact Symmetry Breaking for Tournaments	179
<i>Evan Lohn, Chris Lambert and Marijn Heule</i>	
Enumerative Data Types with Constraints	189
<i>Andrew T Walter, David Greve and Panagiotis Manolios</i>	
Reducing NEXP-complete problems to DQBF	199
<i>Fa-Hsun Chen, Shen-Chang Huang, Yu-Cheng Lu and Tony Tan</i>	
INC: A Scalable Incremental Weighted Sampler	205
<i>Suwei Yang, Victor Liang and Kuldeep S. Meel</i>	
Bounded Model Checking for LLVM	214
<i>Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao and Arie Gurfinkel</i>	
Parameterized Systems and Quantified Reasoning	
Automatic Repair and Deadlock Detection for Parameterized Systems	225
<i>Swen Jacobs, Mouhammad Sakr and Marcus Völz</i>	
Synthesizing Locally Symmetric Parameterized Protocols from Temporal Specifications	235
<i>Ruoxi Zhang, Richard Trefler and Kedar Namjoshi</i>	

Synthesizing Self-Stabilizing Parameterized Protocols with Unbounded Variables	245
<i>Ali Ebneenasir</i>	
The Rapid Software Verification Framework	255
<i>Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovacs and Giles Reger</i>	
Distributed Systems	
ACORN: Network Control Plane Abstraction using Route Nondeterminism	261
<i>Divya Raghunathan, Ryan Beckett, Aarti Gupta and David Walker</i>	
Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+	273
<i>William Schultz, Ian Dardik and Stavros Tripakis</i>	
Awaiting for Godot: Stateless Model Checking that Avoids Executions where Nothing Happens	284
<i>Bengt Jonsson, Magnus Lång and Kostis Sagonas</i>	
Synthesis	
Synthesizing Transducers from Complex Specifications	294
<i>Anway Grover, Rüdiger Ehlers and Loris D’Antoni</i>	
Synthesis of Semantic Actions in Attribute Grammars	304
<i>Pankaj Kumar Kalita, Miriyala Jeevan Kumar and Subhajit Roy</i>	
Reactive Synthesis Modulo Theories using Abstraction Refinement	315
<i>Benedikt Maderbacher and Roderick Bloem</i>	
Learning Deterministic Finite Automata Decompositions from Examples and Demonstrations	325
<i>Niklas Lauffer, Beyazit Yalcinkaya, Marcell Vazquez-Chanlatte, Ameesh Shah and Sanjit A. Seshia</i>	
Reachability and Safety Verification	
Automated Conversion of Axiomatic to Operational Models: Theoretical and Practical Results	331
<i>Adwait Godbole, Yatin A. Manerkar and Sanjit A. Seshia</i>	
Formally Verified Quite OK Image Format	343
<i>Mario Bucev and Viktor Kunčak</i>	
Split Transition Power Abstraction for Unbounded Safety	349
<i>Martin Blicha, Grigory Fedukovich, Antti Hyvärinen and Natasha Sharygina</i>	
Automating Geometric Proofs of Collision Avoidance with Active Corners	359
<i>Nishant Kheterpal, Elanor Tang and Jean-Baptiste Jeannin</i>	
Differential Testing of Pushdown Reachability with a Formally Verified Oracle	369
<i>Anders Schlichtkrull, Morten Konggaard Schou, Jiri Srba and Dmitriy Traytel</i>	
TriCera: Verifying C Programs Using the Theory of Heaps	380
<i>Zafer Esen and Philipp Ruemmer</i>	

The seL4 Verification Journey: How Have the Challenges and Opportunities Evolved

June Andronick

Proofcraft

Kensington, Australia

june.andronick@proofcraft.systems

Abstract—The formal verification journey of the seL4 microkernel is nearing two decades, and still has an busy roadmap for the years ahead. It started as a research project aiming for a highly challenging problem with the potential of significant impact. Today, a whole ecosystem of developers, researchers, adopters and supporters are part of the seL4 community. With increasing uptake and adoption, seL4 is evolving, supporting more platforms, architectures, configurations, and features. This creates both opportunities and challenges: verification is what makes seL4 unique; as the seL4 code evolves, so must its formal proofs. With more than a million lines of formal, machine-checked proofs, seL4 is the most highly assured OS kernel, with proofs of an increasing number of properties (functional correctness, binary correctness, security—integrity and confidentiality—and system initialisation) and for an increasing number of hardware architectures: Arm (32-bit), x86 (64-bit) and RISC-V (64-bit), with proofs now starting for Arm (64-bit). In this talk we will reflect on the evolution of the challenges and opportunities the seL4 verification faced along its long, and continuing, journey.

Why Do Things Go Wrong (or Right)? Applications of Causal Reasoning to Verification

Hana Chockler
King's College London
London, UK
hana.chockler@kcl.ac.uk

Abstract—In this talk I will look at the connections between causality and learning from one side, and verification and synthesis from the other side. I will introduce the relevant concepts and discuss how causality and learning can help to improve the quality of systems and reduce the amount of human effort in designing and verifying systems. I will (briefly) introduce the theory of actual causality as defined by Halpern and Pearl. This theory turns out to be extremely useful in various areas of computer science due to a good match between the results it produces and our intuition. I will illustrate the definitions by examples from formal verification. I will also argue that active learning can be viewed as a type of causal discovery. Tackling the problem of reducing the human effort from the other direction, I will discuss ways to improve the quality of specifications and will focus in particular on synthesis.

On Applying Model Checking in Formal Verification

Håkan Hjort
Cadence Design Systems
Gothenburg, Sweden
hhjort@cadence.com

Abstract—Use of Hardware model checking in the EDA industry is widespread and now considered an essential part of verification. While there are many papers, and books, about SAT, SMT and Symbolic model checking, often very little is written about how these methods can be applied. Choices made when modeling systems can have large impacts on applicability and scalability. There is generally no formal semantics defined for the hardware design languages, nor for the intermediate representations in common use. As unsatisfactory as it may be, industry conventions and behaviour exhibited by real hardware have instead been the guides. In this tutorial we will give an overview of some of the steps needed to apply hardware model checking in an EDA tool. We will touch on synthesis, hierarchy flattening, gate lowering, driver resolution, issues with discrete/synchronous time models, feedback loops and environment constraints, input rating and initialisation/reset.

Design compilation, also known as elaboration and (quick) *synthesis*, is used to create a gate netlist from a hardware description language, commonly System Verilog. When done for implementation this often leverages any semantic freedom in order to create a more efficient implementation. In contrast, for verification we prefer to preserve all possible behaviour of any valid implementation choice. Assertions (properties) are normally handled similarly and translated to an automata representation that is then implemented by a gate netlist.

The gate netlist is a hierarchical representation of gates and their connections (to wires). Removal of *hierarchy* can largely be done replicating the logic. Most gate types represent combinatorial functions, these can be kept as is, or lowered to smaller subset of gate functions (such as in And-Inverter graphs). The state holding gates, (Flip-)Flops (edge sensitive) and Latches (level sensitive) require some more care to model their (as)synchronous behaviour.

Special care is also needed to model Tri-state gates (and weak drivers), which can either drive a value on their output or hold it isolated. Verilog wire uses a domain with 4-values 0,1,X,Z where Z is high-impedance / not-driving. *Resolving* the drivers means replacing the gates that drive a common wire with a model for the resolved logic value (and possibly checks for invalid/bad combinations).

It is common to have configurations, modes of operation and/or parts that should not be validated. Forcing some inputs to a fixed value is referred to as environment *constraints*. Mode complex constraints are instead normally considered part of the verification setup and handled as SV assumptions. The fixed values can be propagated into the gates to remove parts that become constant or disconnected.

For power and performance reasons it is common that designs are multi-clocked, or that clocks are gated (can be turned off and on). To have a global *synchronous model* for verification we need to reduce these multi-clock systems to a single global system (or tool) clock. This is often handled by mux-feedback added to the flops/latches along with logic generating the condition for the muxes. Inputs to the netlist may also have constraints at which rate/phase they can change. Rated inputs are free to take any value but only at certain points, clock generators follow a periodic pattern.

The use of a zero-delay timing model, meaning combinatorial gate output the function of their inputs without any delay, can give rise to problems when there are feedback loops in the netlist. Causing contradictions when a net would have two (or more) values, had there some delay in propagating the values through gates. There are 5 kinds of loops we can occur, through flops (data and clock), through latches (data and enable) and those only going through combinatorial gates. The ones going through flop data are benign, as its effect is mediated by the clock. The others need to be ruled out, or handled by modeling. Introducing some (fractional-)delay/steps seems an attractive approach, but establishing a bound on the number steps needed is challenging (and for some, no bound exists).

Initialisation, also referred to as reset, is commonly done by applying sequence of values to a subset of inputs. This aims to get the design from an arbitrary unknown state into a set of states from which it will have predictable behaviour. Part of the design flops might have asynchronous reset, others can receive values on the data input from other flops and inputs, yet others might be left uninitialised. Automating the computation of an (over-)approximation of the reset states will provide more information to the constructed model checking problem.

Verification of Distributed Protocols: Decidable Modeling and Invariant Inference

Oded Padon
VMware Research
 Palo Alto, CA, USA
 oded.padon@gmail.com

Verification of distributed protocols and systems, where both the number of nodes in the systems and the state-space of each node are unbounded, is a long-standing research goal. In recent years, efforts around the Ivy verification tool [1]–[4] have pushed a strategy of modeling distributed protocols and systems in a new way that enables decidable deductive verification [5]–[8], i.e., given a candidate inductive invariant, it is possible to *automatically* check if it is inductive, and to produce a *finite* counterexample to induction in case it is not inductive. Complex protocols require quantifiers in both models and their invariants, including forall-exists quantifier alternations. Still, it is possible to obtain decidability by enforcing a stratification structure on quantifier alternations, often achieved using modular decomposition techniques, which are supported by Ivy. Stratified quantifiers lead not only to theoretical decidability, but to reliably good solver performance in practice, which is in contrast to the typical instability of SMT solvers over formulas with complex quantification.

Reliable automation of invariant checking and finite counterexamples open the path to automating invariant inference [9]. An invariant inference algorithm can propose a candidate invariant, automatically check it, and get a finite counterexample that can be used to inform the next candidate. For a complex protocol, this check would typically be performed thousands of times before an invariant is found, so reliable automation of invariant checking is a critical enabler. Recently, several invariant inference algorithms [9]–[18] have been developed that can find complex quantified invariants for challenging protocols, including Paxos and some of its most intricate variants.

In the tutorial I will provide an overview of Ivy’s principles and techniques for modeling distributed protocols in a decidable fragment of first-order logic. I will then survey several recently developed invariant inference algorithms for quantified invariants, and present one such algorithm in depth: Primal-Dual Houdini [13]. Primal-Dual Houdini is based on a new mathematical duality, and is obtained by deriving the formal dual of the well-known Houdini algorithm. As a result, Primal-Dual Houdini possesses an interesting formal symmetry between the search for proofs and for counterexamples.

REFERENCES

- [1] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: Safety verification by interactive generalization,” in *PLDI 2016*. [Online]. Available: <https://doi.org/10.1145/2908080.2908118>
- [2] K. L. McMillan and O. Padon, “Deductive verification in decidable fragments with ivy,” in *SAS 2018*. [Online]. Available: https://doi.org/10.1007/978-3-319-99725-4_4
- [3] —, “Ivy: A multi-modal verification tool for distributed algorithms,” in *CAV 2020*. [Online]. Available: https://doi.org/10.1007/978-3-030-53291-8_12
- [4] K. L. McMillan, “Ivy,” <https://github.com/kenmcil/ivy>.
- [5] O. Padon, G. Losa, M. Sagiv, and S. Shoham, “Paxos made EPR: decidable reasoning about distributed protocols,” *OOPSLA 2017*. [Online]. Available: <https://doi.org/10.1145/3140568>
- [6] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos, “Modularity for decidability of deductive verification with applications to distributed systems,” in *PLDI 2018*. [Online]. Available: <https://doi.org/10.1145/3192366.3192414>
- [7] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, “Reducing liveness to safety in first-order logic,” *POPL 2018*. [Online]. Available: <https://doi.org/10.1145/3158114>
- [8] O. Padon, J. Hoenicke, K. L. McMillan, A. Podelski, M. Sagiv, and S. Shoham, “Temporal prophecy for proving temporal properties of infinite-state systems,” *Formal Methods Syst. Des.*, vol. 57, no. 2, pp. 246–269, 2021. [Online]. Available: <https://doi.org/10.1007/s10703-021-00377-1>
- [9] A. Karbyshev, N. S. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham, “Property-directed inference of universal invariants or proving their absence,” *J. ACM*, vol. 64, no. 1, pp. 7:1–7:33, 2017. [Online]. Available: <https://doi.org/10.1145/3022187>
- [10] Y. M. Y. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv, “Inferring inductive invariants from phase structures,” in *CAV 2019*. [Online]. Available: https://doi.org/10.1007/978-3-030-25543-5_23
- [11] J. R. Koenig, O. Padon, N. Immerman, and A. Aiken, “First-order quantified separators,” in *PLDI 2020*. [Online]. Available: <https://doi.org/10.1145/3385412.3386018>
- [12] J. R. Koenig, O. Padon, S. Shoham, and A. Aiken, “Inferring invariants with quantifier alternations: Taming the search space explosion,” in *TACAS 2022*. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_18
- [13] O. Padon, J. R. Wilcox, J. R. Koenig, K. L. McMillan, and A. Aiken, “Induction duality: primal-dual search for invariants,” *POPL 2022*. [Online]. Available: <https://doi.org/10.1145/3498712>
- [14] H. Ma, A. Goel, J. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, “I4: incremental inference of inductive invariants for verification of distributed protocols,” in *SOSP 2019*. [Online]. Available: <https://doi.org/10.1145/3341301.3359651>
- [15] T. Hance, M. Heule, R. Martins, and B. Parno, “Finding invariants of distributed systems: It’s a small (enough) world after all,” in *NSDI 2021*. [Online]. Available: <https://www.usenix.org/conference/nsdi21/presentation/hance>
- [16] A. Goel and K. A. Sakallah, “On symmetry and quantification: A new approach to verify distributed protocols,” in *NFM 2021*. [Online]. Available: https://doi.org/10.1007/978-3-030-76384-8_9
- [17] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, “DistAI: Data-driven automated invariant learning for distributed protocols,” in *OSDI 2021*. [Online]. Available: <https://www.usenix.org/conference/osdi21/presentation/yao>
- [18] J. Yao, R. Tao, R. Gu, and J. Nieh, “DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols,” in *OSDI 2022*. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/yao>

The FMCAD 2022 Student Forum

Mathias Preiner 
 Stanford University
 preiner@cs.stanford.edu

Abstract—The Student Forum at the International Conference on Formal Methods in Computer-Aided Design (FMCAD) gives undergraduate and graduate students the opportunity to introduce their research to the Formal Methods community and receive feedback. In 2022, the event took place in Trento, Italy. Twenty one students were invited to give a short talk and present a poster of their work.

Since 2013, the FMCAD Student Forum provides a platform for undergraduate and graduate students at any career stage to present their research to the audience of the FMCAD conference. The 2022 edition of the FMCAD Student Forum follows the tradition of its predecessors, which took place in:

- Portland, Oregon, USA in 2013 [1]
- Lausanne, Switzerland in 2014 [2]
- Austin, Texas in 2015 [3] and 2018 [4]
- Mountain View, California, USA in 2016 [5]
- Vienna, Austria in 2017 [6]
- San Jose, California, USA in 2019 [7]
- Virtual in 2020 [8] and 2021 [9]

FMCAD 2022 hosted the tenth edition of the Student Forum. Graduate and undergraduate students were invited to submit two-page reports of their current research and ongoing work in the scope of the FMCAD conference. The Student Forum program committee reviewed 25 submissions out of which 21 were accepted. One submission was withdrawn by the student after acceptance resulting in 20 accepted submissions in total. The reviews were based on the overall quality, novelty of the work, its potential impact on the Formal Methods community, as well as the potential positive impact on the student to have the opportunity to participate in the forum. The accepted submissions covered a wide range of topics relevant to the FMCAD community, from foundational aspects of automated reasoning, to analysis and verification of software, hardware, and neural networks, as well as applications of formal methods to security and biology. The following contributions have been accepted¹:

- Guy Amir: *Verification-Driven Ensemble Selection*
- Levente Bajczi: *Axiomatic Analysis of Distributed Systems*
- Mihály Dobos-Kovács: *Lazy abstraction for time in eager CEGAR*
- Bernhard Gstrein: *Tuning the Learning of Circuit-Based Classifiers*
- Ondřej Huvar: *Symbolic Coloured Model Checking for HCTL*

- Omri Isac: *Proof Production for Neural Network Verification*
- Dominik Klumpp: *Commutativity in Concurrent Program Verification*
- Pankaj Kumar Kalita: *GAMBIT: An Interactive Playground for Concurrent Programs Under Relaxed Memory Models*
- Hanna Lachnitt: *Fine-Grained Reconstruction of cvc5 Proofs in Isabelle/HOL*
- Tobias Paxian: *Trading Accuracy For Smaller Cardinality Constraints*
- Siddharth Priya: *SEAURCHIN: Bounded Model Checking for Rust*
- Sarah Sallinger: *A Formalization of Heisenbugs and Their Causes*
- Tiago Soares: *Formal Verification of Algebraic Effects*
- Dániel Szekeres: *Lazy Abstraction for Probabilistic Systems*
- Csanád Telbisz: *Partial Order Reduction for Abstraction-Based Verification of Concurrent Software*
- Muhammad Usama Sardar: *Understanding Trust Assumptions for Attestation in Confidential Computing*
- Daniella Vo: *Formal Approach to Identifying Genes and Microbes Significant to Inflammatory Bowel Disease*
- Amalee Wilson: *Strategies for Parallel SMT Solving*
- Suwei Yang: *Incremental Weighted Sampling*
- Tom Zelazny: *On Optimizing Back-Substitution Methods for Neural Network Verification*

Unlike previous editions of the FMCAD student forum, which invited a subset of the FMCAD program committee to review student submissions, this year's edition nominated an independent program committee (including some members of the FMCAD PC). The 2022 FMCAD Student Forum program committee consisted of *Mathias Preiner (Chair), Armin Biere, Martin Blicha, Rayna Dimitrova, Rohit Dureja, Mathias Fleury, Aman Goel, Stéphane Graham-Lengrand, Antti Hyvärinen, Ahmed Irfan, Martin Jonáš, Daniela Kaufmann, Daniel Larraz, Makai Mann, Alexander Nadel, Andres Noetzli, Mark Santolucito, Nestan Tsiskaridze, Tom van Dijk, and Florian Zuleger*.

We would like to thank the organizers of FMCAD, as well as the FMCAD Student Forum program committee, who have made the FMCAD Student Forum possible. Additionally, we are grateful to the student authors and their research mentors who have contributed their excellent work to the program.

¹Only first authors listed for brevity.

REFERENCES

- [1] T. Wahl, "The FMCAD graduate student forum," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 16–17. [Online]. Available: <https://doi.org/10.1109/FMCAD.2013.7035523>
- [2] R. Piskac, "The FMCAD 2014 graduate student forum," in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014, p. 13. [Online]. Available: <https://doi.org/10.1109/FMCAD.2014.6987589>
- [3] G. Weissenbacher, "The FMCAD 2015 graduate student forum," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, p. 8.
- [4] D. Jovanovic and A. Reynolds, "The FMCAD 2018 graduate student forum," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. S. Björner and A. Gurfinkel, Eds. IEEE, 2018, p. 1. [Online]. Available: <https://doi.org/10.23919/FMCAD.2018.8602995>
- [5] H. Hojjat, "The FMCAD 2016 graduate student forum," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, p. 8. [Online]. Available: <https://doi.org/10.1109/FMCAD.2016.7886654>
- [6] K. Heljanko, "The FMCAD 2017 graduate student forum," in *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, p. 10. [Online]. Available: <https://doi.org/10.23919/FMCAD.2017.8102234>
- [7] G. Fedyukovich, "The FMCAD 2019 student forum," in *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, C. W. Barrett and J. Yang, Eds. IEEE, 2019, p. 1. [Online]. Available: <https://doi.org/10.23919/FMCAD.2019.8894257>
- [8] P. Schrammel, "The FMCAD 2020 student forum," in *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, p. 1. [Online]. Available: https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_6
- [9] M. Santolucito, "The FMCAD 2021 student forum," in *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, p. 1. [Online]. Available: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_8

Proving Robustness of KNN Against Adversarial Data Poisoning

Yannan Li, Jingbo Wang, and Chao Wang

University of Southern California, Los Angeles CA 90089, USA

{yannanli, jingbow, wang626}@usc.edu

Abstract—We propose a method for verifying data-poisoning robustness of the k -nearest neighbors (KNN) algorithm, which is a widely-used supervised learning technique. Data poisoning aims to corrupt a machine learning model and change its inference result by adding polluted elements into its training set. The inference result is considered n -poisoning robust if it cannot be changed by up-to- n polluted elements. Our method verifies n -poisoning robustness by soundly overapproximating the KNN algorithm to consider all possible scenarios in which polluted elements may affect the inference result. Unlike existing methods which only verify the inference phase but not the significantly more complex learning phase, our method is capable of verifying the entire KNN algorithm. Our experimental evaluation shows that the proposed method is also significantly more accurate than existing methods, and is able to prove the n -poisoning robustness of KNN for popular supervised-learning datasets.

I. INTRODUCTION

Data poisoning is an attack aimed to corrupt a machine learning model by polluting its training data, and thus affect the inference results for test data [33]. Prior work shows that even a small amount of polluted data, e.g., $\leq 0.4\%$ of the training set, is enough to affect the inference result [34], [6], [8]. Thus, verifying the robustness of the inference result in the presence of data poisoning is a practically important problem. Specifically, given a potentially-polluted training set T , and the assumption that at most n elements in T are polluted, if we can prove that the inference result for a test input x remains unchanged by any n polluted elements in T , the inference result can still be considered trustworthy.

This work is concerned with n -poisoning robustness of the k -nearest neighbors (KNN) algorithm, which is a widely used supervised learning technique in applications such as e-commerce, video recommendation, document categorization, and anomaly detection [18], [2], [41], [1], [30], [14], [27], [36], [44]. However, the verification problem is challenging for two reasons. First, KNN relies heavily on numerical analysis, which involves a large number of non-linear arithmetic computations and complex statistical analysis techniques such as p -fold cross validation. They are known to be difficult for existing verification techniques. Second, even with a small n , there can be an extremely large number of possible scenarios in which polluted elements in T may affect the trained model and hence the inference result.

Specifically, let $m = |T|$ be the number of elements in T and $i \leq n$ be the actual number of polluted elements in T , the number of clean subsets of T (where polluted elements have

been removed) is $\binom{m}{i}$. Since $i = 1, \dots, n$, the total number of clean subsets of T is $\sum_{i=0}^n \binom{m}{i}$. Thus, it is impractical to explicitly check, for each clean subset $T' \subseteq T$, whether the inference result produced by the model trained using T' remains the same as the inference result produced by the model trained using T .

A practical approach, which is the one used by our method, is to soundly over-approximate the impact of all the clean subsets while analyzing the machine learning algorithm, following the abstract interpretation [9] paradigm for static program analysis. Here, the word soundly means that our method guarantees that, as long as the over-approximated inference result is proved robust, the actual inference result is robust. In addition to being sound, our method is efficient in that, instead of training a model for each clean subset T' , it combines all clean subsets together to compute a set of abstract models in a single pass.

For KNN, in particular, each model corresponds to an optimal value of the parameter K , indicating how many neighbors in T are used to infer the output label of a test input x . Thus, our method computes an over-approximated set of K values, denoted $KSet$. Then, it over-approximates the KNN's inference phase, to check if the output label of x remains the same for all $K \in KSet$. If the output label remains the same, the inference result for x is considered robust against any of the possible n -poisoning attacks of the training set T .

To the best of our knowledge, our method is the first method that can soundly verify n -poisoning robustness of the entire KNN algorithm, consisting of both the learning (K parameter tuning) phase and the inference phase. In the literature, there are two closely related prior works. The first one, by Jia et al. [21], aims to verify the robustness of KNN's inference phase only; in other words, they require the K value to be fixed and given, with the implicit assumption that the optimal K value is not affected by data poisoning. Unfortunately, this is not a valid assumption, as shown by the motivating examples presented in Section II. Furthermore, by fixing the K value, the more challenging part of the verification problem has been sidestepped, which is verifying the p -fold cross validation during KNN's learning phase. How to over-approximate KNN's learning phase soundly and efficiently is a main contribution of our work.

The other closely-related prior work, by Drews et al. [12], aims to prove robustness of a different machine learning technique, namely the decision tree learning (DTL) algo-

rithm. Since DTL differs significantly from KNN in that it relies primarily on logical operations (such as And, Or, and Negation) as opposed to nonlinear arithmetic computations, their verification method relies on a fundamentally different technique (symbolic path exploration) from ours, and is not directly applicable to KNN.

At a high level, our verification method works as follows. Given a tuple $\langle T, n, x \rangle$, where T is the potentially-polluted training set, n is the maximum number of polluted elements in T , and x is a test input, our method tries to prove that, no matter which of the $i \leq n$ elements in T are polluted, the KNN’s inference result for x remains the same. By default, the training set T corresponds to a model M , whose inference result for x is $y = M(x)$. Using an overapproximated analysis, our method checks if the output label $y' = M'(x)$ produced by a model M' corresponding to any clean subset of $T' \subseteq T$ remains the same as the default label $y = M(x)$. If that is the case, our method verifies the robustness of the inference result. Otherwise, it remains inconclusive.

We have implemented our method and conducted experimental evaluation using six popular machine learning datasets, which include both small and large datasets. The small datasets are particularly useful in evaluating the accuracy of the verification result because, when datasets are small, even the baseline approach of explicitly enumerating all clean subsets $T' \subseteq T$ is fast enough to complete and obtain the ground truth. The large datasets, some of which have more than 50,000 training data elements and thus are well beyond the reach of the baseline enumeration approach, are useful in evaluating the efficiency of our method. For comparison, we also evaluated the method of Jia et al. [21] with fixed K values.

Our experimental results show that, for KNN’s inference phase only, our method is significantly more accurate than the method of Jia et al. [21] and as a result, proves robustness for many more cases. Overall, our method is able to achieve similar empirical accuracy as the ground truth on small datasets, while being reasonably accurate on large datasets and several orders-of-magnitudes faster than the baseline method. In particular, our method is the only one that can finish the complete verification of 10,000 test inputs for a training dataset with more than 50,000 elements within half an hour.

To summarize, this paper has the following contributions:

- We propose the first method for soundly verifying data-poisoning robustness of the entire KNN algorithm, consisting of both the learning phase and the inference phase.
- We evaluate the method on popular supervised learning datasets to demonstrate its advantages over both the baseline and a state-of-the-art technique.

The remainder of this paper is organized as follows. First, we review the definition of n -poisoning robustness and the basics of the k -nearest neighbors (KNN) algorithm in Section II. Then, we present the intuition and overview of our method in Section III. Next, we present our method for verifying the KNN learning phase in Section IV and verifying the KNN inference phase in Section V. We present our experimental

results in Section VI, review the related work in Section VII, and give our conclusions in Section VIII.

II. BACKGROUND

A. Data-Poisoning Robustness

Let L be a supervised learning algorithm that takes a set $T = \{(x, y)\}$ of training data elements as input and returns a learned model $M = L(T)$ as output. Within each data element, input $x \in \mathcal{X} \subseteq \mathbb{R}^D$ is an D -dimensional real-valued feature vector, and output $y \in \mathcal{Y} \subseteq \mathbb{N}$ is a natural number that represents a class label. The model is a prediction function $M : \mathcal{X} \rightarrow \mathcal{Y}$ that maps a test input $x \in \mathcal{X}$ to its class label $y \in \mathcal{Y}$. Following Drews et al. [12], we define data-poisoning robustness as follows.

a) *n-Poisoning Model*: Let T be a potentially-polluted training set, $m = |T|$ be the total number of elements in T , and n be the maximum number of polluted elements in T . Assuming that we do not know *which elements in T are polluted*, the set of all possible scenarios is captured by the set of *clean subsets*, denoted $\Delta_n(T) = \{T' \subseteq T : |T \setminus T'| \leq n\}$. In other words, each T' may be the result of removing all of the polluted elements from T .

b) *n-Poisoning Robustness*: We say the inference result $y = M(x)$ for a test input $x \in \mathcal{X}$ is robust to n -poisoning attacks of T if and only if, for all $T' \in \Delta_n(T)$ and the corresponding model $M' = L(T')$, we have $M'(x) = M(x)$. In other words, the predicted label remains the same.

For example, when $T = \{a, b, c, d\}$ and $n = 1$, the clean subsets are $T_1 = \{b, c, d\}$, $T_2 = \{a, c, d\}$, $T_3 = \{a, b, d\}$ and $T_4 = \{a, b, c\}$, which correspond to models $M_1 - M_4$ and inference results $x_1 = M_1(x)$, $x_2 = M_2(x)$, $x_3 = M_3(x)$ and $x_4 = M_4(x)$. Let M be the default model obtained by T and $x = M(x)$ be the default output label. The inference result is 1-poisoning robust if and only if $x_1 = x_2 = x_3 = x_4 = x$.

This robustness definition has two advantages. First, whenever the inference result for a test input x is proved to be robust, it provides a strong guarantee of trustworthiness. Second, the verification procedure does not require the actual label of x to be known, which means it is applicable to unlabeled test data, which are common in practice.

B. k -Nearest Neighbors (KNN)

KNN is a supervised learning algorithm with two phases. During the learning phase, the training set T is used to compute the optimal value of the parameter K , which indicates how many neighbors in T to consider when deciding the output label for a test input x . During the inference phase, given an unlabeled test input $x \in \mathcal{X}$, the K nearest neighbors of x in T are used to compute the most frequent label, which is returned as the output label of x .

The distance between data elements, which is used to find the nearest neighbors of x in T , is defined on the input feature vectors. The most widely used metric is the Euclidean distance: given two elements $x_a, x_b \in \mathcal{X} \subseteq \mathbb{R}^D$, where D is the dimension of the input feature vector, the Euclidean distance is $\sqrt{\sum_{i=1}^D (x_a[i] - x_b[i])^2}$.

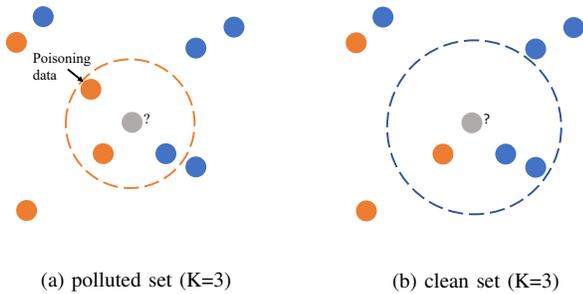


Fig. 1. Example of *direct influence* of the polluted data.

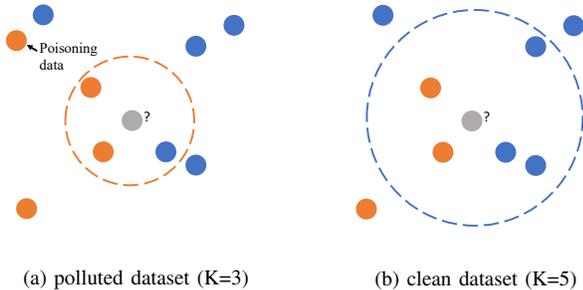


Fig. 2. Example of *indirect influence* of polluted data.

The optimal K value is the one that has the smallest average misclassification error on the training set T . The misclassification error is computed using p -fold *cross validation*, which randomly divides T into p groups of approximately equal size and, for each group, compute the misclassification error by treating this group as the test set and the union of all the other $p - 1$ groups as the training set. Finally, the misclassification errors of the individual groups are used to compute the average misclassification error among all p groups.

III. THE INTUITION AND OVERVIEW OF OUR METHOD

We first present the intuition behind our method, and then give an overview of the method in contrast to the baseline.

A. Two Ways of Affecting the Inference Result

In general, there are two ways in which polluted training elements in T affect the inference result. One of them, called *direct influence*, is to change the neighbors of x and thus their most frequent label. The other one, called *indirect influence*, is to change the parameter K itself.

Fig. 1 shows how polluted data may change the test input’s neighbors and thus the inference result. Here, the gray dot represents the test input x , while the orange and blue dots represent elements in the training set T . There is only one polluted element, which is an orange dot marked in Fig. 1 (a). This element no longer exists in Fig. 1 (b). Assume that the optimal value for the parameter K is 3. For the clean set shown in Fig. 1 (b), the result is ‘blue’ since two of the three nearest neighbors of the test input x are blue. For the polluted set shown in Fig. 1 (a), however, the result is ‘orange’ since two of the three nearest neighbors are orange.

Fig. 2 shows how polluted data may change the inference result by changing the optimal value of the parameter K . In this case, the polluted element in Fig. 2 (a) is far away from the test input x . However, its presence changes the optimal value of the parameter K during the p -fold cross validation phase. While the K value for the clean set is 5, the K value for the polluted set is 3. As a result, the most frequent label of the neighbors is changed from ‘blue’ in Fig. 2 (b) to ‘orange’ in Fig. 2 (a).

These two examples highlight the importance of analyzing both the learning phase and the inference phase of the KNN algorithm. Otherwise, the verification result may be unsound, which is the case for Jia et al. [21] due to their implicit (and incorrect) assumption that K is not affected by polluted elements in T . In contrast, our method soundly verifies both phases of the KNN algorithm.

While verifying the KNN inference phase itself is already challenging, verifying the KNN learning phase is even more challenging, since it uses p -fold cross validation to compute the optimal K value.

B. Overview of Our Method

Before presenting our method, we present a conceptually-simple, but computationally-expensive, baseline method. It will help explain why the verification problem is challenging.

Algorithm 1: Baseline method $\text{KNN_Verify}(T, n, x)$.

```

for each  $T' \in \Delta_n(T)$  do
   $K' \leftarrow \text{KNN\_learn}(T')$ 
   $y' \leftarrow \text{KNN\_predict}(T', K', x)$ 
   $YSet \leftarrow YSet \cup \{y'\}$ 
end
 $robust \leftarrow (|YSet| = 1)$ 

```

a) *The Baseline Method*: This method relies on checking whether the inference result remains the same for all possible ways in which the training set is polluted. Algorithm 1 shows the pseudo code, where T is the training set, n is the maximal polluted number, and x is a test input. For each clean subset $T' \in \Delta_n(T)$, the parameter K is computed using the standard KNN_learn subroutine, and used to predict the label of x using the standard KNN_predict subroutine. Here, $YSet$ stores the set of predicted labels; thus, $|YSet| = 1$ means the prediction result is always the same (and hence robust).

The baseline method is both sound and complete, and thus may be used to obtain the ground truth when the size of the dataset is small enough. However, it is not a practical solution for large datasets because of the combinatorial blowup – it has to explicitly enumerate all $|\Delta_n(T)| = \sum_{i=0}^n \binom{m}{i}$ cases. Even for $m = 100$ and $n = 5$, for example, the number becomes as large as $8 * 10^7$. For realistic datasets, often with tens of thousands of elements, the baseline method would not finish in a billion years.

b) *The Proposed Method*: Our method avoids enumerating the individual scenarios in $\Delta_n(T)$. As shown in Algorithm 2, it first analyzes, in a single pass, the KNN’s learning phase while simultaneously considering the impact of up-to- n

Algorithm 2: Our method $\text{abs_KNN_Verify}(T, n, x)$.

```
 $KSet \leftarrow \text{abs\_KNN\_learn}(T, n)$   
 $YSet \leftarrow \text{abs\_KNN\_predict}(T, n, KSet, x)$   
 $robust \leftarrow (|YSet| = 1)$ 
```

Algorithm 3: Subroutine for the baseline: $\text{KNN_learn}(T)$.

```
Divide  $T$  into  $p$  groups  $\{G_i\}$  of equal size;  
for each  $K \in \text{CandidateKset}$  do  
  for each group  $G_i$  do  
     $errCnt_i^K = 0$   
    for each sample  $(x, y) \in G_i$  do  
       $errCnt_i^K ++$  when  
         $(\text{KNN\_predict}(T \setminus G_i, K, x) \neq y)$ ;  
     $error_i^K = errCnt_i^K / |G_i|$   
   $error^K = \frac{1}{p} \sum_{i=1}^p error_i^K$   
return the  $K$  value with the smallest  $error^K$ 
```

polluted elements in T . The result of this over-approximated analysis is a superset of possibly-optimal K values, stored in $KSet$. Details of the subroutine abs_KNN_learn is presented in Section IV.

Then, for each $K \in KSet$, our method analyzes the KNN's inference phase while considering all possible ways in which up-to- n elements in T may have been polluted. The result of this over-approximated analysis is a superset of possible output labels, denoted $YSet$. We say the inference result for x is robust if the cardinality of $YSet$ is 1; that is, the label of x remains the same regardless of how T may have been polluted. Details of the subroutine abs_KNN_predict is presented in Section V.

IV. ANALYZING THE KNN LEARNING PHASE

To understand why soundly analyzing the KNN learning phase is challenging, we need to compare our method with the the original subroutine, KNN_learn , shown in Algorithm 3, which computes the optimal K value using p -fold cross-validation. Note that both the value of p and the CandidateKset are hyper-parameters of the KNN algorithm itself, not part of the verification method. In practice, they typically do not depend on the size of T (see Section II-B for a detailed explanation).

A. The Algorithm

In contrast, our method shown in Algorithm 4 computes an over-approximated set of K values. The input consists of the training set T and the maximal polluted number n , while the output $KSet$ is a superset of the optimal K values.

Inside Algorithm 4, our method first computes the lower and upper bounds of the misclassification error for each K value, by considering the best case ($errorLB^K$) and the worst case ($errorUB^K$) when up-to- n elements in T are polluted.

After computing the interval $[errorLB^K, errorUB^K]$ for each K value, it computes $minUB$, which is the minimal upper bound among all K values.

Algorithm 4: Subroutine $KSet = \text{abs_KNN_learn}(T, n)$.

```
Divide  $T$  into  $p$  groups  $\{G_i\}$  of equal size;  
for each  $K \in \text{CandidateKset}$  do  
  for each group  $G_i$  do  
     $errCntLB_i^K = errCntUB_i^K = 0$ ;  
    for each sample  $(x, y) \in G_i$  do  
       $errCntLB_i^K ++$  if  
         $(\text{abs\_KNN\_cannot\_obtain\_correct\_label}(T \setminus$   
           $G_i, n, K, x, y) == \text{True})$ ;  
       $errCntUB_i^K ++$  if  
         $(\text{abs\_KNN\_may\_obtain\_wrong\_label}(T \setminus$   
           $G_i, n, K, x, y) == \text{True})$ ;  
     $errorLB_i^K = \max\{0, (errCntLB_i^K - n) / (|G_i| - n)\}$ ;  
     $errorUB_i^K = \min\{errCntUB_i^K / (|G_i| - n), 1\}$ ;  
   $errorLB^K = \frac{1}{p} \sum_{i=1}^p errorLB_i^K$ ;  
   $errorUB^K = \frac{1}{p} \sum_{i=1}^p errorUB_i^K$ ;  
Let  $minUB =$  the smallest  $errorUB^K$  for all  $K$ ;  
 $KSet = \{K \mid errorLB^K \leq minUB\}$ ;
```

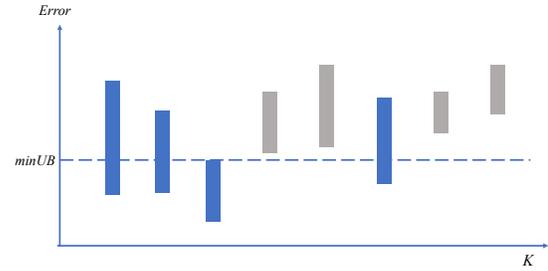


Fig. 3. Example of comparing the error bounds.

Then, by comparing $minUB$ with the $errorLB^K$ for each K , it over-approximates the set of possible K values that may become the optimal K value for some $T' \in \Delta_n(T)$.

Here, the intuition is that, by excluding K values that are *definitely not the optimal* K for any $T' \in \Delta_n(T)$ — they are the ones whose $errorLB^K$ is larger than $minUB$ — we obtain a sound over-approximation in $KSet$.

a) *Example for $minUB$:* Fig. 3 shows an example, where each vertical bar represents the interval $[errorLB^K, errorUB^K]$ of a candidate K value, and the blue dashed line represents $minUB$. The selected K values are those corresponding to the blue bars, since their $errorLB^K$ are smaller than $minUB$. The K values corresponding to the gray bars are dropped, since they definitely cannot have the smallest misclassification error.

b) *The Soundness Guarantee:* To understand why the $KSet$ computed in this manner is an over-approximation, assume that $minUB = errorUB^{K'}$ for some value K' . We now explain why K cannot be the optimal value (with the smallest error) when $errorLB^K > minUB$. Let the actual errors be $error^K \in [errorLB^K, errorUB^K]$ and $error^{K'} \in [errorLB^{K'}, errorUB^{K'}]$. Since we have $errorLB^K > errorUB^{K'}$, we know $error^K$ must be larger than $error^{K'}$. Therefore, K cannot have the smallest error.

To compute the interval $[errorLB^K, errorUB^K]$, we add up the misclassification error for each element $(x, y) \in G_i$,

where $x \in \mathcal{X}$ is the input and $y \in \mathcal{Y}$ is the (correct) label. For each element (x, y) , there is a misclassification error if, for some reason, y differs from the predicted label.

Here, $errCntLB_i^K$ corresponds to the best case scenario — removing n elements from T in such a way that prediction becomes *as correct as possible*. In contrast, $errCntUB_i^K$ corresponds to the worst case scenario — removing n elements from T in such a way that prediction becomes *as incorrect as possible*. These two error counts are computed by two subroutines, which will be presented later in this section.

To convert $errCntLB_i^K$ and $errCntUB_i^K$ to error rates, we consider removing n misclassified elements when computing the lower bound $errorLB_i^K$, and removing n correctly-classified data elements when computing the upper bound $errorUB_i^K$. We assume $n < |G_i|$, which is a reasonable assumption in practice.

To explain subroutines `abs_cannot_obtain_correct_label` and `abs_may_obtain_wrong_label`, we need to introduce some notations, including label counter and removal strategy.

B. The Label Counter

Nearest Neighbors T_x^K . Let T_x^K be a subset of T consisting of the K nearest neighbors of x . For example, given $T = \{(0.1, 0.1), l_2\}, \{(1.1, 0.1), l_1\}, \{(0.1, 1.1), l_1\}, \{(2.1, 3.1), l_3\}, \{(3.3, 3.1), l_3\}\}$, test input $x = (1.1, 1.1)$, and $K = 3$, the set is $T_x^3 = \{(0.1, 0.1), l_2\}, \{(1.1, 0.1), l_1\}, \{(0.1, 1.1), l_1\}$. Here, we assume each neighbor has two real-valued input features and three possible output class labels $l_1 - l_3$.

Label Counter $\mathcal{E}(T_x^K)$. Given any dataset Z , including T_x^K , we use $\mathcal{E}(Z) = \{(l_i : \#l_i)\}$ to represent the label counts, where l_i is a class label, and $\#l_i \in \mathbb{N}$ is the number of elements in Z that have the label l_i . For example, given T_x^3 above, we have $\mathcal{E}(T_x^3) = \{(l_1 : 2), (l_2 : 1)\}$, meaning it has two elements with label l_1 and one with label l_2 .

Most Frequent Label $Freq(\mathcal{E}(T_x^K))$. Given a label counter \mathcal{E} , the most frequent label, denoted $Freq(\mathcal{E})$, is the label with the largest count. Similarly, we can define the second most frequent label. Thus, the KNN inference phase can be described as computing $Freq(\mathcal{E}(T_x^K))$ for the training set T , test input x , and K value.

Tie-Breaker $\mathbb{1}_{(l_i < l_j)}$. If two labels have the same frequency, the KNN algorithm may use their lexicographic order as a tie-breaker to ensure that $Freq(\mathcal{E})$ is unique: Let $<$ be the order relation, $(l_i < l_j)$ must be either true or false. Thus, we define an indicator function, $\mathbb{1}_{(l_i < l_j)}$, to return the numerical value 1 (or 0) when $(l_i < l_j)$ is true (or false).

C. The Removal Strategy

The removal strategy is an abstract way of modeling the impact of polluted data elements. In contrast, the removal set is a concrete way of modeling the impact.

The Removal Set. Given a dataset Z , the removal set $R \subset Z$ can be any subset of Z . Given T_x^3 above, for example, there are 6 possible removal sets: $R_1 = \{(x_1, y_1)\}$, $R_2 = \{(x_2, y_2)\}$, $R_3 = \{(x_3, y_3)\}$, $R_4 = \{(x_1, y_1), (x_2, y_2)\}$,

$R_5 = \{(x_1, y_1), (x_3, y_3)\}$, and $R_6 = \{(x_2, y_2), (x_3, y_3)\}$. In particular, R_1 means removing element (x_1, y_1) from Z .

The Removal Strategy. The removal strategy is simply the label counter of a removal set R , denoted $\mathcal{S} = \mathcal{E}(R)$. In the above example, the six removal sets correspond to only four removal strategies $\mathcal{S}_1 = \{(l_1 : 1)\}$, $\mathcal{S}_2 = \{(l_2 : 1)\}$, $\mathcal{S}_3 = \{(l_1 : 1), (l_2 : 1)\}$, and $\mathcal{S}_4 = \{(l_1 : 2)\}$. In particular, \mathcal{S}_2 means removing an element labeled l_2 ; however, it does not say which of the l_2 elements is removed. Thus, it captures any removal set that has the same label counter.

The Strategy Size. Let the removal strategy be denoted $\mathcal{S} = \{(l_i : \#l_i)\}$, we define the size as $\|\mathcal{S}\| = \sum_{(l_i, \#l_i) \in \mathcal{S}} \#l_i$ — it is the total number of removed elements. For $\mathcal{S}_1 = \{(l_1 : 1)\}$, $\mathcal{S}_2 = \{(l_2 : 2)\}$, and $\mathcal{S}_3 = \{(l_1 : 1), (l_3 : 3)\}$, the strategy size would be $\|\mathcal{S}_1\| = 1$, $\|\mathcal{S}_2\| = 2$, and $\|\mathcal{S}_3\| = 4$.

In the context of the abstract interpretation paradigm [9], the removal sets can be viewed as the *concrete domain* while the removal strategies can be viewed as the *abstract domain*. Focusing on the abstract domain during verification makes our method more efficient. Let $|\mathcal{L}|$ be the total number of class labels, which is often small in practice (e.g., 2 or 10). Since the count of each label in a removal set is at most n , the number of removal strategies is at most $\sum_{i=0}^n \binom{|\mathcal{L}|-1}{i}$. This can be exponentially smaller than the number of possible removal sets, which is $\sum_{i=0}^n \binom{|T|}{i}$.

D. Misclassification Error Bounds

Using the notations defined so far, we present our method for computing the lower and upper bounds, $errCntLB_i^K$ and $errCntUB_i^K$, as shown in Algorithms 5 and 6.

Both bounds rely on computing T_x^{K+n} , the $K+n$ neighbors of x in T , and the label counter $\mathcal{E}(T_x^{K+n})$.

- The first subroutine checks whether it is impossible, even after removing up-to- n elements from T , that the correct label y becomes the most frequent label.
- The second subroutine checks whether it is possible, after removing up-to- n elements from T , that some wrong label becomes the most frequent label.

Before explaining the details, we present Theorem 1, which states the correctness of these checks. It says that, to model the impact of all subsets $T' \in \Delta_n(T)$, we only need to analyze the $(K+n)$ nearest neighbors of x , stored in T_x^{K+n} .

Theorem 1 $\forall T' \in \Delta_n(T)$, we have $Freq(\mathcal{E}((T')^K_x)) \in \{Freq(\mathcal{E}(T_x^{K+n}) \setminus \mathcal{S}) \mid \mathcal{S} \subset \mathcal{E}(T_x^{K+n}), \|\mathcal{S}\| \leq n\}$.

For brevity, we omit the detailed proof. Instead, we give the intuition behind the proof as follows:

- For each clean training subset $T' \in \Delta_n(T)$, we can always find a label counter $\mathcal{E}(T_x^{K+i})$ and a removal strategy $\mathcal{S} \in \mathcal{E}(T_x^{K+i})$, where $\|\mathcal{S}\| = i \leq n$, satisfying $\mathcal{E}(T_x^{K+i} \setminus \mathcal{S}) = \mathcal{E}((T')^K_x)$.
- If we want to check all the predicted labels of x generated by all $T' \in \Delta_n(T)$, we need to search through all of $\mathcal{E}(T_x^K)$, $\mathcal{E}(T_x^{K+1})$, \dots , $\mathcal{E}(T_x^{K+n})$, which is expensive when n is large.

Algorithm 5: Subroutine used in our Algorithm 4 $flag = \text{abs_KNN_cannot_obtain_correct_label}(T, n, K, x, y)$.

Let $\mathcal{E}(T_x^{K+n})$ be the label counter of T_x^{K+n} ;
 Define removal strategy $\mathcal{S} = \{ (y' : \#y' - \#y + \mathbb{1}_{y' < y}) \mid (y' : \#y') \in \mathcal{E}(T_x^{K+n}), y' \neq y, \#y' \geq \#y \}$;
return $(\|\mathcal{S}\| > n)$;

Algorithm 6: Subroutine used in our Algorithm 4 $flag = \text{abs_KNN_may_obtain_wrong_label}(T, n, K, x, y)$.

Let $\mathcal{E}(T_x^{K+n})$ be the label counter of T_x^{K+n} ;
 Let y' be the most frequent label in $\mathcal{E}(T_x^{K+n})$ except the label y ;
 Define removal strategy
 $\mathcal{S} = \{ (y' : \max\{0, \#y - \#y' + \mathbb{1}_{y < y'}\}) \}$;
return $(\|\mathcal{S}\| \leq n)$;

- Fortunately, $\mathcal{E}(T_x^{K+n}) \setminus \mathcal{S}$, where $\|\mathcal{S}\| \leq n$, contains all the possible scenarios denoted by $\mathcal{E}(T_x^{K+i}) \setminus \mathcal{S}$, where $\|\mathcal{S}\| = i$ and $i = 0, \dots, n-1$.

As a result, we only need to analyze $\mathcal{E}(T_x^{K+n})$, which corresponds to the $(K+n)$ nearest neighbors of x ; other elements which are further away from x can be safely ignored.

E. Algorithm 5

To compute the lower bound errCntLB_i^K , Algorithm 5 checks if all the strategies \mathcal{S} satisfying $\text{Freq}(\mathcal{E}(T_x^{K+n}) \setminus \mathcal{S}) = y$ and $\mathcal{S} \subset \mathcal{E}(T_x^{K+n})$ must have $\|\mathcal{S}\| > n$.

Fig. 4 shows two examples. In each example, the gray dot is the test input x and the other dots are neighbors of x in T_x^{K+n} . In Fig. 4 (a), $\#orange = 2$ is the number of orange dots (votes of the correct label). In contrast, $\#blue = 5$ and $\#green = 2$ are votes of the incorrect labels. By assuming the lexicographic order $blue < green < orange$, we define the indicator functions (tie-breakers) as $\mathbb{1}_{blue < orange} = 1$ and $\mathbb{1}_{green < orange} = 1$.

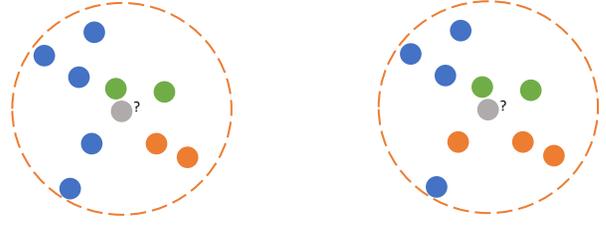
Given the removal strategy $\mathcal{S} = \{(blue : 4), (green : 1)\}$, we know $\|\mathcal{S}\| = 5$ and, since $n = 4$, we have $\|\mathcal{S}\| > n$. Thus, removing up to $n = 4$ dots cannot make the test input x correctly classified (as orange). As a result, $\text{errCntLB}_i^K ++$ is executed to increase the lower bound.

In Fig. 4 (b), however, since $\#blue = 4$, $\#orange = 3$, $\mathbb{1}_{blue < orange} = 1$, and $\mathcal{S} = \{(blue : 2)\}$, we have $\|\mathcal{S}\| = 2$. Since $\|\mathcal{S}\| \leq n$, removing up to $n = 4$ dots can make the test data x correctly classified (as orange). As a result, $\text{errCntLB}_i^K ++$ is not executed.

F. Algorithm 6

To compute the upper bound errCntUB_i^K , Algorithm 6 checks if there exists a strategy \mathcal{S} that satisfies the condition: $\text{Freq}(\mathcal{E}(T_x^{K+n}) \setminus \mathcal{S}) \neq y$, $\mathcal{S} \subset \mathcal{E}(T_x^{K+n})$, and $\|\mathcal{S}\| \leq n$.

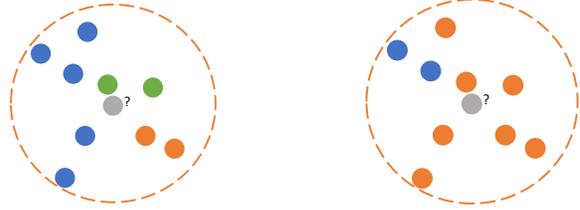
Fig. 5 shows two examples. In Fig. 5 (a), $\#orange = 2$ is the number of correct label, and $\#blue = 5$ is the number of dots with the most frequent wrong label. Thus, $\mathcal{S} = \emptyset$ and since $\|\mathcal{S}\| \leq n$, we know that removing up to $n = 4$ dots can make the test data misclassified. As a result, $\text{errCntUB}_i^K ++$ is executed.



(a) $\mathcal{S} = \{(blue : 4), (green : 1)\}$
and return value is *true*.

(b) $\mathcal{S} = \{(blue : 2)\}$ and return value is *false*.

Fig. 4. Examples for Algorithm 5 with $K = 5$, $n = 4$, and $y = orange$ being the correct label.



(a) $\mathcal{S} = \emptyset$ and return value is *true*.

(b) $\mathcal{S} = \{(orange : 5)\}$ and return value is *false*.

Fig. 5. Example for Algorithm 6 with $K = 5$, $n = 4$, $y = orange$ as correct label, and $y' = blue$ as the most frequent wrong label.

In Fig. 5 (b), $\#orange = 7$ is the number of orange dots, $\#blue = 2$ is the number of dots with the most frequent wrong label. Here, we assume $\mathbb{1}_{orange < blue} = 0$. Thus, $\mathcal{S} = \{(orange : 5)\}$ and since $\|\mathcal{S}\| > n$, we know that removing up to $n = 4$ dots cannot make ‘blue’ (or any other wrong label) the most frequent label. As a result, $\text{errCntUB}_i^K ++$ is not executed.

V. ANALYZING THE KNN INFERENCE PHASE

In this section, we present our method for analyzing the KNN inference phase, implemented in Algorithm 2 as the subroutine $YSet = \text{abs_KNN_predict}(T, n, KSet, x)$, which returns a set of output labels for test input x , by assuming that T contains up-to- n polluted elements.

A. Computing the Classification Labels

Algorithm 7 shows our method, which first checks whether the second most frequent label (y') can become the most frequent one after removing at most n elements. This is possible only if there exists a strategy \mathcal{S} such that (1) it removes at most n elements labeled y , and (2) after the removal, y' becomes the most frequent label. This is captured by the condition $\|\mathcal{S}\| = (\#y - \#y' + \mathbb{1}_{y < y'}) \leq n$. Otherwise, the predicted label is not unique.

We do not attempt to compute more than two labels, as shown by the return statement in the then-branch, because they are not needed by the top-level procedure (Algorithm 2), which only needs to check if $|YSet| = 1$ for the purpose of proving n -poisoning robustness.

Algorithm 7: Method `abs_KNN_predict($T, n, KSet, x$)`.

```
 $YSet = \{ \}$ 
 $visited = \{ \}$ 
while  $\exists K \in (KSet \setminus visited)$  do
  Let  $\mathcal{E}(T_x^{K+n})$  be the label counter of  $T_x^{K+n}$ ;
  Let  $y$  be the most frequent label of  $\mathcal{E}(T_x^{K+n})$ ;
  Let  $y'$  be the second most frequent label of  $\mathcal{E}(T_x^{K+n})$ ;
  Let removal strategy  $\mathcal{S} = \{ (y : \#y - \#y' + \mathbb{1}_{y < y'}) \}$ ;
  if  $|\mathcal{S}| \leq n$  then
     $YSet = YSet \cup \{y, y'\}$ ;
    return  $YSet$ ;
  else
     $YSet = YSet \cup \{y\}$ ;
     $K^{LB} = K - (\#y - \#y' - n - \mathbb{1}_{y' < y})$ ;
     $K^{UB} = K + (\#y - \#y' - n - \mathbb{1}_{y' < y})$ ;
     $visited = visited \cup [K^{LB}, K^{UB}]$ ;
return  $YSet$ ;
```

B. Pruning Redundant K Values

Inside Algorithm 7, after checking $K \in KSet$, our method puts K into the *visited* set to make sure it will never be checked again for the same test input x . In addition, it identifies other values in $KSet$ that are guaranteed to be *equivalent to K* , and prunes away these redundant values. Here, equivalent K values are defined as those with the same inference result for test input x .

To be conservative, we underapproximate the set of equivalent K values. As a result, these K values can be safely skipped since the (equivalent) inference result has been checked. This optimization is implemented using the *visited* set in Algorithm 7. The *visited* set is computed from K and $\mathcal{E}(T_x^{K+n})$ based on the expression $(\#y - \#y' - n - \mathbb{1}_{y' < y})$ over the removal strategy.

a) *The Correctness Guarantee:* We now explain why this pruning technique is safe. The intuition is that, if the most frequent label $Freq(\mathcal{E}(T_x^{K+n}))$ is the label with significantly more counts than the second most frequent label, then it may also be the most frequent label for another value K' . There are two possibilities:

- If $(K' < K)$, then $T_x^{K'+n}$ has $(K - K')$ fewer elements than T_x^{K+n} . Since removing elements from the neighbors will not increase the label count $\#y'$, the only way to change the inference result is decreasing the label count $\#y$. When $(K - K') \leq (\#y - \#y' - n - \mathbb{1}_{y' < y})$, decreasing $\#y$ will not make any difference. Thus, the lower bound of K' is $K - (\#y - \#y' - n - \mathbb{1}_{y' < y})$.
- If $(K' > K)$, then $T_x^{K'+n}$ has $(K' - K)$ more elements than T_x^{K+n} . Since adding elements to the neighbors will not decrease the label count $\#y$, the only way to change the inference result is increasing the label count $\#y'$. However, as long as $(K' - K) \leq (\#y - \#y' - n)$, increasing $\#y'$ will not make any difference. Thus, the upper bound of K' is $K + (\#y - \#y' - n - \mathbb{1}_{y' < y})$.

For example, consider $K = 13$, $n = 2$, and $\mathcal{E}(T_x^{15}) = \{(l_1 : 12), (l_2 : 2), (l_3 : 1)\}$. According to Algorithm 7, $\#y - \#y' - n - \mathbb{1}_{y' < y} = 12 - 2 - 2 = 8$ and thus we compute the interval

TABLE I
STATISTICS OF THE SUPERVISED LEARNING DATASETS.

Name	# training data ($ T $)	# test data ($ XSet $)	# output label (\mathcal{L})	# input dimension (D)
Iris [15]	135	15	3	4
Digits [17]	1,617	180	10	64
HAR [3]	9,784	515	6	561
Letter [16]	18,999	1,000	26	16
MNIST [24]	60,000	10,000	10	36
CIFAR10 [23]	50,000	10,000	10	288

$[13 - 8, 13 + 8] = [5, 21]$. As a result, candidate K values in the set $\{5, 6, 7, \dots, 21\}$ can be safely skipped.

VI. EXPERIMENTS

We have implemented our method in Python and using the machine learning library `scikit-learn 0.24.2`, and evaluated it on two sets of supervised learning datasets. Table I shows the statistics, including the name, size of the training set, size of the test set, number of output class labels, and dimension of the input feature space. For MNIST and CIFAR10, in particular, the features were extracted using the standard histogram of oriented gradients (HOG) method [10].

The first set of datasets consists of Iris and Digits, two small datasets for which even the baseline method as shown in Algorithm 1 can finish and thus obtain the ground truth. We use the ground truth to evaluate the accuracy of our method. The second set of datasets consists of HAR, Letter, MNIST, and CIFAR10, which are larger datasets used to evaluate the efficiency of our method.

For comparison purposes, we also implemented the baseline method in Algorithm 1, and the method of Jia et al. [21], which represents the state of the art. Experiments were conducted on polluted training sets obtained by randomly inserting $\leq n$ input and output mutated samples to the original datasets. Since the same polluted training sets are used to compare all verification methods, and since the verification methods are deterministic, there is no need to run the experiments multiple times and then compute the average. Instead, we run each verification method on each polluted training set once. All experiments were conducted on a computer with a 2 GHz Quad-Core Intel Core i5 CPU and 16 GB of memory.

A. Results on the Small Datasets

We first compared our method with the baseline on the small datasets where the baseline method could actually finish. This is important because the baseline method does not rely on over-approximation, and thus can obtain the ground truth. Here, the ground truth means *which of the test data have inference results that are actually robust against n -poisoning attacks*. By comparing the ground truth with our result, we were able to evaluate the accuracy of our method.

Table II shows the results. Column 1 shows the name of the dataset and the polluted number n . Columns 2-3 show the result of the baseline method, consisting of the number of verified test data and the time taken. Similarly, Columns 4-5

TABLE II

RESULTS OF OUR METHOD AND THE BASELINE METHOD ON THE SMALL DATASETS WITH THE MAXIMAL POLLUTED NUMBER $n=1, 2, \text{ AND } 3$.

Name	Baseline		New Method		Accuracy
	# robust	time (s)	# robust	time (s)	
Iris ($n=1$)	15/15	60	14/15	1	93.3%
iris ($n=2$)	14/15	4,770	13/15	1	92.9%
iris ($n=3$)	-	>9,999	11/15	1	-
Digits ($n=1$)	179/180	8,032	172/180	1	96.1%
Digits ($n=2$)	-	>9,999	170/180	1	-
Digits ($n=3$)	-	>9,999	165/180	1	-

show the result of our method. Column 6 shows the accuracy of our method in percentage.

The results indicate that, for test data that are indeed robust according to the ground truth, our method can successfully verify most of them. In *Iris* ($n=2$), for example, Column 2 shows that 14 of the 15 test data are robust according to the baseline method, and Column 4 shows that 13 out of these 15 test data are verified by our method. Therefore, our method is 92.9% accurate.

Our method is much faster than the baseline. For *Digits* ($n=1$), in particular, our method took only 1 second to verify 172 out of the 180 test data as being robust while the baseline method took 8,032 seconds. As the polluted number n increases, the baseline method ran out of time even for these small datasets. As a result, we no longer have the ground truth needed to directly measure the accuracy of our method. Nevertheless, since all cases verified by our method are guaranteed to be robust, the number of verified test data in Column 4 of Table II serves as a proxy – it decreases slowly as n increases, indicating that the accuracy of our method remains high.

B. Results on the Large Datasets

We also evaluated our method on the large datasets. Table III summarizes the results on these large datasets as well as the two small datasets but with larger polluted numbers (n). Since these verification problems are out of the reach of the baseline method, we no longer have the ground truth. Thus, instead of measuring the accuracy, we measure the percentage of test data that we can verify, shown in Column 3 of Table III.

For example, in *Iris*, $n = 1 \sim 5$ (4%) in Column 2 means that these experiments were conducted for each poisoning number $n = 1, 2, \dots, 5$. Since the training dataset has 135 elements, $n = 5$ means 4% (or 5/135) of these training data may have been polluted. In Column 3, 93.3% is the percentage of verified test data for $n = 1$, while 73.3% is the percentage of verified test data for $n = 5$. Except for *Iris*, which has a small number of training data, we set the poisoning number n to be less than 1% of the training dataset.

Overall, our method remains fast as the sizes of T , $XSet$ and n increase. For *MNIST*, in particular, our method finished analyzing both 10-fold cross validation and KNN inference in 26 minutes, for all of the 60,000 data elements in the training set and 10,000 data elements in the test set. In contrast, the

TABLE III

RESULTS OF OUR METHOD ON LARGE DATASETS, AND ON SMALL DATASETS BUT WITH LARGER POLLUTED NUMBERS.

Name	Polluted Number (n)	Verified Percentage (# robust/ $ XSet $)	Verification Time (s)
Iris	1~5 (4%)	93.3%~73.3%	1 ~ 1
Digits	1~16 (1%)	95.6%~80.6%	1 ~ 2
HAR	1~98 (1%)	99.4%~71.7%	85 ~ 93
Letter	1~190 (1%)	94.0%~5.5%	33 ~ 43
MNIST	1~600 (1%)	99.9%~53.5%	888 ~ 994
CIFAR10	1~500 (1%)	99.2%~2.8%	1,453 ~ 1,559

baseline method failed to verify any of the test data within the 9999-second time limit.

Without the ground truth, the verified percentage provides a lower bound on the number of test data that remain robust against data-poisoning attacks. When $n=1$, the verified percentage in Column 3 is high for all datasets. As the polluted number n increases to 1% of the entire training set T , the verified percentage decreases. Furthermore, the decrease is more significant for some datasets than for other datasets. For example, In *MNIST*, at least 53.5% of the test data remain robust under 1% (or 600) poisoning attacks. In *CIFAR10*, however, only 2.8% of the test data remains robust under 1% (or 500) poisoning attacks. Thus, the relationship between the verified percentage and the polluted number reflects more about the unique characteristics of these datasets. By this, we mean that if one dataset has more *truly-non-robust* cases than another dataset, then the verifier will report more *cannot-be-verified* cases.

The reason why the accuracy is low for *Letter* and *CIFAR10* datasets is because they have larger attack surfaces in the extracted feature space: elements from the same class are not sufficiently concentrated in one area, and the neighbors include many elements from other classes. Thus, small changes to the neighbors can lead to significant changes of the class label. While we believe that the accuracy (measured by the verified percentage) may improve if a better feature extractor is used (to improve the quality of extracted features), it is out of the scope of the verification task.

C. Compared with the Existing Method

While our method is the only one that can verify the entire KNN algorithm, there are existing methods that can verify part of the KNN algorithm. The most recent method proposed by Jia et al. [21], in particular, aims to verify the KNN inference step with a given K value; thus, it can be regarded as functionally equivalent to the subroutine of our method as presented in Algorithm 7. However, our method is significantly more accurate due to its tighter approximation. To experimentally demonstrate the advantage of our method, we used their method to replace Algorithm 7 in our own method before conducting the experimental comparison. Since an open-source implementation of their method is not available, we have implemented it ourselves.

Fig. 6 shows the results, where *blue* lines represent our method and *orange* lines represent their method [21]. Overall,

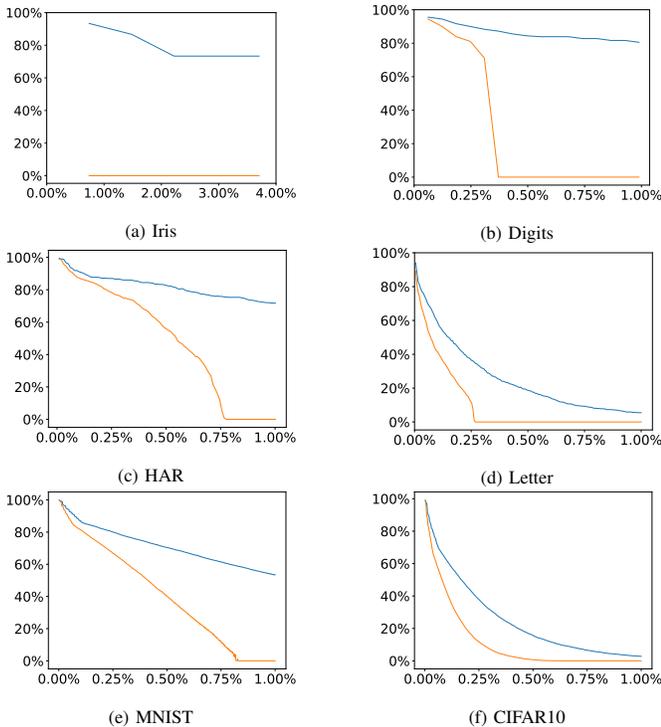


Fig. 6. Comparing our method (blue) with Jia et al. [21] (orange): the x -axis is polluted number n and the y -axis is the percentage of verified test data.

the verified percentage obtained by our method is significantly higher, due to its tighter approximations during the KNN inference phase. For all datasets, the verified percentage obtained by their method drops more quickly than the verified percentage obtained by our method. For *Iris*, in particular, their method cannot verify any of the test data, while our method can verify more than 70% of them as being robust.

VII. RELATED WORK

There is a large body of work on verifying the (local) robustness of machine learning algorithms using formal methods. However, unlike most prior works which focus on adversarial examples in the context of deep neural networks, this work focuses on poisoned datasets for KNN. Unlike neural networks, for which scalability of the verification method typically depends on the network size but not the size of the training data, for KNN, scalability depends on the size of the training data and the number of poisoned elements.

In the context of robustness verification for KNN, our method is a method that can soundly verify n -poisoning robustness of the entire KNN algorithm, while existing methods such as Jia et al. [21] and others [39], [20], [40] are either restricted to a small part of what constitutes a state-of-the-art KNN system or primarily theoretical (and thus not scalable). Since we follow the definition of n -poisoning robustness in Drews et al. [12] instead of Jia et al. [21], our method only handles the removal of elements from already-polluted datasets, but not addition/modification of elements for clean

datasets. Extending our method to handle such cases will be future work.

In addition to this line of research, there is a large body of work on adversarial data poisoning in general.

Data Poisoning in General KNN is not the only type of machine learning techniques found vulnerable to adversarial data poisoning; prior work shows that regression models [29], support vector machines (SVM) [6], [43], [42], clustering algorithms [7], and neural networks [34], [37], [11], [45] are also vulnerable. Unlike our work, this line of research is primarily concerned with showing the security threats and identifying the poisoning sets, which is often formulated as a constrained optimization problem.

Mitigating Data Poisoning Techniques have been proposed to mitigate data poisoning for various machine learning algorithms [35], [38], [19], [13], [5]. There are also techniques [22], [28] for assessing the effectiveness of mitigation techniques such as data sanitization [22] and differentially-private countermeasures [28]. More recently, Bahri et al. [4] propose a method that leverages both KNN and a deep neural network to remove mislabeled data.

Certifying the Defenses Probabilistically There are techniques for certifying the defenses [32], [25] such that accuracy is guaranteed probabilistically. For example, Rosenfeld et al. [32] leverage randomized smoothing to guarantee test-time robustness to adversarial manipulation with high probability. Levine et al. [25] certify robustness of a defense by deriving a lower bound of classification error, which relies on their deep partition aggregation (DPA) learning and is not applicable to typical learning approaches.

Leveraging KNN for Attacks or Defenses Orthogonal to our work, there are techniques that leverage KNN to generate attacks or provide defenses for other machine learning models. For example, Li et al. [26] present a data-poisoning attack that leverages KNN to maximize the effectiveness of malicious behavior while mimicking the user’s benign behavior. Peri et al. [31] use KNN to defend against adversarial input based attacks, although it focuses only on tweaking the test input during the inference phase.

VIII. CONCLUSIONS

We have presented the first method for soundly verifying n -poisoning robustness for the entire KNN algorithm that includes both the learning (K parameter tuning) and the inference (classification) phases. It relies on sound overapproximations to exhaustively and yet efficiently cover the astronomically large number of possible adversarial scenarios. We have demonstrated the accuracy and efficiency of our method, and its advantages over a state-of-the-art method, through experimental evaluation using both small and large supervised-learning datasets. Besides KNN, our method for soundly over-approximating p -fold cross validation may be used to analyze similar cross-validation steps frequently used in other modern machine learning systems.

REFERENCES

- [1] D. A. Adeniyi, Z. Wei, and Y. Yongquan, "Automated web usage data mining and recommendation system using k-nearest neighbor (KNN) classification method," *Applied Computing and Informatics*, vol. 12, no. 1, pp. 90–108, 2016.
- [2] M. Andersson and L. Tran, "Predicting movie ratings using KNN," 2020.
- [3] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones." in *Esann*, vol. 3, 2013, p. 3.
- [4] D. Bahri, H. Jiang, and M. Gupta, "Deep k-nn for noisy labels," in *International Conference on Machine Learning*, 2020, pp. 540–550.
- [5] B. Biggio, I. Corona, G. Fumera, G. Giacinto, and F. Roli, "Bagging classifiers for fighting poisoning attacks in adversarial classification tasks," in *International Workshop on Multiple Classifier Systems*, 2011, pp. 350–359.
- [6] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *International Conference on Machine Learning*, 2012.
- [7] B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, and F. Roli, "Poisoning behavioral malware clustering," in *Workshop on Artificial Intelligent and Security*, 2014, pp. 27–36.
- [8] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.
- [9] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *Journal of Logic and Computation*, vol. 2, no. 4, pp. 511–547, 1992.
- [10] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *International Conference on Computer Vision and Pattern Recognition*, 2005, pp. 886–893.
- [11] A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli, "Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks," in *USENIX Security Symposium*, 2019, pp. 321–338.
- [12] S. Drews, A. Albarghouthi, and L. D'Antoni, "Proving data-poisoning robustness in decision trees," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1083–1097.
- [13] J. Feng, H. Xu, S. Mannor, and S. Yan, "Robust logistic regression and classification," *Advances in Neural Information Processing Systems*, pp. 253–261, 2014.
- [14] I. Firdausi, A. Erwin, A. S. Nugroho *et al.*, "Analysis of machine learning techniques used in behavior-based malware detection," in *International Conference on Advances in Computing, Control, and Telecommunication Technologies*, 2010, pp. 201–203.
- [15] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
- [16] P. W. Frey and D. J. Slate, "Letter recognition using holland-style adaptive classifiers," *Machine learning*, vol. 6, no. 2, pp. 161–182, 1991.
- [17] G. Gates, "The reduced nearest neighbor rule (corresp.)," *IEEE Transactions on Information Theory*, vol. 18, no. 3, pp. 431–433, 1972.
- [18] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "KNN model-based approach in classification," in *International Conferences On the Move to Meaningful Internet Systems*, 2003, pp. 986–996.
- [19] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in *IEEE Symposium on Security and Privacy*, 2018, pp. 19–35.
- [20] J. Jia, X. Cao, and N. Z. Gong, "Intrinsic certified robustness of bagging against data poisoning attacks," *arXiv preprint arXiv:2008.04495*, 2020.
- [21] —, "Certified robustness of nearest neighbors against data poisoning attacks and backdoor attacks," in *AAAI Conference on Artificial Intelligence*, 2022.
- [22] P. W. Koh, J. Steinhardt, and P. Liang, "Stronger data poisoning attacks break data sanitization defenses," *arXiv preprint arXiv:1811.00741*, 2018.
- [23] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," *Technical Report, University of Toronto*, 2009.
- [24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [25] A. Levine and S. Feizi, "Deep partition aggregation: Provable defense against general poisoning attacks," *arXiv preprint arXiv:2006.14768*, 2020.
- [26] B. Li, Y. Wang, A. Singh, and Y. Vorobeychik, "Data poisoning attacks on factorization-based collaborative filtering," *arXiv preprint arXiv:1608.08182*, 2016.
- [27] Y. Li, B. Fang, L. Guo, and Y. Chen, "Network anomaly detection based on TCM-KNN algorithm," in *ACM symposium on Information, Computer and Communications Security*, 2007, pp. 13–19.
- [28] Y. Ma, X. Zhu, and J. Hsu, "Data poisoning against differentially-private learners: Attacks and defenses," *arXiv preprint arXiv:1903.09860*, 2019.
- [29] S. Mei and X. Zhu, "Using machine teaching to identify optimal training-set attacks on machine learners," in *AAAI Conference on Artificial Intelligence*, 2015.
- [30] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Computing*, vol. 20, no. 1, pp. 343–357, 2016.
- [31] N. Peri, N. Gupta, W. R. Huang, L. Fowl, C. Zhu, S. Feizi, T. Goldstein, and J. P. Dickerson, "Deep k-nn defense against clean-label data poisoning attacks," in *European Conference on Computer Vision*, 2020, pp. 55–70.
- [32] E. Rosenfeld, E. Winston, P. Ravikumar, and Z. Kolter, "Certified robustness to label-flipping attacks via randomized smoothing," in *International Conference on Machine Learning*, 2020, pp. 8230–8241.
- [33] A. Schwarzschild, M. Goldblum, A. Gupta, J. P. Dickerson, and T. Goldstein, "Just how toxic is data poisoning? A unified benchmark for backdoor and data poisoning attacks," in *International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., 2021.
- [34] A. Shafahi, W. R. Huang, M. Najibi, O. Suci, C. Studer, T. Dumitras, and T. Goldstein, "Poison frogs! targeted clean-label poisoning attacks on neural networks," *arXiv preprint arXiv:1804.00792*, 2018.
- [35] J. Steinhardt, P. W. Koh, and P. Liang, "Certified defenses for data poisoning attacks," *arXiv preprint arXiv:1706.03691*, 2017.
- [36] M.-Y. Su, "Real-time anomaly detection systems for denial-of-service attacks by weighted k-nearest-neighbor classifiers," *Expert Systems with Applications*, vol. 38, no. 4, pp. 3492–3498, 2011.
- [37] O. Suci, R. Marginean, Y. Kaya, H. Daume III, and T. Dumitras, "When does machine learning FAIL? generalized transferability for evasion and poisoning attacks," in *USENIX Security Symposium*, 2018, pp. 1299–1316.
- [38] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," *arXiv preprint arXiv:1811.00636*, 2018.
- [39] Y. Wang, S. Jha, and K. Chaudhuri, "Analyzing the robustness of nearest neighbors to adversarial examples," in *International Conference on Machine Learning*, 2018, pp. 5133–5142.
- [40] M. Weber, X. Xu, B. Karlas, C. Zhang, and B. Li, "Rab: Provable robustness against backdoor attacks," *arXiv preprint arXiv:2003.08904*, 2020.
- [41] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *Asia-Pacific Software Engineering Conference*, 2011, pp. 389–396.
- [42] H. Xiao, H. Xiao, and C. Eckert, "Adversarial label flips attack on support vector machines," in *ECAI*, 2012, pp. 870–875.
- [43] H. Xiao, B. Biggio, B. Nelson, H. Xiao, C. Eckert, and F. Roli, "Support vector machines under adversarial label contamination," *Neurocomputing*, vol. 160, pp. 53–62, 2015.
- [44] M. Xie, J. Hu, S. Han, and H.-H. Chen, "Scalable hypergrid k-NN-based online anomaly detection in wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1661–1670, 2012.
- [45] C. Zhu, W. R. Huang, H. Li, G. Taylor, C. Studer, and T. Goldstein, "Transferable clean-label poisoning attacks on deep neural nets," in *International Conference on Machine Learning*, 2019, pp. 7614–7623.

On Optimizing Back-Substitution Methods for Neural Network Verification

Tom Zelazny*, Haoze Wu[†], Clark Barrett[†], and Guy Katz*

*The Hebrew University of Jerusalem, Jerusalem, Israel [†]Stanford University, Stanford, California

*{tomz, g.katz}@mail.huji.ac.il [†]{haozewu, barrett}@cs.stanford.edu

Abstract—With the increasing application of deep learning in mission-critical systems, there is a growing need to obtain formal guarantees about the behaviors of neural networks. Indeed, many approaches for verifying neural networks have been recently proposed, but these generally struggle with limited scalability or insufficient accuracy. A key component in many state-of-the-art verification schemes is computing lower and upper bounds on the values that neurons in the network can obtain for a specific input domain — and the tighter these bounds, the more likely the verification is to succeed. Many common algorithms for computing these bounds are variations of the symbolic-bound propagation method; and among these, approaches that utilize a process called back-substitution are particularly successful. In this paper, we present an approach for making back-substitution produce tighter bounds. To achieve this, we formulate and then minimize the imprecision errors incurred during back-substitution. Our technique is general, in the sense that it can be integrated into numerous existing symbolic-bound propagation techniques, with only minor modifications. We implement our approach as a proof-of-concept tool, and present favorable results compared to state-of-the-art verifiers that perform back-substitution.

I. INTRODUCTION

Deep neural networks (DNNs) are dramatically changing the way modern software is written. In many domains, such as image recognition [43], game playing [42], protein folding [2] and autonomous vehicle control [12], [30], state-of-the-art solutions involve deep neural networks — which are artifacts learned automatically from a finite set of examples, and which often outperform carefully handcrafted software.

Along with their impressive success, DNNs present a significant new challenge when it comes to quality assurance. Whereas many best practices exist for writing, testing, verifying and maintaining hand-crafted code, DNNs are automatically generated, and are mostly opaque to humans [24], [25]. Consequently, it is difficult for human engineers to reason about them and ensure their correctness and safety — as most existing approaches are ill-suited for this task. This challenge is becoming a significant concern, with various faults being observed in modern DNNs [5]. One notable example is that of *adversarial perturbations* — small perturbation that, when added to inputs that are correctly classified by the DNN, result in severe errors [20], [48]. This issue, and others, call into question the safety, security and interpretability of DNNs, and could hinder their adoption by various stakeholders.

In order to mitigate this challenge, the formal methods community has taken up interest in DNN verification. In the past few years, a plethora of approaches have been proposed

for tackling the *DNN verification problem*, in which we are given a DNN and a condition about its inputs and outputs; and seek to either find an input assignment to the DNN that satisfies this condition, or prove that it is not satisfiable [1], [8], [10], [14], [21], [27], [29], [31], [33], [39], [51], [57]. The usefulness of DNN verification has been demonstrated in several settings and domains [21], [27], [31], [47], but most existing approaches still struggle with various limitations, specifically relating to scalability.

A key technical challenge in verifying neural networks is to reason about *activation functions*, which are non-linear (e.g., piece-wise linear) transformations applied to the output of each layer in the neural network. Precisely reasoning about such non-linear behaviors requires a case-by-case analysis of the activation phase of each activation function, which quickly becomes infeasible as the number of non-linear activations increases. Instead, before performing such a search procedure, state-of-the-art solvers typically first consider linear abstractions of activation functions, and use these abstractions to over-approximate the values that the activation functions can take in the neural network. Often, these over-approximations significantly curtail the search space that later needs to be explored, and expedite the verification procedure as a whole.

A key operation that is repeatedly invoked in this computation of over-approximations is called *back-substitution* [45], where the goal is to compute, for each neuron in the DNN, lower and upper bounds on the values it can take with respect to the input region of interest. This is done by first expressing the lower and upper bounds of a neuron symbolically as a function of neurons from previous layers, and then concretizing these symbolic bounds with the known bounds of neurons in those previous layers. Such a technique is essential in state-of-the-art solvers (e.g., [32], [45], [54]) and is often able to obtain sufficiently tight bounds for proving the properties with respect to small input regions. However, it tends to significantly lose precision when the input region (i.e., perturbation radius) grows, preventing one from efficiently verifying more challenging problems.

In this work, we seek to improve the precision and scalability of DNN verification techniques, by reducing the over-approximation error in the back-substitution process. Our key insight is that, as part of the symbolic-bound propagation, one can measure the error accumulated by the over-approximations used in back-substitution. Often, the currently computed bound can then be significantly improved by “pushing” it towards the

true function, in a way that maintains its validity. For example, suppose that we upper-bound a function f with a function g , i.e. $\forall x. g(x) \geq f(x)$. If we discover that the minimal approximation error is 5, i.e. $\min_x \{g(x) - f(x)\} = 5$, then $g(x) - 5$ can be used as a better upper bound for f than the original g . By integrating this simple principle into the back-substitution process, we show that we can obtain much tighter bounds, which eventually translates to the ability to verify more difficult properties.

We propose here a verification approach, called *DeepMIP*, that uses symbolic-bound tightening enhanced with our error-optimization method. At each iteration of the back-substitution, DeepMIP invokes an external MIP solver [26] to compute bounds on the error of the current approximation, and then uses these bounds to improve that approximation. As we show, this leads to an improved ability to solve verification benchmarks when compared to state-of-the-art, symbolic-bound tightening techniques. We discuss the different advantages of the approach, as well as the extra overhead that it incurs, and various enhancements that could be used to expedite it further.

The rest of the paper is organized as follows. We begin by presenting the necessary background on DNNs, DNN verification, and on symbolic-bound propagation in Sec. II. Next, in Sec. III we show how one can express the approximation error incurred as part of the back-substitution process. In Sec. IV we present the DeepMIP algorithm, followed by its evaluation in Sec. V. Related work is discussed in Sec. VI, and we conclude in Sec. VII.

II. BACKGROUND

Neural networks. A fully-connected feed-forward neural network with $k + 1$ layers is a function $N : \mathbb{R}^m \rightarrow \mathbb{R}^n$. Given an input $\mathbf{x} \in \mathbb{R}^m$, we use $N_i(\mathbf{x})$ to denote the values of neurons in the i^{th} layer ($0 \leq i \leq k$). The output of the neural network $N(\mathbf{x})$ is defined as $N_k(\mathbf{x})$, which we refer to as the output layer. More concretely, for $1 \leq i \leq k$,

$$N_i(\mathbf{x}) = \sigma(W^{i-1}N_{i-1}(\mathbf{x}) + b^{i-1})$$

where W^{i-1} is a *weight matrix*, b^{i-1} is a *bias vector*, σ is an activation function (in this paper, we focus on the ReLU activation function, defined as $\text{ReLU}(x) = \max\{0, x\}$ and use σ and ReLU interchangeably unless otherwise specified) and $N_0(\mathbf{x}) = \mathbf{x}$. We refer to N_0 as the input layer. Typically, non-linear activations are not applied to the output layer. Thus, when $i = k$, we let σ be the identity function. We note that our techniques are general, and apply to other activation functions (MaxPool, LeakyReLU) and architectures (e.g., convolutional, residual).

Neural network verification. The *neural network verification problem* [31], [39] is defined as follows: given an input domain $\mathcal{D}_i \subseteq \mathbb{R}^m$ and an output domain domain $\mathcal{D}_o \subseteq \mathbb{R}^n$, the goal is to determine whether $\forall \mathbf{x} \in \mathcal{D}_i, N(\mathbf{x}) \in \mathcal{D}_o$. If the answer is affirmative, we say that the verification property pair $\langle \mathcal{D}_i, \mathcal{D}_o \rangle$ holds. In this paper, we assume that the neural network has

a single output neuron and that the verification problem can be reduced to the problem of finding the minimum and/or maximum values for that single output neuron:

$$\min_{\mathbf{x} \in \mathcal{D}_i} (N(\mathbf{x})) \quad \max_{\mathbf{x} \in \mathcal{D}_i} (N(\mathbf{x})) \quad (1)$$

For example, if \mathcal{D}_o is the interval $[-2, 7]$ and we discover that $\min_{\mathbf{x} \in \mathcal{D}_i} (N(\mathbf{x})) = 1$ and $\max_{\mathbf{x} \in \mathcal{D}_i} (N(\mathbf{x})) = 3$, then we are guaranteed that the property holds. We will focus on solving just the maximization problem, although the method that we present next can just as readily be applied towards the minimization problem.

A straightforward way to solve the optimization problem in Eq. 1 is to encode the neural network as a mixed integer programming (MIP) instance [11], [31], [49], and then solve the problem using a MIP solver, which often employs a branch-and-bound procedure. While this approach has proven effective at verifying small DNNs, it faces a scalability barrier when it comes to larger networks. Therefore, before invoking the branch-and-bound procedure, existing solvers typically first seek to prove the property with abstraction-based techniques (symbolic-bound propagation), which have more tractable runtime.

Symbolic-bound propagation. Symbolic-bound propagation [21], [51] is a method of obtaining bounds on the concrete values a neuron may obtain. When applied to a network's output neuron, it enables us to obtain an approximate solution to the optimization problems from Eq. 1, which may be sufficient to determine that the property holds. For example, continuing the example from before, if we are unable to exactly compute that $\max_{\mathbf{x} \in \mathcal{D}_i} (N(\mathbf{x})) = 3$ but can determine that $\max_{\mathbf{x} \in \mathcal{D}_i} (N(\mathbf{x})) < 5$, this is enough for concluding that the property in question holds. The idea underlying symbolic-bound propagation is to start from the bounds for the input layer provided in \mathcal{D}_i , and then propagate them, layer-by-layer, up to the output layer. It has been observed that while affine transformations allow us to precisely propagate bounds from a layer to its successor, activation functions introduce inaccuracies [45].

Before formally defining symbolic bound propagation, we start with an intuitive example using the network in Fig. 1. Let \mathbf{x}^i denote the *pre-activation* values of the neurons in layer i , and let $\mathbf{y}^i = \sigma(\mathbf{x}^i)$ denote their *post-activation* values; similarly, let x_j^i and $y_j^i = \sigma(x_j^i)$ denote the pre- and post-activation values of neuron j in layer i ; and let l_j^i, u_j^i denote the concrete (scalar) lower- and upper-bound for x_j^i , i.e. $l_j^i \leq x_j^i \leq u_j^i$ when the DNN is evaluated on any input from \mathcal{D}_i . Assume that \mathcal{D}_i is the following box domain:

$$\mathcal{D}_i = \{-1 \leq x_i^0 \leq 1 \mid i \in \{0, 1, 2\}\}$$

and that we wish to compute bounds for the single output neuron, x_0^3 .

We begin by propagating the bounds through the first affine layer. According to the network's weights and biases, we get:

$$x_0^1 = x_0^0 + x_1^0, \quad x_1^1 = x_0^0 - x_1^0, \quad x_2^1 = x_2^0$$

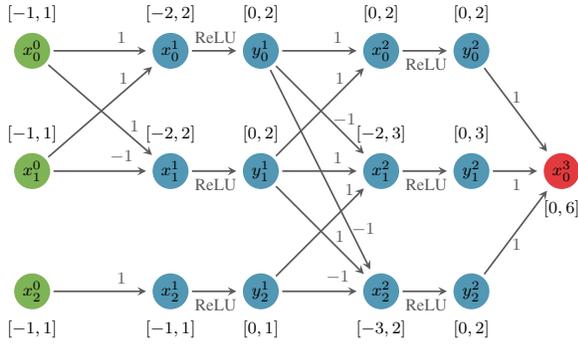


Fig. 1: A neural network.

these equations allow us to compute concrete lower and upper bounds for each of these neurons, by substituting the input neurons (x_0^0, x_1^0, x_2^0) with their corresponding concrete bounds (according to the sign of their coefficients). Using this process, we obtain:

$$x_0^1 \in [-2, 2], \quad x_1^1 \in [-2, 2], \quad x_2^1 \in [-1, 1]$$

this propagation, often referred to as *interval arithmetic* [15], is precise for individual neurons: indeed, x_0^1, x_1^1 and x_2^1 can each take on any value in their respective computed ranges. However, much important information is lost when using just interval arithmetic: for example, it is impossible for x_0^1 and x_1^1 to *simultaneously* be assigned 2. As we will later see, symbolic-bound propagation addresses this issue by capturing some of the dependencies between neurons, and using these dependencies in producing tighter bounds.

For now, we continue propagating our computed bounds to neurons y_0^1, y_1^1 and y_2^1 . The output range of a ReLU is the non-negative part of its input range, which yields:

$$y_0^1 \in [0, 2], \quad y_1^1 \in [0, 2], \quad y_2^1 \in [0, 1]$$

and the next, affine layer is again handled using interval arithmetic. Using the expressions

$$x_0^2 = y_0^1 + y_1^1, \quad x_1^2 = -y_0^1 + y_1^1 + y_2^0, \quad x_2^2 = -y_0^1 + y_1^1 - y_2^0$$

and substituting each y_i^1 with the appropriate bound, we obtain:

$$x_0^2 \in [0, 4], \quad x_1^2 \in [-2, 4], \quad x_2^2 \in [-4, 2]$$

Unfortunately, as we soon show, the bounds computed for x_0^2, x_1^2, x_2^2 are not tight. A better approach is to compute *symbolic bounds*, as opposed to concrete ones, in a way that lets us carry additional information about the dependencies between neurons. In symbolic-bound propagation, we seek to express the upper and lower bounds of each neuron as a linear combination of neurons from earlier layers, using a process known as *back-substitution*. The main difficulty is to propagate these bounds across ReLU layers, which are not convex; and this is performed by using a *triangle relaxation* of the ReLU

function, illustrated in Fig. 2. Assume $x \in [l, u]$; then, using this relaxation, we can deduce the following bounds:

$$\begin{cases} 0 \leq \sigma(x) \leq 0 & \text{if } u \leq 0 \\ x \leq \sigma(x) \leq x & \text{if } l \geq 0 \\ \alpha x \leq \sigma(x) \leq \frac{u}{u-l}(x-l) & \text{otherwise, for any } 0 \leq \alpha \leq 1 \end{cases}$$

Different symbolic bound propagation methods use different heuristics for choosing α [45], [54]; but this is beyond our scope here, and our proposed technique is compatible with any such heuristic. For our running example, we arbitrarily choose the values of α ; and for our implementation, we use an existing heuristic [54].

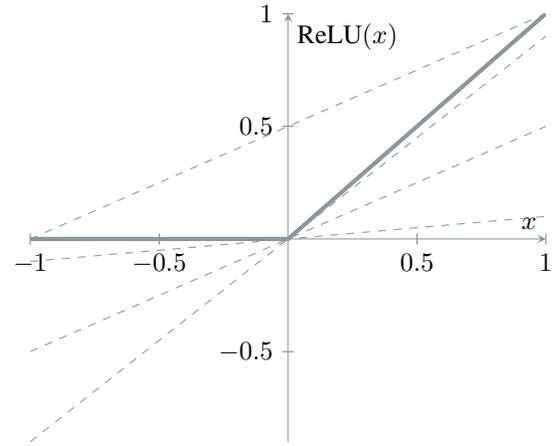


Fig. 2: A triangle relaxation of a ReLU function for $x \in [-1, 1]$. The solid lines correspond to the exact ReLU function, and the dotted lines represent the relaxed lower and upper bounds, for different values of α .

Using this relaxation, we show how to compute symbolic bounds that yield tighter bounds for the x_i^2 neurons. First observe neuron x_0^2 , given as $x_0^2 = y_0^1 + y_1^1 = \sigma(x_0^1) + \sigma(x_1^1)$. To obtain its lower bound we first substitute both $y_0^1 = \sigma(x_0^1)$ and $y_1^1 = \sigma(x_1^1)$ with their corresponding triangle relaxation lower bounds, with the choice of $\alpha = 0$ for both (we note that it is possible to choose different α values for different variables). For the upper bound, we use the linear upper bound from the triangle relaxation. By using the bounds we already know for nodes in previous layers, we get that:

$$\begin{aligned} x_0^2 &\geq 0 \cdot x_0^1 + 0 \cdot x_1^1 = 0 \\ x_0^2 &\leq \frac{1}{2}(x_0^1 + 2) + \frac{1}{2}(x_1^1 + 2) = \frac{1}{2}(x_0^1 + x_1^1) + 2 \\ &= \frac{1}{2}((x_0^0 + x_1^0) + (x_0^0 - x_1^0)) + 2 = x_0^0 + 2 \leq 3 \end{aligned}$$

which indeed produces a tighter upper bound than the one obtained for x_0^2 using interval propagation. Similarly, we get

that for x_1^2 :

$$\begin{aligned} x_1^2 &\geq -\frac{1}{2}(x_0^1 + 2) + 0 \cdot x_1^1 + 0 \cdot x_2^1 \\ &= -\frac{1}{2}(x_0^0 + x_1^0) - 1 = -2 \\ x_1^2 &\leq -0 \cdot x_0^1 + \frac{1}{2}(x_1^1 + 2) + \frac{1}{2}(x_2^1 + 1) \\ &= \frac{1}{2}(x_1^1 + x_2^1) + 1.5 = \frac{1}{2}(x_0^0 - x_1^0 + x_2^0) + 1.5 \leq 3 \end{aligned}$$

and for x_2^2 :

$$\begin{aligned} x_2^2 &\geq -\frac{1}{2}(x_0^1 + 2) + 0 \cdot (x_1^1) - \frac{1}{2}(x_2^1 + 1) \\ &= -\frac{1}{2}(x_0^0 + x_2^0) - 1.5 = -\frac{1}{2}(x_0^0 + x_1^0 + x_2^0) - 1.5 \geq -3 \\ x_2^2 &\leq -0 \cdot x_0^1 + \frac{1}{2}(x_1^1 + 2) - 0 \cdot x_2^1 \\ &= \frac{1}{2}x_1^1 + 1 = \frac{1}{2}(x_0^0 - x_1^0) + 1 \leq 2 \end{aligned}$$

We have thus obtained the following bounds:

$$x_0^2 \in [0, 3], \quad x_1^2 \in [-2, 3], \quad x_2^2 \in [-3, 2]$$

We note that while these bounds are tighter than the ones produced by interval propagation, and are in fact optimal for x_1^2, x_2^2 , this is not the case for x_0^2 (the optimal bounds are displayed in square brackets in Fig. 1). The reason for this sub-optimality is discussed in Section III.

We continue to propagate our bounds through the next layer, obtaining:

$$y_0^2 \in [0, 3], \quad y_1^2 \in [0, 3], \quad y_2^2 \in [0, 2]$$

and finally reach:

$$\begin{aligned} x_0^3 &= y_0^2 + y_1^2 + y_2^2 = \sigma(x_0^2) + \sigma(x_1^2) + \sigma(x_2^2) \\ &\leq x_0^2 + \frac{3}{5}(x_1^2 + 2) + \frac{2}{5}(x_2^2 + 3) \\ &= 2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5} = 2\sigma(x_1^1) + \frac{1}{5}\sigma(x_2^1) + \frac{12}{5} \\ &\leq 2 \cdot \frac{1}{2}(x_1^1 + 2) + \frac{1}{5} \cdot \frac{1}{2}(x_2^1 + 1) + \frac{12}{5} \\ &= x_0^0 - x_1^0 + \frac{1}{10}x_2^0 + 4.5 \leq 6.6 \end{aligned}$$

More generally, the back-substitution process for upper-bounding a neuron x_i^k (assuming we already have valid bounds for all neurons in earlier layers) is iteratively defined as:

$$\begin{aligned} \max(x_i^k) &= \max(W_i^{k-1}\sigma(\mathbf{x}^{k-1})) \\ &\leq \max(W_i^{k-1}R_U^{k-2}\mathbf{x}^{k-1}) \\ &= \max(W_i^{k-1}R_U^{k-2}W^{k-2}\sigma(\mathbf{x}^{k-2})) \\ &\leq \max(W_i^{k-1}R_U^{k-2}W^{k-2}R_U^{k-3}\mathbf{x}^{k-2}) \\ &= \dots \leq \max(W_i^{k-1} \prod_{j=k-2}^0 (R_U^j W^j) \mathbf{x}^0) \end{aligned}$$

(Biases and constants are handled similarly, and are omitted for clarity.) At each step, we can replace the variables of \mathbf{x}^i

by their respective concrete bounds $[l_j^i, u_j^i]$, in an interval-arithmetic fashion, to obtain a valid concrete upper bound for the value of $\max(x_i^k)$. We refer to this operation as *concretization*. We call the matrices R_L^i, R_U^i the respective lower- and upper-bound *relaxation* matrices [54]. These matrices apply the appropriate triangle relaxation to each ReLU, allowing us to replace it with a linear bound, and are defined using the current symbolic bounds for each ReLU as well as the weight matrix of the layer the precedes it. The two matrices are defined such that $\forall \mathbf{x} \in \mathcal{D}_i$:

$$\omega_i R_L^i \mathbf{x} + c_L \leq \omega_i \sigma(\mathbf{x}) \leq \omega_i R_U^i \mathbf{x} + c_U$$

where c_L and c_U are scalar constants; and ω_i is a row vector containing the coefficients of each $\sigma(x_j)$, resulting in linear bounds for the sum of ReLUs. A precise definition of these matrices appears in Sec. A of the Appendix; and a similar procedure can be applied for lower-bounding x_i^k .

At first glance, the iterative back-substitution process may seem counter productive; indeed, in each iteration where we move to an earlier layer of the network, we use a less-than-equals transition, which seems to indicate that the upper bound that we will eventually reach is more loose than the present bound. This, however, is not so; and the reason is the *concretization* process. When we concretize the bounds in some later iteration, it is possible that the known bounds for the variables in that layer of the network will lead to a tighter upper bound than the one that can be derived presently. More generally, this process can be regarded as a trade-off between computing looser expressions for the bound, but being able to concretize them over more exact domains — which could result in tighter bounds [45].

III. ERRORS IN BACK-SUBSTITUTION

As previously mentioned, although symbolic-bound computation using back-substitution can derive tighter bounds than naïve interval propagation, there are cases in which the computed bounds are sub-optimal: for example, while the bounds computed for x_1^2 and x_2^2 were tight (i.e., there exists an input in \mathcal{D}_i for which they are met), the bounds for x_0^2 and x_0^3 were not. In this section, we analyze the reasons behind such sub-optimal bounds. We begin with the following definitions:

Definition 1 (Optimal bias for bound): let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a function and let $U_f(\mathbf{x}) \equiv \omega \mathbf{x} + b$ ($\omega \in \mathbb{R}^n, b \in \mathbb{R}$) be a valid linear upper bound for f over the domain \mathcal{D} , i.e., $\forall \mathbf{x} \in \mathcal{D} : U_f(\mathbf{x}) \geq f(\mathbf{x})$. We say that b is the *optimal bias* for $U_f(\mathbf{x})$ if $\forall b^* : b^* < b$, it holds that $U_f^*(\mathbf{x}) \equiv \omega \mathbf{x} + b^*$ is no longer a valid upper bound for f . The definition for the optimal bias for f 's lower bound is symmetrical.

An example of optimal and sub-optimal upper bounds appears in Fig. 3. In the graph depicted therein, we plot an upper bound for the function $\text{ReLU}(x)$. The bias value of the first bound (in red) is 1; and as we can see, the resulting bound is not tight. When we set the bias value to 1/2, the bound becomes tight, equaling the function at points $x = -1$ and $x = 1$, and so that is the optimal bias value for that bound.

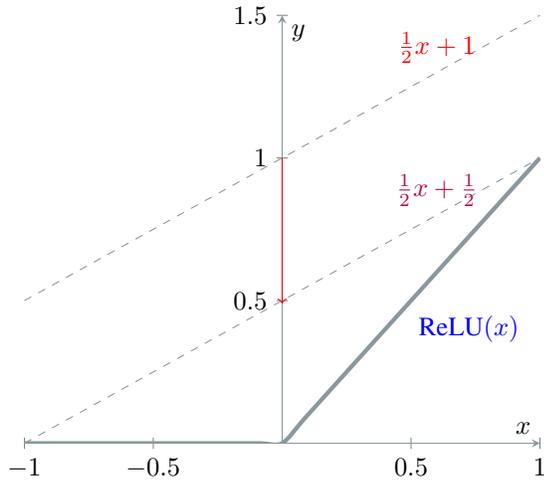


Fig. 3: A simplified illustration of an optimal and sub-optimal bounds for a ReLU function over $x \in [-1, 1]$.

Definition 2 (Bound error): Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$, and let $g(\mathbf{x})$ be an upper bound for f over domain \mathcal{D} , such that we have: $\forall \mathbf{x} \in \mathcal{D} : g(\mathbf{x}) \geq f(\mathbf{x})$. We define the error of g with respect to f as the function: $E(\mathbf{x}) = g(\mathbf{x}) - f(\mathbf{x})$. The case for a lower bound is symmetrical.

We observe that a linear bound g for f over the domain \mathcal{D}_i has *optimal bias* iff $\exists \mathbf{x} \in \mathcal{D}_i : E(\mathbf{x}) = 0$. We refer to any bound that has a sub-optimal bias, i.e. $\forall \mathbf{x} \in \mathcal{D}_i : E(\mathbf{x}) > 0$, as a *detached bound*. We show that these detachments occur naturally as part of the back-substitution process, and are partially responsible for the discovery of sub-optimal concrete bounds.

It is straightforward to see that the aforementioned triangle relaxation for ReLUs produces linear bounds that are bias-optimal for each individual ReLU. However, as it turns out, this may not be the case when multiple ReLUs are involved. In a typical DNN, a neuron’s value is computed as a weighted sum of the ReLUs of values from its preceding layer. Consequently, when we calculate an upper bound for the neuron using back-substitution, we are in fact upper-bounding a sum of ReLUs by summing their individual upper bounds. This can result in a *detached bound*, where, despite the fact that each ReLU was approximated using a bound with an optimal bias, the resulting combined bound does not have optimal bias.

An illustration of this phenomenon appears in Fig. 4. Sub-figures *a* and *b* therein show the graph of ReLU functions, plotted along their triangle-relaxation upper bound (in orange). Sub-figure *c* then shows the graph of the *sum* of the two ReLU functions from sub-figures *a* and *b*, along with the sum of their individual upper bounds (again, in orange). As we can see, although the upper bounds in *a* and *b* touch the functions they are approximating in at least one point (and are hence bias-optimal), the bound in *c* is detached, and is hence not bias-optimal. Each figure in the lower row of Fig. 4 shows the over-approximation error of the figure directly above it.

More formally, the error of the upper bound for $\text{ReLU}(x)$

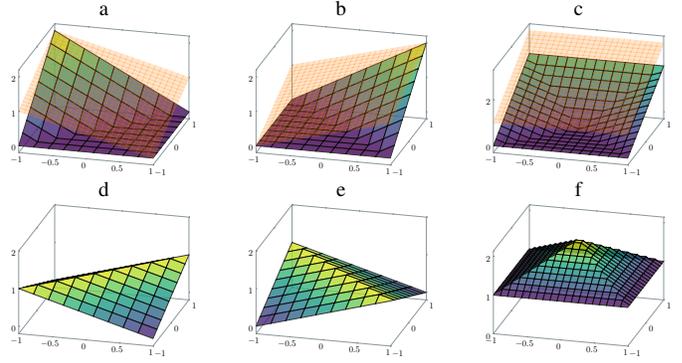


Fig. 4: Illustration of the formation of detached bounds as a result of summed errors. Sub-figures *a* and *b* correspond to $y_0^1 = \text{ReLU}(x_0^0 + x_1^0)$, $y_1^1 = \text{ReLU}(x_0^0 - x_1^0)$ and their relaxed upper bounds (in orange); and sub-figure *c* corresponds to $x_0^2 = y_0^1 + y_1^1$ and its symbolic upper bound, computed using back-substitution.

with current bounds $l < 0 < u$ is:

$$E(x) = \frac{u}{u-l}(x-l) - \sigma(x) \quad x \in [l, u]$$

and we note that $E(l) = E(u) = 0$. In more complex cases, such as the case of the multivariate function $x_0^2 = y_0^1 + y_1^1$ depicted in Fig. 4, the coordinates where the bound error equals zero could be different for y_0^1 and y_1^1 — resulting in the bound obtained for x_0^2 , their sum, becoming detached from the true value of the function. We now show it for the case of x_0^2 in greater detail:

$$x_0^2 = \sigma(x_0^1) + \sigma(x_1^1) = \sigma(x_0^0 + x_1^0) + \sigma(x_0^0 - x_1^0)$$

An upper bound is computed using the relaxations:

$$\begin{aligned} \sigma(x_0^0 + x_1^0) &\leq \frac{1}{2}(x_0^0 + x_1^0 + 2) \\ \sigma(x_0^0 - x_1^0) &\leq \frac{1}{2}(x_0^0 + x_1^0 + 2) \end{aligned}$$

where each relaxation has its own relaxation error:

$$\begin{aligned} E_0^1(x_0^0, x_1^0) &= \frac{1}{2}(x_0^0 + x_1^0 + 2) - \sigma(x_0^0 + x_1^0) \\ E_1^1(x_0^0, x_1^0) &= \frac{1}{2}(x_0^0 + x_1^0 + 2) - \sigma(x_0^0 - x_1^0) \end{aligned}$$

The relaxed linear bound obtained is:

$$x_0^2 \leq \frac{1}{2}(x_0^0 + x_1^0 + 2) + \frac{1}{2}(x_0^0 + x_1^0 + 2) = x_0^0 + 2$$

And its error is the sum of the errors of its summands:

$$\begin{aligned} E_{\text{total}}(x_0^0, x_1^0) &\equiv E_0^1 + E_1^1 \\ &= x_0^0 + 2 - \sigma(x_0^0 + x_1^0) - \sigma(x_0^0 - x_1^0) \end{aligned}$$

We note that:

$$\begin{aligned} \min(E_0^1) &= E_0^1(-1, -1) = E_0^1(1, 1) = 0 \\ \min(E_1^1) &= E_1^1(-1, 1) = E_1^1(1, -1) = 0 \end{aligned}$$

However:

$$\min(E_{\text{total}}) = E_{\text{total}}(-1, x_1^0) = 1$$

The reason for this is that at the coordinates $\langle -1, -1 \rangle$ and $\langle 1, 1 \rangle$ where $E_0^1(-1, -1) = E_0^1(1, 1) = 0$, we have that $E_1^1(-1, -1) = E_1^1(1, 1) = 1$; and vice-versa, for the coordinates $\langle -1, 1 \rangle$ and $\langle 1, -1 \rangle$, where $E_1^1(-1, 1) = E_1^1(1, -1) = 0$ and $E_0^1(-1, 1) = E_0^1(1, -1) = 1$. The optimal linear bound for

$$x_0^2 = \sigma(x_0^0 + x_1^0) + \sigma(x_0^0 - x_1^0)$$

is in fact $x_0^2 \leq x_0^0 + 1$, which is the bias-optimal version of the existing linear bound of $x_0^2 \leq x_0^0 + 2$.

IV. DEEPMIP: MINIMIZING BACK-SUBSTITUTION ERRORS

During a back-propagation execution, the over-approximations of individual ReLUs are repeatedly summed up, which leads to bounds that become increasingly more detached with each iteration — and this results in very loose concrete bounds that hamper verification. We now describe our method, which we term *DeepMIP*, for “tightening” detached bounds, with the goal of eventually obtaining tighter concrete bounds. The idea is to alter the back-propagation mechanism, so that in each iteration it *minimizes* the sum of errors that result from the relaxation of the current activation layer — effectively pushing loose upper bounds down towards the function, by decreasing their bias values (a symmetrical mechanism can be applied for lower bounds). More specifically, we propose to rewrite the general back-substitution rule for a single iteration as follows:

$$\begin{aligned} \max(x_i^k) &= \max(W_i^{k-1} \sigma(\mathbf{x}^{k-1})) \\ &= \max(W_i^{k-1} R_U^{k-2} \mathbf{x}^{k-1} \\ &\quad - (W_i^{k-1} R_U^{k-2} \mathbf{x}^{k-1} - W_i^{k-1} \sigma(\mathbf{x}^{k-1}))) \\ &= \max(W_i^{k-1} R_U^{k-2} \mathbf{x}^{k-1} - E^{k-1}) \\ &\leq \max(W_i^{k-1} R_U^{k-2} \mathbf{x}^{k-1}) - \min(E^{k-1}) \end{aligned}$$

Observe that while $\min(E^{k-1})$ is non-convex, it contains no nested ReLUs, and can often be efficiently solved by MIP solvers [49]. Thus, as DeepMIP performs the iterative back-substitution process, it can invoke a MIP solver to minimize the error in each iteration, and use it to improve the deduced bounds. The pseudo-code for the algorithm appears in the full version of this paper [56]. Observe that MiniMIP can be regarded as a generalization of modern back-substitution methods [45], [54], in the sense that they only use the non-negativity of the error to produce a trivial bound:

$$\min(E^{k-1}) = \min(W_i^{k-1} R_U^{k-2} \mathbf{x}^{k-1} - W_i^{k-1} \sigma(\mathbf{x}^{k-1})) \geq 0$$

which is correct, since the error of an upper bound is non-negative by definition (in the lower bound case, the error is non-positive, and so 0 can be used as a trivial upper bound).

To continue our computation we denote the error caused by the over-approximation of the activation of layer t during back-substitution as:

$$E^t \equiv W_i^{k-1} \prod_{j=k-2}^{t-1} (R_U^j W^j) (R_U^{t-1} \mathbf{x}^t - \sigma(\mathbf{x}^t)) \quad (2)$$

In the definition above, i is the index of the neuron being bounded by the back-substitution. We get:

$$\begin{aligned} \max(x_i^k) &\leq \max(W_i^{k-1} R_U^{k-2} \mathbf{x}^{k-1}) - \min(E^{k-1}) \\ &= \max(W_i^{k-1} R_U^{k-2} W^{k-2} \sigma(\mathbf{x}^{k-2})) - \min(E^{k-1}) \\ &= \max(W_i^{k-1} R_U^{k-2} W^{k-2} R_U^{k-2} \mathbf{x}^{k-2} - E^{k-2}) \\ &\quad - \min(E^{k-1}) \\ &\leq \max(W_i^{k-1} R_U^{k-2} W^{k-2} R_U^{k-2} \mathbf{x}^{k-2}) \\ &\quad - \min(E^{k-2}) - \min(E^{k-1}) \\ &= \dots \\ &\leq \max(W_i^{k-1} \prod_{j=k-2}^0 (R_U^j W^j) \mathbf{x}^0) - \sum_{j=k-1}^0 \min(E^j) \end{aligned}$$

Finally, the maximization problem is transformed into a linear sum over a box domain, which is easy to solve. Since each E^j is shallow (contains no nested ReLUs), it can be minimized efficiently using MIP solvers, and each non-trivial minimum that is found will improve the tightness of the final upper bound. However, we note that the number of MIP problems generated by this process increases linearly with the *depth* of the neuron within the network — i.e., for a neuron in layer k , there are k minimization problems to solve. For deeper networks, especially ones with large domains or ones where many layers only have very loose bounds, minimizing the error terms could become computationally expensive.

Optimization: Direct MIP encoding. As part of its operation, DeepMIP dispatches MIP problems, each corresponding to the over-approximation error of a particular layer. Specifically when it over-approximates the first layer:

$$\begin{aligned} &\max(W_i^{k-1} \prod_{j=k-2}^1 (R_U^j W^j) \sigma(W^0 \mathbf{x}^0)) - \sum_{j=k-2}^1 \min(E^j) \\ &\leq \max(W_i^{k-1} \prod_{j=k-2}^0 (R_U^j W^j) \mathbf{x}^0) - \min(E^0) \\ &\quad - \sum_{j=k-2}^1 \min(E^j) \end{aligned}$$

it will directly solve the linear optimization problem:

$$\max(W_i^{k-1} \prod_{j=k-2}^0 (R_U^j W^j) \mathbf{x}^0)$$

and use a MIP solver to solve:

$$\min(E^0) = \min\left(W_i^{k-1} \prod_{j=k-2}^1 (R_U^j W^j) (R_U^0 \mathbf{x}^t - \sigma(\mathbf{x}^0))\right)$$

We observe that in this particular case, since we reached the input layer, the initial term can instead be directly solved as a separate MIP query:

$$\max(W_i^{k-1} \prod_{j=k-2}^1 (R_U^j W^j) \sigma(W^0 x^0))$$

which may result in tighter bounds, since it prevents any additional imprecision. We note that this optimization to DeepMIP generalizes the common practice of directly finding the concrete bounds of the neurons in the first layer using MIP solvers, and only applying back-substitution from the second layer onward [37], [54].

We illustrate this approach by repeating the back-substitution process for x_0^3 from our running example:

$$\begin{aligned} \max(x_0^3) &= \max(y_0^2 + y_1^2 + y_2^2) \\ &= \max(\sigma(x_0^2) + \sigma(x_1^2) + \sigma(x_2^2)) \\ &= \max\left(\sigma(y_0^1 + y_1^1) + \sigma(-y_0^1 + y_1^1 + y_2^1) \right. \\ &\quad \left. + \sigma(-y_0^1 + y_1^1 - y_2^1)\right) \\ &= \max(A - E_U^2) \leq \max(A) - \min(E_U^2) \end{aligned}$$

where

$$\begin{aligned} A &= (y_0^1 + y_1^1) + \frac{3}{5}(-y_0^1 + y_1^1 + y_2^1) + \frac{2}{5}(-y_0^1 + y_1^1 - y_2^1) + \frac{12}{5} \\ &= 2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5} \end{aligned}$$

and E_U^2 is defined as per Eq. 2:

$$\begin{aligned} E_U^2 &= (y_0^1 + y_1^1) + \frac{3}{5}(-y_0^1 + y_1^1 + y_2^1) \\ &\quad + \frac{2}{5}(-y_0^1 + y_1^1 - y_2^1) + \frac{12}{5} - \sigma(y_0^1 + y_1^1) \\ &\quad - \sigma(-y_0^1 + y_1^1 + y_2^1) - \sigma(-y_0^1 + y_1^1 - y_2^1) \\ &= 2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5} - \sigma(y_0^1 + y_1^1) \\ &\quad - \sigma(-y_0^1 + y_1^1 + y_2^1) - \sigma(-y_0^1 + y_1^1 - y_2^1) \end{aligned}$$

Simplifying these expressions, we get that

$$\begin{aligned} \max(x_0^3) &\leq \max(A) - \min(E_U^2) \\ &= \max(2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5}) - \min(E_U^2) \end{aligned}$$

Using a MIP solver to find the minimum of E_U^2 over the variables of y^1 reveals that $\min(E_U^2) = \frac{2}{5}$. We substitute this, and get:

$$\max(x_0^3) \leq \max(2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5}) - \frac{2}{5}$$

Finally, since we have reached the first layer, we write:

$$\begin{aligned} \max(x_0^3) &\leq \max(2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5}) - \frac{2}{5} \\ &= \max(2\sigma(x_1^1) + \frac{1}{5}\sigma(x_2^1) + \frac{12}{5}) - \frac{2}{5} \\ &= \max(2\sigma(x_0^0 - x_1^0) + \frac{1}{5}\sigma(x_2^0) + \frac{12}{5}) - \frac{2}{5} \end{aligned}$$

and then, using our proposed enhancement, we directly solve this maximization over the input layer instead of back-substituting it any further. The MIP solver replies that:

$$\max(2\sigma(x_0^0 - x_1^0) + \frac{1}{5}\sigma(x_2^0) + \frac{12}{5}) = 6\frac{2}{5}$$

and we then substitute this value to obtain:

$$\max(x_0^3) \leq 6\frac{2}{5} - \frac{2}{5} = 6$$

As we can see, minimizing the errors by using MIP (which is very fast in practice) allows us to back-substitute bounds with optimal bias, which yields tighter bounds for the output variable.

MiniMIP. While DeepMIP produces very strong bounds, for each neuron it must solve multiple MIP instances during back-substitution — many of them for bounds that may already be bias-optimal. This large number of instances to solve can result in a large overhead, and makes it worthwhile to explore heuristics for only solving *some* of these instances.

To illustrate this, we propose a particular, aggressive heuristic that we call *MiniMIP*. Instead of minimizing all error terms during back-substitution, MiniMIP only solves the final query in this series — that is, the query in which the bounds of the current layer are expressed as sums of ReLUs of input neurons. This approach significantly reduces overhead: exactly one MIP instance is solved in each iteration, regardless of the depth of the layer currently being processed. As we later see in our evaluation, even this is already enough to achieve state-of-the-art performance and very tight bounds; and the resulting queries can be solved very efficiently [49].

V. EVALUATION

Implementation. For evaluation purposes, we created a proof-of-concept implementation of our approach in Python. The implementation code, alongside all the benchmarks described in this section, is publicly available online [55]. Our implementation uses the PyTorch library [40] for computing the optimal value of α for each ReLU’s triangle relaxation, as is done in other tools [54]. We use Gurobi [26] as the MIP solver for the minimization of errors and direct concretization of bounds. We ran all experiments on a compute cluster consisting of Xeon E5-2637 CPUs, and a 2-hour timeout per experiment. We note that our implementation currently runs on CPUs only, and extending it to support GPUs is left for future work.

Abstraction refinement cascade. For each verification query, prior to applying our iterative error minimization scheme, we configured our implementation to first run a light-weight, “ordinary” symbolic-bound propagation pass. Specifically, we ran a single pass of the DeepPoly mechanism [45]. A similar technique is applied by other tools [37].

Benchmarks. We evaluated our approach on fully-connected, ReLU networks trained over the MNIST dataset, taken from the ERAN repository [19]. The topologies of the networks we used appear in Table I.

TABLE I: The DNNs used in our evaluation.

Dataset	Model	Type	Neurons	Hidden Layers	Activation
MNIST	6×100	FC	510	5	ReLU
	9×100		810	8	
	6×200		1010	5	
	9×200		1610	8	

For verification queries, we followed standard practice [31], [37], [54], and attempted to prove the *adversarial robustness* of the first 1000 images of the MNIST test set: that is, we used verification to try and prove that ϵ -perturbations to correctly classified inputs in the dataset cannot change the classification assigned by the DNN.

We compared the DeepMIP approach (specifically, Min-iMIP) to two state-of-the-art verification approaches [9]: the PRIMA solver [37], and our implementation of the α -CROWN method [54], which represents the state of the art in symbolic-bound tightening with back-substitution. Indeed, many other verification tools integrate back-substitution with additional techniques, such as search-based techniques [32] or abstraction-refinement [7], making it more difficult to measure the effectiveness of the back-substitution component alone. However, since the α -CROWN implementation in our evaluation also served as the baseline back-substitution method to which we added our methods, any difference between the two is solely due to the addition of our suggested technique. The results of our experiments are summarized in Table II. Recall that symbolic-bound propagation techniques are incomplete, and may fail to prove a given query; the *Solved* columns indicate the number of instances (out of 1000) that each method was able to prove to be robust to adversarial perturbations. The *Time* columns indicate the run time of each method (including timeouts), averaged over the 1000 benchmarks solved.

Our results clearly indicate the superiority of the bounds discovered by DeepMIP: indeed, in all categories, our approach was able to solve the largest number of instances, solving a total of 2378 instances, compared to 2183 instances solved by PRIMA (198 extra instances solved) and 1087 instances solved by α -CROWN (1291 extra instances solved). These improvements come with an overhead, due to the additional MIP queries that need to be solved: our approach is approximately 5.6 times slower than α -CROWN, and 2.5 times slower than PRIMA. Furthermore, DeepMIP timed out on 2 out of the 3829 total benchmarks tested ($\approx 0.05\%$), while PRIMA and α -CROWN did not have any timeouts.

The main conclusions that we draw from these experiments are that (i) the DeepMIP approach has a significant potential for solving queries that other approaches cannot; and (ii) additional work, in the form of improved heuristics, engineering improvements, and support for GPUs is still required to make our approach faster. Our results also indicate that a portfolio-based approach, which starts from light-weight techniques and then progresses towards DeepMIP for difficult queries, could enjoy the benefits of both worlds.

VI. RELATED WORK

The topic of DNN verification has been receiving significant attention from the formal methods community, and various tools and methods have been proposed for addressing it. These include techniques that leverage SMT solvers (e.g., [27], [32], [39], [53]), LP and MILP solvers (e.g., [13], [15], [36], [49]), reachability analysis [47], abstraction-refinement techniques [7], [16], [17], and many others. The techniques most related to DeepMIP are those that rely on the propagation of symbolic bounds using abstract interpretation (e.g., [21], [50]–[52]). Recent work has also extended beyond answering binary questions about DNNs, instead targeting tasks such as automated DNN repair [23], [34], DNN simplification [22], [35], ensemble selection [3], and quantitative verification and optimization [10], [46]; and also the verification of recurrent neural networks [28], [41], [57] and reinforcement-learning based systems [4], [18], [29]. Our proposed techniques could be integrated into any number of these approaches.

Bound propagation has been playing a significant part in DNN verification efforts for the past few years. Starting with interval-arithmetic-based propagation [31] and optimization queries for individual neurons [15], [49], these approaches have progressed to use various relaxations and over-approximations for individual neurons [21], [45], [51] and sets thereof [37], [38], [44], culminating in highly sophisticated approaches [37], [54]. We consider our work as another step in this very promising research direction.

VII. CONCLUSION AND FUTURE WORK

We presented an enhancement to the popular back-substitution procedure, which includes a formulation of the over-approximation errors introduced during back-substitution. These errors can then be minimized, in order to greatly tighten the resulting bounds. Our approach achieves tighter bounds than state-of-the-art approaches, but at the cost of longer running times; and we are currently exploring methods for expediting it. Specifically, moving forward, we intend to focus on adding support for GPUs; on better refinement heuristics; on better MIP encoding [6]; and also on improving the core algorithm to utilize previously calculated bounds and errors. Furthermore, we intend to generalize our methods to other abstract domains, and also to integrate them with search-based techniques.

ACKNOWLEDGEMENTS

The project was partially supported by the Israel Science Foundation (grant number 683/18) and by the Binational Science Foundation (grant number 2020250).

APPENDIX A RELAXATION MATRICES

The matrices R_U^t and R_L^t are how we apply the triangle relaxation during back-substitution over layer t . for example if:

$$x_j^{i+1} = \sigma(x_0^i) - 2\sigma(x_1^i)$$

TABLE II: Comparing DeepMIP to α -CROWN and PRIMA.

Model	ϵ	α -CROWN		PRIMA		DeepMIP (MiniMIP)	
		Solved	Time (seconds)	Solved	Time (seconds)	Solved	Time (seconds)
6×100	0.026	207	38	504	123	581	302
9×100	0.026	223	88	427	252	463	452
6×200	0.015	349	93	652	222	709	801
9×200	0.015	308	257	600	462	625	1121
Total		1087	476	2183	1059	2378	2676

then in order to find a linear upper bound for x_j^{i+1} , we need to replace $\sigma(x_0^i)$ with its triangle-relaxation upper bound (since it has a positive coefficient), and $\sigma(x_1^i)$ with its triangle-relaxation lower bound. This gives rise to:

$$x_j^{i+1} \leq \frac{u_0^i}{u_0^i - l_0^i} (x_0^i - l_0^i) - 2\alpha x_1^i$$

which can be written as (for some constant term d):

$$x_j^{i+1} \leq \frac{u_0^i}{u_0^i - l_0^i} x_0^i - 2\alpha x_1^i + d$$

Written as a vector product:

$$x_j^{i+1} = [1 \quad -2] \cdot \begin{bmatrix} \sigma(x_0^i) \\ \sigma(x_1^i) \end{bmatrix} \leq [1 \quad -2] \cdot \begin{bmatrix} \frac{u_0^i}{u_0^i - l_0^i} & 0 \\ 0 & \alpha \end{bmatrix} \cdot \begin{bmatrix} x_0^i \\ x_1^i \end{bmatrix} + d$$

We use R_U^i to denote the matrix that was used to relax $\sigma(x^i)$, and observe that it depends on the weights/coefficients of each non-linearity about to be relaxed, and also on the existence of $[l^i, u^i]$ in order to compute the corresponding relaxations. Formally we define the matrix $R_U^i(\omega^t, l^t, u^t)$ as:

$$R_U^i(\omega^t, l^t, u^t)[i, j] = 0 \quad i \neq j$$

$$R_U^i(\omega^t, l^t, u^t)[i, i] \equiv \begin{cases} 1 & \text{if } l_i^t \geq 0 \\ 0 & \text{if } u_i^t \leq 0 \\ \frac{u_i^t}{u_i^t - l_i^t} & \text{if } \omega_i^t \geq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \\ \alpha & \text{if } \omega_i^t \leq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \end{cases}$$

where ω^t is a row vector such that ω_i^t contains the coefficient of $\sigma(x_i^t)$, and l^t, u^t are vectors such that $l_i^t \leq x_i^t \leq u_i^t$. Similarly, we define $R_L^t(\omega^t, l^t, u^t)$ as:

$$R_L^t(\omega^t, l^t, u^t)[i, j] = 0 \quad i \neq j$$

$$R_L^t(\omega^t, l^t, u^t)[i, i] \equiv \begin{cases} 1 & \text{if } l_i^t \geq 0 \\ 0 & \text{if } u_i^t \leq 0 \\ \frac{u_i^t}{u_i^t - l_i^t} & \text{if } \omega_i^t \leq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \\ \alpha & \text{if } \omega_i^t \geq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \end{cases}$$

We note that there exists similar matrices for updating the constant term during back-substitution; we omit them to reduce clutter. Furthermore, when it is clear from context, we write R_L^t, R_U^t instead of $R_L^t(\omega^t, l^t, u^t), R_U^t(\omega^t, l^t, u^t)$.

REFERENCES

- [1] M. Akintunde, A. Kevochian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 197–210, 2019.
- [2] M. AlQuraishi. AlphaFold at CASP13. *Bioinformatics*, 35(22):4862–4865, 2019.
- [3] G. Amir, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2022.
- [4] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.
- [5] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. Concrete Problems in AI Safety, 2016. Technical Report. <https://arxiv.org/abs/1606.06565>.
- [6] R. Anderson, J. Huchette, C. Tjandraatmadja, and J. Vielma. Strong Mixed-Integer Programming Formulations for Trained Neural Networks, 2018. Technical Report. <http://arxiv.org/abs/1811.08359>.
- [7] P. Ashok, V. Hashemi, J. Kretinsky, and S. Mohr. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 92–107, 2020.
- [8] G. Awni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.
- [9] S. Bak, C. Liu, and T. Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, 2021. Technical Report. <http://arxiv.org/abs/2109.00498>.
- [10] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks And its Security Applications. In *Proc. 26th ACM Conf. on Computer and Communication Security (CCS)*, 2019.
- [11] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.
- [12] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. <http://arxiv.org/abs/1604.07316>.
- [13] R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 4795–4804, 2018.
- [14] T. Dreossi, D. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. Seshia. VeriAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 432–442, 2019.
- [15] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
- [16] Y. Elboher, E. Cohen, and G. Katz. Neural Network Verification using Residual Reasoning. In *Proc. 20th Int. Conf. on Software Engineering and Formal Methods (SEFM)*, 2022.
- [17] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.
- [18] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group*

- on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 305–318, 2021.
- [19] ERAN. The ERAN Repository, 2022. <https://github.com/eth-sri/eran>.
- [20] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, 2018.
- [21] T. Gehr, M. Mirman, D. Drachler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
- [22] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.
- [23] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.
- [24] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
- [25] D. Gunning. Explainable Artificial Intelligence (XAI), 2017. Defense Advanced Research Projects Agency (DARPA) Project.
- [26] Gurobi. The Gurobi MILP Solver, 2021. <https://www.gurobi.com/>.
- [27] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
- [28] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 57–74, 2020.
- [29] P. Jin, J. Tian, D. Zhi, X. Wen, and M. Zhang. Trainify: A CEGAR-Driven Training and Verification Framework for Safe Deep Reinforcement Learning. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, pages 193–218, 2022.
- [30] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
- [31] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.
- [32] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
- [33] W. Kokke, E. Komendantskaya, D. Kienitz, R. Atkey, and D. Aspinall. Neural Networks, Secure by Construction: An Exploration of Refinement Types. In *Proc. 18th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 67–85, 2020.
- [34] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.
- [35] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.
- [36] A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. <http://arxiv.org/abs/1706.07351>.
- [37] M. Müller, G. Makarchuk, G. Singh, M. Puschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In *Proc. 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2022.
- [38] M. Ostrovsky, C. Barrett, and G. Katz. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In *Proc. 20th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2022.
- [39] L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.
- [40] PyTorch. The PyTorch Library, 2022. <https://pytorch.org/>.
- [41] W. Ryou, J. Chen, M. Balunovic, G. Singh, A. Dan, and M. Vechev. Scalable Polyhedral Verification of Recurrent Neural Networks. In *33rd Int. Conf. on Computer Aided Verification (CAV)*, pages 225–248, 2021.
- [42] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
- [43] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014. Technical Report. <http://arxiv.org/abs/1409.1556>.
- [44] G. Singh, R. Ganvir, M. Puschel, and M. Vechev. Beyond the Single Neuron Convex Barrier for Neural Network Certification. In *Proc. 33rd Conf. on Neural Information Processing Systems (NeurIPS)*, 2019.
- [45] G. Singh, T. Gehr, M. Puschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.
- [46] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. *Journal of Machine Learning*, pages 1–28, 2021.
- [47] X. Sun, K. H., and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.
- [48] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. <http://arxiv.org/abs/1312.6199>.
- [49] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. <http://arxiv.org/abs/1711.07356>.
- [50] H. Tran, S. Bak, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 18–42, 2020.
- [51] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.
- [52] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. Dhillon, and L. Daniel. Towards Fast Computation of Certified Robustness for ReLU Networks, 2018. Technical Report. <http://arxiv.org/abs/1804.09699>.
- [53] H. Wu, A. Zeljić, G. Katz, and C. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 143–163, 2022.
- [54] K. Xu, H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C.-J. Hsieh. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers, 2020. Technical Report. <http://arxiv.org/abs/2011.13824>.
- [55] T. Zelazny, H. Wu, C. Barrett, and G. Katz. DeepMIP Code, 2022. <https://doi.org/10.5281/zenodo.6982973>.
- [56] T. Zelazny, H. Wu, C. Barrett, and G. Katz. On Optimizing Back-Substitution Methods for Neural Network Verification (Full Version), 2022. Technical Report. <https://arxiv.org/abs/2208.07669>.
- [57] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.

Verification-Aided Deep Ensemble Selection

Guy Amir, Tom Zelazny, Guy Katz and Michael Schapira
 The Hebrew University of Jerusalem, Jerusalem, Israel
 {guyam, tomz, guykatz, schapiram}@cs.huji.ac.il

Abstract—Deep neural networks (DNNs) have become the technology of choice for realizing a variety of complex tasks. However, as highlighted by many recent studies, even an imperceptible perturbation to a correctly classified input can lead to misclassification by a DNN. This renders DNNs vulnerable to strategic input manipulations by attackers, and also over-sensitive to environmental noise. To mitigate this phenomenon, practitioners apply joint classification by an *ensemble* of DNNs. By aggregating the classification outputs of different individual DNNs for the same input, ensemble-based classification reduces the risk of misclassifications due to the specific realization of the stochastic training process of any single DNN. However, the effectiveness of a DNN ensemble is highly dependent on its members *not simultaneously erring* on many different inputs. In this case study, we harness recent advances in DNN verification to devise a methodology for identifying ensemble compositions that are less prone to simultaneous errors, even when the input is adversarially perturbed — resulting in more *robustly-accurate* ensemble-based classification. Our proposed framework uses a DNN verifier as a backend, and includes heuristics that help reduce the high complexity of directly verifying ensembles. More broadly, our work puts forth a novel universal objective for formal verification that can potentially improve the robustness of real-world, deep-learning-based systems across a variety of application domains.

I. INTRODUCTION

In recent years, deep learning [33] has emerged as the state-of-the-art solution for a myriad of tasks. Through the automated training of *deep neural networks* (DNNs), engineers can create systems capable of correctly handling previously unencountered inputs. DNNs excel at tasks ranging from image recognition and natural language processing to game playing and protein folding [2], [21], [38], [48], [74], [75], and are expected to play a key role in various complex systems [15], [44].

Despite their immense success, DNNs suffer from severe vulnerabilities and weaknesses. A prominent example is the sensitivity of DNNs to *adversarial inputs* [34], [49], [80], i.e., slight perturbations of correctly-classified inputs that result in misclassifications. The susceptibility of DNNs to input perturbations involves two risks that limit the applicability of deep learning to mission-critical tasks: (1) falling victim to strategic input manipulations by *attackers*, and (2) failing to *generalize* well in the presence of environmental noise. In light of the above, recent work has focused on enhancing the *robustness* of DNN-based classification to adversarial inputs while preserving *accuracy* [13], [29], [62], [82], [97]. Informally, a classifier is *robustly accurate* (aka *astute* [86]) with respect to a given distribution over inputs, if it continues to correctly classify inputs drawn from this distribution, with high

probability, even when these inputs are arbitrarily perturbed (up to some maximally allowed perturbation).

We focus here on a classic technique for improving classification quality [9], [52]: combining the outputs of an *ensemble* [28], [37], [81] of DNN-based classifiers on an input to derive a joint classification decision for that input. By incorporating the outputs of *independently-trained* DNNs, ensembles mitigate the risk of misclassification of a single DNN due to a specific realization of its stochastic training process and the specifics of its training data traversal. For a DNN ensemble to provide a meaningful improvement over utilizing a single DNN, its members should not frequently misclassify *the same* input. Consider, for instance, an extreme example, where an ensemble with $k = 10$ members is used, but for some part of the input space, the 10 DNNs effectively behave identically, making mistakes on the exact same inputs. In this scenario, the ensemble as a whole is no more robust on this input subspace than each of its individual members. Our objective is to demonstrate how recent advances in DNN verification [40], [45] can be harnessed to provide system designers and engineers with the means to avoid such scenarios, by constructing adequately diverse ensembles.

Significant progress has recently been made on formal verification techniques for DNNs [1], [8], [11], [12], [26], [56], [67], [76], [90]. The basic DNN verification query is to determine, given a DNN N , a precondition P , and a postcondition Q , whether there exists an input x such that $P(x)$ and $Q(N(x))$ both hold. Recent verification work has focused on *identifying* adversarial inputs to DNN-based classification, or formally proving that no such inputs exist [30], [35], [58]. We demonstrate the applicability of DNN verification to solving a new kind of queries, pertaining to DNN ensembles, which could significantly boost the robustness of these ensembles (as opposed to just measuring the robustness of individual DNNs). We note that despite great strides in recent years [47], [58], [76], even state-of-the-art DNN verification tools face severe scalability limitations. This renders solving verification queries pertaining to ensembles extremely challenging, since the complexity of this task grows exponentially with the number of ensemble members (see Section III).

In this case-study paper, we propose and evaluate an efficient and scalable approach for verifying that different ensemble members do not tend to err simultaneously. Specifically, our scheme considers *small subsets* of ensemble members,¹

¹While our technique is applicable to subsets of any size, we focused on pairs in our evaluation, as we later elaborate.

and dispatches verification queries to seek perturbations of inputs for which *all* members in the subset err *simultaneously*. By identifying such inputs, we can assign a *mutual error score* to each subset. Using these mutual error scores, we compute, for each individual ensemble member, a *uniqueness score* that signifies how often it errs simultaneously with other ensemble members. This score can be used to detect the “weakest” ensemble members, i.e. those most prone to erring in parallel to others, and replace them with fresh DNNs — thus enhancing the diversity among the ensemble members, and improving the overall robust accuracy of the ensemble.

To evaluate our scheme, we implemented it as a proof-of-concept tool, and used this tool to conduct extensive experimentation on DNN ensembles for classifying digits and clothing items. Our results demonstrate that by identifying the weakest ensemble members (using verification) and replacing them, the robust accuracy of the ensemble as a whole may be significantly improved. Additional experiments that we conducted also demonstrate that our verification-driven approach affords significant advantages when compared to competing, non-verification-based, methods. Together, these results showcase the potential of our approach. Our code and benchmarks are publicly available online [6].

The rest of the paper is organized as follows. Section II contains background on DNN ensembles and DNN verification. In Section III we present our verification-based methodology for ensemble selection, and then present our case study in Section IV. Next, in Section V we compare our verification-based approach to state-of-the-art, gradient-based, methods. Related work is covered in Section VI, and we conclude and discuss future work in Section VII.

II. BACKGROUND

Deep Neural Networks. A deep neural network (DNN) [33] is a directed graph, comprised of layers of nodes (also known as *neurons*). In feed-forward DNNs, data flows sequentially from the first (*input*) layer, through a sequence of intermediate (*hidden*) layers, and finally into an *output* layer. The network’s output is evaluated by assigning values to the input layer’s neurons and computing the value assignment for neurons in each of the following layers, in order, until reaching the output layer and returning its neuron values to the user. In classification networks, which are our subject matter here, each output neuron corresponds to an output *class*; and the output neuron with the highest value represents the class, or label, which the particular input is being classified as.

Fig. 1 depicts a toy DNN. It has an input layer with two neurons, followed by a *weighted sum layer*, which computes an affine transformation of values from its preceding layer. For example, for input $V_1 = [1, -5]^T$, the second layer’s computed values are $V_2 = [-8, 1]^T$. Next is a ReLU layer, which applies the ReLU function $\text{ReLU}(x) = \max(0, x)$ to each individual neuron, resulting in $V_3 = [0, 1]^T$. Finally, the network’s output layer again computes an affine transformation, resulting in the output $V_4 = [6, 3]^T$. Thus, input $[1, -5]^T$ is classified as

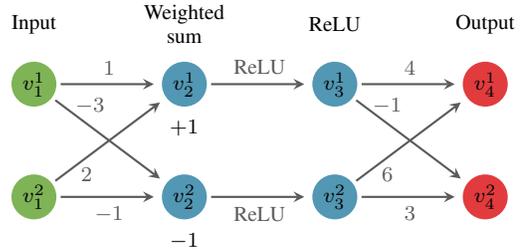


Fig. 1: A toy DNN.

the label corresponding to neuron v_4^1 . For additional details, see [33].

Accuracy, Robustness, and Deep Ensembles. The weights of a DNN are determined through its training process. In supervised learning, we are provided a set of pairs (x_i, l_i) drawn according to some (unknown) distribution D , where x_i is an input point and l_i is a ground-truth label for that input. The goal is to select weights for the DNN N that maximize its *accuracy*, which is defined as: $Pr_{(x,l) \sim D}(N(x) = l)$ (we slightly abuse notation, and use $N(x)$ to denote both the network’s output vector, as well as the label it assigns x).

We restrict our attention to the *classification* setting, in which labels are discrete. The training of a DNN-based classifier is typically a stochastic process. This process is affected, for example, by the initial assignment of weights to the DNN, the order in which training data is traversed, and more. A prominent method for avoiding misclassifications originating from the stochastic training of a single DNN is employing *deep ensembles*. A deep ensemble is a set $\mathcal{E} = \{N_1, \dots, N_k\}$ of k independently-trained DNNs. The ensemble classifies an input by aggregating the individual classification outputs of its members (see Fig. 2). The collective decision is typically achieved by averaging over all members’ outputs. Ensembles have been shown to often achieve better accuracy than their individual members [9], [52], [57], [92].

A critical condition for the success of ensemble-based classifiers is that the ensemble members’ misclassifications are not strongly correlated [53], [63], [79]. This key property is crucial in order to avoid a scenario where many different members of the ensemble frequently make mistakes on the same input, causing the ensemble as a whole to also err on that input. Heuristics for achieving diversity across ensemble members include, e.g., training the members simultaneously with diversity-aware loss [43], [52], randomly initializing different weights for the ensemble members [50], and other methods [63], [73].

Since the discovery of adversarial inputs, practitioners have become interested in DNNs that are not only accurate but also *robustly accurate*. We say that a network N is ϵ -robust around the point x if every input point that is at most ϵ away from x receives the same classification as x : $\|x' - x\| \leq \epsilon \Rightarrow N(x) = N(x')$, where $N(x)$ is the label assigned to x ; and the definition of accuracy is generalized to ϵ -robust

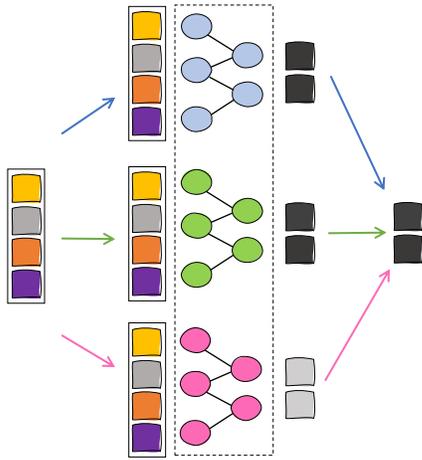


Fig. 2: An ensemble comprising three DNNs. Each input vector is independently classified by all three networks, and the results are aggregated into a final classification.

accuracy as follows: $Pr_{(x,l) \sim D}(\|x' - x\| \leq \epsilon \Rightarrow N(x') = l)$. While improvements in accuracy afforded by ensembles are straightforward to measure, this is typically not the case for robust accuracy, as we discuss in Section III.

DNN Verification. Given a DNN N , a verification query on N specifies a precondition P on N 's input vector x , and a postcondition Q on N 's output vector $N(x)$. A DNN verifier needs to determine whether there exists a concrete input x_0 that satisfies $P(x_0) \wedge Q(N(x_0))$ (the SAT case), or not (the UNSAT case). Typically, P and Q are expressed in the logic of linear real arithmetic. For instance, the ϵ -robustness of a DNN around a point x can be phrased as a DNN verification query, and then dispatched using existing technology [30], [45], [85]. The DNN verification problem is known to be NP-complete [46].

III. IMPROVING ROBUST ACCURACY USING VERIFICATION

A. Directly Quantifying Robust Accuracy is Hard

In order to construct a robustly-accurate ensemble \mathcal{E} with k members, we train a set of $n > k$ DNNs and then seek to select a subset of k DNNs that provides high robust accuracy. This method of training multiple models and then discarding a subset thereof is known as *ensemble pruning*, and is a common practice in deep-ensemble training [14], [98]. In our case, a straightforward approach to do so would be to quantify the robust accuracy for all possible k -sized DNN-subsets, and then pick the best one. This, however, is computationally expensive, and requires an accurate estimate of the robust accuracy of an ensemble.

A natural approach for estimating the ϵ -robust accuracy of a DNN is to verify, for many points in the test data, that the DNN yields an accurate label not only on each data point itself, but also on each and every input derived from that data point via an ϵ -perturbation [30]. The fraction of tested points

for which this is indeed the case can be used to estimate the accuracy of the classifier on the underlying distribution from which the data is generated.

A similar process can be performed for an ensemble $\mathcal{E} = \{N_1, \dots, N_k\}$, by first constructing a single, large DNN $N_{\mathcal{E}}$ that aggregates \mathcal{E} 's joint classification, and then verifying its robustness on a set of points from the test data (see the extended version of this paper [7]). However, this approach faces a significant scalability barrier: the DNN ensemble, $N_{\mathcal{E}}$, comprised of all k member-DNNs is (roughly) k times larger than any of the N_i 's, and since DNN verification becomes exponentially harder as the DNN size increases, $N_{\mathcal{E}}$'s size might render efficient verification infeasible. As we demonstrate later, this is the case even when the constituent networks themselves are fairly small. Our proposed methodology circumvents this difficulty by only solving verification queries pertaining to *very small* sets of DNNs.

B. Mutual Error Scores and Uniqueness Scores

In general, the less likely it is that members of an ensemble err simultaneously with other members, the more accurate the ensemble is. This motivates our definition of mutual error scores below.

Definition 1 (Agreement Points): Given an ensemble $\mathcal{E} = \{N_1, N_2, \dots, N_k\}$, we say that an input point x_0 is an *agreement point* for \mathcal{E} if there is some label y_0 such that $N_i(x_0) = y_0$ for all $i \in [k]$. We let $\mathcal{E}(x_0)$ denote the label y_0 .

As we later discuss, the ϵ -neighborhoods of agreement points are natural locations for detecting hidden tendencies of ensemble members to err together.

Definition 2 (Mutual Errors): Let \mathcal{E} be an ensemble, and let x_0 be an agreement point for \mathcal{E} . Let $B_{x_0, \epsilon}$ be the ϵ -ball around x_0 , $B_{x_0, \epsilon} = \{x \mid \|x - x_0\|_{\infty} \leq \epsilon\}$. We say that N_1 and N_2 have a *mutual error* in B if there exists a point $x \in B_{x_0, \epsilon}$ such that $N_1(x) \neq \mathcal{E}(x_0)$ and $N_2(x) \neq \mathcal{E}(x_0)$.

Intuitively, if N_1 and N_2 have many mutual errors, incorporating both into an ensemble is a poor choice. This naturally gives rise to the following definition:

Definition 3 (Mutual Error Scores): Let A be a finite set of m agreement points in an ensemble \mathcal{E} 's input space, and let B_1, B_2, \dots, B_m denote the ϵ -balls surrounding the points in A . Let N_1, N_2 denote two members of \mathcal{E} . The *mutual error score* of N_1 and N_2 with respect to \mathcal{E} and A is denoted by $ME_{\mathcal{E}, A}(N_1, N_2)$, and defined as:

$$ME_{\mathcal{E}, A}(N_1, N_2) = \frac{|\{i \mid N_1 \text{ and } N_2 \text{ have a mutual error in } B_i\}|}{m}$$

Observe that $ME_{\mathcal{E}, A}(N_1, N_2)$ is always in the range $[0, 1]$. The closer it is to 1, the more mutual errors N_1 and N_2 have, making it unwise to place them in the same ensemble.

Definition 4 (Uniqueness Scores): Given an ensemble $\mathcal{E} = \{N_1, N_2, \dots, N_n\}$ and a set A of agreement points for \mathcal{E} , we define, for each ensemble member N_i , the *uniqueness score* for N_i with respect to \mathcal{E} and A , $\text{US}_{\mathcal{E},A}(N_i)$, as:

$$\text{US}_{\mathcal{E},A}(N_i) = 1 - \frac{\sum_{j \neq i} \text{ME}_{\mathcal{E},A}(N_i, N_j)}{n - 1}$$

The uniqueness score (US) of N_i is the complement of its average mutual error score with the other ensemble members. When this score is close to 0, N_i tends to err simultaneously with other members of the ensemble on points in A . In contrast, the closer the uniqueness score is to 1, the rarer it is for N_i to misclassify the same inputs as other members of the ensemble. Hence, ensemble members with low uniqueness scores are, intuitively, good candidates for replacement.

We point out that our definitions above can naturally be generalized to larger subsets of the ensemble members — thus measuring robust accuracy more precisely, but rendering these measurements more complex to perform in practice.

Computing Mutual Errors. The only computationally complex step in determining the uniqueness scores of individual ensemble members is computing the pairwise mutual errors for the ensemble. To this end, we leverage DNN verification technology. Specifically, given two ensemble members N_1 and N_2 , an agreement point a for the ensemble with label l , and $\epsilon > 0$, an appropriate DNN verification query can be formulated as follows. First, we construct from N_1 and N_2 a single, larger DNN N , which captures N_1 and N_2 simultaneously processing a shared input vector, side-by-side. This network N is then passed to a DNN verifier, with the precondition that the input be restricted to B , an ϵ -ball around a , and the postcondition that (1) among N ’s output neurons that correspond to the outputs of N_1 , the neuron representing l not be maximal, and (2) among N ’s output neurons that correspond to the outputs of N_2 , the neuron representing l not be maximal. Such queries are supported by most available DNN verification engines. We note that this encoding (depicted in Figure 3), where two networks and their output constraints are combined into a single query, is crucial for finding inputs on which both DNNs err *simultaneously*. For additional details, see the extended version of this paper [7].

C. Ensemble Selection using Uniqueness Scores

An Iterative Scheme. Building on our verification-based method for computing mutual error scores, we propose an iterative scheme for constructing an ensemble. Our scheme consists of the following steps:

- 1) independently train a set \mathcal{N} of n DNNs, and identify a set A of m agreement points that are *correctly classified* by all n DNNs.² This is done by sequentially checking points from the validation dataset;
- 2) arbitrarily choose an initial candidate ensemble \mathcal{E} of size $k < n$;

²In our experiments, we arbitrarily chose $k = 5$, $n = 10$ and $m = 200$.

- 3) compute (using a verification engine backend) all mutual error scores for the DNN members comprising \mathcal{E} , with respect to A ;
- 4) compute the uniqueness score for each ensemble member, and identify a DNN member N_l with a low score;
- 5) identify a fresh DNN N_f , not currently in \mathcal{E} , that has a higher uniqueness score than N_l , if one exists, and replace N_l with N_f . Specifically, identify a DNN $N_f \in \mathcal{N} \setminus \mathcal{E}$, such that the uniqueness score of N_f with respect to the ensemble $\mathcal{E} \setminus \{N_l\} \cup \{N_f\}$ and the point set A , namely $\text{US}_{\mathcal{E} \setminus \{N_l\} \cup \{N_f\}, A}(N_f)$, is maximal. If this score is greater than $\text{US}_{\mathcal{E}, A}(N_l)$, replace N_l with N_f , i.e. set $\mathcal{E} := \mathcal{E} \setminus \{N_l\} \cup \{N_f\}$; and
- 6) repeat Steps (3) through (5), until no N_f is found or until the user-provided timeout or maximal iteration count are exceeded.

Intuitively, after starting with an arbitrary ensemble, we run multiple iterations, each time trying to improve the ensemble. Specifically, we identify the “weakest” member of the current ensemble, and replace it with a fresh DNN that obtains a higher uniqueness score relevant to the remaining members — thus ensuring that each change that we make improves the overall robust accuracy on the fixed set of agreement points.

The greedy search procedure is repeated for the new candidate ensemble, and so on. The process terminates after a predefined number of iterations is reached, when the process converges (no further improvement is achievable on the fixed set of agreement points), or when a predefined timeout value is exceeded.

On the Importance of Agreement Points. Our iterative scheme for constructing an ensemble starts with an arbitrary selection of k candidate members, and then computes the uniqueness score for each member. As mentioned, the uniqueness scores are computed with respect to a fixed set of agreement points, pre-selected from the validation data (which is labeled data, not used for training the DNNs).

We point out that agreement points are data points on which there is overwhelming consensus among ensemble members, despite the specific realization of the training process of each member. As such, agreement points correspond to data points that are “easy” to label correctly. Consequently, data points in close proximity of an agreement point are rarely classified differently than the agreement point by an individual ensemble member, let alone by multiple members simultaneously. As our objective is to expose implicit tendencies of ensemble members to err together, the close neighborhood of agreement points is a natural area for seeking joint deviations from the consensual label (which are expected to be extremely rare). In our evaluation, we computed uniqueness scores based solely on *correctly-classified* agreement points and ignored any incorrectly-classified agreement points.³

As we later demonstrate, a small set of correctly-classified agreement points from the validation set can be used, in

³For example, in our MNIST experiments 99.7% of the agreement points were correctly classified by all individual DNNs, and by the ensemble as a whole.

practice, to identify ensemble members that tend to err simultaneously on *other* data points. We note that this is also the case even when the chosen agreement points are all identically labeled.

Monotonicity and Convergence. Using our approach, an ensemble member is replaced with a fresh DNN only if this replacement leads to *strictly* fewer joint errors with the *remaining members* on the fixed set of agreement points. Thus, the total number of joint errors decreases with every replacement; and, as this number is trivially lower-bounded by 0, this (“potential-function” style) argument establishes the process’s monotonicity and convergence.

By iteratively reducing the number of joint errors across all pairs of chosen ensemble members, our iterative process improves the robust accuracy of the resulting ensemble on the fixed set of agreement points. This, however, does not guarantee improved robust accuracy over the entire input domain. Nonetheless, we show in Section IV that such an improvement does typically occur in practice, even on randomly sampled subsets of input points (which are not necessarily agreement points).

IV. CASE STUDY: MNIST AND FASHION-MNIST

Below, we present the evaluation of our methodology on two datasets: the MNIST dataset for handwritten digit recognition [51], and the Fashion-MNIST dataset for clothing classification [91]. Our results for both datasets demonstrate that our technique facilitates choosing ensembles that provide high robust accuracy via relatively few, efficient verification queries.

The considered datasets are conducive for our purposes since they allow attaining high accuracy using fairly small DNNs, which enables us to *directly quantify* the robust accuracy of an entire ensemble, by dispatching verification queries that would otherwise be intractable (see Section III-A). This provides the ground truth required for assessing the benefits of our approach. The scalability afforded by our approach is crucial even for handling the relatively modest-sized DNNs considered: on the MNIST data, for instance, mutual-error verification queries for two ensemble members typically took a few seconds, whereas verification queries involving the full ensemble of five networks often timed out (35% of the queries on the MNIST data timed out after 24 hours, versus only roughly 1% of the pairwise mutual-error queries). As constituent DNN sizes and ensemble sizes increase, this gap in scalability is expected to become even more significant.

Our verification queries were dispatched using the Marabou verification engine [47] (although other engines could also be used). Additional details regarding the encoding of the verification queries, as well as detailed experimental results, appear in the extended version of this paper [7]. We have publicly released our code, as well as all benchmarks and experimental data, within an artifact accompanying this paper [6].

MNIST. For this part of our evaluation, we trained 10 independent DNNs $\{N_1, \dots, N_{10}\}$ over the MNIST dataset [51],

which includes 28×28 grayscale images of 10 handwritten digits (from “0” to “9”). Each of these networks had the same architecture: an input layer of 784 neurons, followed by a fully-connected layer with 30 neurons, a ReLU layer, another fully-connected layer with 10 neurons, and a final softmax layer with 10 output neurons, corresponding to the 10 possible digit labels.⁴ All networks achieved high accuracy rates of 96.29% – 96.57% (see Table I).

After training, we arbitrarily constructed two distinct ensembles with five DNN members each: $\mathcal{E}_1 = \{N_1, \dots, N_5\}$ and $\mathcal{E}_2 = \{N_6, \dots, N_{10}\}$, with an accuracy of 97.8% and 97.3%, respectively. Notice that the ensembles achieve a higher accuracy over the test set than their individual members.

We then applied our method in an attempt to improve the robust accuracy of \mathcal{E}_1 . We began by searching the validation set, and identifying 200 agreement points (the set A),⁵ all correctly labeled as “0” by all 10 networks.⁶ Using the 200 agreement points and 6 different perturbation sizes⁷ $\epsilon \in \{0.01, 0.02, 0.03, 0.04, 0.05, 0.06\}$, we constructed 1200 ϵ -balls around the selected agreement points; and then, for every ball B and for every pair $N_i, N_j \in \mathcal{E}_1$, we encoded a verification query to check whether N_i and N_j have a mutual error in B (see example in Fig. 3). This resulted in $\binom{5}{2} \cdot 200 \cdot 6 = 12000$ verification queries, which we dispatched using the Marabou DNN verifier [47] (each query ran with a 2-hour timeout limit). Finally, we used the results to compute the uniqueness score for each network in \mathcal{E}_1 ; these results, which appear briefly in Table I (for $\epsilon = 0.02$) and appear in full in [7], clearly show that two of the members, N_2 and N_5 , are each relatively prone to erring simultaneously with the remaining four members of \mathcal{E}_1 .

Next, we began searching among the remaining networks, N_6, \dots, N_{10} , for good replacements for N_2 and N_5 . Specifically, we searched for networks that obtained higher US scores than N_2 and N_5 . To achieve this, we began modifying \mathcal{E}_1 , each time removing either N_2 or N_5 , replacing them with one of the remaining networks, and computing the uniqueness scores for the new members (with respect to the four remaining original networks). We observed that for both N_2 and N_5 , network N_9 was a good replacement, obtaining very high US values. For additional details, see the extended version of our paper [7].

Finally, to evaluate the effect of our changes to \mathcal{E}_1 , we constructed the two new ensembles, $\mathcal{E}_1^{2 \rightarrow 9} = \{N_1, N_9, N_3, N_4, N_5\}$ and $\mathcal{E}_1^{5 \rightarrow 9} = \{N_1, N_2, N_3, N_4, N_9\}$. Computing the new ensembles’ robust accuracy over the entire

⁴Although the DNNs all have the same size and architecture, common ensemble training processes randomly initialize their weights, and also randomly pick samples from the same training set (see [50]). This is the cause for diversity among ensemble members, which our algorithm later detects.

⁵In our experiments, we empirically selected 200 agreement points in order to balance between precision (a higher number of points) and verification speed (a smaller number of points). This selection is based on a user’s available computing power.

⁶The “0” label is the label with the highest accuracy among the trained ensemble members, and thus “0”-labeled agreement points represent areas in the input space with extremely high consensus.

⁷ ϵ values which are too small, or too large, render the queries trivial. Thus, we found it to be useful to use a varied selection of ϵ values.

TABLE I: Accuracy and uniqueness scores for the MNIST networks. Uniqueness scores are measured with respect to the ensemble (either \mathcal{E}_1 or \mathcal{E}_2).

	\mathcal{E}_1					\mathcal{E}_2				
	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}
Accuracy	96.42%	96.55%	96.40%	96.46%	96.29%	96.44%	96.48%	96.57%	96.51%	96.46%
US	90.75%	88.38%	90.63%	92.13%	88.63%	97.38%	96.75%	97.5%	98.88%	97.75%

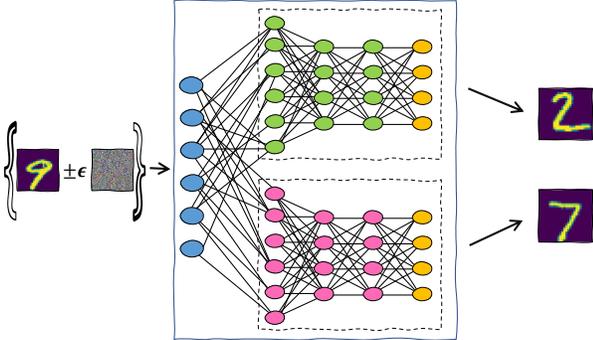


Fig. 3: Checking whether two MNIST digit recognition networks have a mutual error around an agreement point labeled “9”. In this case, the same perturbation causes one network to output the incorrect label “2”, and the other network to output the incorrect label “7”.

test set is computationally expensive, and thus we sampled 200 random points from the test set (these did not necessarily have the same label, nor were they required to be agreement points for the ensemble). For each sample, we created a verification query to check the robust accuracy of the new ensembles around the point, compared to the original ensemble. The results are plotted in Fig. 4, and indicate that the new ensembles demonstrated *significantly higher* robust accuracy on the tested points. These results validate our claim that a scoring metric based on agreement points is useful in improving the ensemble’s robustness also on other, “harder”, input points. Our analysis also indicates that the improved robustness results originated not only from ϵ -balls around inputs labeled as “0”, but from other labels as well. In fact, the gain in robustness was not just in quantity, but also in quality: for almost all cases, whenever \mathcal{E}_1 proved robust around an input, so did $\mathcal{E}_1^{2 \rightarrow 9}$ and $\mathcal{E}_1^{5 \rightarrow 9}$. This indicates that the improved robustness originated from inputs on which \mathcal{E}_1 was prone to err.

Next, we turned our attention to \mathcal{E}_2 , and computed the uniqueness scores for each of its members (see Table I). This time we conducted a “reverse” experiment: we identified the two *best* members of \mathcal{E}_2 , i.e. the two networks that had the highest uniqueness scores, and were consequently the least prone to err simultaneously. These turned out to be networks N_9 and N_{10} . Next, we replaced each of these networks with each of the networks $\{N_1, \dots, N_5\}$, in order to identify a network that, when inserted into \mathcal{E}_2 , achieved a lower score than N_9 and N_{10} . N_4 turned out to be such a network. We created the two modified ensembles, $\mathcal{E}_2^{9 \rightarrow 4} = \{N_6, N_7, N_8, N_4, N_{10}\}$

and $\mathcal{E}_2^{10 \rightarrow 4} = \{N_6, N_7, N_8, N_9, N_4\}$, and compared their robust accuracy to that of \mathcal{E}_2 on 200 random points from the test set. The results, depicted in Fig. 4, indicate that the ensemble’s robust accuracy decreased significantly, as expected.

In both aforementioned experiments, we also computed the *accuracy* (as opposed to *robust accuracy*) of the new ensembles, by evaluating them over the test set. All new ensembles had an accuracy that was on par with that of the original ensembles — specifically, within a range of $\pm 0.2\%$ from the original ensembles’ accuracy.

Fashion-MNIST. For the second part of our evaluation, we trained 10 independent DNNs $\{N_{11}, \dots, N_{20}\}$ over the Fashion-MNIST dataset [91], which includes 28×28 grayscale images of 10 clothing categories (“Coat”, “Dress”, etc.), and is considered more complex than the MNIST dataset. Each DNN had the same architecture as the MNIST-trained DNNs, and achieved an accuracy of 87.05%–87.53% (see Table II). We arbitrarily constructed two distinct ensembles, $\mathcal{E}_3 = \{N_{11}, \dots, N_{15}\}$ and $\mathcal{E}_4 = \{N_{16}, \dots, N_{20}\}$, with an accuracy of 88.22% and 88.48%, respectively.

Next, we again computed the US values of each of the networks. The results, which appear in full in [7], indicate a high variance among the uniqueness scores of the members of \mathcal{E}_4 , as compared to the relatively similar scores of \mathcal{E}_3 ’s members. We thus chose to focus on \mathcal{E}_4 . Based on the computed US values, we identified N_{20} as its least unique DNN; and, by replacing N_{20} with each of the five networks not currently in \mathcal{E}_4 , identified that N_{15} is a good candidate for replacing N_{20} . Performing our validation step over $\mathcal{E}_4^{20 \rightarrow 15}$ revealed that its robust accuracy has indeed increased. Running the “reverse” experiment, in which \mathcal{E}_4 ’s most unique member is replaced with a worse candidate, led us to consider the ensemble $\mathcal{E}_4^{18 \rightarrow 13}$, which indeed demonstrated lower robust accuracy than the original ensemble. For additional details, see the extended version of our paper [7].

For the final step of our experiment, we used our approach to iteratively switch two members of an ensemble. Specifically, after creating $\mathcal{E}_4^{20 \rightarrow 15}$, which had higher robust accuracy than \mathcal{E}_4 , we re-computed the US scores of its members, and identified again the least unique member — in this case, N_{16} . Per our computation, the best candidate for replacing it was N_{12} . The resulting ensemble, namely $\mathcal{E}_4^{20 \rightarrow 15, 16 \rightarrow 12}$, indeed demonstrated higher robust accuracy than both its predecessors. Performing another iteration of the “reverse” experiment yielded ensemble $\mathcal{E}_4^{18 \rightarrow 13, 17 \rightarrow 11}$, with poorer robust accuracy. The results appear in Fig. 5. We note that the only discrepancy, namely the robust accuracy of $\mathcal{E}_4^{20 \rightarrow 15}$ being lower than that

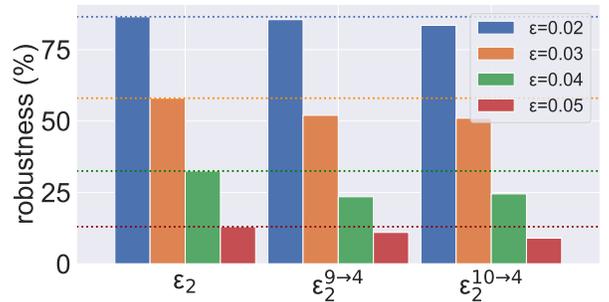
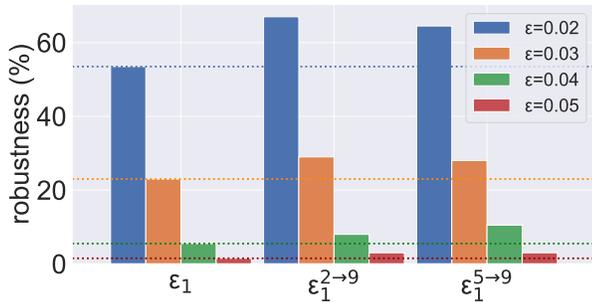


Fig. 4: The average robust accuracy scores for our original and modified ensembles. The results for $\epsilon = 0.01$ and $\epsilon = 0.06$ are trivial (the ensembles achieve near-perfect or near-zero robustness), and are omitted to reduce clutter.

TABLE II: Accuracy and uniqueness scores for the Fashion-MNIST networks. Uniqueness scores are measured with respect to the ensemble (either \mathcal{E}_3 or \mathcal{E}_4).

	\mathcal{E}_3					\mathcal{E}_4				
	N_{11}	N_{12}	N_{13}	N_{14}	N_{15}	N_{16}	N_{17}	N_{18}	N_{19}	N_{20}
Accuracy	87.14%	87.13%	87.53%	87.34%	87.3%	87.05%	87.32%	87.35%	87.34%	87.11%
US	70.63%	71.5%	69.75%	70.88%	73.25%	67.38%	72.38%	80.13%	71.38%	66.75%

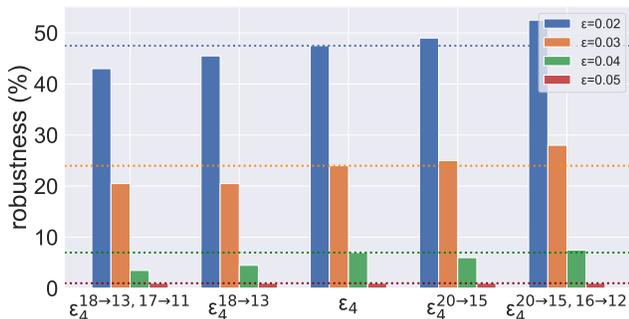


Fig. 5: The original ensemble \mathcal{E}_4 (center), ensembles modified to gain robust accuracy (right), and ensembles modified to reduce robust accuracy (left).

of \mathcal{E}_4 for $\epsilon = 0.04$, is due to timeouts.

Similarly to the MNIST case, the new ensembles in the Fashion-MNIST experiments obtained an accuracy that was on par with that of the original ensembles — specifically, within a range of $\pm 0.17\%$ from the original ensemble’s accuracy.

V. COMPARISON TO GRADIENT-BASED ATTACKS

Current state-of-the-art approaches for assessing a network’s robustness and robust accuracy rely on *gradient-based attacks* — a popular class of algorithms that, like verification methods, are capable of finding adversarial examples for a given neural network. In this section we compare our verification-based approach to these methods.

Gradient-based attacks generate adversarial examples by optimizing (via various techniques) a loss metric over the network’s output, relative to its input. This allows these methods to effectively search the local surroundings of a

fixed input point for local optima, which often constitute adversarial inputs. Gradient-based methods, such as the *fast-gradient sign method* (FGSM) [39], *projected gradient descent* (PGD) [60], and others [49], [59], are in widespread use due to their scalability and relative ease of use. However, as we demonstrate here, they are often unsuitable in our setting.

In order to evaluate the effectiveness of gradient-based methods for measuring the robust accuracy of ensembles, we modified the common FGSM [39] and I-FGSM [49] (“Iterative FGSM”) methods, thus extending them into three novel attacks aimed at finding adversarial examples that can fool multiple ensemble members simultaneously. We refer to these attacks as *Gradient Attack (G.A.) 1, 2, and 3*. For a thorough explanation of these attacks, as well as information about their design and implementation, see the extended version of our paper [7].

Next, we used our three attacks to search for mutual errors of DNN pairs — i.e., adversarial examples that simultaneously affect a pair of DNNs. Specifically, we applied the attacks on both datasets (MNIST and Fashion-MNIST), and searched for adversarial examples within various ϵ -balls around the same set of agreement points used in our previous experiments. This allowed us to subsequently compute, via gradient attacks, the mutual error scores of DNN pairs, and consequently, the uniqueness scores of each constituent ensemble member. The results of the total number of adversarial inputs found (SAT queries) are summarized in Table III. Each gradient attack typically took a few seconds to run. We also provide further details regarding the uniqueness scores computed by the three gradient-based methods in the extended version of this paper [7], and in our accompanying artifact [6].

The results in Table III include a total of 108000 experiments, on all ensemble pairs.⁸ In these experiments, our

⁸The 108000 experiments consist of $\binom{10}{2}$ pairs, times 200 agreement points, times 6 perturbation sizes, times 2 datasets.

TABLE III: The number of SAT queries discovered when searching for an adversarial attack, using the three gradient attack methods (G.A. 1, 2 and 3), and our verification approach.

Experiment	G.A. 1	G.A. 2	G.A. 3	verification
MNIST	1,333	3,886	5,574	16,826
Fashion-MNIST	17,190	21,245	22,129	33,152
Total	18,523	25,131	27,703	49,978

verification-based approach returned 49978 SAT results, while the strongest gradient-based method (gradient attack number 3) returned only 27703 SAT results — a 44% decrease in the number of counterexamples found. This discrepancy is on par with previous research [89], which indicates that gradient-based methods may err significantly when used for adversarial robustness analysis. This phenomenon manifests strongly in our setting, which involves many small and medium-sized perturbations that gradient-based approaches struggle with [24].

The reduced precision afforded by gradient-based approaches can, in some cases, lead to sub-optimal ensemble selection choices when compared to our verification-based approaches. Specifically, even if a gradient-based approach produces a uniqueness score ranking that coincides with the one produced using verification, the dramatic decrease in the number of SAT queries leads to much smaller mutual error scores, and consequently — to uniqueness score values that are overly optimistic, and less capable of distinguishing between poor and superior robust accuracy results.

For example, when observing the first two arbitrary ensembles on the MNIST dataset, \mathcal{E}_1 and \mathcal{E}_2 , the three gradient approaches (G.A. 1, 2 and 3) respectively assign average uniqueness scores of $\langle 95.4\%, 97.8\% \rangle$, $\langle 87.5\%, 94.5\% \rangle$ and $\langle 83.1\%, 92.5\% \rangle$ to the two ensembles (when averaging the US over all ensemble members and all perturbations). This indicates that the robust accuracy of the two ensembles is fairly similar (see appendices in [7]). In contrast, when using the more sensitive, verification-based approach, we find a substantially higher number of mutual errors (see Table III), and consequently, detect a much larger gap between the uniqueness scores of the two ensembles: 55% and 77%.

Another example that demonstrates the increased sensitivity of our method, when compared to gradient-based approaches, is obtained by observing the average uniqueness score of \mathcal{E}_3 and \mathcal{E}_4 on the Fashion-MNIST dataset. The strongest gradient attack that we used assigned almost identical average uniqueness scores to both ensembles (up to a difference of 0.01%), while our approach was sensitive enough to find a 2% difference between the average US of the two ensembles.

Finally, we note that, unlike verification-based approaches, gradient attacks are incomplete, and are consequently unable to return UNSAT. This makes them less suitable for assessing any additional uniqueness metrics based on robust ϵ -balls. We thus argue that, although gradient-based methods are faster

and more scalable than verification, our results showcase the benefits of using verification-based approaches for assessing uniqueness scores and for ensemble selection.

VI. RELATED WORK

Due to its pervasiveness, the phenomenon of adversarial inputs has received a significant amount of attention [27], [34], [61], [65], [66], [80], [99]. More specifically, the machine learning community has put a great deal of effort into measuring and improving the robustness of networks [18]–[20], [29], [36], [54], [60], [68], [71], [72], [87], [94]. The formal methods community has also been looking into the problem, by devising scalable DNN verification, optimization and monitoring techniques [1], [5], [8], [10]–[12], [16], [26], [41], [42], [55], [56], [64], [67], [70], [76], [90], [96]. To the best of our knowledge, ours is the first attempt to apply DNN verification to the setting of DNN ensembles. We note that our approach uses a DNN verifier strictly as a black-box backend, and so its scalability will improve as DNN verifiers become more scalable.

Obtaining DNN specifications to be verified is a difficult problem. While some studies have successfully applied verification to properties formulated by domain-specific experts [3], [4], [22], [25], [45], [78], most research has been focusing on *universal properties*, which pertain to every DNN-based system; specifically, local adversarial robustness [17], [35], [58], [76], fairness properties [83], network simplification [31] and modification [23], [32], [69], [77], [84], [93], and watermark resilience [32].

VII. CONCLUSION AND FUTURE WORK

In this case-study paper, we demonstrate a novel technique for assessing a deep ensemble’s robust accuracy through the use of DNN verification. To mitigate the difficulty inherent to verifying large ensembles, our approach considers pairs of networks, and computes for each ensemble member a score that indicates its tendency to make the same errors as other ensemble members. These scores allow us to iteratively improve the robust accuracy of the ensemble, by replacing weaker networks with stronger ones. Our empiric evaluation indicates the high practical potential of our approach; and, more broadly, we view this work as a part of the ongoing endeavor for demonstrating the real-world usefulness of DNN verification, by identifying additional, universal, DNN specifications.

Moving forward, we plan to tackle the natural open questions raised by our work; specifically, how our methodology for selecting robustly accurate ensembles can be extended beyond the current greedy search heuristic, as well as how ensembles should be selected in the context of other performance objectives, beyond robust accuracy. We also plan on experimenting with multiple stopping conditions for the ensemble member replacement process; as well as explore potential synergies between our verification-based approach and gradient-based approaches for computing mutual error scores. In addition, we note that we are currently extending

our approach to regression learning ensembles and deep reinforcement learning ensembles. Finally, we are in the process of optimizing our approach by using lighter-weight, incomplete verification tools (e.g., [76], [88], [95]), which afford better scalability, and also support parallelization. This will hopefully allow us to handle significantly larger DNNs and more complex datasets.

Acknowledgements. We thank Haoze Wu for his contribution to this project. The first three authors were partially supported by the Israel Science Foundation (grant number 683/18). The first author was partially supported by the Center for Interdisciplinary Data Science Research at The Hebrew University of Jerusalem. The fourth author was partially supported by funding from Huawei.

REFERENCES

- [1] P. Alamdari, G. Avni, T. Henzinger, and A. Lukina. Formal Methods with a Touch of Magic. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 138–147, 2020.
- [2] M. AlQuraishi. AlphaFold at CASP13. *Bioinformatics*, 35(22):4862–4865, 2019.
- [3] G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Verifying Learning-Based Robotic Navigation Systems, 2022. Technical Report. <https://arxiv.org/abs/2205.13536>.
- [4] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.
- [5] G. Amir, H. Wu, C. Barrett, and G. Katz. An SMT-Based Approach for Verifying Binarized Neural Networks. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 203–222, 2021.
- [6] G. Amir, T. Zelazny, G. Katz, and M. Schapira. Supplementary Artifact, 2022. <https://zenodo.org/record/6557083>.
- [7] G. Amir, T. Zelazny, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection, 2022. Technical Report. <https://arxiv.org/abs/2202.03898>.
- [8] G. Anderson, S. Pailoor, I. Dillig, and S. Chaudhuri. Optimization and Abstraction: a Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. 40th ACM SIGPLAN Conf. on Programming Languages Design and Implementations (PLDI)*, pages 731–744, 2019.
- [9] O. Araque, I. Corcuera-Platas, J. Sánchez-Rada, and C. Iglesias. Enhancing Deep Learning Sentiment Analysis with Ensemble Techniques in Social Applications. *Expert Systems with Applications*, 77:236–246, 2017.
- [10] P. Ashok, V. Hashemi, J. Kretinsky, and S. Mohr. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 92–107, 2020.
- [11] G. Avni, R. Bloem, K. Chatterjee, H. T., B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.
- [12] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks and its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1249–1264, 2019.
- [13] R. Bhattacharjee, S. Jha, and K. Chaudhuri. Sample Complexity of Robust Linear Classification on Separated Data. In *Proc. 38th Int. Conf. on Machine Learning (ICML)*, pages 884–893, 2021.
- [14] Y. Bian, Y. Wang, Y. Yao, and H. Chen. Ensemble Pruning Based on Objection Maximization With a General Distributed Framework. *IEEE Transactions on Neural Networks and Learning Systems*, 31(9):3766–3774, 2019.
- [15] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. <http://arxiv.org/abs/1604.07316>.
- [16] R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 4795–4804, 2018.
- [17] N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. <https://arxiv.org/abs/1709.10207>.
- [18] M. Casadio, E. Komendantskaya, M. Daggitt, W. Kokke, G. Katz, G. Amir, and I. Refaeli. Neural Network Robustness as a Verification Property: A Principled Case Study. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, 2022.
- [19] M. Cisse, P. Bojanowski, E. Grave, Y. Dauphin, and N. Usunier. Parseval Networks: Improving Robustness to Adversarial Examples. In *Proc. 34th Int. Conf. on Machine Learning (ICML)*, pages 854–863, 2017.
- [20] J. Cohen, E. Rosenfeld, and Z. Kolter. Certified Adversarial Robustness via Randomized Smoothing. In *Proc. 36th Int. Conf. on Machine Learning (ICML)*, pages 1310–1320, 2019.
- [21] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research (JMLR)*, 12:2493–2537, 2011.
- [22] D. Corsi, R. Yerushalmi, G. Amir, A. Farinelli, D. Harel, and G. Katz. Constrained Reinforcement Learning for Robotics via Scenario-Based Programming, 2022. Technical Report. <https://arxiv.org/abs/2206.09603>.
- [23] G. Dong, J. Sun, J. Wang, X. Wang, and T. Dai. Towards Repairing Neural Networks Correctly, 2020. Technical Report. <http://arxiv.org/abs/2012.01872>.
- [24] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li. Boosting Adversarial Attacks with Momentum. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 9185–9193, 2018.
- [25] S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.
- [26] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
- [27] H. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller. Adversarial Attacks on Deep Neural Networks for Time Series Classification. In *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*, pages 1–8, 2019.
- [28] S. Fort, H. Hu, and B. Lakshminarayanan. Deep Ensembles: A Loss Landscape Perspective, 2019. Technical Report. <http://arxiv.org/abs/1912.02757>.
- [29] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky. Domain-Adversarial Training of Neural Networks. *Journal of Machine Learning Research (JMLR)*, 17(1):2096–2030, 2016.
- [30] T. Gehr, M. Mirman, D. Drachler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
- [31] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.
- [32] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.
- [33] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
- [34] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. <http://arxiv.org/abs/1412.6572>.
- [35] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
- [36] B. Han, Q. Yao, X. Yu, G. Niu, M. Xu, W. Hu, I. Tsang, and M. Sugiyama. Co-teaching: Robust Training of Deep Neural Networks with Extremely Noisy Labels, 2018. Technical Report. <http://arxiv.org/abs/1804.06872>.
- [37] L. Hansen and P. Salamon. Neural Network Ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

- [38] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
- [39] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel. Adversarial Attacks on Neural Network Policies, 2017. Technical Report. <https://arxiv.org/abs/1702.02284>.
- [40] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
- [41] O. Isaac, C. Barrett, M. Zhang, and G. Katz. Neural Network Verification with Proof Production. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2022.
- [42] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 57–74, 2020.
- [43] S. Jain, G. Liu, J. Mueller, and D. Gifford. Maximizing Overall Diversity for Improved Uncertainty Estimates in Deep Ensembles. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 4264–4271, 2020.
- [44] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
- [45] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
- [46] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.
- [47] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
- [48] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.
- [49] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial Examples in the Physical World, 2016. Technical Report. <http://arxiv.org/abs/1607.02533>.
- [50] B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles, 2016. Technical Report. <https://arxiv.org/abs/1612.01474>.
- [51] Y. LeCun. The MNIST Database of Handwritten Digits, 1998. <http://yann.lecun.com/exdb/mnist/>.
- [52] S. Lee, S. Purushwalkam, M. Cogswell, D. Crandall, and D. Batra. Why M Heads are Better than One: Training a Diverse Ensemble of Deep Networks, 2015. Technical Report. <https://arxiv.org/abs/1511.06314>.
- [53] S. Lee, S. Purushwalkam Shiva Prakash, M. Cogswell, V. Ranjan, D. Crandall, and D. Batra. Stochastic Multiple Choice Learning for Training Diverse Deep Ensembles. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2016.
- [54] H. Liu, M. Long, J. Wang, and M. Jordan. Transferable Adversarial Training: A General Approach to Adapting Deep Classifiers. In *Proc. 36th Int. Conf. on Machine Learning (ICML)*, pages 4013–4022, 2019.
- [55] A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. <http://arxiv.org/abs/1706.07351>.
- [56] A. Lukina, C. Schilling, and T. Henzinger. Into the Unknown: Active Monitoring of Neural Networks. In *Proc. 21st Int. Conf. on Runtime Verification (RV)*, pages 42–61, 2021.
- [57] Z. Lyu, N. Gutierrez, A. Rajguru, and W. Beksı. Probabilistic Object Detection via Deep Ensembles. In *Proc. European Conf. on Computer Vision (ECCV)*, pages 67–75, 2020.
- [58] Z. Lyu, C. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 5037–5044, 2020.
- [59] J. Ma, S. Ding, and Q. Mei. Towards More Practical Adversarial Attacks on Graph Neural Networks. In *Proc. 34th Conf. on Neural Information Processing Systems (NeurIPS)*, 2020.
- [60] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks, 2017. Technical Report. <http://arxiv.org/abs/1706.06083>.
- [61] M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [62] M. Moshkovitz, Y. Yang, and K. Chaudhuri. Connecting Interpretability and Robustness in Decision Trees through Separation. In *Proc. 38th Int. Conf. on Machine Learning (ICML)*, pages 7839–7849, 2021.
- [63] G. Nam, J. Yoon, Y. Lee, and J. Lee. Diversity Matters When Learning From Ensembles. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2021.
- [64] M. Ostrovsky, C. Barrett, and G. Katz. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In *Proc. 20th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2022.
- [65] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. Celik, and A. Swami. Practical Black-Box Attacks against Machine Learning. In *Proc. ACM on Asia Conf. on Computer and Communications Security (CCS)*, pages 506–519, 2017.
- [66] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387, 2016.
- [67] P. Prabhakar and Z. Afzal. Abstraction Based Output Range Analysis for Neural Networks, 2020. Technical Report. <https://arxiv.org/abs/2007.09527>.
- [68] C. Qin, J. Martens, S. Gowal, D. Krishnan, K. Dvijotham, A. Fawzi, S. De, R. Stanforth, and P. Kohli. Adversarial Robustness through Local Linearization, 2019. Technical Report. <http://arxiv.org/abs/1907.02610>.
- [69] I. Refaeli and G. Katz. Minimal Multi-Layer Modifications of Deep Neural Networks. In *Proc. 5th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS)*, 2022.
- [70] W. Ruan, X. Huang, and M. Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proc. 27th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2018.
- [71] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. Davis, G. Taylor, and T. Goldstein. Adversarial Training for Free!, 2019. Technical Report. <http://arxiv.org/abs/1904.12843>.
- [72] A. Shafahi, P. Saadatpanah, C. Zhu, A. Ghiasi, C. Studer, D. Jacobs, and T. Goldstein. Adversarially Robust Transfer Learning, 2019. Technical Report. <http://arxiv.org/abs/1905.08232>.
- [73] C. Shui, A. Mozafari, J. Marek, I. Hedhli, and C. Gagné. Diversity Regularization in Deep Ensembles, 2018. Technical Report. <http://arxiv.org/abs/1802.07881>.
- [74] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
- [75] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014. Technical Report. <http://arxiv.org/abs/1409.1556>.
- [76] G. Singh, T. Gehr, M. Puschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.
- [77] M. Sotoudeh and A. Thakur. Correcting Deep Neural Networks with Small, Generalizing Patches. In *Workshop on Safety and Robustness in Decision Making*, 2019.
- [78] X. Sun, K. H., and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.
- [79] M. Svensén and C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer Berlin/Heidelberg, Germany, 2007.
- [80] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. <http://arxiv.org/abs/1312.6199>.
- [81] S. Tao. Deep Neural Network Ensembles. In *Int. Conf. on Machine Learning, Optimization, and Data Science*, pages 1–12, 2019.
- [82] F. Tramer, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel. Ensemble Adversarial Training: Attacks and Defenses, 2017. Technical Report. <http://arxiv.org/abs/1705.07204>.
- [83] C. Urban, M. Christakis, V. Wüstholtz, and F. Zhang. Perfectly Parallel Fairness Certification of Neural Networks. In *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–30, 2020.

- [84] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. Păsăreanu. NNrepair: Constraint-based Repair of Neural Network Classifiers, 2021. Technical Report. <http://arxiv.org/abs/2103.12535>.
- [85] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals, 2018. Technical Report. <http://arxiv.org/abs/1804.10829>.
- [86] Y. Wang, S. Jha, and K. Chaudhuri. Analyzing the Robustness of Nearest Neighbors to Adversarial Examples. In *Proc. 35th Int. Conf. on Machine Learning (ICML)*, pages 5120–5129, 2018.
- [87] E. Wong, L. Rice, and Z. Kolter. Fast is Better than Free: Revisiting Adversarial Training, 2020. Technical Report. <http://arxiv.org/abs/2001.03994>.
- [88] H. Wu, A. Ozdemir, A. Zeljić, A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Păsăreanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.
- [89] H. Wu, A. Zeljić, G. Katz, and C. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 143–163, 2022.
- [90] W. Xiang, H. Tran, and T. Johnson. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2018.
- [91] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNist: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017. Technical Report. <http://arxiv.org/abs/1708.07747>.
- [92] H. Xuan, R. Souvenir, and R. Pless. Deep Randomized Ensembles for Metric Learning. In *Proc. European Conf. on Computer Vision (ECCV)*, 2018.
- [93] X. Yang, T. Yamaguchi, H.-D. Tran, B. Hoxha, T. Johnson, and D. Prokhorov. Neural Network Repair with Reachability Analysis, 2021. Technical Report. <https://arxiv.org/abs/2108.04214>.
- [94] X. Yu, B. Han, J. Yao, G. Niu, I. Tsang, and M. Sugiyama. How does Disagreement Help Generalization against Label Corruption? In *Proc. 36th Int. Conf. on Machine Learning (ICML)*, pages 7164–7173, 2019.
- [95] T. Zelazny, H. Wu, C. Barrett, and G. Katz. On Reducing Over-Approximation Errors for Neural Network Verification. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2022.
- [96] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th Conf. of European Conference on Artificial Intelligence (ECAI)*, 2020.
- [97] H. Zhang, Y. Yu, J. Jiao, E. Xing, L. El Ghaoui, and M. Jordan. Theoretically Principled Trade-off between Robustness and Accuracy. In *Proc. 36th Int. Conf. on Machine Learning (ICML)*, pages 7472–7482, 2019.
- [98] Z. Zhou, J. Wu, and W. Tang. Ensembling Neural Networks: Many Could Be Better Than All. *Artificial Intelligence*, 137(1-2):239–263, 2002.
- [99] D. Zügner, A. Akbarnejad, and S. Günnemann. Adversarial Attacks on Neural Networks for Graph Data. In *Proc. 24th ACM SIGKDD Int. Conf. on Knowledge Discovery & Data Mining (KDD)*, pages 2847–2856, 2018.

Neural Network Verification with Proof Production

Omri Isac*, Clark Barrett[†], Min Zhang[‡] and Guy Katz*

*The Hebrew University of Jerusalem, Jerusalem, Israel [†]Stanford University, Stanford, California, USA

[‡]East China Normal University, Shanghai, China

omri.isac@mail.huji.ac.il, barrett@cs.stanford.edu, zhangmin@sei.ecnu.edu.cn, guykatz@cs.huji.ac.il

Abstract—Deep neural networks (DNNs) are increasingly being employed in safety-critical systems, and there is an urgent need to guarantee their correctness. Consequently, the verification community has devised multiple techniques and tools for verifying DNNs. When DNN verifiers discover an input that triggers an error, that is easy to confirm; but when they report that no error exists, there is no way to ensure that the verification tool itself is not flawed. As multiple errors have already been observed in DNN verification tools, this calls the applicability of DNN verification into question. In this work, we present a novel mechanism for enhancing Simplex-based DNN verifiers with *proof production* capabilities: the generation of an easy-to-check witness of unsatisfiability, which attests to the absence of errors. Our proof production is based on an efficient adaptation of the well-known Farkas’ lemma, combined with mechanisms for handling piecewise-linear functions and numerical precision errors. As a proof of concept, we implemented our technique on top of the Marabou DNN verifier. Our evaluation on a safety-critical system for airborne collision avoidance shows that proof production succeeds in almost all cases and requires only minimal overhead.

I. INTRODUCTION

Machine learning techniques, and specifically deep neural networks (DNNs), have been achieving groundbreaking results in solving computationally difficult problems. Nowadays, DNNs are state-of-the-art tools for performing many safety-critical tasks in the domains of healthcare [29], aviation [45] and autonomous driving [19]. DNN training is performed by adjusting the parameters of a DNN to mimic a highly complex function over a large set of input-output examples (the training set) in an automated way that is mostly opaque to humans.

The Achilles heel of DNNs typically lies in generalizing their predictions from the finite training set to an infinite input domain. First, DNNs tend to produce unexpected results on inputs that are considerably different from those in the training set; and second, the input to the DNN might be perturbed by sensorial imperfections, or even by a malicious adversary, again resulting in unexpected and erroneous results. These weaknesses have already been observed in many modern DNNs [37], [64], and have even been demonstrated in the real world [30] — thus hindering the adoption of DNNs in safety-critical settings.

In order to bridge this gap, in recent years, the formal methods community has started devising techniques for DNN verification (e.g., [2], [11], [13], [31], [32], [40], [41], [53], [58], [61], [62], [66], [68], [73], among many others). Typically, DNN verification tools seek to prove that outputs from a given set of inputs are contained within a safe subspace of the

output space, using various methods such as SMT solving [1], [16], [23], abstract interpretation [32], MILP solving [65], and combinations thereof. Notably, many modern approaches [50], [53], [55], [65] involve a *search* procedure, in which the verification problem is regarded as a set of constraints. Then, various input assignments to the DNN are considered in order to discover a counter-example that satisfies these constraints, or to prove that no such counter-example exists.

Verification tools are known to be as prone to errors as any other program [44], [72]. Moreover, the search procedures applied as part of DNN verification typically involve the repeated manipulation of a large number of floating-point equations; this can lead to rounding errors and numerical stability issues, which in turn could potentially compromise the verifier’s soundness [12], [44]. When the verifier discovers a counter-example, this issue is perhaps less crucial, as the counter-example can be checked by evaluating the DNN; but when the verifier determines that no counter-example exists, this conclusion is typically not accompanied by a witness of its correctness.

In this work, we present a novel proof-production mechanism for a broad family of search-based DNN verification algorithms. Whenever the search procedure returns UNSAT (indicating that no counter-example exists), our mechanism produces a proof certificate that can be readily checked using simple, external checkers. The proof certificate is produced using a constructive version of Farkas’ lemma, which guarantees the existence of a witness to the unsatisfiability of a set of linear equations — combined with additional constructs to support the non-linear components of a DNN, i.e., its piecewise-linear activation functions. We show how to instrument the verification algorithm in order to keep track of its search steps, and use that information to construct the proof with only a small overhead.

For evaluation purposes, we implemented our proof-production technique on top of the Marabou DNN verifier [50]. We then evaluated our technique on the ACAS Xu set of benchmarks for airborne collision avoidance [46], [48]. Our approach was able to produce proof certificates for the safety of various ACAS Xu properties with reasonable overhead (5.7% on average). Checking the proof certificates produced by our approach was usually considerably faster than dispatching the original verification query.

The main contribution of our paper is in proposing a proof-production mechanism for search-based DNN verifiers, which can substantially increase their *reliability* when de-

terminating unsatisfiability. However, it also lays a foundation for a conflict-driven clause learning (CDCL) [74] verification scheme for DNNs, which might significantly improve the performance of search-based procedures (see discussion in Sec. IX).

The rest of this paper is organized as follows. In Sec. II we provide relevant background on DNNs, formal verification, the Simplex algorithm, and on using Simplex for search-based DNN verification. In Sec. III, IV and V, we describe the proof-production mechanism for Simplex and its extension to DNN verification. Next, in Sec. VI, we briefly discuss complexity-theoretical aspects of the proof production. Sec. VII details our implementation of the technique and its evaluation. We then discuss related work in Sec. VIII and conclude with Sec. IX.

II. BACKGROUND

Deep Neural Networks. *Deep neural networks* (DNNs) [36] are directed graphs, whose nodes (neurons) are organized into layers. Nodes in the first layer, called the *input layer*, are assigned values based on the input to the DNN; and then the values of nodes in each of the subsequent layers are computed as functions of the values assigned to neurons in the preceding layer. More specifically, each node value is computed by first applying an affine transformation to the values from the preceding layer and then applying a non-linear *activation function* to the result. The final (output) layer, which corresponds to the output of the network, is computed without applying an activation function.

One of the most common activation functions is the *rectified linear unit* (ReLU), which is defined as:

$$f(b) = \text{ReLU}(b) = \begin{cases} b & b > 0 \\ 0 & \text{otherwise.} \end{cases}$$

When $b > 0$, we say that the ReLU is in the *active* phase; otherwise, we say it is in the *inactive* phase. For simplicity, we restrict our attention here to ReLUs, although our approach could be applied to other piecewise-linear functions (such as *max pooling*, *absolute value*, *sign*, etc.). Non piecewise-linear functions, such as *sigmoid* or *tanh*, are left for future work.

Formally, a DNN $\mathcal{N} : \mathbb{R}^m \rightarrow \mathbb{R}^k$, is a sequence of n layers L_0, \dots, L_{n-1} where each layer L_i consists of $s_i \in \mathbb{N}$ nodes, denoted $v_i^1, \dots, v_i^{s_i}$. The assignment for the j^{th} node in the $1 \leq i < n - 1$ layer is computed as

$$v_i^j = \text{ReLU} \left(\sum_{l=1}^{s_{i-1}} w_{i,j,l} \cdot v_{i-1}^l + p_i^j \right)$$

and neurons in the output layer are computed as:

$$v_{n-1}^j = \sum_{l=1}^{s_{n-2}} w_{n-1,j,l} \cdot v_{n-2}^l + p_{n-1}^j$$

where $w_{i,j,l}$ and p_i^j are (respectively) the predetermined weights and biases of \mathcal{N} . We set $s_0 = m$ and treat v_0^1, \dots, v_0^m as the input of \mathcal{N} .

A simple DNN with four layers appears in Fig. 1. For simplicity, the p_i^j parameters are all set to zero and are ignored.

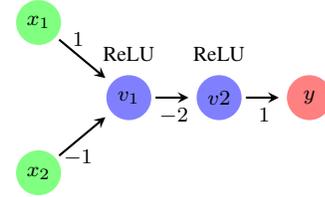


Fig. 1: A toy DNN.

For input $\langle 1, 2 \rangle$, the node in the second layer evaluates to $\text{ReLU}(1 \cdot 1 + 2 \cdot (-1)) = \text{ReLU}(-1) = 0$; the node in the third layer evaluates to $\text{ReLU}(0 \cdot (-2)) = 0$; and the node in the fourth (output) layer evaluates to $0 \cdot 1 = 0$.

DNN Verification and Proofs. Given a DNN $\mathcal{N} : \mathbb{R}^m \rightarrow \mathbb{R}^k$ and a property $P : \mathbb{R}^{m+k} \rightarrow \{\mathbb{T}, \mathbb{F}\}$, the *DNN verification problem* is to decide whether there exist $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^k$ such that $(\mathcal{N}(x) = y) \wedge P(x, y)$ holds. If such x and y exist, we say that the verification query $\langle \mathcal{N}, P \rangle$ is *satisfiable* (SAT); and otherwise, we say that it is *unsatisfiable* (UNSAT). For example, given the toy DNN from Fig. 1, we can define a property $P : P(x, y) \Leftrightarrow (x \in [2, 3] \times [-1, 1]) \wedge (y \in [0.25, 0.5])$. Here, P expresses the existence of an input $x \in [2, 3] \times [-1, 1]$ that produces an output $y \in [0.25, 0.5]$. Later on, we will prove that no such x exists, i.e., the verification query $\langle \mathcal{N}, P \rangle$ is UNSAT.

Typically, P represents the negation of a desired property, and so an input x which satisfies the query is a counterexample — whereas the query’s unsatisfiability indicates that the property holds. In this work, we follow mainstream DNN verification research [53], [68] and focus on properties P that are a conjunction of linear lower- and upper-bound constraints on the neurons of x and y . It has been shown that even for such simple properties, and for DNNs that use only the ReLU activation function, the verification problem is NP-complete [48].

A *proof* is a mathematical object that certifies a mathematical statement. In case a DNN verification query is SAT, the input x for which P holds constitutes a proof of the query’s satisfiability. Our goal here is to generate proofs also for the UNSAT case, which, to the best of our knowledge, is a feature that no DNN verifier currently supports [12].

Verifying DNNs via Linear Programming. *Linear Programming* (LP) [22] is the problem of optimizing a linear function over a given convex polytope. An LP instance over variables $V = [x_1, \dots, x_n]^T \in \mathbb{R}^n$ contains an objective function $c \cdot V$ to be maximized, subject to the constraints $A \cdot V = b$ for some $A \in M_{m \times n}(\mathbb{R}), b \in \mathbb{R}^m$, and $l \leq V \leq u$ for some $l, u \in (\mathbb{R} \cup \{\pm\infty\})^n$. Throughout the paper, we use $l(x_i)$ and $u(x_i)$, to refer to the lower and upper bounds (respectively) of x_i . LP solving can also be used to check the *satisfiability* of constraints of the form $(A \cdot V = b) \wedge (l \leq V \leq u)$.

The *Simplex* algorithm [22] is a widely used technique for solving LP instances. It begins by creating a *tableau*, which is equivalent to the original set of equations $AV = b$.

Next, Simplex selects a certain subset of the variables, $\mathcal{B} \subseteq \{x_1, \dots, x_n\}$, to act as the *basic variables*; and the tableau is considered as representing each basic variable $x_i \in \mathcal{B}$ as a linear combination of non-basic variables, $x_i = \sum_{j \notin \mathcal{B}} c_j \cdot x_j$.

We use $A_{i,j}$ to denote the coefficient of a variable x_j in the tableau row that corresponds to basic variable x_i . Apart from the tableau, Simplex also maintains a variable assignment that satisfies the equations of A , but which may temporarily violate the bound constraints $l \leq V \leq u$. The assignment for a variable x_i is denoted $\alpha(x_i)$.

After initialization, Simplex begins searching for an assignment that simultaneously satisfies both the tableau and bound constraints. This is done by manipulating the set \mathcal{B} , each time swapping a basic and a non-basic variable. This alters the equations of A by adding multiples of equations to other equations, and allows the algorithm to explore new assignments. The algorithm can terminate with a SAT answer when a satisfying assignment is discovered or an UNSAT answer when: (i) a variable has contradicting bounds, i.e., $l(x_i) > u(x_i)$; or (ii) one of the tableau equations $x_i = \sum_{j \notin \mathcal{B}} c_j \cdot x_j$ implies that x_i can never satisfy its bounds. The Simplex algorithm is sound, and is also complete if certain heuristics are used for selecting the manipulations of \mathcal{B} [22]. A detailed calculus for the version of Simplex that we use appears in the extended version of this paper [42].

LP solving is particularly useful in the context of DNN verification, and is used by almost all modern tools (either natively [48], or by invoking external solvers such as GLPK [54] or Gurobi [39]). More specifically, a DNN verification query can be regarded as an LP instance with bounded variables that represents the property P and the affine transformations within \mathcal{N} , combined with a set of piecewise-linear constraints that represent the activation functions. We demonstrate this with an example, and then explain how this formulation can be solved.

Recall the toy DNN from Fig. 1, and property P that is used for checking whether there exists an input x in the range $[2, 3] \times [-1, 1]$ for which \mathcal{N} produces an output y in the range $[0.25, 0.5]$. We use b_1, f_1 to denote the input and output to node v_1 ; b_2, f_2 for the input and output of v_2 ; x_1 and x_2 to denote the network’s inputs, and y to denote the network’s output. The linear constraints of the network yield the linear equations $b_1 = x_1 - x_2$, $b_2 = -2f_1$, and $y = f_2$ (which we name e^1, e^2 , and e^3 , respectively). The restrictions on the network’s input and output are translated to lower and upper bounds: $2 \leq x_1 \leq 3$, $-1 \leq x_2 \leq 1$, $0.25 \leq y \leq 0.5$. The third equation implies that $0.25 \leq f_2 \leq 0.5$, which in turn implies that $b_2 \leq 0.5$. Assume we also restrict: $-0.5 \leq b_2$, $-0.5 \leq b_1 \leq 0.5$, $0 \leq f_1 \leq 0.5$. Together, these constraints give rise to the linear program that appears in Fig. 2. The remaining ReLU constraints, i.e. $f_i = \text{ReLU}(b_i)$ for $i \in \{1, 2\}$, exist alongside the LP instance. Together, query φ is equivalent to the DNN verification problem that we are trying to solve.

Using this formulation, the verification problem can be

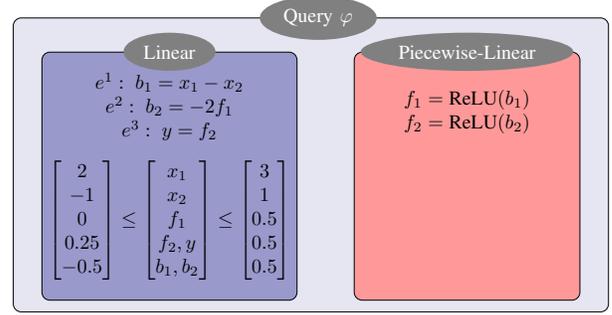


Fig. 2: An example of a DNN verification query φ , comprised of an LP instance and piecewise-linear constraints.

solved using Simplex, enhanced with a *case-splitting* approach for handling the ReLU constraints [17], [48]. Intuitively, we first invoke the LP solver on the LP portion of the query; and if it returns UNSAT, the whole query is UNSAT. Otherwise, if it finds a satisfying assignment, we check whether this assignment also satisfies the ReLU constraints. If it does, then the whole query is SAT. Otherwise, case splitting is applied in order to split the query into two different sub-queries, according to the two phases of the ReLU function.¹ Specifically, in one of the sub-queries, the LP query is adjusted to enforce the ReLU to be in the active phase: the equation $f = b$ is added, along with the bound $b \geq 0$. In the other sub-query, the inactive phase is enforced: $b \leq 0, 0 \leq f \leq 0$. This effectively reduces the ReLU constraint into linear constraints in each sub-query. This process is then repeated for each of the two sub-queries.

Case-splitting turns the verification procedure into a *search tree* [48], with nodes corresponding to the splits that were applied. The tree is constructed iteratively, with Simplex invoked on every node to try and derive UNSAT or find a true satisfying assignment. If Simplex is able to deduce that all leaves in the search tree are UNSAT, then so is the original query. Otherwise, it will eventually find a satisfying assignment that also satisfies the original query. This process is sound, and will always terminate if appropriate splitting strategies are used [22], [48]. Unfortunately, the size of the search tree can be exponential in the number of ReLU constraints; and so in order to keep the search tree small, case splitting is applied as little as possible, according to various heuristics that change from tool to tool [55], [62], [68]. In order to reduce the number of splits even further, verification algorithms apply clever deduction techniques for discovering tighter variable bounds, which may in turn rule out some of the splits a-priori. We also discuss this kind of deduction, which we refer to as dynamic bound tightening, in the following sections.

III. PROOF PRODUCTION OVERVIEW

A Simplex-based verification process of a DNN is tree-shaped, and so we propose to generate a *proof tree* to match

¹The approach is easily generalizable to any piecewise-linear constraint, by splitting the query according to the different linear pieces of the activation function.

it. Within the proof tree, internal nodes will correspond to case splits, whereas each leaf node will contain a proof of unsatisfiability based on all splits performed on the path between itself and the root. Thus, a proof tree constitutes a valid proof of unsatisfiability if each of its leaves contains a proof that demonstrates that all splits so far lead to a contradiction. The proof tree might also include proofs for *lemmas*, which are valid statements for the node in which they reside and its descendants (lemmas are needed for supporting bound tightening, as we discuss later).

As a simple, intuitive example, we depict in Fig. 3 a proof of unsatisfiability for the query φ from Fig. 2. The root of the proof tree represents the initial verification query, which is comprised of LP constraints and ReLU constraints. The fact that this node is not a leaf indicates that the Simplex-based verifier was unable to conclude UNSAT in this state, and needed to perform a case split on the ReLU node v_1 . The left child of the root corresponds to the case where ReLU v_1 is inactive: the LP is augmented with additional constraints that represent the case split, i.e., $f_1 = 0$ and $b_1 \leq 0$. This new fact may now be used by the Simplex procedure, which is indeed able to obtain an UNSAT result. The node then contains a proof of this unsatisfiability: $[-1 \ 0 \ 0]^\top$. This vector instructs the checker how to construct a linear combination of the current tableau's rows, in a way that leads to a bound contradiction, as we later explain in Sec. V.

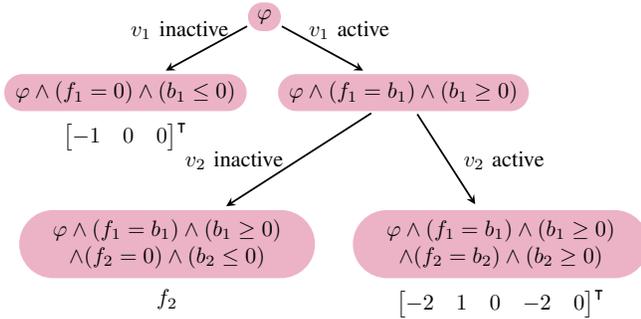


Fig. 3: A proof tree example.

In the right child of the root, which represents v_1 's active phase, the constraints $f_1 = b_1$ and $b_1 \geq 0$ are added by the split. This node is not a leaf, because the verifier performed a second case split, this time on v_2 . The left child represents v_2 's inactive phase, and has the corresponding constraints $f_2 = 0$ and $b_2 \leq 0$. This child is a leaf, and is marked with f_2 , indicating that f_2 is a variable whose bounds led to a contradiction. Specifically, $f_2 \geq 0.25$ from φ and $f_2 = 0$ from the case split are contradictory.

The last node (the rightmost leaf) represents v_2 's active phase, and has the constraints $f_2 = b_2$ and $b_2 \geq 0$. Here, the node indicates that a contradiction can be reached from the current tableau, using the vector $[-2 \ 1 \ 0 \ -2 \ 0]^\top$. In Sec. IV, we explain how this process works.

Because each leaf of the proof tree contains a proof of unsatisfiability, the tree itself proves that the original query

is UNSAT. Note that many other proof trees may exist for the same query. In the following sections, we explain how to instrument a Simplex-based verifier in order to extract such proof trees from the solver execution.

IV. SIMPLEX WITH PROOFS

A. Producing proofs for LP

We now describe our approach for creating proof trees, beginning with leaf nodes. We start with the following lemma:

Lemma 1. *If Simplex returns UNSAT, then there exists a variable with contradicting bounds; that is, there exists a variable $x_i \in V$ with lower and upper bounds $l(x_i)$ and $u(x_i)$, for which Simplex has discovered that $l(x_i) > u(x_i)$.*

This lemma justifies our choice of using contradicting bounds as proofs of unsatisfiability in the leaves of the proof tree. The lemma follows directly from the derivation rules of Simplex. Specifically, there are only two ways to reach UNSAT: when the input problem already contains inconsistent bounds $l(x_i) > u(x_i)$, or when Simplex finds a tableau row $x_i = \sum_{j \notin B} c_j \cdot x_j$ that gives rise to such inconsistent bounds. The complete proof appears in the extended version of this paper [42].

We demonstrate this with an example, based on the query φ from Fig. 2. Suppose that, as part of its Simplex-based solution process, a DNN verifier performs two case splits, fixing the two ReLUs to their active states: $f_1 = b_1 \wedge b_1 \geq 0$ and $f_2 = b_2 \wedge b_2 \geq 0$. This gives rise to the following (slightly simplified) system of equations:

$$b_1 = x_1 - x_2 \quad b_2 = -2f_1 \quad y = f_2 \quad f_1 = b_1 \quad f_2 = b_2$$

Which corresponds to the tableau and variables

$$A = \begin{bmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \end{bmatrix} \quad V = \begin{bmatrix} x_1 \\ x_2 \\ b_1 \\ b_2 \\ f_1 \\ f_2 \\ y \end{bmatrix}^\top$$

such that $AV = \bar{0}$, with the corresponding bound vectors:

$$l = [2 \ -1 \ 0 \ 0 \ 0 \ 0.25 \ 0.25]^\top \\ u = [3 \ 1 \ 0.5 \ 0.5 \ 0.5 \ 0.5 \ 0.5]^\top$$

Then, the Simplex solver iteratively alters the set of basic variables, which corresponds to multiplying various equations by scalars and summing them to obtain new equations. At some point, the equation $b_2 = -2x_1 + 2x_2$ is obtained (by computing $[-2 \ 1 \ 0 \ -2 \ 0]^\top \cdot A \cdot V$), with a current assignment of $\alpha(V)^\top = [2 \ 1 \ 1 \ -2 \ 1 \ -2 \ -2]$.

At this point, the Simplex solver halts with an UNSAT notice. The reason is that b_2 is currently assigned the value -2 , which is below its lower bound of 0 , and so its value needs to be increased. However, the equation, combined with the fact that x_1 is pressed against its lower bound, while x_2 is

pressed against its upper bound, indicates that there is no slack remaining in order to increase the value of b_2 (this corresponds to the Failure_1 rule in the Simplex calculus described in the extended version of this paper [42]). The key point is that the same equation could be used in deducing a tighter bound for b_2 :

$$b_2 \leq -2l(x_1) + 2u(x_2) = -2 \cdot 2 + 2 \cdot 1 = -2$$

and a contradiction could then be obtained based on the contradictory facts $0 = l(b_2) \leq b_2 \leq -2$. In other words, and as we formally prove in the extended version of this paper [42], any UNSAT answer returned by Simplex can be regarded as a case of conflicting lower and upper bounds.

Given Lemma 1, our goal is to instrument the Simplex procedure so that whenever it returns UNSAT, we are able to produce a proof which indicates that $l(x_i) > u(x_i)$ for some variable x_i . To this end, we introduce the following adaptation of *Farkas' Lemma* [67] to the Simplex setting, which states that a linear-sized proof of this fact exists.

Lemma 2. *Given the constraints $A \cdot V = 0$ and $l \leq V \leq u$, where $A \in M_{m \times n}(\mathbb{R})$ and $l, V, u \in \mathbb{R}^n$, exactly one of these two options holds:*

- 1) *The SAT case: $\exists V \in \mathbb{R}^n$ such that $A \cdot V = 0$ and $l \leq V \leq u$.*
- 2) *The UNSAT case: $\exists w \in \mathbb{R}^m$ such that for all $l \leq V \leq u$, $w^\top \cdot A \cdot V < 0$, whereas $0 \cdot w = 0$. Thus, w is a proof of the constraints' unsatisfiability.*

Moreover, these vectors can be constructed during the run of the Simplex algorithm.

This Lemma is actually a corollary of Theorem 3, which we introduce later. For a complete proof, see the extended version of this paper [42].

In our previous, UNSAT example, one possible vector is $w = [-2 \ 1 \ 0 \ -2 \ 0]^\top$. Indeed, $w \cdot A \cdot V = 0$ gives us the equation $-2x_1 + 2x_2 - b_2 = 0$. Given the lower and upper bounds for the participating variables, the largest value that the left-hand side of the equation can obtain is:

$$-2l(x_1) + 2u(x_2) - l(b_2) = -2 \cdot 2 + 2 \cdot 1 - 0 = -2 < 0$$

Therefore, no variable assignment within the stated bounds can satisfy the equation, indicating that the constraints are UNSAT.

Given Lemma 2, all that remains is to instrument the Simplex solver in order to produce the proof vector w on the fly, whenever a contradiction is detected. In case a trivial contradiction $l(x_i) > u(x_i)$ is given as part of the input query for some variable x_i , we simply return " x_i " as the proof (we later discuss also how to handle this case in the presence of dynamic bound tightenings). Otherwise, a non-trivial contradiction is detected as a result of an equation $e \equiv x_i = \sum_{j \notin \mathcal{B}} c_j \cdot x_j$, which contradicts one of the input bounds of x_i . In this case, no assignment can satisfy the equivalent equation $\sum_{j \notin \mathcal{B}} c_j \cdot x_j - x_i = 0$. Since the Simplex algorithm applies only linear operations to the input tableau,

e is given by a linear combination of the original tableau rows. Let $\text{coef}(e)$ denote the Farkas vector of the equation e , i.e., the column vector such that $\text{coef}(e)^\top \cdot A = e$, and which proves unsatisfiability in this case. Our framework simply keeps track, for each row of the tableau, of its coefficient vector; and if that row leads to a contradiction, the vector is returned.

B. Supporting dynamic bound tightening

So far, we have only considered Simplex executions that do not perform any bound tightening steps; i.e., derive UNSAT by finding a contradiction to the original bounds. However, in practice, modern DNN solvers perform a great deal of dynamic bound tightening, and so this needs to be reflected in the proof.

We use the term *ground bounds* to refer to variable bounds that are part of the LP being solved, whether they were introduced by the original input, or by successive case splits, as we will explain in Sec. V. This is opposed to *dynamic bounds*, which are bounds introduced on the fly, via bound tightening. The ground bounds, denoted $l, u \in \mathbb{R}^n$, are used in explaining dynamic bounds, denoted $l', u' \in \mathbb{R}^n$, via Farkas vectors.

For simplicity, we consider here a simple and popular version of bound tightening, called *interval propagation* [25], [48]. Given an equation $x_i = \sum_{j \notin \mathcal{B}} c_j \cdot x_j$ and current bounds $l'(x)$ and $u'(x)$ for each of the variables (whether these are the ground bounds or dynamically tightened bounds themselves), a new upper bound for x_i can be derived:

$$u'(x_i) := \sum_{x_j \notin \mathcal{B}, c_j > 0} c_j \cdot u'(x_j) + \sum_{x_j \notin \mathcal{B}, c_j < 0} c_j \cdot l'(x_j) \quad (1)$$

(provided that the new bound is tighter, i.e., smaller, than the current upper bound for x_i). A symmetrical version exists for discovering lower bounds.

A naive approach for handling bound tightening is to store, each time a new bound is discovered, a separate proof that justifies it; for example, a Farkas vector for deriving the equation that was used in the bound tightening. However, a Simplex execution can include many thousands of bound tightenings — and so doing this would strain resources. Even worse, many of the intermediate bound tightenings might not even participate in deriving the final contradiction, and so storing them would be a waste.

In order to circumvent this issue, we propose a scheme in which we store, for each variable in the query, a single column vector that justifies its current lower bound, and another for its current upper bound. Whenever a tighter bound is dynamically discovered, the corresponding vector is updated; and even if other, previously discovered dynamic bounds were used in the derivation, the vector that we store indicates how the same bound can be derived using the ground bounds. Thus, the proof of the tightened bounds remains compact, regardless of the number of derived bounds; specifically, it requires only $O(n \cdot m)$ space overall. Formally, we have the following result:

Theorem 3. *Let $A \cdot V = 0$ such that $l \leq V \leq u$ be an LP instance, where $A \in M_{m \times n}(\mathbb{R})$ and $l, V, u \in \mathbb{R}^n$.*

Let $u', l' \in \mathbb{R}^n$ represent dynamically tightened bounds of V . Then $\forall i \in [n] \exists f_u(x_i), f_l(x_i) \in \mathbb{R}^m$ such that $f_u(x_i)^\top \cdot A$ and $f_l(x_i)^\top \cdot A$ can be used to efficiently compute $u'(x_i), l'(x_i)$ from l and u . Moreover, vectors $f_u(x_i)$ and $f_l(x_i)$ can be constructed during the run of the Simplex algorithm.

When a Simplex procedure with bound tightening reaches an UNSAT answer, it has discovered a variable x_i with $l'(x_i) > u'(x_i)$. The theorem guarantees that in this case we have two column vectors, $f_u(x_i)$ and $f_l(x_i)$, which explain how $u'(x_i)$ and $l'(x_i)$ were discovered. We refer to these vectors as the *Farkas vectors* of the upper and lower bounds of x_i , respectively. Because $u'(x_i) - l'(x_i)$ is negative, the column vector $w = f_u(x_i) - f_l(x_i)$ creates a tableau row which is always negative, making $w \in \mathbb{R}^m$ a proof of unsatisfiability. The formal, constructive proof of the theorem appears in the extended version of this paper [42].

In order to maintain $f_u(x_i)$ and $f_l(x_i)$ during the execution of Simplex, whenever a tighter upper bound is tightened using Eq. 1, we update the matching Farkas vector:

$$f_u(x_i) := \sum_{j \neq i, c_j > 0} c_j \cdot f_u(x_j) + \sum_{j \neq i, c_j < 0} c_j \cdot f_l(x_j) + \text{coef}(e),$$

where e is the linear equation used for tightening, and $\text{coef}(e)$ is the column vector such that $\text{coef}(e)^\top \cdot A = e$. The lower bound case is symmetrical. To demonstrate the procedure, consider again the verification query from Fig. 2. Assume the phases of v_1, v_2 have both been set to active, and that consequently two new equations have been added: $e^4 : f_1 = b_1$, $e^5 : f_2 = b_2$. In this example, we have five linear equations, so we initialize a zero vector of size five for each of the variable bounds. Now, suppose Simplex tightens the lower bound of b_1 using the first equation e^1 :

$$l'(b_1) := l(x_1) - u(x_2) = 2 - 1 = 1$$

and thus we update

$$\begin{aligned} f_l(b_1) &:= f_l(x) - f_u(y) + \text{coef}(e^1) \\ &= [0 \ 0 \ 0 \ 0 \ 0]^\top + [0 \ 0 \ 0 \ 0 \ 0]^\top \\ &\quad + [1 \ 0 \ 0 \ 0 \ 0]^\top \\ &= [1 \ 0 \ 0 \ 0 \ 0]^\top \end{aligned}$$

since all f_l and f_u vectors have been initialized to $\bar{0}$ and $\text{coef}(e) = [1 \ 0 \ 0 \ 0 \ 0]^\top$ — which indicates that e^1 is simply the first row of the tableau.

We can now tighten bounds again, using the fourth row $f_1 = b_1$, and get $l'(f_1) := l'(b_1) = 1$. We update $f_l(f_1)$:

$$\begin{aligned} f_l(f_1) &:= f_l(b_1) + \text{coef}(e^4) \\ &= [1 \ 0 \ 0 \ 0 \ 0]^\top + [0 \ 0 \ 0 \ 1 \ 0]^\top \\ &= [1 \ 0 \ 0 \ 1 \ 0]^\top \end{aligned}$$

To see that the Farkas vector can indeed explain the dynamically tightened bound, observe that the combination $[1 \ 0 \ 0 \ 1 \ 0]^\top$ of tableau rows gives the equation $f_1 = x_1 - x_2$. We can then tighten the lower bound of f_1 , using the

ground bounds: $l'(f_1) := l(x_1) - u(x_2) = 2 - 1 = 1$. This bound matches the one that we had discovered dynamically, though we derived it using ground bounds only.

V. DNN VERIFICATION WITH PROOFS

A. Producing a proof-tree

We now discuss how to leverage the results of Sec. IV in order to produce the entire proof tree for an UNSAT DNN verification query. Recall that the main challenge lies in accounting for the piecewise-linear constraints, which affect the solving process by introducing case-splits.

Each case split performed by the solver introduces a branching in the proof tree — with a new child node for each of the linear phases of the constraint being split on — and introduces new equations and bounds. In the case of ReLU, one child node represents the active branch, through the equation $f = b$ and bound $b \geq 0$; and another represents the inactive branch, with $b \leq 0$ and $0 \leq f \leq 0$. These new bounds become the *ground bounds* for this node: their Farkas vectors are reset to zero, and all subsequent Farkas vectors refer to these new bounds (as opposed to the ground bounds of the parent node). A new node inherits any previously-discovered dynamic bounds, as well as the Farkas vectors that explain them, from its parent; these vectors remain valid, as ground bounds only become tighter as a result of splitting (see the extended version of this paper [42]).

For example, let us return to the query from Fig. 2 and the proof tree from Fig. 3. Initially, the solver decides to split on v_1 . This adds two new children to the proof tree. In the first child, representing the inactive case, we update the ground bounds $u(b_1) := 0$, $u(f_1) := 0$, and reset the corresponding Farkas vectors $f_u(b_1)$ and $f_u(f_1)$ to $\bar{0}$. Now, Simplex can tighten the lower bound of b_1 using the first equation e^1 :

$$l'(b_1) := l(x_1) - u(x_2) = 2 - 1 = 1$$

resulting in the updated $f_l(b_1) = [1 \ 0 \ 0]^\top$, as shown in Sec. IV, where we use vectors of size three since in this search state we have three equations. Observe this bound contradicts the upper ground bound of b_1 , represented by the zero vector. We can then use the vector

$$f_u(b_1) - f_l(b_1) = \bar{0} - [1 \ 0 \ 0]^\top = [-1 \ 0 \ 0]^\top$$

as a proof for contradiction. Indeed, the matrix A' , which is obtained using the first three rows and columns of A as defined in Sec. III, corresponds to the tableau before adding any new equations. Observe that $[-1 \ 0 \ 0]^\top \cdot A' \cdot V = 0$ gives the equation $-x_1 + x_2 + b_1 = 0$. Given the current ground bounds, the largest value of the left-hand side is:

$$-l(x_1) + u(x_2) + u(b_1) = -2 + 1 + 0 = -1$$

which is negative, meaning that no variable assignment within these bounds can satisfy the equation. This indicates that the proof node representing v_1 's inactive phase is UNSAT.

In the second child, representing v_1 's active case, we update the ground bound $l(b_1) := 0$ and the Farkas vector $f_l(b_1) := \bar{0}$.

We also add the equation $e^4 : f_1 = b_1$. Next, the solver performs another split on v_2 , adding two new children to the tree. In the first one (representing the inactive case) we update the ground bounds $u(b_2) := 0$, $u(f_2) := 0$, and reset the corresponding Farkas vectors $f_u(b_2)$ and $f_u(f_2)$ to $\bar{0}$. In this node, we have a contradiction already in the ground bounds, since $u(f_2) := 0$ but $l(f_2) := 0.25$. The contradiction in this case is comprised of a symbol for f_2 .

We are left with proving UNSAT for the last child, representing the case where both ReLU nodes v_1, v_2 are active. For this node of the proof tree, we update the ground bound $l(b_2) := 0$ and Farkas vector $f_l(b_2) := \bar{0}$, and add the equation $e^5 : f_2 = b_2$. Recall that previously, we learned the tighter bound $l'(f_1) = 1$. With the same procedure as described in Sec. IV, we can update $f_l(f_1) = [1 \ 0 \ 0 \ 1 \ 0]^\top$. Now, we can use $e^2 : b_2 = -2f_1$ to tighten $u'(b_2) := -2l'(f_1) = -2$, and consequently update the Farkas vector:

$$\begin{aligned} f_u(b_2) &= -2 \cdot f_l(f_1) + \text{coef}(e^2) \\ &= -2 \cdot [1 \ 0 \ 0 \ 1 \ 0]^\top + [0 \ 1 \ 0 \ 0 \ 0]^\top \\ &= [-2 \ 1 \ 0 \ -2 \ 0]^\top \end{aligned}$$

The bound $u'(b_2) = -2$, explained by $[-2 \ 1 \ 0 \ -2 \ 0]^\top$ contradicts the ground bound $l(b_2) = 0$ explained by the zero vector. Therefore, we get the vector

$$[-2 \ 1 \ 0 \ -2 \ 0]^\top - \bar{0} = [-2 \ 1 \ 0 \ -2 \ 0]^\top$$

as the proof of contradiction for this node.

B. Bound tightenings from piecewise-linear constraints

Modern solvers often use sophisticated methods [25], [50], [62] to tighten variable bounds using the piecewise-linear constraints. For example, if $f = \text{ReLU}(b)$, then in particular $b \leq f$, and so $u(b) \leq u(f)$. Thus, if initially $u(b) = u(f) = 7$ and it is later discovered that $u'(f) = 5$, we can deduce that also $u'(b) = 5$. We show here how such tightening can be supported by our proof framework, focusing on some ReLU tightening rules as specified in the extended version of this paper [42]. Supporting additional rules should be similar.

We distinguish between two kinds of ReLU bound tightenings. The first are tightenings that can be explained via a Farkas vector; these are handled the same way as bounds discovered using interval propagation. The second, more complex tightenings are those that cannot be explained using an equation (and thus a Farkas vector). Instead, we treat these bound tightenings as *lemmas*, which are added to the proof node along with their respective proofs; and the bounds that they tighten are introduced as ground bounds, to be used in constructing future Farkas vectors. The proof for a lemma consists of Farkas vectors explaining any current bounds that were used in deducing it; as well as an indication of the tightening rule that was used. The list of allowed tightening rules must be agreed upon beforehand and provided to the checker; in the extended version of this paper [42], we present the tightening rules for ReLUs that we currently support.

For example, if $f = \text{ReLU}(b)$ and $u'(f) = 5$ causes a bound tightening $u'(b) = 5$, then this new bound $u'(b) = 5$ is stored as a lemma. Its proof consists of the Farkas vector $f_u(f)$ which explains why $u'(f) = 5$, and an indication of the deduction rule that was used (in this case, $u'(b) \leq u'(f)$).

VI. PROOF CHECKING AND NUMERICAL STABILITY

Checking the validity of a proof tree is straightforward. First, the checker must read the initial query and confirm that it is consistent with the LP and piecewise-linear constraints stored at the root of the tree. Next, the checker begins a depth-first traversal of the proof tree. Whenever it reaches a new inner node, it must confirm that that node’s children correspond to the linear phases of a piecewise-linear constraint present in the query. Further, the checker must maintain a list of current equations and lower and upper bounds, and whenever a new node is visited — update these lists (i.e., add equations and tighten bounds as needed), to reflect the LP stored in that node. Additionally, the checker must confirm the validity of lemmas that appear in the node — specifically, to confirm that they adhere to one of the permitted derivation rules. Finally, when a leaf node is visited, the checker must confirm that the Farkas vector stored therein does indeed lead to a contradiction when applied to the current LP — by ensuring that the linear combination of rows created by the Farkas vector leads to a matrix row $\sum c_j \cdot x_j = 0$, such that for any assignment of the variables, the left-hand side will have a negative value.

The process of checking a proof certificate is thus much simpler than verifying a DNN using modern approaches, as it consists primarily of traversing a tree and computing linear combinations of the tableau’s columns. Furthermore, the proof checking process does not require using division for its arithmetic computations, thus making the checking program more stable arithmetically [44]. Consequently, we propose to treat the checker as a trusted code-base, as is commonly done [15], [49].

Complexity and Proof Size. Proving that a DNN verification query is SAT (by providing a satisfying assignment) is significantly easier than discovering an UNSAT witness using our technique. Indeed, this is not surprising; recall that the DNN verification problem is NP-complete, and that yes-instances of NP problems have polynomial-size witnesses (i.e., polynomial-size proofs). Discovering a way to similarly produce polynomial proofs for no-instances of DNN verification is equivalent to proving that NP = coNP, which is a major open problem [8] and might, of course, be impossible.

Numerical Stability. Recall that enhancing DNN verifiers with proof production is needed in part because they might produce incorrect UNSAT results due to numerical instability. When this happens, the proof checking will fail when checking a proof leaf, and the user will receive warning. There are, however, cases where the query is UNSAT, but only the proof produced by the verifier is flawed. To recover from these cases

and correct the proof, we propose to use an external SMT solver to re-solve the query stored in the leaf in question.

SMT solvers typically use sound arithmetic (as opposed to DNN verifiers), and so their conclusions are generally more reliable. Further, if a proof-producing SMT solver is used, the proof that it produces could be plugged into the larger proof tree, instead of the incorrect proof previously discovered. Although using SMT solvers to directly verify DNNs has been shown to be highly ineffective [48], [59], in our evaluation we observed that leaves typically represented problems that were significantly simpler than the original query, and could be solved efficiently by the SMT solver.

VII. IMPLEMENTATION AND EVALUATION

Implementation. For evaluation purposes, we instrumented the Marabou DNN verifier [50], [69] with proof production capabilities. Marabou is a state-of-the-art DNN verifier, which uses a native Simplex solver, and combines it with other modern techniques — such as abstraction and abstract interpretation [26], [27], [57], [62], [68], [71], advanced splitting heuristics [70], DNN optimization [63], and support for varied activation functions [6]. Additionally, Marabou has been applied to a variety of verification-based tasks, such as verifying recurrent networks [43] and DRL-based systems [3], [5], [28], [51], network repair [34], [60], network simplification [33], [52], and ensemble selection [4].

As part of our enhancements to Marabou’s Simplex core, we added a mechanism that stores, for each variable, the current Farkas vectors that explain its bounds. These vectors are updated with each Simplex iteration in which the tableau is altered. Additionally, we instrumented some of Marabou’s Simplex bound propagation mechanisms — specifically, those that perform interval-based bound tightening on individual rows [25], to record for each tighter bound the Farkas vector that justifies it. Thus, whenever the Simplex core declares UNSAT as a result of conflicting bounds, the proof infrastructure is able to collect all relevant components for creating the certificate for that particular leaf in the proof tree. Due to time restrictions, we were not able to instrument all of Marabou’s many bound propagation components; this is ongoing work, and our experiments described below were run with yet-unsupported components turned off. The only exception is Marabou’s preprocessing component, which is not supported, but is run before proof production starts.

In order to keep track of Marabou’s tree-like search, we instrumented Marabou’s *SmtCore* class, which is in charge of case splitting and backtracking [50]. Whenever a case-split was performed, the corresponding equations and bounds were added to the proof tree as ground truths; and whenever a previous split was popped, our data structures would backtrack as well, returning to the previous ground bounds.

In addition to the instrumentation of Marabou, we also wrote a simple proof checker that receives a query and a proof artifact — and then checks, based on this artifact, that the query is indeed UNSAT. That checker also interfaces with the

cvc5 SMT solver [14] for attempting recovery from numerical instability errors.

Evaluation. We used our proof-producing version of Marabou to solve queries on the ACAS-Xu family of benchmarks for airborne collision avoidance [45]. We argue that the safety-critical nature of this system makes it a prime candidate for proof production. Our set of benchmarks was thus comprised of 45 networks and 4 properties to test on each, producing a total of 180 verification queries. Marabou returned an UNSAT result on 113 of these queries, and so we focus on them. In the future, we intend to evaluate our proof-production mechanism on other benchmarks as well.

We set out to evaluate our proof production mechanism along 3 axes: (i) *correctness*: how often was the checker able to verify the proof artifact, and how often did Marabou (probably due to numerical instability issues) produce incorrect proofs?; (ii) *overhead*: by how much did Marabou’s runtime increase due to the added overhead of proof production?; and (iii) *checking time*: how long did it take to check the produced proofs? Below we address each of these questions.

Correctness. Over 1.46 million proof-tree leaves were created and checked as part of our experiments. Of these, proof checking failed for only 77 leaves, meaning that the Farkas vector written in the proof-tree leaf did not allow the proof checker to deduce a contradiction. Out of the 113 queries checked, 97 had all their proof-tree leaves checked successfully. As for the rest, typically only a tiny number of leaves would fail per query, but we did identify a single query where a significant number of proofs failed to check (see Fig. 4). We speculate that this query had some intrinsic numerical issues encoded into it (e.g., equations with very small coefficients [20]).

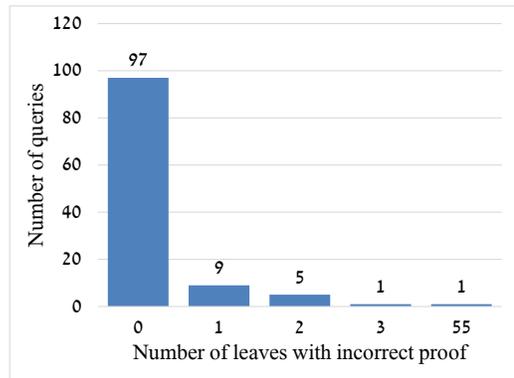


Fig. 4: Number of queries per number of leaves with incorrect proofs.

Next, when we encoded each of the 77 leaves as a query to the cvc5 SMT solver [14], it was able to show that all queries were indeed UNSAT, in under 20 seconds per query. From this we learn that although some of the proof certificates produced by Marabou were incorrect, the ultimate UNSAT result was correct. Further, it is interesting to note how quickly each of the queries could be solved. This gives rise to an

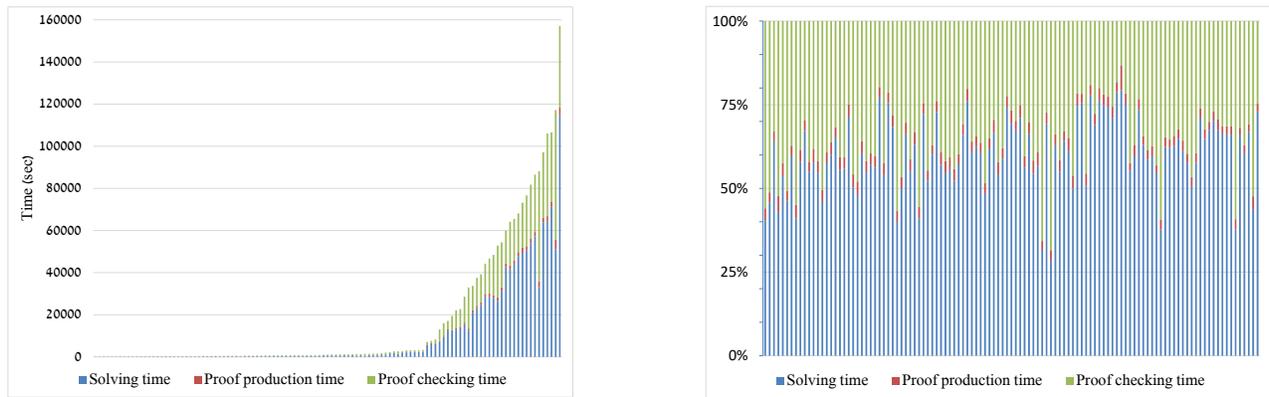


Fig. 5: Proof production and checking time comparison — absolute (left) and relative (right)

interesting verification strategy: use modern DNN verifiers to do the “heavy-lifting”, and then use more precise SMT solvers specifically on small components of the query that proved difficult to solve accurately.

Overhead and Checking Time. In Fig. 5, we compare the running time of vanilla Marabou, the overhead incurred by our proof-production extension to Marabou, and the checking time of the resulting proof certificates. We can see that the overhead of proof production time is relatively small for all queries (an average overhead of 5.7%), while the certification time is non-negligible, but shorter than the time it takes to solve the queries by a factor of 66.5% on average.

VIII. RELATED WORK

The importance of proof production in verifiers has been repeatedly recognized, for example by the SAT, SMT, and model-checking communities (e.g., [15], [21], [38]). Although the risks posed by numerical imprecision within DNN verifiers have been raised repeatedly [12], [44], [48], [47], we are unaware of any existing proof-producing DNN verifiers.

Proof production for various Simplex variants has been studied previously [56]. In [24], Dutertre and de Moura study a Simplex variant similar to ours, but without explicit support for dynamic bound tightening. Techniques for producing Farkas vectors have also been studied [10], but again without support for dynamic bound tightening, which is crucial in DNN verification. Other uses of Farkas vectors, specifically in the context of interpolants, have also been explored [9], [18].

Other frameworks for proof production for machine learning have also been proposed [7], [35]; but these frameworks are interactive, unlike the automated mechanism we present here.

IX. CONCLUSION AND FUTURE WORK

We presented a novel framework for producing proofs of unsatisfiability for Simplex-based DNN verifiers. Our framework constructs a proof tree that contains lemma proofs in internal nodes and unsatisfiability proofs in each leaf. The certificates of unsatisfiability that we provide can increase the reliability of

DNN verification, particularly when floating-point arithmetic (which is susceptible to numerical instability) is used.

We plan to continue this work along two orthogonal paths: (i) extend our mechanism to support additional steps performed in modern verifiers, such as preprocessing and additional abstract interpretation steps [53], [62]; and (ii) use our infrastructure to allow learning succinct *conflict clauses*. During search, the Farkas vectors produced by our approach could be used to generate conflict clauses on-the-fly. Intuitively, conflict clauses guide the verification algorithm to avoid any future search for a satisfying assignment within subspaces of the search space already proven to be UNSAT. Such clauses are a key component in modern SAT and SMT solvers, and are the main component of CDCL algorithms [74] — and could significantly curtail the search space traversed by DNN verifiers and improve their scalability.

Acknowledgments. This work was supported by the Israel Science Foundation (grant number 683/18), the ISF-NSFC joint research program (grant numbers 3420/21 and 62161146001), the Binational Science Foundation (grant numbers 2017662 and 2020250), and the National Science Foundation (grant number 1814369).

REFERENCES

- [1] E. Abraham and G. Kremer. SMT Solving for Arithmetic Theories: Theory and Tool Support. In *Proc. 19th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNAS)*, pages 1–8, 2017.
- [2] M. Akintunde, A. Kevochian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 197–210, 2019.
- [3] G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Verifying Learning-Based Robotic Navigation Systems, 2022. Technical Report. <https://arxiv.org/abs/2205.13536>.
- [4] G. Amir, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2022.
- [5] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.
- [6] G. Amir, H. Wu, C. Barrett, and G. Katz. An SMT-Based Approach for Verifying Binarized Neural Networks. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 203–222, 2021.

- [7] C. Anil, G. Zhang, A. Wu, and R. Grosse. Learning to Give Checkable Answers with Prover-Verifier Games, 2021. Technical Report. <https://arxiv.org/abs/2108.12099>.
- [8] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [9] S. Asadi, M. Blicha, A. Hyvärinen, G. Fedyukovich, and N. Sharygina. Farkas-Based Tree Interpolation. In *Proc. 27th Int. Static Analysis Symposium (SAS)*, pages 357–379, 2020.
- [10] D. Avis and B. Kaluzny. Solving Inequalities and Proving Farkas’s Lemma Made Easy. *The American Mathematical Monthly*, 111(2):152–157, 2004.
- [11] G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.
- [12] S. Bak, C. Liu, and T. Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, 2021. Technical Report. <http://arxiv.org/abs/2109.00498>.
- [13] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks And its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1249–1264, 2019.
- [14] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 415–442, 2022.
- [15] C. Barrett, L. de Moura, and P. Fontaine. Proofs in Satisfiability Modulo Theories. *All about Proofs, Proofs for All*, 55(1):23–44, 2015.
- [16] C. Barrett and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.
- [17] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.
- [18] M. Blicha, A. Hyvärinen, J. Kofroň, and N. Sharygina. Decomposing Farkas Interpolants. In *Proc. 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 3–20, 2019.
- [19] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. <http://arxiv.org/abs/1604.07316>.
- [20] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.
- [21] S. Conchon, A. Mebsout, and F. Zaidi. Certificates for Parameterized Model Checking. In *Proc. 20th Int. Symposium on Formal Methods (FM)*, pages 126–142, 2015.
- [22] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- [23] L. de Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011.
- [24] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. 18th Int. Conf. on Computer Aided Verification (CAV)*, pages 81–94, 2006.
- [25] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
- [26] Y. Elboher, E. Cohen, and G. Katz. Neural Network Verification using Residual Reasoning. In *Proc. 20th Int. Conf. on Software Engineering and Formal Methods (SEFM)*, 2022.
- [27] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.
- [28] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 305–318, 2021.
- [29] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean. A Guide to Deep Learning in Healthcare. *Nature medicine*, 25(1):24–29, 2019.
- [30] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, 2018.
- [31] D. Fremont, J. Chiu, D. Margineantu, D. Osipychiev, and S. Seshia. Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VERIFAI. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 122–134, 2020.
- [32] T. Gehr, M. Mirman, D. Drachler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, pages 3–18, 2018.
- [33] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.
- [34] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.
- [35] S. Goldwasser, G. Rothblum, J. Shafer, and A. Yehudayoff. Interactive Proofs for Verifying Machine Learning. In *Proc. 12th Innovations in Theoretical Computer Science Conf. (ITCS)*, 2021.
- [36] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
- [37] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. <http://arxiv.org/abs/1412.6572>.
- [38] A. Griggio, M. Roveri, and S. Tonetta. Certifying Proofs for SAT-Based Model Checking. *Formal Methods in System Design*, 57(2):178–210, 2021.
- [39] The Gurobi Optimizer. <https://www.gurobi.com/>.
- [40] P. Henriksen and A. Lomuscio. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 2513–2520, 2020.
- [41] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
- [42] O. Isac, C. Barrett, M. Zhang, and G. Katz. Neural Network Verification with Proof Production, 2022. Technical Report. <https://arxiv.org/abs/2206.00512>.
- [43] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 57–74, 2020.
- [44] K. Jia and M. Rinard. Exploiting Verified Neural Networks via Floating Point Numerical Error. In *Proc. 28th Int. Static Analysis Symposium (SAS)*, pages 191–205, 2021.
- [45] K. Julian, M. Kochenderfer, and M. Owen. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.
- [46] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
- [47] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
- [48] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMDS)*, 2021.
- [49] G. Katz, C. Barrett, C. Tinelli, A. Reynolds, and L. Hadarean. Lazy Proofs for DPLL(T)-Based SMT Solvers. In *Proc. 16th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 93–100, 2016.
- [50] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
- [51] Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 83–89, 2019.
- [52] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.
- [53] Z. Lyu, C.-Y. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc.*

- 34th AAAI Conf. on Artificial Intelligence (AAAI), pages 5037–5044, 2020.
- [54] A. Makhorin. GLPK (GNU Linear Programming Kit). <https://www.gnu.org/software/glpk/glpk.html>.
- [55] M. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In *Proc. 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2022.
- [56] G. Necula. *Compiling with Proofs*. Carnegie Mellon University, 1998.
- [57] M. Ostrovsky, C. Barrett, and G. Katz. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In *Proc. 20th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2022.
- [58] L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.
- [59] L. Pulina and A. Tacchella. Challenging SMT Solvers to Verify Neural Networks. *AI Communications*, 25(2):117–135, 2012.
- [60] I. Refaeli and G. Katz. Minimal Multi-Layer Modifications of Deep Neural Networks. In *Proc. 5th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS)*, 2022.
- [61] S. Sankaranarayanan, S. Dutta, and S. Mover. Reaching Out Towards Fully Verified Autonomous Systems. In *Proc. 13th Int. Conf. on Reachability Problems (RP)*, pages 22–32, 2019.
- [62] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 1–30, 2019.
- [63] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. *Journal of Machine Learning*, pages 1–28, 2021.
- [64] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. <http://arxiv.org/abs/1312.6199>.
- [65] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. <http://arxiv.org/abs/1711.07356>.
- [66] H.-D. Tran, S. Bak, W. Xiang, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 18–42, 2020.
- [67] R. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, Berlin, 1996.
- [68] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, pages 1599–1614, 2018.
- [69] H. Wu, A. Ozdemir, A. Zeljić, A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Păsăreanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.
- [70] H. Wu, A. Zeljić, K. Katz, and C. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 143–163, 2022.
- [71] T. Zelazny, H. Wu, C. Barrett, and G. Katz. On Reducing Over-Approximation Errors for Neural Network Verification. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2022.
- [72] C. Zhang, T. Su, Y. Yan, F. Zhang, G. Pu, and Z. Su. Finding and Understanding Bugs in Software Model Checkers. In *Proc. 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 673–773, 2019.
- [73] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.
- [74] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*, pages 279–285, 2001.

TBUDDY: A Proof-Generating BDD Package

Randal E. Bryant 

Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
Email: Randy.Bryant@cs.cmu.edu

Abstract—The TBUDDY library enables the construction and manipulation of reduced, ordered binary decision diagrams (BDDs). It extends the capabilities of the BUDDY BDD package to support *trusted* BDDs, where the generated BDDs are accompanied by proofs of their logical properties. These proofs are expressed in a standard clausal framework, for which a variety of proof checkers are available. Building on TBUDDY via its application-program interface (API) enables developers to implement automated reasoning tools that generate correctness proofs for their outcomes. In some cases, BDDs serve as the core reasoning mechanism for the tool, while in other cases they provide a bridge from the core reasoner to proof generation. A Boolean satisfiability (SAT) solver based on TBUDDY achieves polynomial scaling when generating unsatisfiability proofs for a number of problems that yield exponentially-sized proofs with standard solvers. It performs particularly well for formulas containing parity constraints, where it can employ Gaussian elimination to systematically simplify the constraints.

I. INTRODUCTION

Proof generation has become a core requirement for Boolean satisfiability (SAT) solvers when they encounter an unsatisfiable problem. The SAT solver generates a detailed proof in a standard proof format. An independent proof checker can then affirm that the problem is indeed unsatisfiable, ruling out any false negative results due to a bug in the SAT solver’s algorithms or implementation. Most modern solvers are based on conflict-driven clause-learning (CDCL) algorithms, and these can readily be extended to generate proofs in the *Deletion Resolution Asymmetric Tautology* (DRAT) proof framework [1], [2]. Like resolution proofs [3], a DRAT proof is a *clausal proof* consisting of a sequence of clauses, each of which preserves the satisfiability of the preceding clauses. An unsatisfiability proof starts with the clauses of the input formula and ends with an empty clause, indicating logical falsehood. The fact that this clause can be derived from the original formula proves that the original formula cannot be satisfied.

Although a number of SAT solvers based on Binary Decision Diagrams (BDDs) have been implemented over the years [4]–[8], most of these predated the era when proof generation became a priority. In 2006, Biere, Jussila, and Sinz demonstrated that the underlying logic behind standard BDD algorithms can be encoded as steps in an extended resolution framework [9], [10]. Extended resolution [11], [12] augments standard resolution by allowing proofs to introduce *extension variables*, serving as abbreviations for Boolean formulas over the input and other extension variables. This can yield proofs that are exponentially more compact than standard resolution

proofs [13]. Biere, Jussila, and Sinz use this capability by introducing an extension variable for each BDD node generated. The logic for each recursive step of standard BDD operations, based on the Apply algorithm [14], can then be expressed with a short sequence of proof steps. TBUDDY builds on this work.

The DRAT framework also supports extension variables. Our solver PGBDD [15], [16] (for “proof-generating BDD”) demonstrated that a BDD-based SAT solver can generate DRAT proofs of unsatisfiability by integrating proof generation into the BDD package. Our second solver PGPBS (for “proof-generating pseudo-Boolean solver”) augments the SAT solver with a *pseudo-Boolean constraint* solver, enabling it to generate DRAT proofs of unsatisfiability for problems where the input formula, described in conjunctive normal form (CNF), encodes parity and cardinality constraints [17]. PGPBS relies on the constraint solver to detect that the formula is unsatisfiable. BDDs serve only as a mechanism to prove that 1) each of the extracted constraints is implied by the input formula, and 2) each step of the solver preserves satisfiability. These two solvers achieved polynomial scaling while generating unsatisfiability proofs for a number of challenging SAT problems.

The prototype solvers PGBDD and PGPBS demonstrated that BDDs can provide a useful framework for proof-generating automated reasoning tools, but their performance, in terms of both speed and capacity, was limited by their Python implementations. In this work, we describe TBUDDY, a high performance library for constructing and manipulating trusted BDDs. TBUDDY builds on BUDDY, a BDD package written by Jørn Lind-Nielsen while he was a PhD student at the Technical University of Denmark in the late 1990s [18]. It has subsequently been used and modified by a number of others, although the current version (2.4) has been unchanged on Sourceforge since 2014. BUDDY is written in C but has a C++ interface that provides more convenient memory management. These features were carried over to the implementation of TBUDDY.

Although there are a number of BDD packages available, we chose to implement our proof-generating library by extending BUDDY for several reasons:

- Multiple studies have shown that BUDDY generally performs as well as other BDD packages [19]–[21].
- BUDDY references nodes as integer indices into an array, rather than as pointers to a node data structure. As a result, it can manage BDDs with up to two billion (2^{31})

nodes using four-byte references, rather than the eight-byte pointers required for modern, 64-bit machines.

- BUDDY does *not* use complement pointers [22], [23] to denote Boolean negation. Although these can reduce BDD sizes and enable constant-time complementation, they would greatly complicate adding proof generation. Complement pointers rely on a symmetry between True and False that is not present in clausal representations.
- The BUDDY code is clear and concise. The complete package, prior to our modifications, consists of around 13,000 lines of code. By contrast, the core of the popular CUDD package [24] has over 72,000 lines of code. CUDD includes many features that are not relevant for this work but would require updating as the core data structures are changed.
- BUDDY supports dynamic variable ordering [25]. We do not use that feature directly, since it would be challenging to keep the proof information updated as variables are swapped in the BDD. However, it enables maintaining a distinction between the numbering of variables in the input file and the ordering of those variables within the BDD. We have found this capability vital for achieving good performance on some benchmarks.

This paper describes the design and implementation of TBDD, as well as TBSAT, a proof-generating SAT solver implemented using TBDD. It presents experimental results for several scalable benchmarks that are intractable for current CDCL solvers. A complete version of the code is available at <https://github.com/rebryant/tbuddy-artifact>.

II. PROOF GENERATION WITH BDDs

Our immediate goal is to support the operations of a BDD-based SAT solver, generating one or more solutions when the formula is satisfiable and an unsatisfiability proof when it is not. Future uses of a proof-generating BDD package include a variety of automated reasoning tasks that would benefit from the assurances provided by checkable proofs of correctness.

A. Notation

Formulas are defined over a set of Boolean variables $X = \{x_1, x_2, \dots, x_n\}$. The symbols u, v and w also denote Boolean variables, possibly with subscripts. The notation \bar{u} denotes complement of variable u . A *literal* ℓ is either a variable or its complement. A clause C consists of a set of literals, and a formula ϕ consists of a set of clauses. We denote a clause as a disjunction of literals, enclosed in square brackets, e.g., $[\bar{u} \vee \bar{v} \vee w]$. A clause consisting of a single literal ℓ , denoted $[\ell]$, is a *unit clause*.

An assignment α is a mapping from the input variables X to the set $\{0, 1\}$, where 0 represents false, and 1 represents true. Assignment α is said to satisfy clause C if there is some literal $\ell \in C$ such that $\ell = x$ and $\alpha(x) = 1$, or $\ell = \bar{x}$ and $\alpha(x) = 0$. Assignment α satisfies formula ϕ if it satisfies every clause in ϕ . A formula ϕ is said to be satisfiable if it has a satisfying assignment and to be unsatisfiable if no satisfying

TABLE I
DEFINING CLAUSES FOR EXTENSION VARIABLE u REPRESENTING BDD
NODE u

Notation	Formula	Clausal Representation		
		Nonterm. child	Child is 1	Child is 0
HD(u)	$x \rightarrow (u \rightarrow u_1)$	$[\bar{x} \vee \bar{u} \vee u_1]$	1	$[\bar{x} \vee \bar{u}]$
LD(u)	$\bar{x} \rightarrow (u \rightarrow u_0)$	$[x \vee \bar{u} \vee u_0]$	1	$[x \vee \bar{u}]$
HU(u)	$x \rightarrow (u_1 \rightarrow u)$	$[\bar{x} \vee \bar{u}_1 \vee u]$	$[\bar{x} \vee u]$	1
LU(u)	$\bar{x} \rightarrow (u_0 \rightarrow u)$	$[x \vee \bar{u}_0 \vee u]$	$[x \vee u]$	1

assignment exists. A formula containing the empty clause $[\]$ cannot be satisfied.

A clausal proof consists of a sequence of clauses $C_1, C_2, \dots, C_m, C_{m+1}, \dots, C_t$ where the first m clauses are those of the input formula ϕ , while the subsequent clauses have the property that they preserve the satisfiability of the preceding clauses. That is, for all $m \leq i < t$, if the formula consisting of clauses $\{C_1, \dots, C_i\}$ is satisfiable, then so is the formula $\{C_1, \dots, C_i, C_{i+1}\}$. A proof of unsatisfiability has an empty clause as its final clause. The fact that this clause can be derived via a sequence of the steps from the input formula proves that the formula is unsatisfiable.

B. BDD Extension Variables and Defining Clauses

The BDD package maintains a directed acyclic graph consisting of a set of nodes, where each node u is either *terminal* or *nonterminal*. There are just two terminal nodes: T_0 , representing false, and T_1 , representing true. Nonterminal node u has an associated variable $\text{Var}(u) \in X$ as well as child nodes $\text{Low}(u)$ and $\text{High}(u)$. Each BDD node u represents a Boolean function, denoted $\llbracket u \rrbracket$. Terminal nodes represent constant functions: $\llbracket T_0 \rrbracket = 0$, and $\llbracket T_1 \rrbracket = 1$. The function for nonterminal node u is defined recursively using the *ITE* operator (short for “if-then-else”), where $\text{ITE}(u, v, w) = (u \wedge v) \vee (\neg u \wedge w)$:

$$\llbracket u \rrbracket = \text{ITE}\left(\text{Var}(u), \llbracket \text{High}(u) \rrbracket, \llbracket \text{Low}(u) \rrbracket\right) \quad (1)$$

The DRAT proof system supports an *extension* rule, similar to that of extended resolution [11], [12]. That is, the proof can define and reference *extension variables* serving as abbreviations for Boolean formulas over input variables and previous extension variables. Extension variable u encoding Boolean formula F is introduced by including a set of *defining clauses* in the proof encoding the formula $u \leftrightarrow F$. This capability is key to proof generation with BDDs, with an extension variable defined for every nonterminal node in the BDD.

An assignment α over the input variables can be uniquely extended to assign values to the extension variables. Extension variable u is assigned the value resulting from applying its defining formula F to the values assigned to the input and previous extension variables. For assignment α and extension variable u , we therefore have $\alpha(u) \in \{1, 0\}$.

As with the approach of Biere, Sinz, and Jussila [9], [10], each nonterminal BDD node has an associated extension variable. Nodes are denoted by boldface letters, possibly with subscripts, e.g., u, v , and v_1 , while their corresponding extension

variables are denoted with a normal face, e.g., u , v , and v_1 . The extension variables associated with the nonterminal nodes of the BDD provide the proof with a semantic definition of how BDDs encode Boolean functions according to Equation 1. More precisely, for nonterminal node v , let $\text{Ex}(v) = v$ be the extension variable associated with the node. For the two terminal nodes, define $\text{Ex}(T_0) = 0$ and $\text{Ex}(T_1) = 1$. For nonterminal node u , let $x = \text{Var}(u)$, $u_1 = \text{Ex}(\text{High}(u))$, and $u_0 = \text{Ex}(\text{Low}(u))$. Then the defining clauses for u encode the formula $u \leftrightarrow \text{ITE}(x, u_1, u_0)$. These clauses are shown in Table I. As can be seen, when both children are nonterminal, there will be four clauses, each containing three literals. When one or more children are terminal nodes, some of the formulas for the defining clauses degenerate into tautologies (indicated by table entry 1.) These are not included among the defining clauses. Others have just two literals. For BDD node u , we let $\text{Def}(u)$ denote the set of defining clauses for all nodes in the subgraph with root u .

Consider assignment α over the input variables extended to assign values to the extension variables. We will say that assignment α satisfies BDD root u with associated extension variable u if $\alpha(u) = 1$. This will occur precisely for those assignments where $\llbracket u \rrbracket$, the Boolean function associated with u , evaluates to 1.

C. RUP Proof Steps

Each logical inference for the subset of the DRAT proof system we use is based on an application of the *reverse unit propagation* (RUP) rule [26], [27]. RUP provides an easily checkable way to combine a linear sequence of resolution steps with subsumption. Let $C = [\ell_1 \vee \ell_2 \vee \dots \vee \ell_p]$ be a clause to be proved and let clauses D_1, D_2, \dots, D_k be a sequence of supporting *antecedent* clauses occurring earlier in the proof. The RUP step proves that $\bigwedge_{1 \leq i \leq k} D_i \rightarrow C$ by showing that a combination of the antecedents plus the negation of C leads to a contradiction. The negation of C is the formula $\bar{\ell}_1 \wedge \bar{\ell}_2 \wedge \dots \wedge \bar{\ell}_p$ having a CNF representation consisting of unit clauses $[\bar{\ell}_i]$ for $1 \leq i \leq p$. A RUP check processes the clauses of the antecedent in sequence, inferring additional unit clauses. In processing clause D_i , if all but one literal in the clause is the negation of one of the accumulated unit clauses, then we can add this literal to the accumulated set. The final step, with clause D_k , must cause a contradiction, i.e., all of its literals are falsified by the accumulated unit clauses.

D. The Trusted BDD API

The TBUDDY package supports the generation of *trusted* BDDs (TBDDs). These are ones that have been formally certified to be implied by the input formula. More precisely, for a trusted BDD with root node u and associated extension variable u , any assignment α to the input variables that satisfies the input formula must also assign 1 to u . This can be written as $\phi, \text{Def}(u) \models u$. This property is proved by generating a sequence of proof clauses leading to a proof of the *validating clause*, consisting of unit clause $[u]$. We use the notation \dot{u} to indicate that node u is trusted.

```

/* Generate TBDD from input clause */
tbdd tbdd_from_clause_id(int i);

/* Form conjunction of two TBDDs */
tbdd tbdd_and(tbdd u, tbdd v);

/* Upgrade BDD v to TBDD */
tbdd tbdd_validate(bdd v, tbdd u);

/* Generate proof of clause */
int tbdd_validate_clause(ilst lits, tbdd u);

```

Fig. 1. Trusted BDD API Function Prototypes

The TBUDDY API provides several procedures that enable the generation of TBDDs. Their prototypes are shown in Figure 1. In these, data types `bdd` and `tbdd` represent BDDs and TBDDs, respectively, as is described in Section III-A. Data type `ilst` is the API’s representation of integer lists.

The `tbdd_from_clause_id` operation generates the BDD representation u_i of input clause C_i , as well as a proof of unit clause $[u_i]$. The BDD representation of a clause is a linear chain. The proof that $C_i, \text{Def}(u) \models u_i$ consists of a single RUP step, with C_i plus a subset of the defining clauses for the nodes in the chain as antecedents [10].

Given trusted BDDs \dot{u} and \dot{v} , the `tbdd_and` operation first generates the BDD representation w of their conjunction. It also generates a proof that $u \wedge v \rightarrow w$, given by the clause $[\bar{u} \vee \bar{v} \vee w]$. It then uses a RUP step with this clause plus unit clauses $[u]$ and $[v]$ to prove the unit clause $[w]$, upgrading node w to \dot{w} . As is described below, the BDD construction and the proof generation are performed by a version of the BDD APPLYAND operation that generates both a BDD node and a sequence of proof steps [15], [16].

The standard version of the APPLYAND procedure recursively traverses the nodes for the two arguments and generates intermediate result nodes [14]. It maintains an *operation table* of previously computed results to ensure polynomial complexity. Given arguments u and v , it directly handles the cases where one argument is a terminal node. Failing this, it looks in the table with key $\langle u, v, \text{And} \rangle$ and returns any stored result. Otherwise, a set of recursive calls is required. The program chooses variable x as the least (in the BDD variable ordering) among variables $\text{Var}(u)$ and $\text{Var}(v)$ and splits into two cases, given by nodes u_1 and v_1 , and nodes u_0 and v_0 . It recursively computes nodes w_1 and w_0 as the conjunctions of u_1 and v_1 , and of u_0 and v_0 , respectively. When $w_1 = w_0$, this becomes the returned result w . Otherwise node w is created having $\text{Var}(w) = x$, $\text{High}(w) = w_1$, and $\text{Low}(w) = w_0$. Before returning, an entry with key $\langle u, v, \text{And} \rangle$ and result w is added to the table.

The modified version of APPLYAND operation follows this recursive structure, such that a recursive call generating node w as the conjunction for nodes u and v also generates a proof of the clause $[\bar{u} \vee \bar{v} \vee w]$, i.e., that $u \wedge v \rightarrow w$. We refer to this proof step as the *justifying clause* for the operation. The recursive calls will have generated proofs of the clauses

$[\bar{w}_1 \vee \bar{v}_1 \vee w_1]$ and $[\bar{w}_0 \vee \bar{v}_0 \vee w_0]$. In general, the desired result can require two RUP steps. The first generates a proof of the intermediate result $x \rightarrow (u \wedge v \rightarrow w)$ given by clause $[\bar{x} \vee \bar{u} \vee \bar{v} \vee w]$ using as antecedents the defining clauses $\text{HD}(u)$, $\text{HD}(v)$, and $\text{HU}(w)$, as well as the recursive result $[\bar{u}_1 \vee \bar{v}_1 \vee w_1]$. The second step proves the target clause using as antecedents the intermediate result, defining clauses $\text{LD}(u)$, $\text{LD}(v)$, and $\text{LU}(w)$, and the recursive result $[\bar{u}_0 \vee \bar{v}_0 \vee w_0]$. For special cases, such as when some of the arguments are terminal nodes, only a subset of these antecedents is required. In some cases, the desired proof degenerates to a single proof step. The proof generation code in TBUDY attempts to generate a single-step proof when one of the recursive results is a tautology. When this fails, or for the more general case, it generates a two-step proof. A built-in RUP checker determines which clauses to use as antecedents and can detect whether the proof succeeds or fails. The intermediate clause generated in a two-step proof can be deleted immediately after the second clause is added, and therefore there is a single justifying clause associated with each recursive operation.

Observe that to reuse results from the operation table, the program needs to reference its justifying clause. This requires augmenting the table entry with a field to hold an identifier for the justifying clause, as is discussed in Section III-A.

The `tbdd_validate` operation enables an ordinary BDD with root v to be upgraded to trusted node \hat{v} based on trusted node \hat{u} . When called, the program first generates a proof of the implication $u \rightarrow v$, given by the clause $[\bar{u} \vee v]$. It then uses a RUP step with this clause plus unit clause $[u]$ to prove the unit clause $[v]$. The implication proof is generated by `PROVEIMPLICATION` [15], an operation that traverses the BDD and generates proof steps without adding any nodes. At each step on arguments u' and v' , it generates a proof of the justifying clause $[\bar{u}' \vee v']$, i.e., that $u' \rightarrow v'$, using a simplified version of the proof structure used for the conjunction operation.

Some applications of TBDDs combine BDD and clausal reasoning, alternating between the two forms. The `tbdd_validate_clause` operation transfers the trust embodied in TBDD node \hat{u} to a clause C , generating a proof of $\text{Def}(u), u \models C$: This function requires TBUDY to generate a sequence of proof steps, concluding with a RUP step with the specified clause. In some cases, the step can be performed directly by tracing a path in the BDD from u down to node T_0 and listing some of the defining clauses along the way as antecedents. In cases where the path is not unique, the prover must first generate a BDD representation v of the clause, validate v , and then trace the path from v to T_0 .

E. Proof File Format

There are several different file formats for encoding a DRAT proof, representing different trade-offs between the level of detail that must be supplied by the proof generator, versus the effort required to check the validity of the proof. With the *LRAT* format [28], each proof step must be accompanied

by a *hint*. For a RUP step, the hint specifies the sequence of antecedent clauses. These proofs can be checked efficiently by the program `LRAT-CHECK`. There are also several formally verified checkers for LRAT proofs [28], [29]. By contrast, no hints are given with the *DRAT* format [2]. For each RUP step, the checker must identify a sequence of prior clauses that can serve as the antecedent. This format is accepted by the widely used `DRAT-TRIM` checker. Internally, `DRAT-TRIM` operates by adding the hints and then invoking an LRAT checker. The *FRAT* format [30] spans the two extremes of hints versus no hints by making the hints optional. It also operates by adding hints and invoking an LRAT checker. TBUDY can generate proofs in any of these formats. Here we describe properties of the LRAT file format that influence how BUDDY encodes and stores proof information. Generating proofs in other formats requires storing additional information. For long executions, the proofs can range up to one billion clauses. These would be far too long for the `DRAT-TRIM` checker, due to the high cost of generating hints. In practice, therefore, it is best to either generate LRAT proofs or to generate FRAT proofs where the steps involving BDD operations include hints.

Following the conventions of the DIMACS format for encoding CNF formulas, the proof clauses for a formula with n variables and m clauses are encoded using signed integers to represent literals, where variable x_i is represented as the value i , and its complement as $-i$. Each clause in the proof is assigned a numeric *clause ID*, with the first m of these corresponding to the input clauses (which are not included in the proof file). Clause IDs must be in ascending order, but they need not be consecutive. Extension variables are represented by integers with values greater than n . RUP proof steps are encoded by giving the clause ID, the literals of the clause, and a list of the antecedent clause IDs. LRAT also supports *clause deletion*, where a list of clause IDs is provided, indicating that the proof will no longer use these clauses as antecedents. Deleting clauses whenever possible is critical for the proof checker, since it must retain copies of all *active* clauses, i.e., those that have been added but not yet deleted.

III. IMPLEMENTATION

With this as background, we can now describe how the BUDDY BDD package was modified to support proof generation. As we have seen, the key requirements are:

- Each time a new BDD node is created, it must be assigned an extension variable and its defining clauses must be added to the proof.
- For each input clause C_i , its BDD representation u_i must be generated, along with a proof of validating clause $[u_i]$.
- Every recursive step of the `APPLYAND` and `PROVEIMPLICATION` operations must generate one or two proof steps.
- The result nodes and proof steps generated by BDD operations must be stored for later reuse.
- A RUP step is required to prove validating clause $[u]$ when BDD root u is generated by conjunction or implication testing.

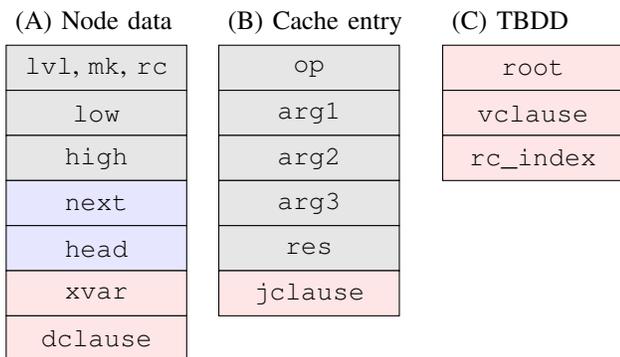


Fig. 2. Data structures for nodes (A) cache entries (B), and TBDDs (C). Each rectangle represents four bytes. Proof generation requires adding the fields shown in red.

- The defining clauses for the nodes and the clauses generated by RUP steps should be deleted when they are no longer required for subsequent proof steps.

These capabilities can all be incorporated into the basic BDD operations, as well as the supporting operations to manage the data structures.

A. Data Structures

Figure 2(A) and (B) show the fields in the two major data structures for BUDDY, with added fields (shown in red) to support proof generation. It also shows the representation for a TBDD (C). A BDD node in BUDDY is indicated by an integer, providing an index into an array of node structures, each having the fields shown in (A). Nodes T_0 and T_1 are represented by indices 0 and 1, respectively. Each rectangle in the figure represents four bytes. The node array integrates the set of BDD nodes with the *unique table*, providing a mapping from the children and variable for each node to the node itself. In the node data structure (A), the fields indicated in gray encode the node. Three values are packed into the first four-byte word: `lvl`, encoding the position of the node variable in the BDD variable ordering, `rc`, a reference count used to track external references to the node, and `mk`, a single bit used to support mark-sweep garbage collection. The indices for the two children `low` and `high` occupy the second and third words. The fields shown in blue encode the unique table, with the `next` field forming a link in the linked list implementing a hash table bucket, and the `head` field providing the head of the linked list for all nodes that hash to this index.

As mentioned earlier, to support dynamic variable ordering, BUDDY distinguishes between the level of a variable, giving its position in the BDD variable ordering, and the integer representation of the variable, with permutation vectors providing the mapping between these two. We use this feature to allow the BDD variable ordering to be independent of the numbering of variables in the input file.

Supporting proof generation requires adding two fields to the node data structure. The `xvar` field gives the associated extension variable, encoded as an integer having a value greater than the number of input variables n . When a node

is created, the next four clause IDs are assigned to its defining clauses, even if only some subset of these is added to the proof. The `dclause` field stores the first of these—the remaining three can be computed as offsets from this field. In skipping some possible clause IDs, we add some sparseness to the ID space. Considering that we can only encode around two billion ($2^{31} - 1$) clause IDs, and proofs can routinely reach one billion clauses, this might seem wasteful. However, only a small fraction of the nodes in large BDDs will have terminal nodes as children, and so the vast majority of nodes will require the full complement of four defining clauses.

Like other BDD packages [22], BUDDY stores its table of previously computed results as a direct-mapped cache indexed by a hash of the operation and arguments.¹ Before performing the recursive steps of an APPLY operation, the table is first referenced to see if a suitable result has already been generated. When a new result is added to the table, any previous result that hashes to the same position is overwritten. The entries in the cache are shown in Figure 2(B). The standard entries (shown in gray) encode the operation, arguments (up to three), and the result node, each given as a four-byte integer. In the event the operation is either APPLYAND or PROVEIMPLICATION, reusing the cached result also requires the ID of the justifying clause. This is stored in the field `jclause`.

The added fields enable TBUDDY to track the clause IDs of the defining clauses for the active BDD nodes and the justifying clauses of the cache entries. Significantly, TBUDDY need not keep copies of the clauses themselves. When actual clauses are required to support proof generation, they can be recreated based on other information stored with the node or the cache entry.

We can see that the node data structure expands from 20 bytes to 28 in order to support proof generation. Cache entries require 24 bytes with or without proof generation, since an eight-byte field is used to store results for operations that return floating-point numbers. We configured the program to maintain a cache size that has 1/8 the number of entries as the node array. Therefore, adding proof generation required growing these two data structures from combined total of 23 bytes per node to 31 bytes per node, an increase of 1.35 \times . These are the only two data structures that grow in proportion to the number of BDD nodes.

Figure 2(C) shows the representation of a TBDD. It consists of three integers. The first identifies the root node and the second gives the clause identifier for the validating clause. The third field, labeled `rc_index`, supports reference counting of TBDDs. This count is distinct from the reference count for the root node, since there may be references to a BDD node that are independent of its use in a TBDD. The reference count for a TBDD tracks references to possible uses of the validating clause in proof generation. Once the count drops to zero, the clause can be deleted. Since TBDD structures

¹The standard BUDDY package maintains seven separate caches to support different operations. We combined these into a single, unified cache.

are passed by value, they cannot hold actual reference counts. Instead, a separate table of reference counts is maintained, with the `rc_index` field providing an index into this table. In typical applications, fewer than 1% of the BDD nodes serve as TBDD roots, and so the space required by this table is negligible.

As can be seen, the modifications to support proof generation are fairly modest. In terms of code, the original BUDDY package contains 13,186 lines of source code. The TBDD package expands this to 18,030, with 1,061 lines added to existing files, 2,715 lines in new files to support proof generation and TBDDs, and 1,068 in new files to support parity reasoning. As noted above, the memory used increases by around 1.35 \times . The impact on runtime is more variable; we show experimental results in Section V.

B. BDD Management

BUDDY represents all of the nodes as a single array. This array starts with an initial allocation and is expanded as more nodes are added. Each expansion requires allocating a larger array, copying over existing nodes, and reconstructing the unique table and free list. Before expanding, it attempts to free existing nodes by performing garbage collection, reclaiming nodes that cannot be reached by any reference external to the data structure. Garbage collection is supported by 1) having each node store a reference count indicating the number of external references to the node, and 2) performing mark-sweep garbage collection to determine which nodes are unreachable. Nodes with nonzero reference counts provide the starting points of the marking phase. Both resizing the node array and performing garbage collection cause the entire cache to be flushed, with all entries marked as invalid. Garbage collection can occur at any point during the program operation, including in the middle of a series of recursive calls. To support this capability, a stack is maintained indicating intermediate nodes that may be required at future points in the outstanding calls. These nodes are also incorporated into the marking phase.

Garbage collection and cache flushing provide the means to manage the active clauses in a proof. That is, when a node is reclaimed during the sweep phase, its defining clauses are deleted. When a cache entry is evicted, either because it is overwritten or the cache is flushed, its justifying clause is deleted. To support the ability to perform garbage collection in the middle of a sequence of recursive calls, the deletion steps are not added to the proof directly. Rather, they are added to a list, which is cleared as the top-level of the recursion completes. As mentioned earlier, the validating clauses for TBDDs are managed via a separate set of reference counts. The C++ interfaces to the package automatically handle the reference counting for both BDDs and TBDDs.

IV. CAPABILITIES SUPPORTED BY TBDD

Building on the basic support for TBDDs, we have created several additional libraries and a BDD-based SAT solver. We describe these capabilities here and present some experimental results in Section V.

A. Parity Reasoning

Parity constraints arise in a variety of contexts, but they are not well handled by current CDCL solvers. A parity constraint is an equation of the form:

$$x_{i_1} \oplus x_{i_2} \oplus \dots \oplus x_{i_k} = p \quad (2)$$

The variables in this constraint are a subset of the input variables, and the phase p is 1 for odd parity and 0 for even. Adding two parity constraints creates a new parity constraint. Gaussian or Gauss-Jordan elimination systematically adds constraints to yield a reduced set [31]. It can determine when the set of constraints cannot be satisfied. When the constraints are satisfiable, it can be used to derive a satisfying assignment.

Manipulating parity constraints is especially efficient for BDDs. The BDD representation of a constraint with k variables contains $2k + 1$ nodes, independent of the BDD variable ordering. As we have demonstrated [17], a set of parity constraints encoded in CNF can be automatically extracted from an input formula, and BDD-based proofs of unsatisfiability can be generated using Gaussian elimination. The TBDD package provides the necessary support for the proof generation portion of this task.

Our constraint library represents a parity constraint as a list of integer variable IDs, a phase, and a TBDD giving the BDD representation of the constraint as well as the ID of a validating clause justifying that this constraint is implied by the input formula. An input constraint is converted into this representation by 1) forming the TBDD representations of the input clauses that encode it, 2) conjuncting them, and 3) using this TBDD to validate a BDD representation of the constraint. Each time constraints having TBDD representations \dot{u} and \dot{v} are summed to form a constraint with BDD representation \dot{t} , we use the conjunction operation to generate TBDD \dot{w} representing the conjunction of the constraints and validate the sum by calling `tbdd_validate(\dot{t} , \dot{w})`.

Applying Gaussian elimination requires first running a preprocessor to identify how the clauses encode parity constraints [17]. The program creates a *schedule* listing equations of the form of Equation 2 and identifying which clauses encode each of these. It also provides a list of the *internal* variables, i.e., those appearing only in parity constraints. Implicitly, all other variables are *external*. Gaussian elimination reduces the set of constraints to a smaller set over only the external variables. If the reduced set contains a constraint of the form $0 = 1$, then the original set cannot be satisfied. Otherwise, any solution of the reduced set can be expanded into a solution of the original set. In either case, the reduced constraints have TBDD representations and can therefore be used in proof generation.

Our Gaussian elimination routine attempts to preserve the sparseness found in typical parity constraint problems, where the number of variables in the constraints is far less than the total number of variables in the problem. Maintaining sparseness requires a successful strategy for *pivot selection*. Consider a set of parity constraints P_1, P_2, \dots, P_m , each of the form of

Equation 2. Let the notation $x_j \in P_i$ indicate that constraint P_i contains variable x_j . Each elimination step requires selecting a pivot constraint P_s and a pivot variable $x_t \in P_s$. It then eliminates variable x_t from all other constraints P_i for which $x_t \in P_i$ by replacing P_i with the sum $P_i \oplus P_s$. Our routine uses a greedy pivot selection strategy attributed to Markowitz [32], [33]: Let c_s be the number of nonzero variables in constraint P_s and r_t be the number of constraints containing variable x_t . Then a constraint P_s and variable $x_t \in P_s$ are selected such that the cost function $(c_s - 1) \cdot (r_t - 1)$ is minimized. That cost is an upper bound on the net number of variables that will be added to the constraints when generating the sums $P_i \oplus P_s$.

B. The TBSAT SAT Solver

The TBSAT solver builds on the TBUDDY library. It can generate multiple solutions for satisfiable formulas and proofs of unsatisfiability for unsatisfiable formulas. It starts by reading the input clauses and forming their TBDD representations. The overall control flow is determined by the combination of an optional input schedule file and bucket elimination, expanding on the capabilities implemented in our prototype solvers PGBDD [15] and PGPBS [17]. The schedule file can serve two different roles. In one, it specifies a sequence of conjunction and existential quantification operations using a stack-based notation. This mode can be effective when the user has some problem-dependent strategy for solving a particular problem. In the other form, it identifies sets of clauses forming parity constraints. These constraints are converted into TBDDs and simplified using Gaussian elimination. In some cases, a TBDD with root node T_0 will be generated while processing the schedule file. That indicates the formula is unsatisfiable and the proof of unsatisfiability will be complete. Otherwise, the TBDDs remaining, including those of unused input clauses, are processed using bucket elimination. When no schedule file is provided, all clauses are processed in this manner.

Bucket elimination [8], [9], [34] processes the TBDDs according to some ordering of the variables. Our implementation makes the simplifying assumption that buckets are ordered according to the BDD variable ordering, with bucket i associated with input variable x_i . Each TBDD is stored in a list (the “bucket”) according to its root node variable. Buckets are processed from the least to the greatest. For bucket i , a conjunction of the TBDDs in the bucket is computed to yield TBDD \hat{u}_i . A new BDD is computed as $v_i = \text{Low}(\hat{u}_i) \vee \text{High}(\hat{u}_i)$, existentially quantifying x_i from \hat{u}_i . This BDD is validated using TBDD \hat{u}_i , since any Boolean function f and variable x satisfies $f \rightarrow \exists x f$. The resulting TBDD \hat{v}_i is then placed in the bucket corresponding to its root node variable. This process continues until either 1) the TBDD \hat{T}_0 is generated, or 2) all buckets are processed with the final step yielding $v_n = T_1$. In the former case, the formula is unsatisfiable and the unsatisfiability proof is complete. In the latter case, the formula is satisfiable and the next task is to generate one or more solutions.

To generate a solution, the solver starts with an empty assignment and works in reverse order, adding assignments

to variables x_n through x_1 . Let $\alpha_{n+1} = \emptyset$. For bucket i , it can assume that α_{i+1} satisfies v_i , and we must assign a value to x_i . Let $\mathbf{u}_1 = \text{High}(\mathbf{u}_i)$ and $\mathbf{u}_0 = \text{Low}(\mathbf{u}_i)$. Assignment α must satisfy at least one of these. In the event that just \mathbf{u}_1 is satisfied, assign 1 to x_i . If just \mathbf{u}_0 is satisfied, then assign 0 to x_i . Otherwise, x_i can be assigned an arbitrary value. No further BDD generation is required to find a solution.

To generate a solution where some of the variables have been eliminated by Gaussian elimination, the solver first continues the elimination process to simplify the intermediate parity constraints via Gauss-Jordan elimination [31]. It uses BDD representations of these constraints to generate assignments for the internal variables. To generate multiple solutions, a new clause is created as the negation of the generated assignment, and the buckets are reprocessed in forward order. If this processing yields BDD node T_0 , then no further solutions exist. Otherwise, the bottom-up generation of an assignment will be guaranteed to find a new solution.

V. EXPERIMENTAL EVALUATION

As a general purpose SAT solver, TBSAT is no match for state-of-the-art CDCL solvers. Among benchmarks used in recent SAT competitions, it succeeds only on the TSEITIN-GRID parity constraint problems [35]. On the other hand, it handles classes of problems for which CDCL solvers fare poorly. BDD-based approaches can best complement CDCL, rather than compete with it.

Table II shows the performance of proof-generating SAT solvers on several scalable, unsatisfiable challenge problems. It compares different operating modes of TBSAT to KISSAT, a state-of-the-art CDCL solver [36]. It shows a progression of problem sizes, with the most difficult benchmark for one approach becoming the starting point for the next. All experiments were performed on a 3.2 GHz Apple M1 Max processor with 64 GB of memory and running the OS X operating system. The proofs were checked using DRAT-TRIM for the proofs generated by KISSAT and LRAT-CHECK for those generated by TBSAT. For LRAT proofs over 500 million clauses, we used a modified version of LRAT-CHECK that better exploits the sparseness in the proof structure that arises when a large fraction of the clauses is deleted. The column labeled “SAT Time” indicates the time (in seconds) taken by the solver, and the column labeled “Check Time” indicates the time taken by the checker. The column labeled “Proof Clauses” indicates the number of clauses in the generated proof. Entries marked “—” indicate a failure by the program to complete. The following benchmark problems were evaluated:

- *Mutilated chessboard*: Tile an $n \times n$ chessboard with dominos. Two opposite corners are removed from the chessboard, making the task impossible [37]. The problem size, in terms of the number of variables and clauses, scales as $O(n^2)$.
- *Pigeonhole*: Assign $n+1$ pigeons to n holes such that no hole contains more than one pigeon [38]. The at-most-one constraints are encoded using auxiliary variables [39]. The problem size scales as $O(n^2)$.

TABLE II
PERFORMANCE OF KISSAT AND TBSAT ON UNSATISFIABLE CHALLENGE PROBLEMS

Solver	Method	Problem Size	Variables	Clauses	SAT Time	Proof Clauses	Check Time
Mutilated Chessboard							
KISSAT	CDCL	16	476	1,592	358.7	12,621,694	618.5
KISSAT	CDCL	18	608	2,044	1314.9	38,083,824	1295.8
TBSAT	Column scan	18	608	2,044	0.1	111,163	0.1
TBSAT	Column scan	368	270,108	943,544	898.2	568,261,363	568.8
Pigeonhole							
KISSAT	CDCL	13	351	508	1116.1	66,263,560	2041.8
KISSAT	CDCL	14	406	589	6077.2	331,858,919	—
TBSAT	Column scan	14	406	589	0.1	92,687	0.1
TBSAT	Column scan	254	129,286	193,549	898.5	898,819,648	993.5
Chew-Heule parity formulas							
KISSAT	CDCL	40	114	304	334.3	29,133,644	594.2
KISSAT	CDCL	44	126	336	3103.6	227,489,490	8254.9
TBSAT	Bucket elim.	44	126	336	0.1	24,492	0.1
TBSAT	Bucket elim.	8,666	25,992	69,312	894.7	505,637,209	523.4
TBSAT	Gauss. elim.	8,666	25,992	69,312	4.6	5,066,914	5.2
TBSAT	Gauss. elim.	699,051	2,097,147	5,592,392	645.3	575,600,179	656.1
Urquhart-Li parity formulas							
KISSAT	CDCL	3	153	408	—	—	—
TBSAT	Bucket elim.	3	153	408	0.1	38,598	0.1
TBSAT	Bucket elim.	35	25,305	67,480	784.6	349,400,890	230.8
TBSAT	Gauss. elim.	35	25,305	67,480	3.8	4,232,657	4.3
TBSAT	Gauss. elim.	316	2,093,184	5,581,824	529.3	484,548,938	346.9

- *Chew-Heule*: Enforce both odd and even parity constraints on the n input variables. Each constraint is encoded linearly using $n - 1$ auxiliary variables, with the second constraint using a random permutation of the variables [40]. The problem size scales as $O(n)$.
- *Urquhart-Li*: A parity constraint problem devised by Urquhart [41], defined over a bipartite graph with $2m^2$ nodes. The problem size scales as $O(m^2)$. We use the benchmark generator implemented by Li [42].

The formulas were evaluated for different values of the scaling parameter n or m . Runs of TBSAT were limited to 900 seconds—longer runs generally produced proofs that exceeded the capacity of the proof checker. KISSAT was allowed to run for up to 7200 seconds.

The limitations of CDCL solvers for these problems are clearly indicated by the results for KISSAT. It can only handle relatively small instances. We also found that allowing longer run times does not have a significant effect, due to the exponential scaling. For example, KISSAT completes the mutilated chessboard problem for $n = 16$ in 360 seconds, but once it reaches $n = 20$, the solver runs for over two hours without completing. Similarly, KISSAT completes the pigeonhole problem for $n = 12$ in just 42 seconds, but once it reaches $n = 14$, it requires nearly 1.7 hours and generates a proof that is too large for DRAT-TRIM to check. For the Chew-Heule formulas, KISSAT can only complete $n \leq 44$ within the 7200-second time limit. We ran KISSAT for over 16 hours on the smallest instance of the Urquhart-Li benchmark, having $m = 3$, but it did not complete. It is remarkable that a problem with just 153 variables and 408 clauses could be so challenging for CDCL solvers.

By contrast, TBSAT achieves polynomial scaling for all four benchmarks. In earlier work [15], we presented *column scanning* to efficiently generate unsatisfiability proofs of the mutilated chessboard and pigeonhole problems. This approach performs a sequence of conjunction and quantification steps to effectively sweep through the columns of the chessboard or the pigeons in the pigeonhole problem in a manner inspired by symbolic model checking. TBSAT can also apply column scanning, easily handling the limiting instances for KISSAT. It can scale to $n = 368$ for the mutilated chessboard problem and to $n = 254$ for the pigeonhole problem within the 900-second time limit. Even though the generated proofs are very large, they can be verified by the modified version of LRAT-CHECK. It remains to be seen whether column scanning can be made more general and with automatic generation of the schedule and variable order.

TBSAT can apply bucket elimination to the two parity problems with good effect. It can easily handle the limiting instances for KISSAT, and it scales to the Chew-Heule benchmark for $n \leq 8666$ and the Urquhart-Li benchmark for $m \leq 35$ within a 900-second time limit.

Perhaps the most striking results are those using Gaussian elimination. By exploiting the sparse structure of the formulas, TBSAT can solve very large instances of the Chew-Heule and Urquhart benchmarks quickly. The limiting factor for both of these problems is that BUDDY allocates only 21 bits for the level field in each BDD node (Figure 2(A)), limiting it to a maximum of $2^{21} - 1$ (2,097,151) input variables. This prevents it from going beyond $n = 699,051$ for Chew-Heule and $m = 316$ for Urquhart, each having over two million input variables and five million clauses. Obtaining these results

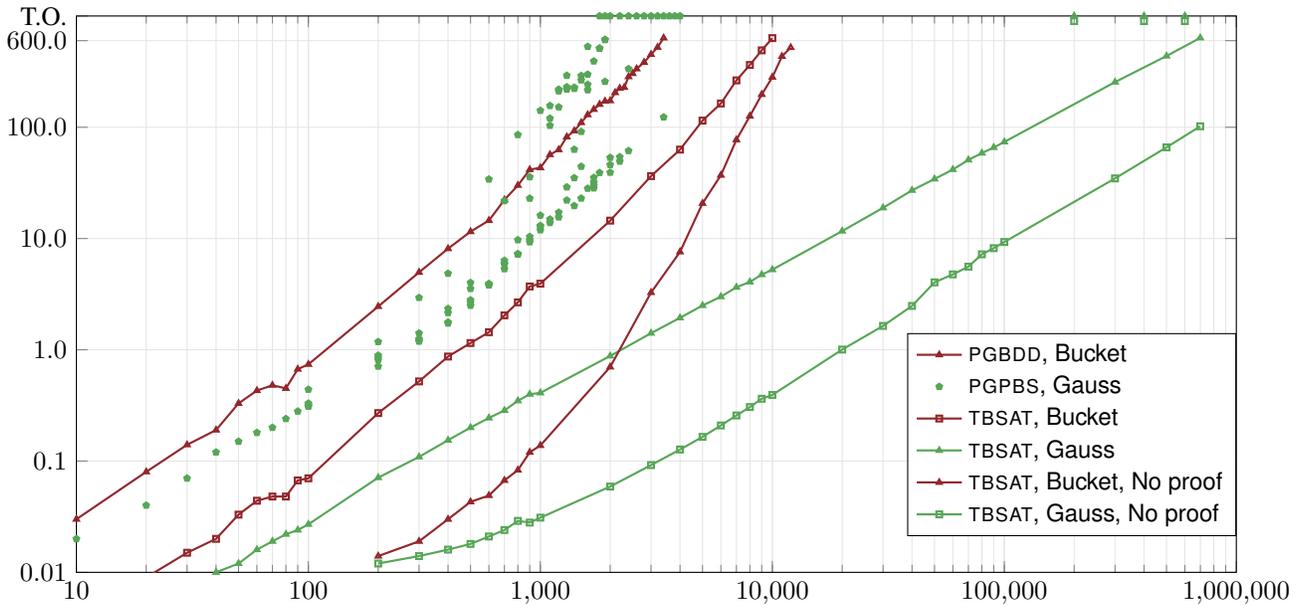


Fig. 3. Elapsed times (in seconds) for different solvers and solution methods on Chew-Heule parity formulas, as function of problem size n

requires no guidance for the user, and it is insensitive to the BDD variable ordering.

Figure 3 presents more runtime data for the Chew-Heule parity formula benchmark as a function of problem size n , enabling us to compare the relative performance and scaling of different solvers and solution methods. The red lines show three different versions of solving via bucket elimination. The top red line shows the performance of our prototype solver PGBDD, while the middle line shows the times for TBSAT. As can be seen, TBSAT consistently ran 10–12 \times faster. This can be attributed to the advantage of compiled C/C++ code versus interpreted Python. The lower red line shows the performance of TBSAT when proof generation is not required. This mode performs only the conjunction and quantification BDD operations, without generating proof clauses or writing them to a file. For smaller values of n , the runtime can be up to 33 \times faster, but this advantage drops to just a factor of 2 \times for larger values. For large values of n , the cost of garbage collection becomes a more dominant concern.

The data shown in green give results for three different versions of solving via Gaussian elimination. The data points at the top show the performance of our prototype pseudo-Boolean solver PGPBS. We found that the runtimes and generated proof sizes varied widely depending on the random permutation of the second parity constraint, and so the plot shows the raw data for five different random seeds for each value of n , including timeouts. The variation depends on whether or not the greedy pivot selections kept the constraints sparse. The middle green line shows the performance of TBSAT using Gaussian elimination. As noted before, it scales very well, nearly reaching its upper limit of $n = 699,051$ within the 600-second time limit. Compared to even the best data points for PGPBS, we see that TBSAT achieves much better

scaling despite using very similar algorithms. However, like PGPBS, its ability to maintain sparseness depends on both the particular permutation of the second parity constraint, as well as the random tie breaking done during pivot selection. Consequently, some data points yielded timeouts. The lower green line shows the performance of TBSAT using Gaussian elimination, but without proof generation. In this mode, it need not perform any BDD operations and hence can be very fast, reaching a maximum of 15.3 \times faster for $n = 3,000$, but dropping off to 6.2 \times as n approaches its limiting value.

Overall, these measurements show that 1) TBSAT greatly outperforms the prototype implementations, 2) adding proof generation can slow performance considerably, but the penalty diminishes for larger benchmarks, 3) Gaussian elimination greatly increases the speed and capacity of the solver for parity constraint problems, and 4) careful pivot selection is required to maintain sparseness during Gaussian elimination.

VI. CONCLUSIONS AND ACKNOWLEDGEMENTS

The TBUDDY library provides a powerful framework for creating automated reasoning tools that generate proofs of correctness. Building on an established BDD package, it can generate clausal proofs justifying the correctness of each step in its recursive algorithms. The TBSAT solver is especially strong for handling problems with parity constraints. We have also incorporated its proof-generation capability into a CDCL solver that uses Gauss-Jordan elimination for parity reasoning [43]. We anticipate implementing other automated reasoning tools using TBUDDY.

Thanks to Marijn Heule for his continued advice and for creating a high capacity version of LRAT-CHECK. This work was supported by the U. S. National Science Foundation under grant CCF-2108521.

REFERENCES

- [1] M. J. H. Heule, W. A. Hunt, Jr., and N. D. Wetzler, “Verifying refutations with extended resolution,” in *Conference on Automated Deduction (CADE)*, ser. LNCS, vol. 7898, 2013, pp. 345–359.
- [2] N. D. Wetzler, M. J. H. Heule, and W. A. Hunt Jr., “DRAT-trim: Efficient checking and trimming using expressive clausal proofs,” in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 8561, 2014, pp. 422–429.
- [3] J. A. Robinson, “A machine-oriented logic based on the resolution principle,” *JACM*, vol. 12, no. 1, pp. 23–41, January 1965.
- [4] R. Damiano and J. Kukula, “Checking satisfiability of a conjunction of BDDs,” in *Design Automation Conference (DAC)*, June 2003, pp. 818–923.
- [5] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet, “SBSAT: a state-based, BDD-based satisfiability solver,” in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 2919, 2004, pp. 398–410.
- [6] J. Huang and A. Darwiche, “Toward good elimination orders for symbolic SAT solving,” in *International Conference on Tools for Artificial Intelligence (ICTAI)*, 2004, pp. 566–573.
- [7] H. Jin and F. Somenzi, “CirCUs: A hybrid satisfiability solver,” in *Theory and Applications of Satisfiability Testing (SAT)*, ser. Lecture Notes in Computer Science, vol. 3542, 2005, pp. 211–223.
- [8] G. Pan and M. Y. Vardi, “Search vs. symbolic techniques in satisfiability solving,” in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 3542, 2005, pp. 235–250.
- [9] T. Jussila, C. Sinz, and A. Biere, “Extended resolution proofs for symbolic SAT solving with quantification,” in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 4121, 2006, pp. 54–60.
- [10] C. Sinz and A. Biere, “Extended resolution proofs for conjoining BDDs,” in *Computer Science Symposium in Russia (CSR)*, ser. LNCS, vol. 3967, 2006, pp. 600–611.
- [11] O. Kullmann, “On a generalization of extended resolution,” *Discrete Applied Mathematics*, vol. 96–97, pp. 149–176, 1999.
- [12] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” in *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Springer, 1983, pp. 466–483.
- [13] S. A. Cook, “A short proof of the pigeon hole principle using extended resolution,” *SIGACT News*, vol. 8, no. 4, pp. 28–32, Oct. 1976.
- [14] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.
- [15] R. E. Bryant and M. J. H. Heule, “Generating extended resolution proofs with a BDD-based SAT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Part I, ser. LNCS, vol. 12651, 2021, pp. 76–93.
- [16] —, “Generating extended resolution proofs with a BDD-based SAT solver,” *CoRR*, vol. abs/2105.00885, 2021.
- [17] R. E. Bryant, A. Biere, and M. J. H. Heule, “Clausal proofs for pseudo-Boolean reasoning,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, 2022.
- [18] J. Lind-Nielsen, *BuDDy: a Binary Decision Diagram Package*. Department of Information Technology, Technical University of Denmark, 1996.
- [19] R. M. Jensen, “A comparison study between the CUDD and BuDDy OBDD package applied to AI-planning problems,” Carnegie Mellon University, Tech. Rep. CMU-CS-02-173, September 2002.
- [20] R. Pohl, K. Lauenroth, and K. Pohl, “A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models,” in *International Conference on Automated Software Engineering (ASE)*, 2011, pp. 313–322.
- [21] T. van Dijk, E. M. Hahn, D. N. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini, and L. Zhang, “A comparative study of BDD packages for probabilistic symbolic model checking,” in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, ser. LNCS, vol. 9409, 2015, pp. 35–51.
- [22] K. S. Brace, R. L. Rudell, and R. E. Bryant, “Efficient implementation of a BDD package,” in *Design Automation Conference (DAC)*, June 1990, pp. 40–45.
- [23] S.-I. Minato, N. Ishiura, and S. Yajima, “Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation,” in *Design Automation Conference (DAC)*, June 1990, pp. 52–57.
- [24] F. Somenzi, “Efficient manipulation of decision diagrams,” *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 171–181, 2001.
- [25] R. L. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” in *International Conference on Computer-Aided Design (ICCAD)*, November 1993, pp. 139–144.
- [26] E. I. Goldberg and Y. Novikov, “Verification of proofs of unsatisfiability for CNF formulas,” in *Design, Automation and Test in Europe (DATE)*, 2003, pp. 886–891.
- [27] A. Van Gelder, “Producing and verifying extremely large propositional refutations,” *Annals of Mathematics and Artificial Intelligence*, vol. 65, no. 4, pp. 329–372, 2012.
- [28] M. J. H. Heule, W. A. Hunt, M. Kaufmann, and N. D. Wetzler, “Efficient, verified checking of propositional proofs,” in *Interactive Theorem Proving*, ser. LNCS, vol. 10499, 2017, pp. 269–284.
- [29] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, “cake_lpr: Verified propagation redundancy checking in CakeML,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Part II, ser. LNCS, vol. 12652, 2021, pp. 223–241.
- [30] S. Baek, M. Carneiro, and M. J. H. Heule, “A flexible proof format for SAT solver-elaborator communication,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Part I, ser. LNCS, vol. 12651, 2021, pp. 59–75.
- [31] T. Laitinen, T. Junttila, and I. Niemelä, “Extending clause learning SAT solvers with complete parity reasoning,” in *International Conference on Tools with Artificial Intelligence*, 2012, pp. 65–72.
- [32] I. S. Duff and J. K. Reid, “A comparison of sparsity orderings for obtaining a pivotal sequence in Gaussian elimination,” *IMA Journal of Applied Mathematics*, vol. 14, no. 3, pp. 281–291, 1974.
- [33] H. M. Markowitz, “The elimination form of the inverse and its application to linear programming,” *Management Science*, vol. 3, no. 3, pp. 213–284, 1957.
- [34] R. Dechter, “Bucket elimination: A unifying framework for reasoning,” *Artificial Intelligence*, vol. 113, no. 1–2, pp. 41–85, 1999.
- [35] J. Ellfers and J. Nordström, “Documentation of some combinatorial benchmarks,” in *Proceedings of the SAT Competition 2016*, 2016.
- [36] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020,” in *Proc. of SAT Competition 2020—Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.
- [37] M. Alekhovich, “Mutilated chessboard problem is exponentially hard for resolution,” *Theoretical Computer Science*, vol. 310, no. 1–3, pp. 513–525, Jan. 2004.
- [38] A. Haken, “The intractability of resolution,” *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985.
- [39] C. Sinz, “Towards an optimal CNF encoding of Boolean cardinality constraints,” in *Principles and Practice of Constraint Programming (CP)*, ser. LNCS, vol. 3709, 2005, pp. 827–831.
- [40] L. Chew and M. J. H. Heule, “Sorting parity encodings by reusing variables,” in *Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 12178, 2020, pp. 1–10.
- [41] A. Urquhart, “Hard examples for resolution,” *JACM*, vol. 34, no. 1, pp. 209–219, 1987.
- [42] C.-M. Li, “Equivalent literal propagation in the DLL procedure,” *Discrete Applied Mathematics*, vol. 130, no. 2, pp. 251–276, 2003.
- [43] R. E. Bryant and M. Soos, “Proof generation for CDCL solvers using Gauss-Jordan elimination,” 2022.

Stratified Certification for k -Induction

Emily Yu* 
zhengqi.yu@jku.at

Nils Froleyks* 
nils.froleyks@jku.at

Armin Biere† 
biere@cs.uni-freiburg.de

Keijo Heljanko‡§ 
keijo.heljanko@helsinki.fi

*Johannes Kepler University, Linz, Austria

†Albert–Ludwigs–University, Freiburg, Germany

‡Helsinki Institute for Information Technology and

§University of Helsinki, Helsinki, Finland

Abstract—Our recently proposed certification framework for bit-level k -induction-based model checking has been shown to be quite effective in increasing the trust of verification results even though it partially involved quantifier reasoning. In this paper we show how to simplify the approach by assuming reset functions to be stratified. This way it can be lifted to word-level and in principle to other theories where quantifier reasoning is difficult. Our new method requires six simple SAT checks and one polynomial-time check, allowing certification to remain in co-NP while the previous approach required five SAT checks and one QBF check. Experimental results show a substantial performance gain for our new approach. Finally we present and evaluate our new tool CERTIFAIGER-WL which is able to certify k -induction-based word-level model checking.

I. INTRODUCTION

Over the past several years, there has been growing interest in system verification using word-level reasoning. Satisfiability Modulo Theories (SMT) solvers for the theory of fixed-size bit-vectors are widely used for word-level reasoning [1], [2]. For example, word-level model checking has been an important part of the hardware model checking competitions since 2019. Given the theoretical and practical importance of word-level verification, a generic certification framework for it is necessary. As quantifiers in combination with bit-vectors are challenging for SMT solvers and various works have focused on eliminating quantifiers in SMT [2]–[4], a main goal of this paper is to generate certificates without quantification.

Temporal induction (also known as k -induction) [5] is a well-known model checking technique for verifying software and hardware systems. An attractive feature of k -induction is that it is natural to integrate it with modern SAT/SMT solvers, making it popular in both bit-level model checking and beyond [6]–[8], including word-level model checking.

Certification helps gaining confidence in model checking results, which is important for both safety- and business-critical applications. There have been several contributions focusing on generating proofs for SAT-based model checking [9]–[15]. For example [16] and [14] proposed an approach to certify LTL properties and a few preprocessing techniques by generating deductive proofs. In this paper, we focus on finding an inductive invariant for k -induction. Unlike other SAT/SMT-based techniques such as IC3 [17] and interpolation [18], [19], k -induction does not automatically generate an inductive

invariant that can be used as a certificate [20]. In previous research [21], certification of k -induction can be achieved via five SAT checks together with a one-alternation QBF check, redirecting the certification problem to verifying an inductive invariant in an extended model that simulates the original one.

At the heart of the present contribution is the idea of reducing the certification method of k -induction to pure SAT checks, *i.e.*, eliminating the quantifiers. This enables us to complete the certification procedure at a lower complexity, and to directly apply the framework to word-level certification. We introduce the notion of stratified simulation which allows us to reason about the simulation relation between two systems.

This stratified simulation relation can be verified by three SAT and a polynomial-time check. The latter checks whether the reset function is indeed stratified. In addition, we present a witness circuit construction which simulates the original under the stratified simulation relation thus creating a simpler and more elegant certification construction for k -induction.

While the previous work only focused on bit-level model checking, we also lift our method to word-level by implementing a complete toolsuite CERTIFAIGER-WL, where the experiments show the practicality and effectiveness of our certification method for word-level models.

II. BACKGROUND

This paper extends previous work in certification for k -induction-based bit-level model checking [21]. In this section, we present essential concepts and notations.

For the sake of simplicity we work with *functions* represented as interpreted terms and formulas over fixed but arbitrary theories which include an equality predicate. We further assume a finite sorted set of variables L where each variable $l \in L$ is associated with a finite domain of possible values. We also include Boolean variables as variables with a domain of $\{\top, \perp\}$, for which we keep standard notations.

For two sets of variables I and L , we also write I, L to denote their union. Given two functions $f(V), g(V')$ where $V \subseteq V'$ (represented as interpreted terms over our fixed but arbitrary theories) we call them *equivalent*, written $f(V) \equiv g(V')$, if for every assignment to variables in V and V' that matches on the shared set of variables V , the functions $f(V), g(V')$ have the same values. Additionally, we use “ \simeq ” for syntactic equivalence [22], “ \rightarrow ” for syntactic

Funded by FWF project W1255-N23, the LIT AI Lab funded by the State of Upper Austria, and Academy of Finland project 336092.

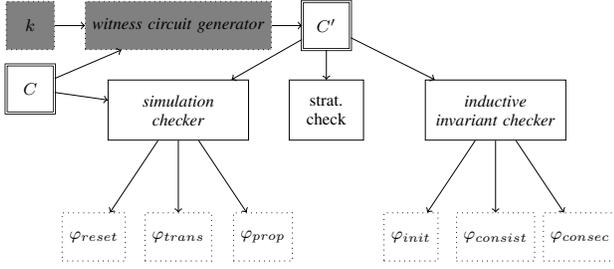


Fig. 1: An outline of the certification approach. Given some value of k and a model C , C' is the resulting witness circuit. The coloured area is specific to our approach for k -induction, and the rest corresponds to the general certification flow.

implication, and “ \Rightarrow ” for semantic implication. To define semantical concepts or abbreviations we stick to equality “ $=$ ”. We use $vars(f)$ to denote the set of variables occurring in the syntactic representation of a function f .

In word-level model checking operations are applied to fixed-size bit-vectors. We introduce the notion of word-level circuits where we model inputs and latches as finite-domain variables.

Definition 1 (Circuit). A circuit is a tuple $C = (I, L, R, F, P)$ such that:

- I is a finite set of input variables.
- L is a finite set of latch variables.
- $R = \{r_l(L) \mid l \in L\}$ is a set of reset functions.
- $F = \{f_l(I, L) \mid l \in L\}$ is a set of transition functions.
- $P(I, L)$ is a function that evaluates to a Boolean output, encoding the (good states) property.

By Def. 1 a circuit represents a hardware system in a fully symbolic form. In order to talk about the reset functions of a subset of latches $L'' \subseteq L$, we also write

$$R(L'') = \bigwedge_{l \in L''} (l \simeq r_l(L)).$$

The following four definitions are adapted from our previous work [21] for completeness of exposition.

Definition 2 (Unrolling). For an unrolling depth $m \in \mathbb{N}$, the unrolling of a circuit $C = (I, L, R, F, P)$ of length m is defined as $U_m = \bigwedge_{i \in [0, m]} (L_{i+1} \simeq F(I_i, L_i))$.

Definition 3 (Inductive invariant). Given a circuit C with a property P , $\phi(I, L)$ is an inductive invariant in C if and only if the following conditions hold:

- 1) $R(L) \Rightarrow \phi(I, L)$, “initiation”
- 2) $\phi(I, L) \Rightarrow P(I, L)$, and “consistency”
- 3) $U_1 \wedge \phi(I_0, L_0) \Rightarrow \phi(I_1, L_1)$. “consecution”

As a generalisation of the notion of an inductive invariant, k -induction checks k steps of unrolling instead of 1. In the following, to verify that a property is an inductive invariant, we consider it as the special case of k -induction with $k = 1$ and $\phi(I, L) = P(I, L)$.

Definition 4 (k -induction). Given a circuit C with a property P , P is called k -inductive in C if and only if the following two conditions hold:

- 1) $U_{k-1} \wedge R(L_0) \Rightarrow \bigwedge_{i \in [0, k)} P(I_i, L_i)$, and “BMC”
- 2) $U_k \wedge \bigwedge_{i \in [0, k)} P(I_i, L_i) \Rightarrow P(I_k, L_k)$. “consecution”

Definition 5 (Combinational extension).

A circuit $C' = (I', L', R', F', P')$ combinationaly extends a circuit $C = (I, L, R, F, P)$ if $I = I'$ and $L \subseteq L'$.

III. CERTIFICATION

In this section we introduce and formalise our certification approach which reduces the certification problem to six SAT checks and one polynomial stratification check.

The certification approach is outlined in Fig. 1. Intuitively, a *witness circuit* is generated from a given value of k (provided by the model checker) and a model (either bit-level or word-level). The witness circuit simulates the original circuit while allowing more behaviours (we formally define it as the *stratified simulation relation*). In practice, the witness circuit would be required to be provided by model checkers as the certificate in hardware model checking competitions.

We also perform a polynomial-time stratification check on the witness circuit. The check requires that the definition of the reset function is stratified, *i.e.*, no cyclic dependencies between the reset definitions of the variables exist. This is the case for all hardware model checking competition benchmarks. Even though cyclic definitions have been the subject of study in several papers [23]–[25], they are usually avoided due to the complexity of their analysis and subtle effects on semantics.

The approach in [21] can handle cyclic resets but at the cost of QBF quantification, and thus [21] not being able to be efficiently adapted to the context of word-level verification. Furthermore, the witness circuit includes an inductive invariant which serves as a proof certificate, which is verified by another three SAT checks as defined in Def. 3.

We begin by defining stratified reset functions.

Definition 6. (Dependency graph.) Given a set of latches L and a set of reset functions $R = \{r_l \mid l \in L\}$, the dependency graph G_R has latch variables L as nodes and contains a directed edge (a, b) from a to b iff $a \in vars(r_b)$ and $r_b \neq b$.

Latches with undefined reset value are common in applications. We simply set $r_b = b$ for some uninitialised latch b in such a case (as in AIGER and BTOR) to avoid being required to reason about ternary logic or partial functions. Thus the syntactic condition “ $r_b \neq b$ ” in the last definition simply avoids spurious self-loops in the dependency graph for latches with undefined reset values.

Definition 7. (Stratified resets.) Given a set of latches L , and a set of reset functions $R = \{r_l \mid l \in L\}$. R is said to be stratified iff G_R is acyclic.

TABLE I: Summary of certification results for the bit-level TIP suite.

Benchmarks	φ_{init}		$\varphi_{consist}$		φ_{consec}		φ_{trans}		φ_{prop}		φ_{reset}	
	t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2
c.periodic	7.78	0.06	0.06	0.06	56.82	56.29	0.15	0.14	0.05	0.05	84.04	0.00
n.guidance ₁	0.19	0.01	0.01	0.01	3.73	3.79	0.12	0.12	0.01	0.01	1.21	0.00
n.guidance ₇	4.09	0.02	0.02	0.02	18.40	18.17	0.12	0.12	0.02	0.02	25.22	0.00
n.tcas ₂	0.17	0.01	0.01	0.01	2.64	2.68	0.23	0.23	0.01	0.02	1.79	0.00
n.tcas ₃	0.11	0.01	0.01	0.01	1.82	1.70	0.23	0.26	0.02	0.02	1.01	0.00
v.prodcell ₁₂	2.35	0.03	0.03	0.03	59.05	59.22	0.12	0.12	0.03	0.03	8.48	0.00
v.prodcell ₁₃	0.22	0.01	0.01	0.01	2.99	2.99	0.12	0.12	0.01	0.01	0.20	0.00
v.prodcell ₁₄	0.64	0.02	0.02	0.02	13.69	13.69	0.12	0.12	0.02	0.02	1.45	0.00
v.prodcell ₁₅	2.22	0.02	0.03	0.03	32.66	32.28	0.12	0.12	0.02	0.02	2.26	0.00
v.prodcell ₁₆	0.01	0.01	0.01	0.01	1.19	1.20	0.12	0.12	0.01	0.01	0.06	0.00
v.prodcell ₁₇	2.34	0.03	0.03	0.03	48.51	48.17	0.12	0.12	0.03	0.03	6.86	0.00
v.prodcell ₁₈	0.67	0.01	0.01	0.01	8.67	8.78	0.12	0.12	0.02	0.02	0.79	0.00
v.prodcell ₁₉	1.66	0.02	0.02	0.03	31.98	31.78	0.12	0.12	0.03	0.03	3.73	0.00
v.prodcell ₂₄	3.32	0.04	0.04	0.04	112.12	115.18	0.12	0.12	0.04	0.04	17.64	0.00

Columns report the benchmark names, and the time (in seconds) used for each SAT check by CERTIFAIGER (t_1) and CERTIFAIGER++ (t_2) respectively.

Interestingly, the SAT solving time for the new reset check is close to zero, which checks the equality of the reset functions between the shared set of latches and the latches in the original circuit. This is because all latches in the benchmark set are initialized to \perp , thus making the SAT checks rather trivial.

Definition 8. (Stratified circuit.) A circuit $C = (I, L, R, F, P)$ is said to be stratified iff R is stratified.

The stratification check can be done in polynomial time using Def. 7 and it is enforced syntactically in the two hardware description formats AIGER and BTOR2.

Definition 9. (Stratified simulation.) Given two stratified circuits C and C' , where C' combinationally extends C . There is a stratified simulation between C' and C iff,

- 1) $r_l(L) \equiv r'_l(L')$ for $l \in L$, “reset”
- 2) $f_l(I, L) \equiv f'_l(I, L')$ for $l \in L$, and “transition”
- 3) $P'(I, L') \Rightarrow P(I, L)$. “property”

In essence, the crucial change here compared to the combinational simulation definition in [21] is the reset condition, whose simplification was possible under the stratification assumption. The above three conditions are encoded into SAT/SMT formulas ($\varphi_{reset}, \varphi_{trans}, \varphi_{prop}$ in Fig. 1) which are then checked by a solver for validity. In the rest of the paper, we simply refer to the stratified simulation relation as simulation relation. Proofs of the presented theoretical results can be found in an extended version of this paper [26].

Theorem 1. Given two circuits C and C' , where C' simulates C . If C' is safe, then C is also safe.

Next, we introduce the witness circuit construction. This is similar to the construction in [21] but differs in several details, e.g., the reset function definition is stratified and significantly simplified compared to [21].

Definition 10. (Witness circuit.) Given a circuit $C = (I, L, R, F, P)$ and an integer $k \in \mathbb{N}^+$, its witness circuit $C' = (I', L', R', F', P')$ is defined as follows:

- 1) $I' = I$ (also referred to as X^{k-1}),
- 2) $L' = L^{k-1} \cup \dots \cup L^0 \cup X^{k-2} \cup \dots \cup X^0 \cup B$ where,
 - $L^{k-1} = L$, the other variables sets are copies of I and L respectively with the same variable domains.
 - $B = \{b^{k-1}, \dots, b^0\}$ are Booleans.

3) R' :

- for $l \in L^{k-1}$, $r'_l = r_l(L^{k-1})$.
- for $l \in L^0 \cup \dots \cup L^{k-2} \cup X^0 \cup \dots \cup X^{k-2}$, $r'_l = l$.
- $r'_{b^{k-1}} = \top$.
- for $i \in [0, k-1)$, $r'_{b^i} = \perp$.

4) F' :

- for $l \in L^{k-1}$, $f'_l = f_l(I', L^{k-1})$.
- $f'_{b^{k-1}} = b^{k-1}$.
- for $i \in [0, k-1)$, $l^i \in (L^i \cup X^i \cup \{b^i\})$, $f'_{l^i} = l^{i+1}$.

5) $P' = \bigwedge_{i \in [0, 4]} p_i(I', L')$ where

- $p_0(I', L') = \bigwedge_{i \in [0, k-1)} (b^i \rightarrow b^{i+1})$.
- $p_1(I', L') = \bigwedge_{i \in [0, k-1)} (b^i \rightarrow (L^{i+1} \simeq F(X^i, L^i)))$.
- $p_2(I', L') = \bigwedge_{i \in [0, k)} (b^i \rightarrow P(X^i, L^i))$.
- $p_3(I', L') = \bigwedge_{i \in [1, k)} ((\neg b^{i-1} \wedge b^i) \rightarrow R(L^i))$.
- $p_4(I', L') = b^{k-1}$.

Here we extend a given circuit to a witness circuit, which has k copies of the original latches and inputs, and additional k latches of B that we refer to as the initialisation bits. We refer to the $\{k-1\}th$ as the most recent, and the 0th as the oldest. Intuitively the most recent copy unrolls in the same way as the original circuit, with the older copies copying the previous values of the younger copies. When all initialisation bits are \top , we say the machine has reached a “full initialisation” state.

Lemma 1. Given a circuit C with reset function R and its witness circuit C' with reset function R' . If R is stratified, then R' is also stratified.

Theorem 2. Given a circuit C and its witness circuit C' . C' simulates C .

We now present the main theorem of this paper.

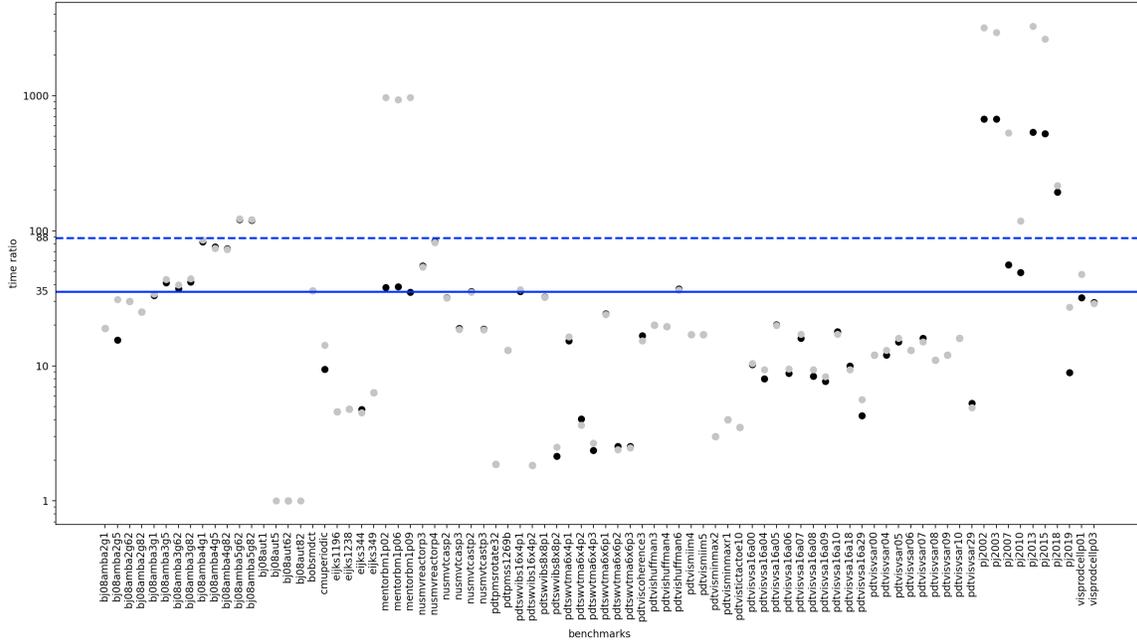


Fig. 2: Bit-level: the experimental results of the HWMCC 2010. The benchmark names are shown on the x-axis. The time ratio on the y-axis is calculated by computing certification time divided by model checking time (ran on the model checker McAiger [27]). The black dots in the graph are the results obtained from CERTIFAIGER++ and the grey dots are from CERTIFAIGER. The straight line and the dashed line are the calculated means for CERTIFAIGER++ and CERTIFAIGER respectively. As we can see from the plot, especially for the instances with certification time greater than 500 seconds, the new implementation significantly improved the certification performance.

Theorem 3. Given a circuit $C = (I, L, R, F, P)$ and its witness circuit $C' = (I', L', R', F', P')$. P is k -inductive in C iff P' is 1-inductive in C' .

IV. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented the proposed certification approach into two complete toolkits [28]: CERTIFAIGER++ for bit-level, and CERTIFAIGER-WL for word-level. We evaluate the performance of our tools against several benchmark sets from previous literature and the model checking competitions.

A. Bit-level

Our toolkit CERTIFAIGER++ extends the certification toolkit CERTIFAIGER [21]. Note that the AIGER format only allows stratified resets by default. All experiments were performed on a workstation with an Intel® Core™ i9-9900 CPU 3.60GHz computer with 32GB RAM running Manjaro with Linux kernel 5.4.72-1.

To determine the speedups of the new implementation proposed in this paper, we performed experiments on the same sets of the benchmarks used in [21]. The results are reported in Table I. There are significant overall gains in the initiation checks (φ_{init}) as well as the reset checks (φ_{reset}). For the initiation check which checks the invariant holds in all initial

states, the performance improvement is largely due to the simplification of the reset functions in the new witness circuit construction.

The results in Fig. 2 demonstrate that CERTIFAIGER++ in general is much faster than CERTIFAIGER during the overall certification process. Compared to CERTIFAIGER, CERTIFAIGER++ achieved overall speedups of 2.46 times. We observe performance gains in most benchmarks, as the previous performance bottleneck for certain benchmarks is the QBF solving time for the reset check. For other instances, the bottleneck is the SAT solving time for the consecution check, which is also improved due to a simpler reset construction (as part of the inductive invariant).

B. Word-level

We further lifted the method to certifying word-level model checking by implementing an experimental toolkit called CERTIFAIGER-WL. CERTIFAIGER-WL follows the same architecture design as CERTIFAIGER++ and uses Boolector [29] as the underlying SMT solver. All models and SMT encodings are in BTOR2 [29] format, which is the standard word-level model checking format used in hardware model checking competitions.

TABLE II: Summary of certification results word-level benchmarks from the HWMCC20

Benchmarks	k	#model	#witness	ModelCh.	Certifi.	Consec.	Ratio
paper_v3	256	35	12801	10.25	1.14	0.90	0.11
VexRiscv-regch0-15-p0	17	2149	43077	10.31	4.04	3.29	0.39
zipcpu-pfcache-p02	37	1818	105874	13.95	4.40	2.73	0.32
zipcpu-pfcache-p24	37	1818	105874	14.35	4.49	2.83	0.31
zipcpu-busdelay-p43	101	950	145466	15.29	6.14	3.86	0.40
dspfilters_fastfir_second-p42	15	6732	115388	16.11	14.80	12.96	0.92
zipcpu-pfcache-p01	41	1818	117434	18.33	6.34	4.47	0.35
dspfilters_fastfir_second-p10	11	6732	84348	24.56	9.76	8.44	0.40
zipcpu-busdelay-p15	101	950	145466	58.17	8.18	5.89	0.14
qspiflash_dualfexpress_divfive-p120	97	3100	394412	63.58	22.07	14.58	0.35
zipcpu-pfcache-p22	93	1818	267714	166.07	23.66	19.06	0.14
VexRiscv-regch0-20-p0	22	2149	55862	240.50	16.76	15.76	0.07
dspfilters_fastfir_second-p14	15	6732	115388	354.01	21.27	19.44	0.06
dspfilters_fastfir_second-p11	21	6732	161948	627.69	46.88	44.30	0.07
dspfilters_fastfir_second-p45	17	6732	130908	1094.11	30.14	28.06	0.03
VexRiscv-regch0-30-p1	32	2150	81464	1444.47	83.38	81.95	0.06
dspfilters_fastfir_second-p43	19	6732	146428	2813.61	58.02	55.69	0.02

To select the benchmarks presented, we first ran AVR with a timeout of 5000 seconds. We display the results here that are of particular interest with a running time of more than ten seconds (there are 7 instances with $k = 1$ which were certified and solved under 0.2s). Columns report the benchmark names, the value of k , the size of the model (measured in number of instructions) and the generated witness, the model checking time, and certification time (in seconds). Additionally we list the time Boolector took to solve the consecution check, as well as the ratio of model checking vs. certification time. We only list the consecution check (Consec.) here as it takes up the majority of the certification time.

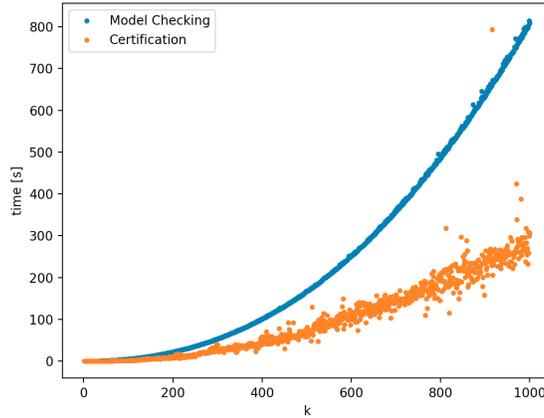


Fig. 3: Word-level: model checking vs. certification time for the Counter example (with 500 bits) with increasing values of k . For the experiments, we fixed the modulo bound at 32 and scaled the inductive depth up to 1000. The certification time is significantly smaller than the model checking time. As the value of k increases, on average the certification time is proportionally lower.

We ran benchmarks of the Counter example [21] on AVR [30] to get the values of k . Fig. 3 shows the experimental results obtained with CERTIFAIGER-WL under the same setting as Section IV-A. Interestingly, the certification time is much lower than the model checking time as can be seen in the diagram, meaning certification is at a low cost.

In Table II we report the experimental results obtained on a superset of the hardware model checking competition 2020 [31] benchmarks. We observe that the certification time is much lower than model checking time. Including certification would increase the runtime of AVR on the model checking benchmarks by less than 6%.

V. CONCLUSION AND FUTURE WORK

We have presented a new certification framework which allows certification for k -induction to be done by six SAT checks and a polynomial-time check. We further lifted our approach to word-level, and implemented our method in both contexts. Experimental results demonstrate the effectiveness and computational efficiency of our toolkits. The removal of the QBF quantifiers has reduced the theoretical complexity of the problem compared to [21] and also reduced the overall runtime overhead of the certification. Additionally, in future work we plan to obtain formally verified certificate checkers by using theorem proving. Finally, how to certify liveness properties is another important avenue of further research.

REFERENCES

- [1] A. Niemetz, M. Preiner, and A. Biere, “Precise and complete propagation based local search for satisfiability modulo theories,” in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9779. Springer, 2016, pp. 199–217. [Online]. Available: https://doi.org/10.1007/978-3-319-41528-4_11
- [2] —, “Propagation based local search for bit-precise reasoning,” *Formal Methods Syst. Des.*, vol. 51, no. 3, pp. 608–636, 2017. [Online]. Available: <https://doi.org/10.1007/s10703-017-0295-6>
- [3] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. W. Barrett, and C. Tinelli, “Towards satisfiability modulo parametric bit-vectors,” *J. Autom. Reason.*, vol. 65, no. 7, pp. 1001–1025, 2021.
- [4] A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli, “Solving quantified bit-vectors using invertibility conditions,” in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10982. Springer, 2018, pp. 236–255. [Online]. Available: https://doi.org/10.1007/978-3-319-96142-2_16
- [5] M. Sheeran, S. Singh, and G. Stålmarck, “Checking safety properties using induction and a SAT-solver,” in *FMCAD*, ser. Lecture Notes in Computer Science, vol. 1954. Springer, 2000, pp. 108–125.
- [6] A. Champion, A. Mebsout, C. Stickel, and C. Tinelli, “The Kind 2 model checker,” in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 9780. Springer, 2016, pp. 510–517.
- [7] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari, “SAL 2,” in *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, R. Alur and D. A. Peled, Eds., vol. 3114. Springer, 2004, pp. 496–500. [Online]. Available: https://doi.org/10.1007/978-3-540-27813-9_45
- [8] D. Jovanovic and B. Dutertre, “Property-directed k-induction,” in *FMCAD*. IEEE, 2016, pp. 85–92.
- [9] S. Conchon, A. Mebsout, and F. Zaïdi, “Certificates for parameterized model checking,” in *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, ser. Lecture Notes in Computer Science, N. Bjørner and F. S. de Boer, Eds., vol. 9109. Springer, 2015, pp. 126–142.
- [10] A. Gurfinkel and A. Ivrii, “K-induction without unrolling,” in *FMCAD*. IEEE, 2017, pp. 148–155.
- [11] T. Kuismin and K. Heljanko, “Increasing confidence in liveness model checking results with proofs,” in *HaiFa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 8244. Springer, 2013, pp. 32–43.
- [12] K. S. Namjoshi, “Certifying model checkers,” in *CAV*, ser. Lecture Notes in Computer Science, vol. 2102. Springer, 2001, pp. 2–13.
- [13] L. G. Wagner, A. Mebsout, C. Tinelli, D. D. Cofer, and K. Slind, “Qualification of a model checker for avionics software verification,” in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, ser. Lecture Notes in Computer Science, C. W. Barrett, M. Davies, and T. Kahsai, Eds., vol. 10227, 2017, pp. 404–419.
- [14] A. Griggio, M. Roveri, and S. Tonetta, “Certifying proofs for LTL model checking,” in *FMCAD*. IEEE, 2018, pp. 1–9.
- [15] Z. Yu, A. Biere, and K. Heljanko, “Certifying hardware model checking results,” in *ICFEM*, ser. Lecture Notes in Computer Science, vol. 11852. Springer, 2019, pp. 498–502.
- [16] A. Griggio, M. Roveri, and S. Tonetta, “Certifying proofs for SAT-based model checking,” *Formal Methods Syst. Des.*, vol. 57, no. 2, pp. 178–210, 2021.
- [17] A. R. Bradley, “SAT-based model checking without unrolling,” in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 6538. Springer, 2011, pp. 70–87.
- [18] K. L. McMillan, “Interpolation and SAT-based model checking,” in *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13. [Online]. Available: https://doi.org/10.1007/978-3-540-45069-6_1
- [19] —, “An interpolating theorem prover,” *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 101–121, 2005. [Online]. Available: <https://doi.org/10.1016/j.tcs.2005.07.003>
- [20] Z. Manna and A. Pnueli, *Temporal verification of reactive systems - safety*. Springer, 1995.
- [21] E. Yu, A. Biere, and K. Heljanko, “Progress in certifying hardware model checking results,” in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 12760. Springer, 2021, pp. 363–386.
- [22] A. Degtyarev and A. Voronkov, “Equality reasoning in sequent-based calculi,” in *Handbook of Automated Reasoning (in 2 volumes)*, J. A. Robinson and A. Voronkov, Eds. Elsevier and MIT Press, 2001, pp. 611–706.
- [23] S. Malik, “Analysis of cyclic combinational circuits,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 13, no. 7, pp. 950–956, 1994. [Online]. Available: <https://doi.org/10.1109/43.293952>
- [24] J. R. Jiang, A. Mishchenko, and R. K. Brayton, “On breakable cyclic definitions,” in *2004 International Conference on Computer-Aided Design, ICCAD 2004, San Jose, CA, USA, November 7-11, 2004*. IEEE Computer Society / ACM, 2004, pp. 411–418. [Online]. Available: <https://doi.org/10.1109/ICCAD.2004.1382610>
- [25] M. D. Riedel, *Cyclic combinational circuits*. California Institute of Technology, 2004.
- [26] E. Yu, N. Froleyks, A. Biere, and K. Heljanko, “Stratified certification for k-induction,” 2022. [Online]. Available: <https://arxiv.org/abs/2208.01443>
- [27] A. Biere and R. Brummayer, “Consistency checking of all different constraints over bit-vectors within a SAT solver,” in *FMCAD*. IEEE, 2008, pp. 1–4.
- [28] Certifaiger++ and Certifaiger-wl, “Certifaiger++ and Certifaiger-wl,” 2022, <http://fmv.jku.at/certifaiger>.
- [29] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, “Btor2 , btormc and boolector 3.0,” in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 587–595. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_32
- [30] A. Goel and K. A. Sakallah, “AVR: abstractly verifying reachability,” in *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Biere and D. Parker, Eds., vol. 12078. Springer, 2020, pp. 413–422. [Online]. Available: https://doi.org/10.1007/978-3-030-45190-5_23
- [31] M. Preiner, A. Biere, and N. Froleyks, “Hardware model checking competition 2020,” 2020, <http://fmv.jku.at/hwmcc20/>.

Reconstructing Fine-Grained Proofs of Rewrites Using a Domain-Specific Language

Andres Nötzli*^{id}, Haniel Barbosa[†]^{id}, Aina Niemetz*^{id}, Mathias Preiner*^{id},
Andrew Reynolds[†]^{id}, Clark Barrett*^{id}, and Cesare Tinelli[†]^{id}

*Stanford University, Stanford, USA, ✉{noetzli, niemetz, preiner, barrett}@cs.stanford.edu

[†]The University of Iowa, Iowa City, USA, ✉{andrew-reynolds, cesare-tinelli}@uiowa.edu

[‡]Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, ✉hbarbosa@dcc.ufmg.br

Abstract—Satisfiability modulo theories (SMT) solvers are widely used to prove security and safety properties of computer systems. For these applications, it is crucial that the result reported by an SMT solver be correct. Recently, there has been a renewed focus on producing independently checkable proofs in SMT solvers, partly with the aim of addressing this risk. These proofs record the reasoning done by an SMT solver and are ideally detailed enough to be easy to check. At the same time, modern SMT solvers typically implement hundreds of different term-rewriting rules in order to achieve state-of-the-art performance. Generating detailed proofs for applications of these rules is a challenge, because code implementing rewrite rules can be large and complex. Instrumenting this code to additionally produce proofs makes it even more complex and makes it harder to add new rewrite rules. We propose an alternative approach to the direct instrumentation of the rewriting module of an SMT solver. The approach uses a domain-specific language (DSL) to describe a set of rewrite rules declaratively and then reconstructs detailed proofs for specific rewrite steps on demand based on those declarative descriptions.

I. INTRODUCTION

Satisfiability modulo theories (SMT) solvers are widely used to reason about the security and safety of critical systems [1, 2, 10, 13]. These applications require a high level of trust in the correctness of the underlying solver. SMT solvers, however, are complex pieces of software, in some cases consisting of hundreds of thousands of lines of code. As with any other large and complex software project, they are not immune to bugs [17], which may, in the worst case, cause incorrect results. Due to the size and complexity of SMT solvers and the fact that most of them continue to be in active development, their full verification is currently still out of reach. As a consequence, the best one can do is to check their individual answers based on evidence provided by the solvers themselves.

For quantifier-free inputs reported to be satisfiable, SMT solvers are typically capable of producing as evidence a satisfying model, which can then be used to validate the claim. Note that for quantified formulas, model validation for satisfiable queries is usually still possible although more complex. For unsatisfiable inputs, there have been efforts in recent years

towards producing independently checkable *proofs*, which record the reasoning steps required to deduce unsatisfiability. These steps can later be replayed and checked efficiently by a *proof checker*. Proofs can also be used to automatically discharge proof obligations in interactive theorem provers such as Coq [25] and Isabelle [19]. For this use case, the SMT solver acts as an automated tactic. The proof obligation is encoded as an SMT problem and the proof generated by the SMT solver is then used, in essence, to reconstruct a proof in the proof assistant’s native proof representation.

Producing and checking proofs for unsatisfiable problems requires considerably more effort than generating and validating models for satisfiable inputs. Additionally, proofs can be produced in many different forms, each with its own trade-offs. When it comes to the form of a proof, one characteristic of interest is the proof’s *granularity*. *Fine-grained* proofs enable efficient proof checking since the proofs are detailed enough to not require any search during checking. Similarly, proof reconstruction for interactive theorem provers requires detailed proofs to minimize *holes* that must be proved manually. However, fine-grained proofs are generally more costly to produce. *Coarse-grained* proofs, on the other hand, are cheaper to produce but require more computation to check. Regardless of the proof form, the traditional approach for generating proofs is to instrument each component of the SMT solver to record its reasoning steps, and then consolidate the relevant recorded steps into a single proof.

Instrumentation can be particularly challenging and tedious for the components of the solver that implement *rewriting*. Modern SMT solvers implement hundreds of rewrite rules for normalizing and simplifying terms to achieve state-of-the-art performance. Because rewriting is an essential part of the reasoning done by the solver, a proof must contain a record of the rewriting steps performed. Previous work [6] has described how to generate rewriting proofs whose only holes are *atomic rewrites*, i.e., an application of a single rewrite step to a single term. Such proofs use a single generic rule for all atomic rewrites. This approach has two major drawbacks, however: (i) the proof checker has to guess or search for the rule to apply or trust that the rewriting was done correctly; and (ii) if used in a proof assistant, each rewrite step becomes a proof obligation that must be discharged by the user. On the other

This work was supported in part by DARPA (award no. FA8650-18-2-7861), the Stanford Agile Hardware Center, and by a gift from Amazon Web Services.

hand, if occurrences of atomic rewrites are proven using a fixed set of specific rules, we can prove the correctness of the rules in this set once and for all and then use those proofs during proof checking or during replay in a proof assistant.

As mentioned above, instrumenting rewriting code for proof generation is difficult and tedious. Additionally, since rewriting is applied not only as a preprocessing step but also repeatedly during the solving process, rewriting code (including any instrumentation) must be efficient. In this work, *we propose an alternative approach that does not rely on instrumenting the original rewriter*. Instead, our approach treats the rewriter as a black box and relies on a post-processing phase to expand coarse-grained rewriting steps occurring in proofs into fine-grained proofs. We use a generic reconstruction algorithm that consults a separate database of core rewrite rules in order to produce the detailed proof using as input only the terms before and after an atomic rewrite. The core rewrite rules need not include every atomic rewrite. It is enough for every atomic rewrite to be reconstructable using one or more of the core rewrite rules. This simplifies the task of populating the database, as the rules used can be fewer and simpler than what is actually done in the solver. To specify the set of rules in the database, we propose the use of a high-level, domain-specific language (DSL) designed to succinctly express a set of core rules to be used in proofs. We have used this approach to reconstruct detailed proofs for the theory of strings in the SMT solver *CVC5* [4]. In our experience, this approach greatly reduces the burden of proof production for rewriting code, as it allows a solver developer to quickly and incrementally define core rewrite rules to help fill holes in proofs. Also, note that rewrite steps are typically equality-preserving. Because we treat the rewriter as a black box (i.e., independently from any specific solver or implementation), our approach is quite general and could be used to produce or complete proofs for any tool or situation where proofs of equivalence are needed. By providing a DSL for specifying rewrites and an automatic reconstruction algorithm for coarse-grained atomic rewrites, we expect to greatly improve the flexibility and usability of proofs from SMT solvers. Our contributions are as follows:

- We propose an SMT-LIB-like domain specific language for defining rewrite rules.
- We describe an algorithm that can use such rules to reconstruct detailed proofs for rewrites in an SMT solver.
- We implement our approach in *CVC5* and report on a case study reconstructing detailed proofs for rewrites in the theory of strings.
- We evaluate our implementation and show that it has reasonable performance in practice.

In the remainder of the paper, we provide an overview of our approach (Section II) and then describe the language (Section III) and the proof reconstruction algorithm (Section IV) in more detail. We then present a case study of using the approach to produce detailed proofs for the theory of strings in *CVC5* (Section V) and evaluate our approach (Section VI) experimentally. Finally, we conclude with some future direc-

tions for the language and our approach (Section VII).

A. Related Work

Barbosa et al. [5] introduced a framework for modularizing the production of proofs for formula processing and term rewriting, a long-standing challenge for SMT solvers. A similar and more general framework for overall proof production [6] was recently implemented in *CVC5*. However, both frameworks produce proofs that are coarse-grained with respect to atomic rewrites, i.e., each atomic rewriting step is a single proof step without further justification.

In the integration between the *veriT* solver [11] and the Isabelle/HOL proof assistant [23], which leverages the framework from [5], the Sledgehammer tool [8] sends proof goals to *veriT* and then reconstructs proofs from those emitted by *veriT* in the Alethe proof format [22]. The reconstructed proofs can then be used to prove the original Isabelle/HOL proof goals. An initial version of this framework was similarly coarse-grained: every atomic rewrite applied by the solver was justified with a single Alethe proof rule. As shown by Schurr et al. [23], this led to failures and performance issues in the Isabelle/HOL reconstruction of Alethe proofs. One approach to address this issue is to extend the Alethe format to contain finer-grained rules for atomic rewrites, and to integrate each of these rules into both *veriT* and Sledgehammer. This has been shown to increase the success rate of proof reconstruction, but the process is fully manual: every new rule added requires updating the solver, the format, and the reconstruction.

Nötzli [20] proposed a language for rewrite rules in SMT solvers with the goal of automatically generating executable code that replaces parts of an existing rewriter. The DSL presented in this work is an evolution of that language and is focused on the needs of proof reconstruction. Our dedicated rewrite language bears some similarity to equational specification languages such as Maude [12], ELAN [9], and CafeOBJ [14]. In contrast to those more general-purpose languages, the DSL presented in this work has a much more narrow scope and includes specific features to support its use in proof reconstruction.

B. Formal Preliminaries

We formalize our work within the setting of many-sorted logic with equality (see e.g., [15, 26]). Let S be a set of *sort symbols*. For every sort $\tau \in S$, we assume an infinite set of variables of that sort. A *signature* Σ consists of a set $\Sigma^s \subseteq S$ of sort symbols and a set Σ^f of function symbols. Constants are treated as 0-ary functions. We assume that Σ includes a sort *Bool*, interpreted as the Boolean domain, and the *Bool* constants \top (*true*) and \perp (*false*). Signatures do not contain separate predicate symbols and use instead function symbols that return a *Bool* value. We further assume that for all sorts $\tau \in S$, Σ contains an equality symbol $\approx: \tau \times \tau \rightarrow \text{Bool}$, interpreted as the identity relation. Finally, we assume the usual definitions of well-sorted terms, literals, and formulas.

A Σ -*interpretation* I maps: each $\tau \in \Sigma^s$ to a distinct non-empty set of values τ^I (the *domain* of τ in I); each variable

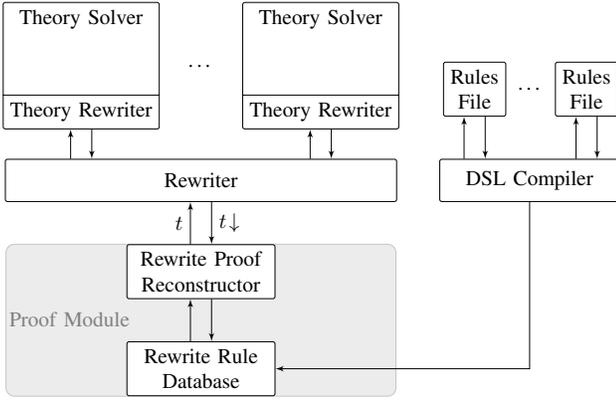


Fig. 1: Overview of the components of our approach

x of sort τ to an element $x^I \in \tau^I$; and each $f^{\tau_1 \dots \tau_n \tau} \in \Sigma^f$ to a total function $f^I : \tau_1^I \times \dots \times \tau_n^I \rightarrow \tau^I$ if $n > 0$, and to an element in τ^I if $n = 0$. We use the usual notion of a satisfiability relation \models between Σ -interpretations and Σ -formulas. A Σ -theory T is a non-empty class of Σ -interpretations closed under variable reassignment (i.e., every interpretation that only disagrees with an interpretation in T on how it interprets variables is also in T). A Σ -formula φ is T -satisfiable (resp., T -unsatisfiable, T -valid) if it is satisfied by some (resp., no, all) interpretations in T . We write $\models_T \varphi$ when φ is T -valid. We say that φ_1 T -entails φ_2 , and write $\varphi_1 \models_T \varphi_2$, when $\models_T \varphi_1 \Rightarrow \varphi_2$.

II. OVERVIEW

In this paper, we assume a fixed theory T and consider only rewrite rules that preserve equivalence in T . Formally, let $t \downarrow_a$ denote the result of performing atomic rewrite a on term t . Then, we require that $\models_T t \approx t \downarrow_a$.

Figure 1 shows an overview of our proposed approach. Modern SMT solvers implement a large number of theory-specific rewrite rules. Conceptually, the implementation of these theory-specific rewrite rules can be seen as *theory rewriter* modules of the individual theory solvers. A *rewriter* is a module that traverses a given term and invokes the appropriate theory rewriter on each subterm. To determine which theory rewriter to call, the rewriter looks at the top-most symbol of the subterm and calls the theory whose signature contains that symbol. The *proof module*, which manages proofs, utilizes the *rewrite proof reconstructor* to fill in the missing subproofs for rewrites. The rewrite proof reconstructor bases its reconstruction on a set of rewrite rules, stored in the *rewrite rule database*. This database is generated at compile-time from a set of rewrite rules written in our DSL RARE (described in Section III). These rewrite rules are stored in text files, which are compiled to C++ code using the DSL compiler. The compiled code populates a discrimination tree [16] which is an index used for matching terms with applicable rewrite rules during proof reconstruction. Assuming that the rewrite rules in the rewrite rule database are correct, our reconstruction is sound since only these rules are used to construct the proofs.

```

(rule) ::= ( define-rule <symbol> ( <par>* )
           <expr> <expr> )
        | ( define-cond-rule <symbol> ( <par>* )
           <expr> <expr> <expr> )
        | ( define-rule* <symbol> ( <par>* )
           <expr> <expr> [ <expr> ] )

<par> ::= <symbol> <sort> <attr>*

<sort> ::= ? | <symbol> | ( <symbol> <sort>+ )
         | ( _ <symbol> <idx>+ )

<idx> ::= ? | <numeral>

<expr> ::= <const> | <id> | ( <id> <expr>+ ) | <let>

<id> ::= <symbol> | ( _ <symbol> <idx>+ )

<let> ::= ( let ( <binding>+ ) <expr> )

<binding> ::= ( <symbol> <expr> )

```

Fig. 2: Overview of the grammar of RARE.

The output of the proof module consists of the proof with the subproofs for rewrites completed.

The rule database may also play a role in proof *checking*. In particular, a stand-alone proof checker may use the database to automatically generate code that can check whether a rule in the database is used correctly. While the syntax is checked in this scheme, the T -validity of the rules in the database is trusted. Checking the rules for T -validity is another task which can (and should) be done separately, perhaps using a proof assistant. We do not address these issues in this paper, but instead focus on the RARE language and the algorithm at the core of the rewrite proof reconstructor.

III. THE LANGUAGE

In this section, we describe the scope, design goals, syntax, and semantics of RARE, our domain-specific language for rewrites, automatically reconstructed. To reduce the cost of introducing such a new language into the development workflow of an existing SMT solver, we identify several requirements:

Succinctness: Writing rewrite rules should be simple and concise. Adding new rules should be far less costly than instrumenting existing code.

Expressiveness: The language should be able to express the majority of the rewrite rules used in a state-of-the-art SMT solver.

Accessibility: The language should be easy to parse and understand.

There is an inherent tension between making a DSL succinct and making it expressive. We designed RARE to be as expressive as possible without sacrificing succinctness. To aid with accessibility, its syntax reuses the syntax of the SMT-LIB [7] language standard whenever possible.

As we discuss in Section V, we do not aim for full generality, because certain rewrites, such as polynomial normalization, are less amenable to our approach. Similarly, we assume that constant folding is built into the reconstruction algorithm and therefore does not have to be explicitly defined with rewrite rules.

An input file for RARE consists of a list of rewrite rules whose syntax is defined by the BNF grammar in Figure 2. Rewrite rules are written as S-expressions. For symbols and concrete constants (e.g., integer numbers, string literals), RARE uses the same syntax as the SMT-LIB language. In contrast to SMT-LIB, parameterized sorts such as arrays and bit-vectors do not need to be concrete. Instead, RARE is gradually typed and allows the parameters of such sorts to remain abstract. This allows users to specify rewrites that are, e.g., independent of the bit-width or the sorts of indices and elements in arrays. In the following, we discuss all the different constructs of the language in detail.

Basic Syntax. As indicated in Figure 2, $\langle rule \rangle$ defines three different types of rewrite rules: basic rules (`define-rule`), conditional rules (`define-cond-rule`), and fixed-point rules (`define-rule*`). A *basic rule* consists of a *name*, a list of match *parameters*, the *match* expression, and the *target* expression. The name identifies the rewrite rule and is later used to label steps in the rewrite proof; the list of parameters $\langle par \rangle^*$ introduces the term variables that appear in the rule, along with their sorts; the match expression defines the syntactic shape of terms the rewrite rule applies to; and the target expression defines how a matched term is rewritten. Both the match expression and the target expression have the same syntax as SMT-LIB terms. All the variables that appear in a rewrite rule must either be declared as a parameter or introduced locally with the `let` binder.

Basic rules define simple rewrite rules without preconditions. The following example shows such a rule, which defines the rewrite $\text{substr}("", m, n) \rightsquigarrow ""$ from a term denoting the substring from position m to position n of the empty string to just the empty string, regardless of the value of m and n .

```
(define-rule substr-empty ((m Int) (n Int))
  (str.substr "" m n) "")
```

In this example, the match expression specifies that the rule applies to string terms of the form $\text{substr}("", s, t)$ where the first argument of `substr` is the empty string, the second argument s is matched by m , and the third argument t is matched by n . The compiler and the proof reconstruction algorithm have built-in knowledge of theory symbols such as `substr` as defined in the SMT-LIB standard.

Matching. If a variable x appears multiple times in a match expression, the rewrite rule only applies if each occurrence of x matches syntactically identical terms. For example, the match expression $(= (\text{str}.\text{++ } x1 x2) x2)$ with variables $x1$ and $x2$ matches $a \text{++ } b \approx b$, but not $a \text{++ } b \approx c$. For a rewrite rule to apply, a term matched by a declared variable must be of the expected sort. We use `?` to denote that a term

can be of any sort, or to match an arbitrary sort parameter. The following example illustrates the use of multiple variable occurrences and abstract sorts.

```
(define-rule eq-refl ((t ?)) (= t t) true)
```

This rule rewrites equalities of syntactically equivalent terms to \top , regardless of the sort of the term matched by variable t .

Lists. Some operators defined in SMT-LIB, e.g., string concatenation, can be applied to two or more terms. We use variables declared with the `:list` attribute to match an arbitrary number of arguments of an operator. The following example shows a rule for flattening string concatenations.

```
(define-rule str-concat-flatten (
  (xs String :list) (s String)
  (ys String :list) (zs String :list))
  (str.++ xs (str.++ s ys) zs) ; match
  (str.++ xs s ys zs) ; target)
```

This rule applies to any string concatenation with another string concatenation as a subterm. The prefix xs and the suffix zs may be empty (although not at the same time in this case).

Conditional Rules. The previous rewrite rule examples rely on purely syntactic matching. To make matching more expressive, RARE supports the conditional matching of terms using `define-cond-rule`. Such rules have an additional argument, the *precondition*, before the match expression. That is either a single condition, expressed by a literal, or a conjunction of them capturing all conditions that must be met for the rule to apply. When reconstructing a proof, these conditions introduce new proof obligations. The following example illustrates the use of conditional rules.

```
(define-cond-rule concat-clash (
  (s1 String) (s2 String :list)
  (t1 String) (t2 String :list))
  (and (= (str.len s1) (str.len t1))
        (not (= s1 t1)))
  (= (str.++ s1 s2) (str.++ t1 t2))
  false)
```

This rule rewrites a word equation $s_1 \text{++ } s_2 \approx t_1 \text{++ } t_2$ to \perp , provided that two conditions are met: the lengths of the prefixes s_1 and t_1 are the same and the prefixes are distinct in the theory T . For example, this rule applies to the equality $\text{"abc"} \text{++ } x \approx \text{"def"} \text{++ } y$ since both $|\text{"abc"}| \approx |\text{"def"}| \approx 3$ and $\text{"abc"} \not\approx \text{"def"}$ hold in the theory of strings. Note that the precondition $|s_1| \approx |t_1|$ does not require the evaluation of $|s_1|$ and $|t_1|$. Instead, it just requires some proof that they are equal. In practice, we prove the precondition by applying additional rewrite rules. This allows us to show that the precondition holds for equalities such as $|x \text{++ } y| \approx |y \text{++ } x|$, for instance.

Fixed-Point Rules. As an optimization, RARE allows the definition of fixed-point rules with `define-rule*`. These rules are repeatedly applied until they no longer apply. They are most useful for rewrite rules that effectively iterate over arguments of n-ary operators, as we demonstrate in the example below. Fixed-point rules take a match expression, a target

```

rc( $t \approx s, d$ )
1: if  $d < 0$  then return  $\perp$ 
2: if  $t \approx s \in \mathcal{P}$  then
3:   if  $\mathcal{P}[t \approx s] = (\text{fail}, e)$  and  $d \leq e$  then return  $\perp$ 
4:   if  $\mathcal{P}[t \approx s] \neq (\text{fail}, e)$  then return  $\top$ 
5: if  $t \downarrow_e = s \downarrow_e$  then  $\mathcal{P}[t \approx s] := \text{eval}$ , return  $\top$ 
6: if  $(t \approx s) \downarrow = \perp$  then  $\mathcal{P}[t \approx s] := (\text{fail}, \infty)$ , return  $\perp$ 
7:  $\mathcal{P}[t \approx s] := (\text{fail}, d)$ 
8: if  $(t, s) = (f(\vec{u}), f(\vec{v}))$  and
9:    $\text{rc}(u \approx v, d)$  for all  $u \approx v \in \vec{u} \approx \vec{v}$  then
10:   $\mathcal{P}[t \approx s] := \text{cong}$  return  $\top$ 
11: if  $t = f(\vec{u})$  and  $\vec{u} \downarrow = \vec{c}$  and  $f(\vec{c}) \downarrow_e = s \downarrow_e$  and
12:   $\text{rc}(\vec{u} \approx \vec{c}, d)$  then
13:   $\mathcal{P}[t \approx s] := \text{ceval}$  return  $\top$ 
14: foreach  $(r, \vec{p} \approx \vec{q} \Rightarrow u \approx v) \in \mathcal{R}$ 
15:   s.t.  $t = \sigma(u)$  for some  $\sigma$  do
16:   if  $\text{rc}(\sigma(v) \approx s, d - 1)$  and
17:      $\text{rc}(\sigma(p \approx q), d - 1)$  for all  $p \approx q \in \vec{p} \approx \vec{q}$  then
18:      $\mathcal{P}[t \approx s] := r$ , return  $\top$ 
19: return  $\perp$ 

```

Fig. 3: The algorithm for reconstructing a proof sketch \mathcal{P} from rule database \mathcal{R} . Calling $\text{rc}(t \approx s, d)$ returns true if the proof of $t \approx s$ having depth at most d can be constructed.

expression, and, optionally, a context expression as arguments. The *target* expression indicates the recursion step, i.e., the term that should be rewritten next. The *context* expression indicates how to use the result of the recursion step to construct the final result. It is a term with a placeholder $_$ for the location of the result of the recursion step. Omitting the context expression is the same as providing a context of $_$, which indicates that the result of the recursion step is also the final result. The following example defines a rewrite rule that distributes the string length operator over the elements in a concatenation:

```

(define-rule* str-len-concat-rec (
  (s1 String) (s2 String)
  (rest String :list))
  (str.len (str.++ s1 s2 rest))
  (str.len (str.++ s2 rest))
  (+ (str.len s1) _))

```

This rule specifies that we rewrite $|s_1 ++ s_2 ++ \dots|$ to $|s_1| + t$, where t is the result of recursively applying the rule to the term $|s_2 ++ \dots|$.

Annotating rules to be fixed-point rules reduces the search space during reconstruction, because the reconstruction algorithm always applies these rules until a fixed-point is reached, without considering possible interleavings of other rules. This improves efficiency at the cost of not considering some possible reconstructions. Thus, there is a trade-off, and this feature must be used carefully.

IV. RECONSTRUCTING PROOFS

In this section, we describe our approach for constructing proofs of rewrites $t \approx t \downarrow_a$ using rules from a *rewrite rule*

$$\begin{array}{c}
\text{eval} \frac{}{t \approx t \downarrow_e} \quad \text{trans} \frac{r \approx s \quad s \approx t}{r \approx t} \\
\text{cong} \frac{\vec{s} \approx \vec{t}}{f(\vec{s}) \approx f(\vec{t})} \quad \text{ceval} \frac{\vec{s} \downarrow \approx \vec{t} \downarrow}{f(\vec{s}) \approx (f(\vec{t})) \downarrow_e}
\end{array}$$

Fig. 4: The basic proof rules; $t \downarrow_e$ is the evaluated form of t .

database \mathcal{R} obtained by compiling RARE rules. To simplify the presentation, we do not consider fixed-point rules for now, postponing the general case to later in this section. The database \mathcal{R} stores a set of labeled implications of the form $(r, \vec{p} \approx \vec{q} \Rightarrow t \approx s)$, where r is a rule identifier, $\vec{p} \approx \vec{q}$ is a conjunction of term equalities, and $\vec{p} \approx \vec{q} \models_T t \approx s$. Operationally, the rule specifies that a term t can be rewritten to a term s when the premises $\vec{p} \approx \vec{q}$ hold. Note that using just equalities in the premises is without loss of generality since an arbitrary formula φ can be expressed as a premise of the form $\varphi \approx \top$. Unconditional rules are represented using the single, valid premise $\top \approx \top$.

Our proof reconstruction for an equality $t \approx t \downarrow_a$ based on the rule database \mathcal{R} consists of two phases. In the first one, captured by the algorithm in Figure 3, we search for a *proof sketch* \mathcal{P} , which is a map from term equalities to rules that can be used to prove them in a final proof. In the second, the discovered proof sketch, if any, is transformed into a full proof, which may consist of the application of multiple rules from \mathcal{R} , as described later in this section.

A. Finding Proof Sketches

Figure 3 shows our algorithm rc for recursively finding proof sketches for equalities $t \approx s$. The inputs are the (oriented) equality to prove and an integer d specifying an upper bound on the *depth* of rc 's recursive calls. Some recursive calls are generated by the algorithm's attempt to justify the use of a conditional rule from \mathcal{R} to prove the input equality. In that case, the algorithm attempts to prove the premises of the conditional rule, but does so for a decreased depth. The rationale behind the depth limit on the search is that there is no guarantee that preconditions are simpler than the current equality to be proved, and so there is no guarantee of termination in general. The depth limit can be chosen by the user at runtime to maximize the chances of successfully reconstructing a proof for a rewrite while minimizing the amount of work spent on unsuccessful parts of the search space. Note that d is decremented only in recursive calls over the premises of conditional rules. For other recursive calls, which are over subterms of the input equality, termination is ensured by the reduction in the size of the new input equality.

The algorithm returns \top if it finds a proof sketch for $t \approx s$ within the given depth restriction d . During its search, it updates a (global) proof sketch map \mathcal{P} from term equalities to rules r that can be used in the final proof, or to pairs (fail, e) indicating that no proof for that equality can be found within

depth e . We use the array-like notation $P[t \approx s]$ to refer to the value that P associates with $t \approx s$. A few of the rewrite rules stored in P are built-in, the rest are from the database \mathcal{R} . The built-in rules are provided in Figure 4 in the style of inference rules. Note that trans is actually not used for proof sketches, but only for the construction of final proofs.

Going through the algorithm line by line, we see that it first returns \perp if the given depth d is negative. Then, on line 2, it checks if a proof sketch for $t \approx s$ has already been determined. If so and the value was (fail, e) , then no proof was found for $t \approx s$ using depth e . If e is at least d , then it is impossible to construct a proof with depth d , and \perp is returned to indicate failure. On the other hand, if a proof already exists, then \top is returned, indicating success.

If none of these quick-return cases hold, the algorithm tries to prove the equality using several techniques, which we informally call proof *tactics*. First, the algorithm checks if the equality can be quickly (dis)proven. Specifically, on line 5 the simplest tactic checks whether the equality can be proven by *evaluation*, and returns \top if so. We write $t \downarrow_e$ to denote the *evaluated* form of t , typically a concrete constant c equivalent to t , if one can be determined by recursively evaluating (i.e., constant-folding) subterms of t , or t itself otherwise. If the evaluated form of t and s are the same, the algorithm stores in P the information that $t \approx s$ can be proven by evaluation, denoted by built-in rule eval . This case applies for instance to simple equalities such as $1 + 3 \approx 2 + 2$. On line 6, the global rewriter of an SMT solver (denoted as \downarrow) is used as an oracle to check whether the current equality can be rewritten to \perp , which means that the search for a proof sketch is futile. In this case, failure is stored as (fail, ∞) , indicating that a proof for $t \approx s$ cannot exist because $\models_T t \not\approx s$. This is a fast albeit incomplete check which is useful when the input $t \approx s$ is a precondition of some other rule. If that check fails, the search continues because the global rewriter is incomplete, and thus a proof for $t \approx s$ may still exist. On line 7, $t \approx s$ is tentatively marked in P as (fail, d) , but then an attempt is made to prove $t \approx s$ using the remaining proof tactics. The equality is marked as a failure *before* running these tactics to avoid infinite recursion when $t \approx s$ happens to be a premise in some recursive call.

Line 9 gives our tactic for proving the given equality by congruence, which we associate with a proof rule cong . If t and s have the same top symbol f and our reconstruction algorithm succeeds in proving equalities pairwise for each of their arguments $\vec{u} \approx \vec{v}$, we mark $t \approx s$ as proven and return \top . Line 12 gives our tactic for congruence plus evaluation, which we associate with a proof rule ceval . This tactic uses the global rewriter again as an oracle to check whether all the arguments \vec{u} of t can be rewritten to some constant values \vec{c} , i.e., whether $\vec{u} \downarrow = \vec{c}$. If additionally the evaluation of the top symbol f on \vec{c} is equal to the evaluation of s , then the algorithm tries to construct a proof for equalities $\vec{u} \approx \vec{c}$ using a recursive call. If it finds a proof, then $t \approx s$ is marked proven and \top is returned. Failing this, the algorithm applies the main proof tactic, which checks whether there is a rule r

in rewrite rule database \mathcal{R} whose conclusion's left-hand side u matches t under some substitution σ . In this case, it calls itself recursively, attempting to prove that: (i) the right-hand side s is equivalent to u ; and (ii) each premise of that rule holds in the theory under the same substitution. If both of these checks succeed, $t \approx s$ is marked as proven by rule r . Note that the matching does not automatically take into consideration the commutativity of operators. Instead, the algorithm relies on the commutativity of operators being expressed as additional rewrite rules.

Database Implementation. The algorithm is implemented by using a discrimination tree data structure to index the conclusions of all rules in \mathcal{R} . When a rule is added to \mathcal{R} , it is normalized so that its variables are taken from a global list and assigned based on a left-to-right traversal of the conclusion. For example, $x+y \approx y+x$ is normalized to $x_1+x_2 \approx x_2+x_1$, where the global list of integer variables is (x_1, x_2, \dots) . We enumerate matches for $t \approx s$ based on a single traversal of the discrimination tree, which both constructs the matching substitution and ends at the rewrite rule identifier.

Optimizations and Extensions. Our actual algorithm includes several optimizations and extensions not shown in Figure 3. First, our tactics use a fast failure heuristic that avoids making recursive calls for a set of equalities $\vec{u} \approx \vec{v}$ if a single $u_i \approx v_i$ can be shown to fail without recursion. For example, our congruence tactic for $f(u, 0) \approx f(v, 1)$ fails early since $(0 \approx 1) \downarrow = \perp$. Second, we extend our techniques for evaluation of arithmetic equalities to incorporate *polynomial normalization*, where, for example, the arithmetic term $y + x + x$ can be shown to be equal to $2 * x + y$. Third, we use additional built-in tactics for Booleans, e.g., that prove $(t \approx s) \approx \top$ if $t \approx s$ can be proven. Finally, we account for fixed-point rules from \mathcal{R} (as described in Section III) by an extension to the tactic in line 15. In particular, when considering a fixed point rule r with conclusion $u \approx v$ that matches $t \approx s$ with substitution σ , we immediately check if the subterm of $\sigma(v)$ occurring at the placeholder position denoted by r also produces a match using the rule r . If so, we store the proof sketch for $t \approx \sigma(v)$ and continue this process until we have proven the equality $t \approx v'$ for some v' . We then attempt to prove $s \approx v'$ along with the required preconditions for the application(s) we used to derive $t \approx v'$.

B. From Proof Sketches to Proofs

We now return to the question of how to transform a proof sketch into a final proof. A proof is built out of *proof nodes*. A proof node is a triple (r, \vec{q}, \vec{t}) , where r is a proof rule identifier, \vec{q} is a list of proof nodes, and \vec{t} is a list of terms. A *proof checker* for a proof rule r is a function taking a list of formulas $\vec{\varphi}$ and a list of terms \vec{t} , and returning either a *conclusion* formula ψ or failure. Intuitively, the proof checker returns ψ if r concludes ψ from premises $\vec{\varphi}$ and a side condition depending on terms \vec{t} . A well-formed proof in a proof system \mathcal{S} is a directed acyclic graph over proof nodes whose conclusions can be assigned based on the proof checkers for the rules in \mathcal{S} . In

particular, a proof node (r, \vec{q}, \vec{t}) can be assigned a conclusion ψ if the proof nodes in \vec{q} are well-formed with conclusions $\vec{\varphi}$ and the proof checker for r on $(\vec{\varphi}, \vec{t})$ returns ψ .

Overall, the algorithm in Figure 3 maintains the invariant that equalities $t \approx s$ map to a rule r by the proof sketch P only if entries for the preconditions \vec{p} of rule r also have been successfully added to P , and moreover these dependencies are acyclic. Thus, we can transform the proof sketch P into a final proof by first recursively reconstructing the proofs of the preconditions to the current rule. For equalities $t \approx s$ marked with the `eval` rule, we construct a proof whose proof rule is reflexivity or evaluation. For equalities $f(\vec{u}) \approx f(\vec{v})$ marked with the `cong` rule, we first construct proofs for each of $\vec{u} \approx \vec{v}$, and then construct the proof of $f(\vec{u}) \approx f(\vec{v})$ by congruence. For equalities $f(\vec{u}) \approx s$ marked `ceval`, after reconstructing the proofs of $\vec{u} \approx \vec{c}$, we prove $f(\vec{u}) \approx f(\vec{c})$ by congruence, $f(\vec{c}) \approx s$ by evaluation, and then $f(\vec{u}) \approx s$ by transitivity of these two equalities using the `trans` rule from Figure 4. For equalities $t \approx s$ marked with a rule r from our database having conclusion $u \approx v$, we reconstruct the substitution σ such that $t = \sigma(u)$ by matching. We prove $t \approx \sigma(v)$ by rule r , which implies the existence of a proof of $\sigma(v) \approx s$ (due to the recursive call on line 16), and we finally combine them to a proof for $t \approx s$ by transitivity.

Example 1: Suppose we wish to prove the correctness of the rewrite `substr(substr("abc", 4, 1), m, n) \rightsquigarrow ""`. Furthermore, assume our rewrite database \mathcal{R} contains:

```
(define-cond-rule substr-empty-s (
  (s String) (m Int) (n Int))
  (= s "") (str.substr s m n) "")
```

We call the method `rc` from Figure 3 on the equality `substr(substr("abc", 4, 1), j, k) \approx ""` with a chosen depth $d=3$. Assume that the proof sketch map P is initially empty. For this input, none of the conditions on lines 1-6 apply. On line 7, we provisionally set $P[\text{substr}(\text{substr}(\text{"abc"}, 4, 1), j, k) \approx \text{""}]$ to $(\text{fail}, 3)$. The conditions on lines 8 and 10 also do not apply. In the loop on line 14, we find that the match term `substr("", j, k)` from rule `substr-empty-s` matches the left-hand side of our equality with substitution $\sigma = \{s \mapsto \text{substr}(\text{substr}(\text{"abc"}, 4, 1), m \mapsto j, n \mapsto k)\}$. On lines 16 and 17, we recursively call `rc` on $(\sigma(\text{""}) \approx \text{""}, 2)$ and on $(\sigma(s \approx \text{""}), 2)$, respectively. Both recursive calls succeed trivially on line 5, where the latter equality is `substr("abc", 4, 1) \approx ""`. Thus, we successfully prove the conditions for applying `substr-empty-s` to our input equality. We denote this in P and return \top , where P is the mapping $\{\text{""} \approx \text{""} \mapsto \text{eval}, \text{substr}(\text{substr}(\text{"abc"}, 4, 1) \approx \text{""} \mapsto \text{eval}, \text{substr}(\text{substr}(\text{"abc"}, 4, 1), j, k) \approx \text{""} \mapsto \text{substr-empty-s}\}$. The proof of the original equality can then be constructed trivially based on this mapping, where, overall, the proof involves an application of `substr-empty-s` whose premise is proven by `eval`.

V. IMPLEMENTATION

We implemented both a compiler for RARE and the reconstruction algorithm, and integrated them with CVC5 [4], a

state-of-the-art SMT solver, most of which is instrumented to produce proofs [6]. Notably, the rewriter is not instrumented, so proof reconstruction is an attractive option for CVC5. Our initial implementation focuses on the theory of strings, both because it is used in practical applications such as reasoning about access policies in the cloud [2], and because it presents a challenge due to the large number of complex rules in the strings theory rewriter, which are required to achieve good performance [21]. The theory of strings is frequently combined with the theory of linear integer arithmetic to reason about the length and indices of strings. Thus, reconstructing rewrite proofs for string problems requires reasoning about Boolean, linear integer arithmetic, and string terms. None of these theories require parameterized sorts, so the current implementation uses concrete types. Supporting rewrite rules with partially specified types is left for future work.

In the following, we discuss the integration of our approach in the existing proof infrastructure and our experience using RARE to define a set of rewrite rules. We implemented our reconstruction algorithm as a module in the existing proof infrastructure of CVC5. At compile-time, our compiler for RARE populates the rewrite rule database (referred to as \mathcal{R} in the previous section). As mentioned earlier, RARE aims at being a compromise between succinctness and expressiveness. The limited expressiveness of RARE means that some desirable rewrite rules cannot be expressed in it. To overcome this limitation, our reconstruction module supports mixing RARE rules with rules implemented in C++. We use this feature, for example, for certain integer arithmetic rewrites, as discussed below. Reconstructing the proofs for rewrites happens during post-processing of the overall proof. If a proof for a given atomic rewrite cannot be reconstructed, a generic *theory rewrite* proof rule is used instead.

The proof module of CVC5 supports the production of proof certificates in different proof formats. One of the proof formats that is well-supported is LFSC [24]. Proofs in LFSC use the same language to define both the proof rules and the proofs themselves. As part of our implementation, we extended CVC5's LFSC back end to automatically generate LFSC proof rules for each rewrite that appears in a given proof.

The string theory rewriter in CVC5 is complex—its implementation, not including any of the helper functions, amounts to over 3,000 lines of C++ code and distinguishes over 200 different rewrite rules. Moreover, not all of those rules can be expressed as a single rewrite rule in RARE. In view of these difficulties, we took a pragmatic approach to proof reconstruction for the theory of strings: instead of trying to implement all of the rewrite rules in RARE, we focused on a set of challenging string benchmarks (see Section VI) of practical interest, and then defined rules on demand to fill in missing subproofs. We ended up with 40 RARE rules for the theory of strings.

The structure of the CVC5 theory rewriter for arithmetic, on the other hand, is quite different. Instead of a large number of different rewrite rules, most of the rewriting boils down to normalizing polynomials. Thus, for normalizing polynomials

we implemented a single rule, which is complemented with 25 rules for arithmetic that do not concern this normalization.

Finally, the rewriter for Booleans is far simpler than rewriters for other theories—its implementation is less than 350 lines of C++ code. For reconstructing Boolean rewrite rules, we took a similar approach to the one for string rewrites and defined RARE rules on demand to fill in missing subproofs on problems of interest. This led to 22 Boolean rules in RARE.

While using RARE is not possible or desirable for all rewrite rules, it did enable us to iterate quickly to cover the majority of missing subproofs for our target benchmarks.

VI. EVALUATION

Using our implementation in CVC5, we evaluated the following research questions:

- Can we generate fine-grained proofs for rewrites?
- What is the performance impact of generating fine-grained proofs?

We considered two benchmark sets, both over the theory of strings. The first consists of 25 unsatisfiable industrial benchmarks that are representative of challenging queries in a specific production environment. The second set consists of 26,626 unsatisfiable benchmarks from the logics QF_S and QF_SLIA in the SMT-LIB benchmark library. To determine the set of unsatisfiable benchmarks, we used the results from an artifact [3] of an earlier evaluation of CVC5, which ran the competition configuration of CVC5 for 1200s.

For the evaluation, we ran all benchmarks with three configurations of CVC5: CVC5, which does not generate any proofs; CVC5-C, which generates proofs with coarse-grained steps for rewrites; and CVC5-F, which uses our approach to generate fine-grained proofs for rewrites. For the proof reconstruction, we set the depth d to 3. The configurations involved in our evaluation are all variants of CVC5 since to the best of our knowledge, no other SMT solvers generate proofs for nontrivial theory rewrites. In particular, no other solver can generate fine-grained proofs for the theory of strings.

We ran all experiments on a cluster equipped with Intel Xeon E5-2620 v4 CPUs running Ubuntu 16.04. We allocated one physical CPU core and 8GB of RAM for each solver-benchmark pair and used a 900 seconds time limit.

To measure the effectiveness of our reconstruction, we analyzed the generated proofs of benchmarks that were solved by all configurations. The proofs for the industrial benchmarks contain 43,819 rewrite steps, and the proofs for the SMT-LIB benchmarks contain 2,806,761. For those steps, CVC5-F reconstructed fine-grained proofs in terms of our current rewrite rule database for 95% of the rewrite steps for the industrial set, and for 92% of the rewrite steps for SMT-LIB. The lower rate in SMT-LIB can be explained by our greater focus on the rewrite steps from proofs of the industrial benchmarks. We expect that the SMT-LIB rate can be improved to the level of the industrial set without significant challenges, i.e., primarily by adding more rules to the rewrite rule database. We also note that for 20% (5 out of 25) benchmarks in the industrial set, CVC5-F manages to produce fine-grained proofs for *all*

TABLE I: Number of solved benchmarks and cumulative solving times in seconds on commonly solved benchmarks, with the slowdown versus CVC5-C in parentheses.

Division		CVC5	CVC5-C	CVC5-F	
Industrial (25)	Solved	25	25	25	(1.09×)
	Time	238	715	779	
SMT-LIB (26,626)	Solved	26,615	26,614	26,609	(3.18×)
	Time	34,028	35,932	114,330	
Total (26,651)	Solved	26,640	26,639	26,634	(3.14×)
	Time	34,266	36,647	115,109	

rewrites, whereas for SMT-LIB, 22% of CVC5-F’s proofs with rewrite steps (5,945 out of 26,418) are fully fine-grained.

Table I summarizes the overhead incurred by our approach grouped by benchmark set. Figure 6 shows a cactus plot that compares the performance of the different configurations. In this experiment, we use CVC5 as a reference point to measure the general overhead of proof production, and to compare that overhead with the additional overhead of generating fine-grained proofs. Table I shows that the overhead on the industrial benchmarks for generating proofs is significant, but the additional overhead of generating the fine-grained proofs is negligible. For the benchmarks from SMT-LIB, the opposite is the case: the overhead for generating coarse-grained proofs is relatively small, but the overhead of generating fine-grained proofs is significant. For a better understanding of the origin of the overhead, we provide three scatter plots in Figure 5. Figure 5a compares the performance of CVC5-C with the performance of CVC5-F and shows that for benchmarks that are solved quickly with CVC5-C, there are cases where the overhead of the proof reconstruction is significant. For longer running benchmarks, the overhead seems to be less pronounced. In Figure 5b, we plot the solving time in relationship with the relative number of atomic rewrites in proofs generated by CVC5-C. The plot shows that atomic rewrites are featured more prominently in proofs of benchmarks that are solved quickly. This may explain part of the overhead for easy benchmarks: a larger portion of the proof is being post-processed with the reconstruction algorithm. Finally, Figure 5c shows the relationship between the difference in solving time between CVC5-F and CVC5-C and the number of atomic rewrites. The plot indicates two trends: more atomic rewrites lead to more overhead and—more surprisingly—there seems to be a large number of benchmarks with a relatively small number of rewrites that have a significant amount of overhead.

Overall, we find that our approach does not significantly affect the number of solved benchmarks. Additionally, it works well for the industrial use case that we originally targeted with our approach. Some of the SMT-LIB benchmarks, on the other hand, make use of complex rewrites such as the ones described in earlier work [21], which we have not explicitly optimized our current implementation for.

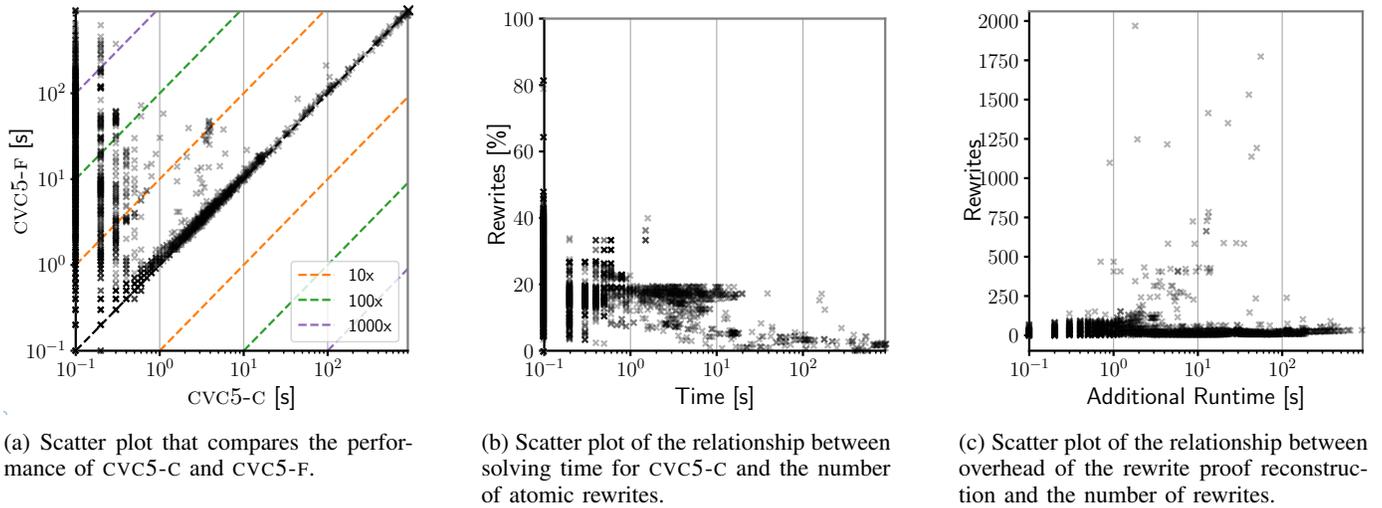


Fig. 5: Scatter plots that analyze the overhead of our rewrite proof reconstruction.

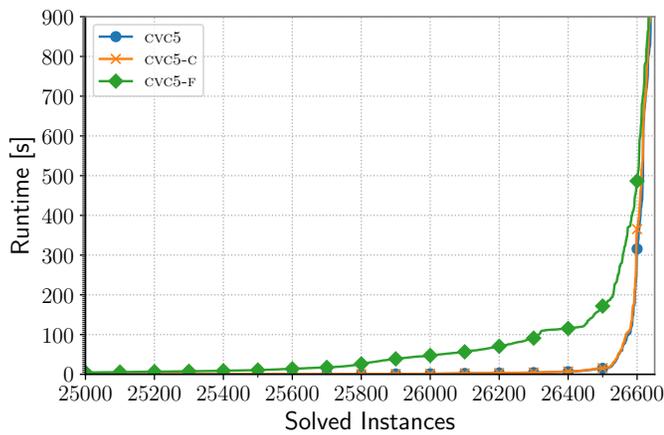


Fig. 6: Cactus plot that shows the general performance impact of generating proofs and the performance impact of generating fine-grained proofs for rewrites.

VII. CONCLUSION

We presented a DSL-based approach for reconstructing fine-grained proofs of rewrite rules. For the future, we plan to expand our implementation to other theories in CVC5, including theories with parameterized sorts, which will require adding support for gradual typing. The DSL proposed in this work is independent of the discussed use case and can be used to express rewrite rules for SMT solvers in other contexts.

Another direction for future work is to expand the DSL compiler to generate efficient code to replace parts of existing theory rewriters, i.e., code that actually performs the rewrites. This could make it much easier to explore different sets of rewrite rules. It would also make the rewriting code easier to understand and maintain. However, since the rewriter is called frequently during solving, its performance is critical. Therefore, integrating automatically generated code needs to be done carefully. Our primary targets in that context are the

theories of fixed-size bit-vectors and floating-point arithmetic.

Another back end for the DSL could be used to generate verification conditions for the T -validity of rewrite rules. These conditions could be discharged using a third-party tool such as a proof assistant or another SMT solver. An interesting challenge here is that SMT solvers generally only support reasoning about fixed-size bit-vectors, whereas rewrite rules for the theory of bit-vectors are parameterized by the bit-width. We plan to explore approaches for bit-width independent verification (e.g., [18]) to discharge these verification conditions.

REFERENCES

- [1] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. S. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan. Stratified abstraction of access control policies. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2020.
- [2] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *FMCAD*, pages 1–9. IEEE, 2018.
- [3] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. M. Y. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. Artifact for Paper cvc5: A Versatile and Industrial-Strength SMT Solver, Nov. 2021.
- [4] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.
- [5] H. Barbosa, J. C. Blanchette, M. Fleury, and P. Fontaine. Scalable fine-grained proofs for formula processing. *J. Autom. Reason.*, 64(3):485–510, 2020.
- [6] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, Y. Zohar, C. Tinelli, and C. W. Barrett. Flexible proof production in an industrial-strength SMT solver. In J. Blanchette, L. Kovács, and D. Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022.

- [7] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
- [8] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.
- [9] P. Borovanský, C. Kirchner, H. Kirchner, and P. Moreau. ELAN from a rewriting logic point of view. *Theor. Comput. Sci.*, 285(2):155–185, 2002.
- [10] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta, C. Schlesinger, C. Stephens, C. Varming, and A. Warfield. Block public access: trust safety verification of access control policies. In P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 281–291. ACM, 2020.
- [11] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
- [12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
- [13] B. Cook. Formal reasoning about the security of amazon web services. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2018.
- [14] R. Diaconescu and K. Futatsugi. *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
- [15] H. Enderton and H. B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.
- [16] W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9(2):147–167, 1992.
- [17] A. Niemetz, M. Preiner, and C. W. Barrett. Murxla: A modular and highly extensible API fuzzer for SMT solvers. In S. Shoham and Y. Vitez, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2022.
- [18] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. Barrett, and C. Tinelli. Towards satisfiability modulo parametric bit-vectors. *Journal of Automated Reasoning*, 65(7):1001–1025, Oct. 2021.
- [19] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.
- [20] A. Nötzli. *Towards better simplifications in SMT solvers with applications in string solving*. PhD thesis, Stanford University, 2021.
- [21] A. Reynolds, A. Nötzli, C. W. Barrett, and C. Tinelli. High-level abstractions for simplifying extended string constraints in SMT. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2019.
- [22] H. Schurr, M. Fleury, H. Barbosa, and P. Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). *CoRR*, abs/2107.02354, 2021.
- [23] H. Schurr, M. Fleury, and M. Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In A. Platzer and G. Sutcliffe, editors, *Proc. Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.
- [24] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. Smt proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.
- [25] T. C. D. Team. The coq proof assistant, Jan. 2022.
- [26] C. Tinelli and C. G. Zarba. Combining decision procedures for sorted theories. In J. J. Alferes and J. Leite, editors, *Logics in Artificial Intelligence*, pages 641–653, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Small Proofs from Congruence Closure

Oliver Flatt^{*}, Samuel Coward[†], Max Willsey[‡], Zachary Tatlock[§], Pavel Panchekha[¶]

^{*} University of Washington, Seattle WA 98195, USA, Email: oflatt@cs.washington.edu

[†] Numerical Hardware Group, Intel Corporation, Email: samuel.coward@intel.com

[‡] University of Washington, Seattle WA 98195, USA, Email: mwillsey@cs.washington.edu

[§] University of Washington, Seattle WA 98195, USA, Email: ztatlock@cs.washington.edu

[¶] University of Utah, Salt Lake City, UT 84112, USA, Email: pavpan@cs.utah.edu

Abstract—Satisfiability Modulo Theory (SMT) solvers and equality saturation engines must generate proof certificates from e-graph-based congruence closure procedures to enable verification and conflict clause generation. Smaller proof certificates speed up these activities. Though the problem of generating proofs of minimal size is known to be NP-complete, existing proof minimization algorithms for congruence closure generate unnecessarily large proofs and introduce asymptotic overhead over the core congruence closure procedure. In this paper, we introduce an $O(n^5)$ time algorithm which generates optimal proofs under a new relaxed “proof tree size” metric that directly bounds proof size. We then relax this approach further to a practical $O(n \log(n))$ greedy algorithm which generates small proofs with no asymptotic overhead. We implemented our techniques in the `egg` equality saturation toolkit, yielding the first certifying equality saturation engine. We show that our greedy approach in `egg` quickly generates substantially smaller proofs than the state-of-the-art Z3 SMT solver on a corpus of 3760 benchmarks.

I. INTRODUCTION

Congruence closure procedures based on e-graphs [1] are a central component of equality saturation engines [2], [3] and SMT solvers [4], [5]. Sophisticated optimizations like deferred congruence [3] and incremental e-matching [6] make such tools faster, but also make guaranteeing correctness more difficult [7], [8].

Engineers sidestep the challenge of directly verifying high-performance congruence implementations by instead extending procedures to generate *proof certificates* [8], [9]. Proof certificates provide the sequence of equalities that the congruence procedure used to establish that two terms are equivalent. Clients can safely use results from an untrusted procedure by checking its proofs. For example, several proof assistants adopt this strategy to provide “hammer tactics” [10] which dispatch proof obligations to SMT solvers and then reconstruct the resulting SMT proofs back into the proof assistant’s logic, thus improving automation without trusting solver implementations.

Proof *size* can be especially important when extending existing verification tools with untrusted solvers. For example, in a case study on six Intel-provided Register Transfer Level (RTL) circuit design benchmarks [11], an untrusted equality saturation engine took under 1 minute to optimize, but the existing verification tool took 4.7 hours to replay and check the large proof certificates generated by existing techniques [9].

Unfortunately, finding proofs of minimal size is an NP-complete problem [12].

In this paper, we explore efficient generation of small proof certificates for e-graph-based congruence procedures. We first introduce the problem of *finding minimal size proofs* for congruence closure procedures. We define the space of admissible proofs and give an integer linear programming formulation for finding a proof with minimal size. Next, we introduce a relaxed metric called *proof tree size*, which directly bounds the size of the proof, and develop TreeOpt, an $O(n^5)$ time algorithm for finding a proof with minimal proof tree size. Unfortunately, the $O(n^5)$ algorithm is still too expensive for practical use, since congruence closure procedures often consider thousands of equations. Thus we also developed an $O(n \log(n))$ time greedy approach using subproof size estimates. Our algorithm incurs no asymptotic overhead relative to congruence closure and finds small proofs in practice.

We evaluate our approach by implementing both proof generation and greedy proof minimization in the state-of-the-art `egg` equality saturation toolkit [3], yielding the first certifying equality saturation engine. We compare our greedy algorithm against the state-of-the-art SMT solver Z3, which performs proof reduction (see Section II) to find smaller proofs. Where we can run Z3 (Z3 times out in 5.0% of cases), our proofs are only 72.8% as big as Z3’s on average (15.0% in the best case). Our proofs are also only 107.8% as big as TreeOpt’s on average, compared to 147.6% for Z3. Using our greedy proof minimizer, we were able to reduce proof replaying time in the Intel-provided RTL verification case study from 4.7 hours down to 2.3 hours.

In this paper, we first define the problem of finding the minimal proof and provide an ILP formulation (Section III). We then introduce the proof tree size metric and an optimal $O(n^5)$ time algorithm for finding proofs of minimal tree size (Section IV). Finally, we demonstrate a practical greedy algorithm for finding proofs of small tree size with no asymptotic overhead (Section V).

II. BACKGROUND AND RELATED WORK

Congruence is the property that $a = b$ implies $f(a) = f(b)$. Congruence closure refers to building a model of a set of equalities that satisfies congruence; these models can be used for determining whether other equalities are true (as is common in SMT solvers) or for finding new equivalent forms of

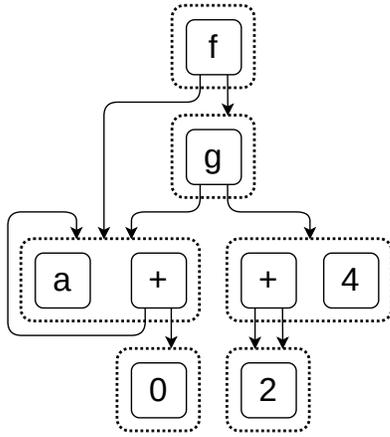


Fig. 1: A e-graph model of the equalities $a + 0 = a$ and $2 + 2 = 4$ and the expression $f(a + 0, g(a + 0, 2 + 2))$. Note that the top e-class contains both the expression $f(a + 0, g(a + 0, 2 + 2))$ and the expression $f(a, g(a, 4))$, which proves that these two expressions are equal modulo the equalities.

an expression (as is common in equality saturation engines). For example, consider the equalities $a + 0 = a$ and $2 + 2 = 4$; a model of these two equalities should permit queries like whether $f(a + 0, g(a + 0, 2 + 2))$ has a simpler form or whether it is equal to $f(a, g(a, 4))$.

A congruence closure model is typically represented as an e-graph, which is a collection of e-nodes and e-classes.¹ Each e-node represents a single function being applied and an e-class for each argument; each e-class, meanwhile, is a set of equivalent e-nodes. Any expression can be inserted into the e-graph by converting it recursively into e-nodes, while equalities can be added into the e-graph by merging the e-classes for the equality’s left and right hand side. For example, given the equalities $a + 0 = a$ and $2 + 2 = 4$, one can determine whether $f(a + 0, g(a + 0, 2 + 2)) = f(a, g(a, 4))$ by inserting these two expression into an e-graph and then adding the two equalities. The resulting e-graph is shown in Figure 1. The two expressions end up in the same e-class, so they have been proven to be equal.

Congruence procedures must handle queries quickly, with tens or hundreds of thousands of equalities. The large number of equalities means that e-graphs can contain hundreds of thousands or even millions of e-nodes, with the resulting e-graph taking significant time to construct. A substantial literature [3], [6], [13] describes numerous optimizations to e-graphs. Past work shows that an e-graph for n equalities can be constructed in $O(n \log n)$ time [14].

Congruence Proofs Proof certificates for e-graphs allow checking that two terms are equal without reconstructing the e-graph. Instead, for an equality $E_1 = E_2$ witnessed by the e-graph, a proof certificate is a list of given equalities that

¹Depending on the author, the “e” in “e-graph” can stand for “expression”, “equivalence”, or “equality”.

can be applied in order, one after another, as rewrite rules to transform E_1 into E_2 . Some of these equalities are applied at the root of the expression being rewritten, while others apply to subexpressions (via congruence). In our running example, we can prove $f(a + 0, g(a + 0, 2 + 2)) = f(a, g(a, 4))$ as follows:

$$\begin{aligned}
 & f(a + 0, g(a + 0, 2 + 2)) \\
 & \xrightarrow{a+0=a} f(a, g(a + 0, 2 + 2)) \\
 & \xrightarrow{2+2=4} f(a, g(a + 0, 4)) \\
 & \xrightarrow{a+0=a} f(a, g(a, 4))
 \end{aligned}$$

Note that some equalities may be reused, as in this example.

Over time, proof certificates have grown increasingly important. In SMT solvers, proof certificates correspond to conflict clauses and enable *non-chronological backtracking*, a key component of modern SMT solvers [15]. In proof automation, proof certificates bridge foundational logics and unverified automated theorem provers, as in the “hammer” style of proof tactics [10]. In equality saturation engines, replaying proof certifications enables the combination of slow verification procedures with fast equality saturation engines.

To produce proofs certificates, e-graph implementations maintain a spanning tree for each e-class, with each edge of the tree justifying the equality of the two e-nodes it connects [16]. This justification is either one of the (quantifier-free) equalities provided as input or a congruence edge that refers to other connected nodes in the tree. This spanning tree is maintained alongside the union-find structure used for efficiently merging e-classes, so there is no algorithmic overhead to maintaining it. Producing a proof for the equality of two e-nodes in the same e-class is then a simple recursive procedure which traverses the path between two e-nodes, recursively finding subproofs for each congruence edge. In a spanning tree, there is a unique path between any two e-nodes, so this recursive algorithm is quite fast, taking $O(n \log n)$ time for n equalities.

Shrinking Congruence Proofs Most uses of proof certificates, including generating conflict clauses and replaying and checking proofs, take longer as more unique equalities are used in the proof certificate. The standard approach to finding smaller proof certificates, implemented in SMT solvers such as Z3 [5], is based on the observation [16] that proof certificates can contain redundant equations; for example, if the given equalities include $a = b$, $a = c$, and $b = c$, a proof certificate may include all three. By attempting to re-prove the same equation while excluding one of the equalities, a proof certificate can thereby be shrunk. If the initial proof certificate has length k , this proof reduction procedure takes $O(k^2 \log k)$ (as checking the validity of each new proof takes $O(k \log k)$ time using an e-graph).

This state of the art algorithm is limited in two ways. First, when $k \in o(\sqrt{n})$, it introduces an asymptotic slowdown over the rest of the congruence closure algorithm, which can answer queries and generate proofs in $O(n \log n)$ time

(where n is the number of equalities). Second and more importantly, proof reduction is ultimately limited by the choice of the proof to reduce. Since proof reduction is too slow to consider the entire e-graph, a valid initial proof is generated before applying proof reduction, discarding many (potentially useful) equalities right away. This means that, while it results in shorter proof certificates, those proof certificates are still longer than optimal. This paper addresses both concerns.

III. OPTIMAL DAG SIZE

Because proof certificates often contain repeated subproofs, we propose a measure for a proof’s size in terms of the number of *unique* equalities it uses. We call this measure *DAG size* because equalities may be reused in the proof. DAG size is also the same as the size of a conflict set in the context of SMT solvers. The problem of finding a proof of minimal DAG size is also NP-complete [12]. This section formalizes a *DAG size* measure of proof length which accounts for subproof reuse, and gives an ILP formulation for finding the proof of optimal DAG size.

A. C-graphs

Traditionally, each equivalence class in an e-graph is represented by a spanning tree. Each edge in the spanning tree is either a single equality between two terms or equality via congruence. Any additional equalities between nodes already connected are discarded, since there is already a way to prove the two terms are equal. However, these equalities may enable a significantly smaller proof. For example, an e-graph can be constructed from the equalities $a = b$, $b = c$, and $a = c$. The e-graph constructs a spanning tree with edges $a = b$ and $b = c$, discarding $a = c$. Now the e-graph will admit a proof between a and c that has a size of 2.

Since these additional equalities can be used to produce shorter proofs, our algorithm requires storing them. We call the resulting structure a c-graph, which maintains a graph, not a spanning tree, for each equivalence class. Storing these additional edges merely requires recording information on every e-graph merge operation, so can be done without changing the complexity of the congruence closure algorithm. The c-graph can be substituted directly for an e-graph without changing the complexity of the congruence closure algorithm. In practice, a c-graph uses the same representation and algorithms as an e-graph, but additionally has an adjacency list for each node storing this graph of equalities. In the context of producing proofs, we define a simple version of a c-graph below:

Definition 1. A c-graph is an undirected graph $G = (V, E)$, where nodes V represent expressions and edges E represent equalities, along with a justification $j(e)$ for edge e . A justification is either an equality $v_1 = v_2$ between the vertices or a congruence subproof $c_1 = c_2$, where c_i is a child of v_i .

For convenience, we write C for the set of congruence edges in E . An edge justified by an equality connects the left and right-hand sides of the equality directly, while an edge justified by a congruence $c_1 = c_2$ connects terms which are equal

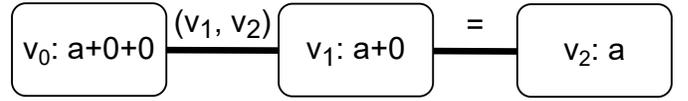


Fig. 2: A c-graph proof that $a + 0 + 0 = a$. There is one congruence edge (v_0, v_1) with $j((v_0, v_1)) = (v_1, v_2)$. Since v_0 and v_2 are e-connected, the proof holds.

by congruence over c_1 and c_2 (e.g. $f(c_1)$ and $f(c_2)$). If two terms are equal due to the congruence of multiple children, the c-graph contains one congruence edge per argument (one per child). This keeps the encoding simple, as each congruence edge corresponds to one proof of congruence. All functions have a bounded arity, so this transformation does not affect complexity results.

For a c-graph to be a valid proof, all congruence edges must refer to e-connected nodes:

Definition 2. A congruence edge $e \in E$ with $j(e) = (c_1 = c_2)$ is *valid* if the congruent children c_1 and c_2 are **e-connected** in the reduced c-graph (G', j) , where $G' = (V, E \setminus \{e\})$. All non-congruence edges are valid.

Definition 3. Two vertices v_s and v_t are **e-connected** in a c-graph (G, j) if there is a path between them consisting of **valid** edges in E .

A c-graph then proves $s = t$ if the corresponding vertices v_s and v_t are e-connected. The particular path showing that v_s and v_t are e-connected, along with proofs for each congruence edge along the path, represents a particular proof. The definition of e-connectedness and edge validity are mutually recursive; the base case occurs when two vertices are connected by a set of non-congruence edges.

The c-graph structure allows for a simple definition of the DAG size metric:

Definition 4. The *DAG size* of a c-graph (G, j) is $|E \setminus C|$, the number of non-congruence edges it contains.

Each non-congruence edge $e \in E \setminus C$ could also be assigned a positive, real-numbered weight $w(e)$, giving a weighted DAG size: $\sum_{e \in E \setminus C} w(e)$. Applications could leverage these weights in order to sample proofs that minimize an alternative objective function, such as the run-time of verifying the steps of the proof. The algorithms in this paper easily support weighted DAG size, but we will use the simpler definition of DAG size with each non-congruence edge assigned a weight of 1.

B. Minimal DAG Size

The key to finding shorter proofs is to keep track of a c-graph of possible proofs during congruence closure, from which a short proof can eventually be extracted. Traditional congruence closure algorithms store only one proof of equality between any two terms (they generate c-graphs shaped like forests) because they discard any equalities they discover between already-equal terms. Instead, we will store these redundant edges, producing a c-graph shaped like a full graph,

EDGES	$S[i, j] \leq (i, j) \in E \setminus C$	$S[i, j] = S[j, i]$
CONGRUENCE	$M[i, j, l, r] \leq (i, j) \in E \wedge j((i, j)) = (l = r)$	$M[i, j, l, r] = M[j, i, r, l]$
PATHS	$P[i, i, j] = 0$ $C[i, j] = \sum_k P[i, k, j]$	$P[i, k, j] \leq V[i, j]$ $P[i, k, j] \leq C[k, j]$
VALIDITY	$V[i, j] \leq S[i, j] + \sum_{l, r} M[i, j, l, r]$	
NO CYCLES	$0 \leq D[i, j] \leq \ell$ $(1 - P[i, k, j])\ell + (D[i, j] - D[k, j]) \geq D[i, k]$ $(1 - M[i, j, l, r])\ell + D[i, j] \geq D[l, r]$	$D[i, j] \geq 1$ if $i \neq j$
GOAL	$C[v_s, v_t] = 1$	$\min \sum_{i, j} S[i, j]$

Fig. 3: An integer linear programming formulation of the minimum DAG size problem. Variables S , M , V , and P are sets of boolean variables, while D is integer-valued. Variables are indexed by i , j , and k , which represent nodes in the c-graph. Decision variables S and M define which non-congruence and congruence edges of E are selected respectively. $\ell = |C|^{|C|+1}|E|$ bounds the maximum length of a valid non-cyclic path.

and will then later search this c-graph for a sub-c-graph of minimal size. We will also discover any extra opportunities for proofs of congruence between terms, adding these to the c-graph as congruence edges.

Definition 5. Consider a c-graph (G, j) , all of whose edges are valid. We write $(G', j) \subseteq (G, j)$ when $G' \subseteq G$ and all edges in (G', j) are valid.

The goal is then to find the sub-c-graph of minimal size in which two terms s and t remain e-connected.

Definition 6 (The Minimum DAG size Problem). Given a c-graph (G, j) and two e-connected terms s and t , find a $(G', j) \subseteq (G, j)$ in which s and t remain e-connected with minimal DAG size.

Note that a sub-c-graph is defined by which edges in G it keeps; this allows us to phrase the minimum DAG size problem as an integer linear programming problem with one decision variable per edge in E . The full linear programming problem is given in Figure 3. It defines selected edges via S and M , paths P and e-connectedness C (via edge validity V), and breaks cycles using distance measure D ; it is similar to the standard formulation of graph connectedness as an ILP problem, except with extra constraints for the validity of congruence edges. These constraints require the selected edges S and M to form a sub-c-graph of (G, j) with all edges valid. Finally, v_s and v_t are asserted to be e-connected to ensure that the sub-c-graph proves $s = t$ and then DAG size is minimized. While this ILP formulation is solvable by industry-standard ILP solvers for very small instances, it is NP-complete in general [12].

IV. OPTIMAL TREE SIZE

What makes the minimal DAG size problem NP-complete is the fact that the e-connectedness of multiple congruence edges can rely on the same edges. This sharing means that the cost of using a congruence edge depends on equalities other congruence edges rely on—global information about the sub-c-graph of the solution as a whole. Instead of finding the optimal solution, we optimize for a different metric to achieve a practical algorithm for proof length minimization. The distance metric $D[i, j]$ in the ILP formulation, which we call the *tree size* of a c-graph, is an effective metric for this purpose.

The tree-size of a c-graph is computed by summing the length of the proof, without sharing. Specifically, given a c-graph (G, j) that proves $s = t$, its tree size is the tree size of the path from v_s to v_t :

Definition 7. Consider a path P that e-connects v_i to v_j in a c-graph. The tree size of P is the number of non-congruence edges in P plus, for each congruence edge justified by $(v_l = v_r)$, the tree size of the path from v_l to v_r .

If a c-graph has minimal DAG size, its DAG size is the number of non-congruence edges in the graph. Its tree size, meanwhile, may count each more than once, so presents an upper bound on the DAG size.² We can thereby hope that the c-graph of minimal tree size will also have a small DAG size.

Definition 8 (The Minimum Tree Size Problem). Given a c-graph (G, j) that proves $s = t$, find the $(G', j) \subseteq (G, j)$ that proves $s = t$ and has minimal tree size.

²We chose the name “DAG size” and “tree size” because the relationship between these two metrics is similar to the relationship between a DAG and a tree containing the same parent-child relationships.

```

1 def optimal_tree_size(start, end):
2   for i in G.vertices:
3     dist[(i, i)] = 0
4
5   for (l, r) in E \ C:
6     dist[l, r] = 1
7
8   for i in range(|C|):
9     for (l, r) in C:
10      dist[l, r] = shortest_path(l, r, dist)
11  return shortest_path(start, end, weights=dist)

```

Fig. 4: Pseudocode for the optimal proof tree size algorithm. The algorithm keeps a dictionary $dist[a, b]$, the length of the shortest tree size from a to b found so far.

A. Minimum Proof Tree Size Algorithm

Unlike DAG size, tree size does not have the problem of shared edges. Finding a proof of optimal tree size thus does not require global reasoning about the surrounding context: using the same edges with another part of the proof does not reduce the tree size. As a result, it is possible to solve the minimum tree size problem in polynomial time.

Finding a proof of optimal tree size is not a simple graph search. The key problem is that congruence edges may contain other congruence edges in their subproofs, and the tree size of those subproofs is initially unknown. Moreover, often a congruence edge (v_1, v_2) can be proven in terms of another congruence edge (v_3, v_4) and vice versa. Our algorithm tackles this problem by computing the size of proofs of congruence bottom up, in multiple passes. At the i -th pass, it constructs proofs of equalities between vertices where congruence subproofs only go i layers deep. These proofs form an upper bound on the optimal tree size, decreasing in size until the optimal proof is found. When the algorithm reaches a fixed point, the proof of optimal tree size is discovered. The algorithm for finding the size of the optimal proof is given in Figure 4. With more bookkeeping, it can be easily extended to yield the specific proof the optimal size corresponds to.

In each pass, this algorithm computes the shortest path for each proof of congruence. Non-congruence edges have a weight of 1, and congruence edges are initialized to have infinite weight. A fixed point is guaranteed after $|C|$ iterations, because each subproof for a congruence edge e cannot use the same edge e again (else its tree size would increase). The overall running time of the algorithm is bounded by $O(|C|^2|E|)$, with $|C|^2$ being the number of calls to the shortest path algorithm and $|E|$ being the complexity of finding a shortest path given the weights. Since there may be n^2 congruence edges for n nodes in the graph, the overall running time is also bounded by $O(n^5)$. However, in practice the number of congruence edges is some constant multiple of n , and in this case the running time is $O(n^3)$.

V. GREEDY OPTIMIZATION OF PROOF TREE SIZE

The optimal algorithm of Section IV finds the proof with minimal tree size, but it does so at an unacceptable cost: its running time dominates the $O(n \log n)$ running time of

```

1 def greedy(start, end, pf_size_estimates):
2   todo = Queue((start, end))
3   fuel = T
4
5   while len(todo) > 0:
6     (start, end) = todo.pop()
7     path = shortest_path(start, end, pf_size_estimates)
8     for edge in path:
9       match edge:
10        congruence(l, r) ->
11          if fuel > 0:
12            todo.push(l, r)
13            fuel = fuel - 1
14          else:
15            add_to_proof(optimized_proof(l, r))
16        axiom(a) ->
17          add_to_proof(a)

```

Fig. 5: Pseudocode for the greedy optimization of proof tree size. The algorithm either recurs for congruence edges if fuel allows, or it uses the estimates for each congruence edge. Unlike TreeOpt, the algorithm is top-down and terminates after T steps.

congruence closure itself [1]. In the context of c -graphs, $n = |E \setminus C|$, the set of input equalities to congruence closure. This section thus proposes a greedy algorithm for proof tree size, which reduces tree size and DAG size significantly in practice, though it is not optimal with respect to either metric.

A. Greedy Optimization

The key insight behind the greedy algorithm is that the multiple passes of the optimal algorithm are only necessary to compute the minimal cost of congruence edges. If the tree size for each congruence edge were known, the proof with optimal tree size could be found by a simple shortest path algorithm. The greedy algorithm is a simple breadth-first search shortest path algorithm that takes estimated costs for congruence edges as an input. The closer the estimates are to the proof of optimal tree size, the better the results of the greedy algorithm.

Defer for now the challenge of estimating the tree size for each congruence edge, and focus on the greedy algorithm itself. The algorithm is simple: use a breadth-first search to choose a path from the start vertex s to the end vertex t of minimal length, using the estimates for each congruence edge. However, those estimates may not be optimal, so the algorithm then recurses for each congruence edge. Note the difference between the optimal algorithm (which first optimizes congruence edges) and the greedy algorithm (which first finds a shortest path). If the recursion were performed until all congruences are optimized, this algorithm would take time $O(|C|(n + |C|))$, which is still too high compared to the $O(n \log(n))$ runtime of congruence closure. Instead, only T expansions of congruence edges are permitted; in practice, we choose $T = 10$, which seems to work well. After T expansions, there may be sub-proofs which have not been generated. In this case, the algorithm defaults to a generic proof production algorithm for the remaining sub-proofs [16]. Figure 5 lists the greedy algorithm.

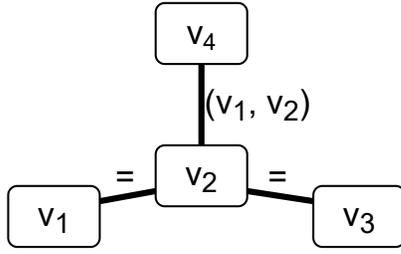


Fig. 6: An example reduced c-graph with a single congruence edge. The root of the tree is the vertex labeled v_4 at the top, and there is a single congruence edge (v_1, v_2) in the spanning tree. The proof of congruence between vertices 1 and 2 has a tree size of two because the proof between the congruent children involves two equalities.

B. Estimating Tree Sizes

The main challenge to instantiating the greedy algorithm is generating size estimates for congruence edges. However, there is a simple way to do so: reduce the c-graph to a forest (G_t, j) with one tree per connected component, in such a way that all edges remain valid. Luckily, the traditional congruence closure proof production algorithm generates such reduced c-graphs by omitting any unions which connect already-equal terms. Now, the tree size of a proof of congruence can be estimated by directly calculating the tree size of a proof in the reduced instance. In such a reduced c-graph, there is only one possible path between any two nodes, so the proof is unique.

Computing the tree sizes of all proofs in the reduced c-graph requires some care to stay within the necessary asymptotic bounds. First, each tree in (G_t, j) is arbitrarily rooted. Given a vertex a , let $\text{size}[a]$ be the size of the proof between a and the root of its tree. Then the tree size of the proof between any two vertices a and b can be calculated

$$\text{size}[a] + \text{size}[b] - 2 * \text{size}[\text{lca}(a, b)],$$

where lca computes the least common ancestor of a and b in the tree. The lca function can be pre-computed for all relevant proofs in $O(n)$ time using Tarjan’s off-line algorithm [17].

Figure 7 shows the pseudocode for calculating proof tree sizes given (G_t, j) . To avoid an infinite loop in proof length calculation, the algorithm builds each tree in (G_t, j) incrementally using a union-find structure (using the `parent` array). Consider the example in Figure 6, in which the path to the root node v_4 contains a congruence edge. The tree size of the proof between nodes v_2 and v_4 , written $\text{tree_size}(v_2, v_4)$, involves calculating the size of the congruence proof $\text{tree_size}(v_1, v_3)$. So $\text{tree_size}(v_2, v_4)$ cannot be computed using v_4 as the root of the tree, since the path to the root involves the congruence edge. Instead, the algorithm uses least common ancestor v_2 to compute $\text{tree_size}(v_1, v_3)$. Because the proof is e-connected, any congruence edges on the path to the least common ancestor can be computed recursively without diverging.

```

1
2 def path_compress(vertex):
3     if parent[vertex] != vertex:
4         path_compress(parent[vertex])
5         parent[vertex] = parent[parent[vertex]]
6         size[vertex] = size[vertex] + size[parent[vertex]]
7
8 def traverse_to_ancestor(v, ancestor):
9     while parent[vertex] != ancestor:
10        edge = parent_edge(parent[vertex], G)
11        parent[edge.start] = edge.end
12        if is_congruence(edge):
13            traverse(j(edge).start, j(edge).end)
14            estimate_size(edge)
15        path_compress(vertex)
16
17 def traverse(start, end):
18     path_compress(start)
19     path_compress(end)
20     ancestor = argmin(
21         (lca(start, end), parent[start], parent[end]),
22         distance_to_root)
23     path_compress(ancestor)
24
25     # Ensure that start, end, and their lca share a parent
26     traverse_to_ancestor(start, ancestor)
27     traverse_to_ancestor(end, ancestor)
28     estimate_tree_size(start, end)
29
30 def estimate_tree_size(start, end):
31     tree_size[(start, end)] = size[start] + size[end]
32                             - 2 * size[lca(start, end)]
33
34 def estimate_size(edge):
35     match edge:
36         congruence(left, right) ->
37             size[edge.start] = tree_size[(left, right)]
38         axiom(a) ->
39             size[edge.start] = 1
40
41 for i in G.vertices:
42     parent[i] = i
43     size[i] = 0
44
45 for (start, end) in congruence_edges(G):
46     traverse(start, end)

```

Fig. 7: Pseudocode for computing tree sizes of all congruence proofs given (G_t, j) . The algorithm efficiently computes these tree sizes by storing a union-find datastructure that keeps track of size, the size of the proof between a node and its parent. Computing the size of a proof involves traversing the proof, updating the union-find whenever the size of a sub-proof is discovered. The pseudocode uses the function `distance_to_root` to denote the number of edges from v to the root of its tree. It also makes use of `lca`, a function that returns the lowest common ancestor of two vertices.

Each congruence edge results in at most one recursive call to `traverse`, while non-congruence edges are added to the union-find data structure directly. Ultimately, each edge in the c-graph contributes at most five union-find operations: three find operations at the start of `tree_size`, one union operation to add it to the union-find data structure, and one more find in `traverse_to_ancestor`. A sequence of m operations on a union-find data structure with h nodes can be executed in $O(m \log(h))$ time [18]. This means the overall cost of estimating sizes for congruence edges is $O(n \log(n))$ since n bounds both m and h (recall $n = |E \setminus C|$). Adding

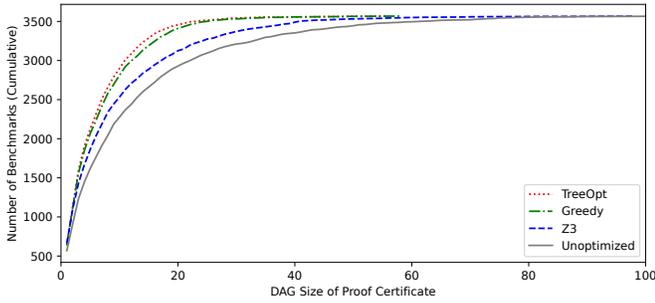


Fig. 8: This CDF compares the unoptimized (gray solid), Z3 (blue dashed), greedy (green dash-dotted), and TreeOpt (red dotted) proof generation algorithms on the same 3571 benchmarks where Z3 does not time out. Each line shows the number of benchmarks whose proofs are at most the size indicated on the horizontal axis. Our greedy approach (green) closely tracks the size of TreeOpt’s (red) proof certificates, showing that its certificates are difficult to shrink further. Five outliers with an unoptimized DAG size of more than 100 are omitted.

on $O(n + |C|)$ cost for the greedy algorithm itself yields an overall runtime of $O(n \log(n) + n + |C|) = O(n \log(n) + |C|)$. Limiting the number of congruence edges C to a multiple of n results in a $O(n \log(n))$ runtime, introducing no asymptotic overhead compared to congruence closure alone.³

VI. EVALUATION

This section compares an implementation of our greedy proof generation algorithm in the `egg` equality saturation toolkit [3] to Z3’s proof generation [19]. As described in Section II, Z3 applies *proof reduction* to the first proof it finds, which substantially reduces proof size. Our greedy approach instead attempts to extract a minimal proof from the e-graph. We found that, even without a proof reduction post-pass, our greedy approach can quickly find significantly smaller proofs than Z3 (Figure 8).

A. Comparing `egg` to Z3

We use Z3 version 4.8.12 and `egg` version 0.7.1 compiled with Rust 1.51.0. `egg` is a state-of-the-art equality saturation library that implements the rebuilding algorithm for speeding up equality saturation workloads. It is used by projects like Herbie [20], Ruler [21] and Szalinski [22]. Z3 is a state-of-the-art automated theorem prover and is optimized for theorem proving workloads. To create a realistic benchmark set, we used the Herbie 1.5 numerical program synthesis tool [20]. Herbie uses equality saturation for program optimization and comes with a standard benchmark suite of programs drawn from textbooks, research papers, and open-source software. We extracted Herbie’s set of quantified equalities and recorded all inputs and outputs from its equality saturation procedure.

³In practice, $|C|$ is typically a small constant factor larger than n . We use a constant factor of $10n$ as a reasonable limit on the number of congruence edges.

TABLE I: Data comparing `egg` to Z3 using different proof production algorithms: `egg` with proofs of optimal tree size, `egg` with greedy optimization, `egg` with traditional proof reduction (see section II), Z3, and `egg` without any optimization. Note that proof reduction’s analysis is in terms of k , the size of the unoptimized proof, while n is the size of the entire c-graph instance. In practice, k is often small relative to n .

Algorithm	TreeOpt	Ave Time (ms)	Complexity
TreeOpt	100.0%	1008.60	$O(n^3)$
Greedy	105.9%	39.33	$O(n \log(n))$
Egg Reduc.	138.7%	23.01	$O(n \log(n) + k^2 \log(k))$
Z3	147.3%	130.69	$O(n \log(n) + k^2 \log(k))$
Egg	185.9%	22.15	$O(n \log(n))$

This results in 3760 input/output pairs, of which we focus on the 3571 where Z3 did not produce an answer after 2 minutes.

For the Z3 baseline, we converted each input/output pair into a satisfiability query by asserting each quantified equality (with a trigger for the left hand side of the equality) and then asserting that the input and output are not equal. Z3 then attempts to prove the input and output are equal using an e-graph and the quantified equalities (the theory of uninterpreted functions). We then computed the DAG size by counting the number of calls to its `quant-inst` command [23] in its proof scripts. We ran `egg` exactly how it is used by Herbie, and then optimized proof length using the greedy algorithm of Section V and measured DAG size by counting proof nodes. Z3 times out after 2 minutes for 5.0% of the input/output pairs, and completes in 213.25 milliseconds on average for the remainder. `egg` does not time out, and runs for an average of 39.57 milliseconds. To measure DAG size for the resulting proofs, we ran both `egg` and Z3 in proof-producing mode and examined the resulting proofs.

Figure 8 contains the results: the proofs produced by `egg` are 72.8% as big as Z3’s on average, despite Z3’s use of a proof reduction algorithm. Moreover, the effect of proof length optimization is greater for longer proofs: queries with Z3 DAG size over 10 see an average 36.0% reduction, while queries with Z3 DAG size over 50 see an average 49.7% reduction.

B. Detailed Analysis

In this section, we perform a more detailed ablation study comparing `egg`’s results using different algorithms. We implement proof reduction for `egg` and the optimal tree width algorithm described in Section IV. The ILP solution is not feasible to run, so we use Z3 as a baseline.

Table I summarizes the results. Z3 and `egg` are optimized for different workloads and so use different underlying congruence closure algorithms, and so produce different proofs. Using proof reduction, `egg` finds slightly shorter proofs than Z3. It also performs better than Z3-style proof reduction implemented in `egg`. Using the greedy algorithm, `egg` finds proofs which are even shorter, and which are also quite close to proofs of optimal tree size. The data in Table I consists of the 3571 out of 3760 where Z3 did not time out, the same set used in Figure 8.

TABLE II: RTL design benchmark results. Total runtime includes equality saturation and proof production runtimes but excludes any formal verification time.

Benchmark	Tree Size			DAG Size			Runtime (sec)		
	Orig	Greedy	Reduce	Orig	Greedy	Reduce	Total	Proof	Proof %
Datapath 1	174	90	48%	67	61	9%	37.5	2.58	7%
Datapath 2	561	92	84%	98	46	53%	34.5	2.08	6%
Datapath 3	14	13	7%	13	12	8%	5.13	0.49	9%
Datapath 4	4402	202	95%	223	120	46%	76.4	32.80	43%
Datapath 5	271	95	65%	101	72	29%	105	0.18	0.2%
Datapath 6	155	83	46%	67	49	27%	280	168.00	60%

While we would ideally use the minimal DAG size proofs as a baseline in our evaluation, we found the ILP formulation was infeasible to run on real queries. However, the $O(n^5)$ TreeOpt algorithm, which runs in $O(n^3)$ time when the number of congruences is bounded, performs well enough to run on all of the examples. We found that in 81.1% of these cases, the greedy algorithm in fact found the proof with optimal tree size. Moreover, across all of these benchmarks our greedy algorithm’s overall performance closely tracks that of TreeOpt, showing that the greedy algorithm’s proof certificates are difficult to shrink further.

C. Case Study

Typically, proof production is necessary in equality saturation to perform translation validation. In this case, the shorter proofs produced by proof length optimization reduce the number of translation validation steps that must be performed and thus result in faster end-to-end results. A practical application that benefits from this reduction is hardware optimization performed using `egg` by researchers at Intel Corporation [11]. Translation validation is used to ensure that the `egg` optimized hardware designs are formally equivalent to the input. Extremely high assurance is needed for hardware designs because of the high cost of actual hardware manufacturing. For *each* step in the tree proof two Register Transfer Level (RTL) designs are generated, which are proven to be formally equivalent by Synopsys HECTOR technology, an industrial formal equivalence checking tool. The intermediate steps generate a chain of reasoning proving the equivalence of the input and optimized designs, necessary because the tools can fail to prove equivalence of significantly transformed designs. The tree proof is used to ensure that HECTOR can prove each step with no user input as it is a simpler check than a DAG proof step.

The results of evaluating this paper’s greedy optimization algorithm on six Intel-tested RTL design benchmarks are shown in Table II. On average, proof lengths decreased by 29%, with the best case showing a 53% reduction, while proof production took only 34 seconds on average, miniscule compared to multi-hour translation validation times. Moreover, these reductions in proof length resulted in shorter translation validation times. The optimized constant multiplication hardware design described in Figure 9 was generated by `egg`,

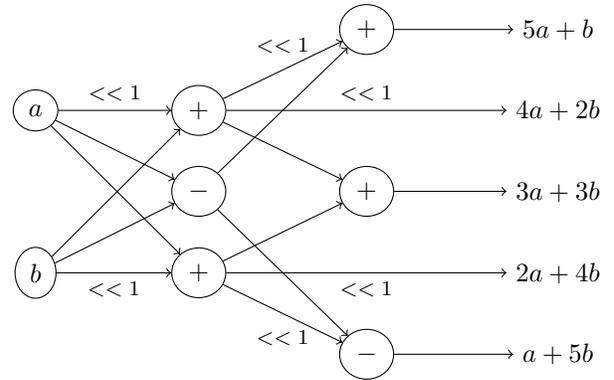


Fig. 9: Dataflow graph of an optimized multiple constant multiplication circuit design generated by `egg`.

starting from an initial naive implementation. Running the complete verification flow for the original and greedy proofs, the runtime was reduced from 4.7 hours to 2.3 hours. In more complex examples we expect that days of computation could be saved. For parameterizable RTL, where a design must typically be re-verified for every possible parameterization, these gains add up quickly.

VII. CONCLUSION AND FUTURE WORK

This paper examined the problem of finding minimal congruence proofs from first principles. Since finding the optimal solution is infeasible, we introduced a relaxed metric for proof size called *proof tree size*, and gave an $O(n^5)$ algorithm for optimal solutions in that metric. While the optimal algorithm is too expensive in practice, it provides a reasonable baseline for small congruence problems, and inspired a practical $O(n \log(n))$ greedy algorithm which generates proofs which are 107.8% as big on average.

We implemented proof generation in the `egg` equality saturation toolkit, making it the first equality saturation engine with this capability. Since equality saturation toolkits—unlike SMT solvers—support optimization directly, this opens the door to certifying the results of much recent work in optimization and program synthesis [3], [20]–[22], [24]–[26].

Looking forward, we are especially eager for the community to explore more applications of proof certificates in congruence closure procedures. For example, it should be possible

to use proofs to tune rewrite rule application schedules in e-matching, improve debugging of subtle equality saturation issues, and enable equality-saturation-based “hammer” tactics in proof assistants. It may also be possible to further improve on the greedy proof generation algorithm with better heuristics for estimating proof sizes, or to enable more efficient prover state serialization via smaller proofs.

VIII. ACKNOWLEDGEMENTS

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA and supported by the National Science Foundation under Grant No. 1749570.

REFERENCES

- [1] C. G. Nelson, “Techniques for program verification,” Ph.D. dissertation, Stanford, CA, USA, 1980, aAI8011683.
- [2] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: A new approach to optimization,” in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’09. New York, NY, USA: ACM, 2009, pp. 264–276. [Online]. Available: <http://doi.acm.org/10.1145/1480881.1480915>
- [3] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, “Egg: Fast and extensible equality saturation,” *Proc. ACM Program. Lang.*, vol. 5, no. POPL, jan 2021. [Online]. Available: <https://doi.org/10.1145/3434304>
- [4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, “Cvc4,” in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV’11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 171–177.
- [5] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’08/ETAPS’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1792734.1792766>
- [6] L. Moura and N. Bjørner, “Efficient e-matching for smt solvers,” in *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, ser. CADE-21. Berlin, Heidelberg: Springer-Verlag, 2007, p. 183–198. [Online]. Available: https://doi.org/10.1007/978-3-540-73595-3_13
- [7] D. Winterer, C. Zhang, and Z. Su, “Validating smt solvers via semantic fusion,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 718–730. [Online]. Available: <https://doi.org/10.1145/3385412.3385985>
- [8] D. Oe, A. Reynolds, and A. Stump, “Fast and flexible proof checking for smt,” in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 6–13. [Online]. Available: <https://doi.org/10.1145/1670412.1670414>
- [9] L. de Moura and N. Bjørner, “Proofs and refutations, and Z3,” in *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, ser. CEUR Workshop Proceedings, P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, and S. Schulz, Eds., vol. 418. CEUR-WS.org, 2008. [Online]. Available: <http://ceur-ws.org/Vol-418/paper10.pdf>
- [10] L. Czajka and C. Kaliszyk, “Hammer for coq: Automation for dependent type theory,” *Journal of Automated Reasoning*, vol. 61, 06 2018.
- [11] S. Coward, G. A. Constantinides, and T. Drane, “Automatic datapath optimization using e-graphs,” vol. abs/2204.11478, 2022. [Online]. Available: <https://arxiv.org/abs/2204.11478>
- [12] A. Fellner, P. Fontaine, and B. Woltzenlogel Paleo, “Np-completeness of small conflict set generation for congruence closure,” *Formal Methods in System Design*, vol. 51, 12 2017.
- [13] Y. Zhang, Y. R. Wang, M. Willsey, and Z. Tatlock, “Relational e-matching,” *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: <https://doi.org/10.1145/3498696>
- [14] P. J. Downey, R. Sethi, and R. E. Tarjan, “Variations on the common subexpression problem,” *J. ACM*, vol. 27, no. 4, p. 758–771, oct 1980. [Online]. Available: <https://doi.org/10.1145/322217.322228>
- [15] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Dpll(t): Fast decision procedures,” in *CAV*, 2004.
- [16] R. Nieuwenhuis and A. Oliveras, “Proof-producing congruence closure,” in *Proceedings of the 16th International Conference on Term Rewriting and Applications*, ser. RTA’05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 453–468. [Online]. Available: https://doi.org/10.1007/978-3-540-32033-3_33
- [17] H. N. Gabow and R. E. Tarjan, “A linear-time algorithm for a special case of disjoint set union,” in *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, ser. STOC ’83. New York, NY, USA: Association for Computing Machinery, 1983, p. 246–251. [Online]. Available: <https://doi.org/10.1145/800061.808753>
- [18] R. E. Tarjan, “Efficiency of a good but not linear set union algorithm,” *J. ACM*, vol. 22, no. 2, p. 215–225, Apr. 1975. [Online]. Available: <https://doi-org.offcampus.lib.washington.edu/10.1145/321879.321884>
- [19] L. de Moura and N. Bjørner, “Proofs and refutations, and z3,” vol. 418, 01 2008.
- [20] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, “Automatically improving accuracy for floating point expressions,” *SIGPLAN Not.*, vol. 50, no. 6, p. 1–11, Jun. 2015. [Online]. Available: <https://doi.org/10.1145/2813885.2737959>
- [21] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, “Rewrite rule inference using equality saturation,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: <https://doi.org/10.1145/3485496>
- [22] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock, “Synthesizing structured CAD models with equality saturation and inverse transformations,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 31–44. [Online]. Available: <https://doi.org/10.1145/3385412.3386012>
- [23] S. Böhme, “Proof reconstruction for Z3 in Isabelle/HOL,” in *7th International Workshop on Satisfiability Modulo Theories (SMT ’09)*, 2009.
- [24] Y. Yang, P. M. Phothilimtha, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar, “Equality saturation for tensor graph superoptimization,” in *Proceedings of Machine Learning and Systems*, 2021.
- [25] A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, *Vectorization for Digital Signal Processors via Equality Saturation*. New York, NY, USA: Association for Computing Machinery, 2021, p. 874–886. [Online]. Available: <https://doi.org/10.1145/3445814.3446707>
- [26] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciuc, “SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra,” *Proceedings of the VLDB Endowment*, 2020.

Proof-Stitch: Proof Combination for Divide-and-Conquer SAT Solvers

Abhishek Nair , Saranyu Chattopadhyay , Haoze Wu , Alex Ozdemir , and Clark Barrett 
Stanford University, Stanford, USA.

{aanair, saranyuc, haozewu, aozdemir, barrettc}@stanford.edu

Abstract—With the increasing availability of parallel computing power, there is a growing focus on parallelizing algorithms for important automated reasoning problems such as Boolean satisfiability (SAT). Divide-and-Conquer (D&C) is a popular parallel SAT solving paradigm that partitions SAT instances into independent sub-problems which are then solved in parallel. For unsatisfiable instances, state-of-the-art D&C solvers generate DRAT refutations for each sub-problem. However, they do not generate a single refutation for the original instance. To close this gap, we present *Proof-Stitch*, a procedure for combining refutations of different sub-problems into a single refutation for the original instance. We prove the correctness of the procedure and propose optimizations to reduce the size and checking time of the combined refutations by invoking existing trimming tools in the proof-combination process. We also provide an extensible implementation of the proposed technique. Experiments on instances from last year’s SAT competition show that the optimized refutations are checkable up to seven times faster than unoptimized refutations.

Index Terms—Parallel SAT, Divide and Conquer, Refutation Checking

I. INTRODUCTION

Boolean satisfiability (SAT) solvers have improved dramatically in recent years. They are now regularly used in a wide variety of application areas including hardware verification [1], computational biology [2] and decision planning [3].

With the emergence of cloud-computing and improvements in multi-processing hardware, the availability of parallel computing power has also increased dramatically. This has naturally led to an increased focus on parallelizing important algorithms, and SAT is no exception. There are two traditional approaches to parallel SAT solving - the Divide-and-Conquer (D&C) approach [4]–[6] and the portfolio approach [7]. In the D&C approach, the original SAT instance is partitioned into independent sub-problems to be solved in parallel, while in the portfolio approach multiple SAT solvers are independently run on the original instance. Although the portfolio approach in combination with clause sharing performs well for small portfolio sizes, the D&C approach scales better in environments with large parallel computing power such as the cloud. Several implementations of D&C solvers exist [4]–[6], [8]. Every implementation uses: a *divider* to split up the original instance into sub-problems, and a *base SAT solver* to solve the

independent sub-problems. For example, *ggSAT* [8] uses *CadiCaL* [9] as its base solver.

If a SAT problem is unsatisfiable, a proof of unsatisfiability (or *refutation*) can be produced and independently checked to validate the result. Since 2013, the annual SAT competition has required SAT solvers to generate refutations. The most commonly supported refutation format today is the DRAT format [10]. Existing D&C SAT solvers produce refutations for each sub-problem independently. However, even if the refutation for each sub-problem passes the proof-checker, this is not a formal guarantee that the original instance also admits a refutation, as there could have been an error in the partitioning strategy. For example, a buggy solver may incompletely partition the SAT instance $(\neg l_1) \wedge (l_2 \vee l_3) \wedge (\neg l_2 \vee l_3)$ into sub-problems with cubes l_1 and $\neg l_2$. Both of these sub-problems are unsatisfiable, even though the instance is satisfiable. Transient errors in the underlying distributed system may also cause sub-problem refutations to be truncated or missing. To address these challenges, we introduce *Proof-Stitch*, which implements a strategy for combining DRAT refutations for sub-problems into a single refutation for the original instance, a process we call *refutation stitching*. Our contributions are:

- We describe an algorithm for combining DRAT refutations of partitions of problems into a single refutation for the original problem and provide an open-source implementation on GitHub [11].
- We describe an optimization technique leveraging existing trimming tools (e.g., *drat-trim* [12]) to improve the quality of the combined refutations.
- We evaluate our implementation on benchmarks from last year’s SAT competition [13]. Our results show that trimmed refutations are checkable up to seven times faster than untrimmed refutations.

The rest of this paper is organized as follows. Section II discusses background and related work. Section III presents the *Proof-Stitch* algorithm and theoretically justifies our method of combining refutations. We also describe an optimization technique that reduces the checking time and the size of the combined refutations. Section IV details our tool implementation. Results are presented in Section V, and Section VI concludes.

This work was partially funded by a gift from Amazon Web Services’ Automated Reasoning group.

II. BACKGROUND AND RELATED WORK

A. Propositional refutations

We assume familiarity with the basic concepts of CDCL SAT algorithms (see, e.g., [14]). We also assume that a base SAT solver can produce a *DRAT* refutation, which we define below (following [15]).

Throughout the paper we model clauses as *sets* of literals and formulas as *multisets* of clauses. By $\cdot \cup \cdot$, we denote the standard union operation on sets, and the multiplicity-summing union on multisets.

Let $F = \{C_1, \dots, C_n\}$ be a formula. F *unit propagates* on ℓ to $F' = \{C \setminus \{-\ell\} : C \in F, \ell \notin C\} \cup \{\ell\}$ (written $F \rightarrow_\ell F'$) if there exists a clause $\{\ell, \ell_1, \dots, \ell_k\} \in F$ such that $\{-\ell_i\} \in F$ for $i \in [1, k]$. If $F \rightarrow_\ell F'$ for some ℓ , then $F \rightarrow F'$. We say that $F \rightarrow \perp$ if F contains an empty clause. Let the relation \rightarrow^* denote the reflexive, transitive closure of \rightarrow . We say that $F \mapsto F'$ when $F \rightarrow^* F'$ and there is no $F'' \neq F'$ such that $F' \rightarrow F''$. One can show that the \mapsto relation is a function. We say that $C = \{\ell_1, \dots, \ell_k\}$ has *asymmetric tautology* (AT) with respect to F if $F \cup \{-\ell_1\} \cup \dots \cup \{-\ell_k\} \mapsto \perp$. We say that C has *resolution asymmetric tautology* (RAT) with respect to literal $\ell_1 \in C$ and F if for all $C' \in F$ containing $\neg \ell_1$, $C \cup (C' \setminus \{-\ell_1\})$ has AT.

Let o_i denote an operation. Consider a sequence of operation-clause pairs $\pi = ((o_1, C_1), \dots, (o_m, C_m))$, where each o_i indicates either the addition (\oplus) or deletion (\ominus) of a clause from a formula.

Let ϕ denote a CNF formula. Define ϕ_i recursively: $\phi_0 = \phi$, and ϕ_{i+1} is $\phi_i \cup \{C_{i+1}\}$ when o_{i+1} is \oplus , or $\phi_i \setminus \{C_{i+1}\}$ otherwise. The sequence π is a *DRAT refutation* of ϕ if when $o_{i+1} = \oplus$ then C_{i+1} has RAT with respect to ϕ_i , and if the last element in π is (\oplus, \emptyset) .

B. Divide-and-Conquer SAT solving

One parallel SAT solving paradigm is *Divide-and-Conquer*: a SAT instance is divided into simpler SAT instances (sub-problems), which are then solved in parallel. Typically, the sub-problems represent partitions of the search space, such that the disjunction of all the sub-problems is equisatisfiable with the original problem. The sub-problems are derived from the original instance by assigning Boolean values to literals. The set of literals that are assigned (decided) for a particular sub-problem is called the *cube* of the sub-problem and the number of literals in the cube is the *depth* of the sub-problem. There are many D&C-based solvers [4]–[6], including: Psato [16], Painless [17], and AMPHAROS [18]. One prominent D&C approach, Cube-and-Conquer [19], uses a lookahead solver to divide instances and a CDCL solver to solve sub-problems. This approach has been successful for large mathematical problems [20] and is implemented by tools such as Paracooba [21] and gg-sat [8].

D&C SAT solvers generate separate DRAT refutations for each sub-problem. There has been little work on combining these refutations into a single refutation for the original instance. One work [22] considers proof composition, but its parallel

composition rule does not apply to DRAT refutations. Another work [23] gives an alternate proof calculus for parallel solvers.

III. METHODOLOGY

In this section, we present an algorithm to combine sub-problem refutations into a refutation for the original Boolean instance. Then we show the algorithm’s correctness. Finally, we present a technique to optimize the combined refutations.

A. Algorithm

The first step in the *Proof-Stitch* algorithm is to construct a decision tree representing the steps taken by the D&C solver. The root of the tree represents the original instance, and the leaves represent the sub-problems. Figure 1 shows the decision tree for an example instance.

Algorithm 1: Stitching algorithm

In : Instance: ϕ ,

Decision literal: x ,

Refutations of:

$\phi \cup \{\{x\}\}$: $\pi = ((o_1, C_1), \dots, (o_n, C_n))$,

$\phi \cup \{\{-x\}\}$: $\pi' = ((o'_1, C'_1), \dots, (o'_m, C'_m))$,

Out: Refutation of ϕ

procedure *stitching* (ϕ, x, π, π')

return

$$\left((o_1, C_1 \cup \{-x\}), \dots, (o_n, C_n \cup \{-x\}), \right. \\ \left. (o'_1, C'_1 \cup \{x\}), \dots, (o'_m, C'_m \cup \{x\}), (\oplus, \emptyset) \right)$$

Next, *Proof-Stitch* performs a sequence of *stitching* operations to produce a single refutation for the original SAT instance. A stitching operation (Algorithm 1) reads in a SAT instance ϕ , a decision variable x and two refutations π and π' corresponding to the sub-problems $\phi \cup \{\{x\}\}$ and $\phi \cup \{\{-x\}\}$ respectively. It produces a single refutation corresponding to the instance ϕ . The refutation for instance ϕ contains the clauses from refutation π appended with the literal $\neg x$ and the clauses from refutation π' appended with the literal x . More generally, the clauses from a refutation are appended with the negation of the decision literal used to generate the sub-problem. Figure 2 illustrates the stitching operation.

As an example of the proof combination process, consider Figure 3. First the refutations π_{00} and π_{01} are combined. Then π_{10} and π_{11} are combined, and finally, π_0 and π_1 are combined to produce the refutation π corresponding to the original instance. In *Proof-Stitch*, the stitching operations are ordered according to the following rule: A stitching operation to combine a pair of refutations π and π' can only occur after all refutations with greater depth have been combined. Informally, this means that refutations are combined in decreasing order of their depth, as shown in Figure 3. Stitching operations at the same depth are independent and can occur in parallel.

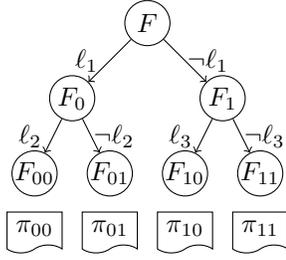


Fig. 1: Decision tree of an example unsatisfiable SAT instance.

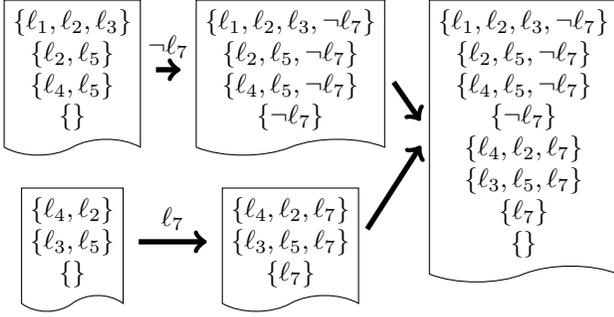


Fig. 2: Stitching operation on example refutations

B. Justification for the stitching operation

We now show that Algorithm 1 is correct: given suitable inputs, it produces a DRAT refutation for ϕ .

Definition 1. A DRAT refutation π is **preserving** if for all C , (\ominus, C) occurs at most as many times in π as (\oplus, C) .

Lemma 1. Let ϕ be a CNF formula, x be a variable, and π and π' be preserving DRAT refutations of $\phi \cup \{\{x\}\}$ and $\phi \cup \{\{\neg x\}\}$ respectively. Then, $\text{stitching}(\phi, x, \pi, \pi')$ outputs a preserving DRAT refutation of ϕ .

Proof. Let π^* be the output of *stitching*. Let $\pi = ((o_1, C_1), \dots, (o_n, C_n))$ and $\pi' = ((o'_1, C'_1), \dots, (o'_n, C'_n))$. Let $\psi = \phi \cup \{\{x\}\}$ and $\psi' = \phi \cup \{\{\neg x\}\}$. Define ψ_i recursively, by $\psi_0 = \psi$ and $\psi_{i+1} = \psi_i \cup \{C_{i+1}\}$ when o_{i+1} is an addition, and $\psi_{i+1} = \psi_i \setminus \{C_{i+1}\}$ otherwise. Define ψ'_i (respectively ϕ_i) analogously, based on formula ψ' (resp. ϕ) and refutation π' (resp. π^*).

By construction, π^* 's final step is (\oplus, \emptyset) . Moreover, since π and π' are preserving and formulas are clause *multisets*, π^* is preserving. Thus, our main task is to show that each addition (\oplus, C_{i+1}^*) in π^* has RAT with respect to ϕ_i . C_{i+1}^* is either derived from a clause in π , derived from a clause in π' , or is the final empty clause. We begin with the first case: $C_{i+1}^* = C_{j+1} \cup \{\neg x\}$.

First, we show that if C_{j+1} has AT with respect to ψ_j , then C_{i+1}^* has AT with respect to ϕ_i . Note that $\psi_j \cup \{\{\neg l_1\}, \dots, \{\neg l_k\}\} = F' \cup \{\{x\}\} \cup \{\{\neg l_1\}, \dots, \{\neg l_k\}\} \rightarrow_x F'' \cup \{\{x\}\} \cup \{\{\neg l_1\}, \dots, \{\neg l_k\}\} \mapsto \perp$. Now, consider $F''' = \phi_i \cup \{\{x\}, \{\neg l_1\}, \dots, \{\neg l_k\}\}$. If $F''' \mapsto \perp$, then C_{i+1}^* has the desired property. Observe that $F''' \rightarrow_x F'' \cup \{\{x\}\} \cup$

$\{\{\neg l_1\}, \dots, \{\neg l_k\}\}$; thus, since the latter propagates to bottom, F''' does too.

Second, we show that if C_{j+1} has RAT with respect to literal l and formula ψ_j , then $C_{i+1}^* = \{\neg x\} \cup C_{j+1}$ has RAT with respect to literal l and formula ϕ_i . Let C^* be a clause in ϕ_i that contains $\neg l$. If $C_{i+1}^* \cup (C^* \setminus \{\neg l\})$ has AT with respect to ϕ_i , we are done. Since C_{i+1}^* is a clause in ϕ_i , there is some C in ψ_j such that $C \cup \{\neg x\} = C^*$ or $C = C^*$. Thus, $C_{i+1}^* \cup (C^* \setminus \{\neg l\}) = \{\neg x\} \cup C_{j+1} \cup (C \setminus \{\neg l\})$. Let $\neg x, l_1, \dots, l_k$ be the literals of this clause. As before, since $\psi_j \cup \{\{\neg l_1\}, \dots, \{\neg l_k\}\}$ unit propagates to bottom, $\phi_i \cup \{\{x\}, \{\neg l_1\}, \dots, \{\neg l_k\}\}$ does too.

In the case that $C_{i+1}^* = C'_{j+1} \cup \{x\}$ (i.e., C_{i+1}^* is derived from π'), the argument is similar. The key insight is that an initial propagation on $\neg x$ in any AT check removes all the clauses added by π . Since π deletes no clauses from the original formula, this leaves an intermediate propagation result that shows C'_{j+1} is RAT.

The final step in π^* is (\oplus, \emptyset) . It has AT because ϕ_{n+m} contains both $\{x\}$ and $\{\neg x\}$. Since π^* 's added clauses all have the AT or RAT properties, and the final step adds an empty clause, π^* is a valid DRAT refutation of ϕ . \square

In *Proof-Stitch*, the final refutation is built through stitching operations on DRAT refutations of the sub-problems. Since each stitching operation produces a preserving DRAT refutation, recursive application of Lemma 1 proves that the final refutation is a valid DRAT refutation of the original instance.

C. Optimization

Empirically, we have observed that refutations created through stitching operations contain a large number of clauses that are not needed during validation ("redundant" clauses). Identifying and removing these clauses reduces the time required to check the refutation and the storage space required to save the refutation. One approach to remove such redundant clauses is by identifying the "unsatisfiable core" as described in [24]. This approach optimizes the refutation by only retaining clauses that are essential for validation by a proof-checker. Our implementation optimizes refutations by using *drat-trim* to extract the unsatisfiable core after every stitching operation.

However, aggressively invoking the optimization technique (e.g., after every stitching operation) could incur significant runtime overhead in the refutation generation process. This calls for a heuristic to decide when to apply the optimization technique. Empirically we observe that refutations with larger clauses (more literals) require longer to check. We hypothesize that this occurs because larger clauses are less likely to contribute to unit-propagation while simultaneously consuming more memory in the cache of the refutation checker. Therefore, optimizing refutations with large clauses should yield the greatest benefit. To implement this, we introduce a threshold parameter CL_{avg} . After each stitching step, the refutation is optimized only if the average clause length in the refutation is greater than CL_{avg} .

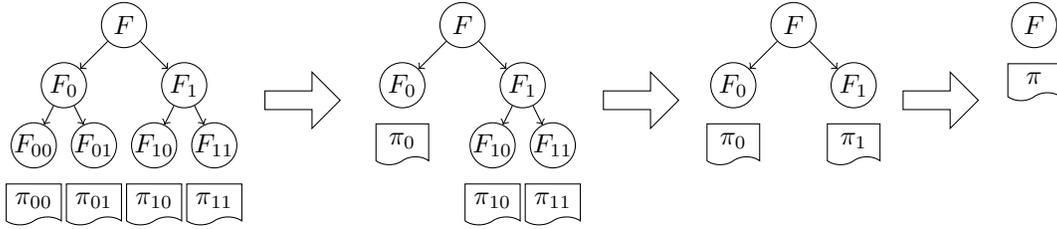


Fig. 3: Refutation stitching process for the SAT instance shown in Figure 1. The decision literals are omitted.

IV. IMPLEMENTATION

In this section, we describe our implementation of the *Proof-Stitch* algorithm. *Proof-Stitch* is implemented in Python and uses *drat-trim* [12] to optimize refutations. Our tool comprises of just under 300 lines of Python code and is available on GitHub [11].

The tool inputs are the original SAT instance in CNF form, the refutations and cubes for each sub-problem, and the threshold value CL_{avg} . Our implementation requires that the cube of each sub-problem be encoded in the name of the corresponding refutation file. For example, the refutation file corresponding to refutation π_{00} in Figure 1 is named $\ell_1\text{-}\ell_2\text{-proof}$. The output is a single file containing a refutation of the original instance. As noted in section III, stitching operations at the same depth of the decision tree are independent and their combined refutations can be optimized in parallel. Our tool supports this. Setting the parameter $CL_{avg} = 0$ enables optimization after every stitching operation and $CL_{avg} = -1$ turns off optimization (only stitching is performed). We denote refutations combined with $CL_{avg} = 0$ as "fully optimized" and refutations combined with $CL_{avg} = -1$ as "unoptimized".

V. EXPERIMENTS

To evaluate *Proof-Stitch*, we run it on six benchmarks from the parallel track of last year’s SAT competition [13]. The chosen benchmarks can be solved by *Paracooba* [21] within 1 minute of run-time. We also attempted running the tool on harder instances from the parallel track. While unoptimized proofs can be produced quickly (within a few minutes) on those instances, proof-checking and optimization are both computationally prohibitive due to the limitation of the underlying proof-checker (e.g., *drat-trim* fails to validate the combined refutations on harder instances even with a 24 hour time limit). For large refutations, the proof-checker faces memory and run-time bottlenecks on almost all the intermediate optimization steps. Therefore, we do not consider harder instances in our evaluation, but note that the proposed techniques in principle apply to larger instances once the scalability of the underlying proof-checker improves.

In our experiments, we compare the checking time and size of unoptimized refutations against fully optimized refutations to show the benefit of optimization. We also report the tool run-time to demonstrate that *Proof-Stitch* does not introduce unacceptable overheads. Finally, we analyze the average checking time and tool run-time for $CL_{avg} = 10$, a value

TABLE 1: Refutation checking time (T_c) (s), tool run-time (T_g) (s), and size of refutation file (S_g) (MB) for six benchmarks from last year’s SAT competition [13]

Benchmarks	Un-optimized			Fully Optimized		
	T_c (s)	T_g (s)	S_g (MB)	T_c (s)	T_g (s)	S_g (MB)
p01_lb_05	987	271	1700	141	686	184
kf_TF-4.tf_2_0.02_18	212	78	385	76	600	77
satch2ways12u	1370	275	1600	272	836	655
pb_300_10_lb_06	163	107	536	36	459	27
mp1-Nb6T06	241	106	586	44	201	222
E02F17	417	223	1500	112	467	294

empirically determined to perform well. We perform our evaluation on an Intel Xeon E5-2640 v3 machine with 128 GBytes of DRAM and 16 cores.

Table 1 shows the time required for *drat-trim* to check the final refutations for the benchmarks (T_c), tool execution time to combine refutations (T_g), and the size of the combined refutations (S_g). The time required to check refutations reduces by between $(2.7 - 7)\times$ for all the benchmarks when full optimization is performed. Full optimization also results in smaller refutation file sizes, but increases the tool run-time.

Figure 4 compares the average run-time to combine refutations (denoted “merging” time) and the average run-time to check refutations for unoptimized, $CL_{avg} = 10$, and fully optimized refutations. Interestingly, running our tool with $CL_{avg} = 10$ decreases the total validation time (merging + checking) compared to the unoptimized case. This points to the benefit of optimizing refutations in parallel—the overhead associated with optimizing refutations can be amortized by the savings in refutation checking time. Another important observation is that setting $CL_{avg} = 10$ reduces the time required to combine refutations compared to the unoptimized case. We believe the reason is as follows: optimizing refutations decreases their size. When $CL_{avg} = 10$, we optimize all intermediate refutations with average clause length greater than 10. Since the intermediate refutations are now smaller, the next stitching operation on this refutation takes lesser time. The time spent in optimizing refutations is mitigated by the savings in stitching time.

VI. CONCLUSION

We have presented *Proof-Stitch*, a technique that complements Divide-and-Conquer SAT solvers by combining sub-problem refutations into a single refutation for the original

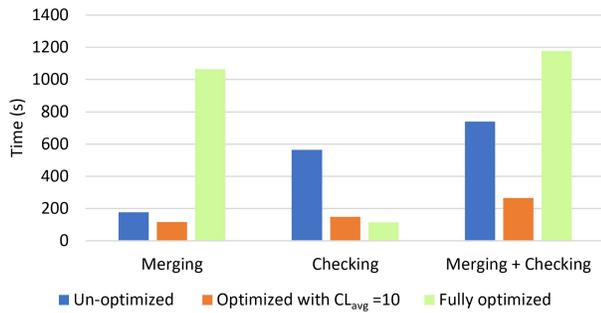


Fig. 4: Average merging time and refutation checking time when the refutations are not optimized, optimized with $CL_{avg} = 10$ and fully optimized

instance. *Proof-Stitch* also uses existing proof-trimming tools to optimize the combined refutation.

Future Work: *Proof-Stitch*'s run-time overhead can be reduced by performing more stitching operations in parallel. Currently, only stitching operations at the same tree depth are parallelized, while in principle, any two independent stitching operations could be parallelized. Another potential future direction would be to incorporate parallelism in the refutation checker itself, likely requiring extension of the DRAT format to incorporate structural information of the search tree. Finally, it would be interesting to evaluate alternative measures for guiding the optimization process, such as Literal Block Distance [25], and to look into additional ways to reduce refutation sizes.

Acknowledgement: This work began as a course project for Caroline Trippel's CS357S (Fall 2021) at Stanford University.

REFERENCES

- [1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.
- [2] A. Graça, J. Marques-Silva, I. Lynce, and A. L. Oliveira, "Efficient haplotype inference with pseudo-boolean optimization," in *Proceedings of the 2nd International Conference on Algebraic Biology*, ser. AB'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 125–139.
- [3] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, 1992, pp. 359–363.
- [4] W. Blochinger, C. Sinz, and W. Küchlin, "Parallel propositional satisfiability checking with distributed dynamic learning," *Parallel Computing*, vol. 29, no. 7, pp. 969–994, 2003.
- [5] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä, "Partitioning sat instances for distributed solving," in *Logic for Programming, Artificial Intelligence, and Reasoning*, C. G. Fermüller and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 372–386.
- [6] A. E. Hyvärinen, T. Junttila, and I. Niemelä, "A distribution method for solving sat in grids," in *International conference on theory and applications of satisfiability testing*. Springer, 2006, pp. 430–435.
- [7] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.
- [8] A. Ozdemir, H. Wu, and C. Barrett, "Sat solving in the serverless cloud," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 241–245.
- [9] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.
- [10] M. J. Heule and A. Biere, "Proofs for satisfiability problems," *All about Proofs, Proofs for all*, vol. 55, no. 1, pp. 1–22, 2015.
- [11] "Proof-stitch," <https://github.com/abhisheknaair1729/Proof-Stitch/commit/d93a0c33b6114044413eb22962c677b06308b00e>, 2022.
- [12] N. Wetzler, M. J. Heule, and W. A. Hunt, "Drat-trim: Efficient checking and trimming using expressive clausal proofs," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2014, pp. 422–429.
- [13] "Sat competition 2021," <https://satcompetition.github.io/2021/>, 2021.
- [14] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability: Second Edition*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. [Online]. Available: <https://books.google.com/books?id=dUAvEAAAQBAJ>
- [15] M. Heule, M. Järvisalo, and A. Biere, "Clause elimination procedures for cnf formulas," in *LPAR*, 2010.
- [16] H. Zhang, M. P. Bonacina, and J. Hsiang, "Psato: a distributed propositional prover and its application to quasigroup problems," *Journal of Symbolic Computation*, vol. 21, no. 4, pp. 543–560, 1996.
- [17] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon, "Modular and efficient divide-and-conquer sat solver on top of the painless framework," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 135–151.
- [18] S. Nejati, Z. Newsham, J. Scott, J. H. Liang, C. Gebotys, P. Poupart, and V. Ganesh, "A propagation rate based splitting heuristic for divide-and-conquer solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2017, pp. 251–260.
- [19] M. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 7261. Springer, 2011, pp. 50–65.
- [20] M. J. H. Heule, O. Kullmann, and V. W. Marek, "Solving and verifying the boolean pythagorean triples problem via cube-and-conquer," in *SAT*, ser. Lecture Notes in Computer Science, vol. 9710. Springer, 2016, pp. 228–245.
- [21] M. Heisinger, M. Fleury, and A. Biere, "Distributed cube and conquer with paracooba," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2020, pp. 114–122.
- [22] M. J. Heule and A. Biere, "Compositional propositional proofs," in *Logic for Programming, Artificial Intelligence, and Reasoning*, 2015.
- [23] T. Philipp, "Unsatisfiability proofs for parallel sat solver portfolios with clause sharing and inprocessing," in *GCAI*, 2016, pp. 24–38.
- [24] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for cnf formulas," in *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 886–891.
- [25] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.

Reconciling Verified-Circuit Development and Verilog Development

Andreas Lööw
Imperial College London
London, UK

Abstract—In software development, verified compilers like the CompCert compiler and the CakeML compiler enable a methodology for software development and verification that allows software developers to establish program-correctness properties on the verified compiler’s target level. Inspired by verified compilers for software development, the verified Verilog synthesis tool Lutsig enables the same methodology for Verilog hardware development. In this paper, we address how Verilog features that must be understood as *hardware constructs*, rather than as *software constructs*, fit into hardware development methodologies, such as Lutsig’s, inspired the development methodology enabled by software compilers. We explore this issue by extending the subset of Verilog supported by Lutsig with one such feature: `always_comb` blocks. In extending Lutsig’s Verilog support with this, seemingly minor, feature, we are, perhaps surprisingly, required to revisit Lutsig’s methodology for circuit development and verification; this revisit, it turns out, requires reconciling traditional Verilog development and the traditional program-verification methodology offered by verified software compilers. All development for this paper has been carried out in the HOL4 theorem prover.

Index Terms—hardware development, hardware synthesis, Verilog

I. INTRODUCTION

In software development, verified compilers enable the following interactive-theorem-proving-based verified-program development (VPD) methodology:

- 1) *develop and compile* your program in the same way as when using an unverified compiler;
- 2) *prove* a source-level correctness theorem about your program (by whatever means you have available – the methodology is independent of how the correctness theorem is established); and, lastly,
- 3) *transport* the source-level program-correctness theorem down to your verified compiler’s target level by simple composition of the source-level program-correctness theorem and the compiler’s (program-independent) correctness theorem.

VPD has been successfully deployed in many different software contexts, such as e.g. imperative programming [1], functional programming [2], concurrent programming [3], just-in-time compilation [4], [5], compiler-implementation correctness (by compiler bootstrapping) [2], [6], usability such as compositional/separate compilation [7], security such as constant-time preservation [8], and performance such as time/space reasoning [9]–[11].

In this paper, however, our interest lies in hardware development rather than software development. Previous work on verified hardware-synthesis tools [12]–[15] – also known as hardware compilers – show that VPD is equally applicable to hardware contexts, thereby providing a methodology for circuit development and verification. In this paper, we augment existing work on VPD in hardware contexts by considering source-level language Verilog features that must be understood as *hardware constructs* rather than as *software constructs*.

To handle such hardware constructs, we propose a hardware development methodology combining VPD and traditional Verilog development (TVD). While radical methodological redesign is certainly a worthwhile enterprise [16]–[26], we here dedicate our energy towards an enterprise in which we want to maintain as much as possible of the look-and-feel of both VPD and TVD. Specifically, as we further elaborate in the next section (Sec. II), we want to maintain both (1) VPD’s ability to transport source-level correctness theorems down to the compiler’s target level and (2) TVD’s synthesis-modeling-idiom-based approach to synthesis.

We validate the proposed methodology combining VPD and TVD by adapting and extending Lutsig [14], a verified synthesis tool for synchronous Verilog designs, for the methodology. Specifically, we extend Lutsig’s Verilog support with one of Verilog’s features that must be understood as a hardware construct: `always_comb` blocks, which allows hardware designers to declare that certain parts of their behavioral Verilog code are to be synthesized to combinational logic. Combinational logic is stateless logic and stands in contrast to sequential logic (modeled as e.g. `always_ff` blocks), which is stateful logic.

All in all, we make the following two contributions:

- We propose a development methodology combining VPD, i.e. the traditional development methodology based on verified compilers, and TVD, i.e. traditional Verilog development, in a way that inherits the strengths of both and simultaneously avoids their main weaknesses.
- We validate the methodology by showing that it allows us to add support for `always_comb` blocks to Lutsig, the Verilog semantics used in Lutsig, and a proof-producing Verilog code generator connected to Lutsig.

All the work for this paper has been carried out in the HOL4 theorem prover [27]. All source code and proofs are available at <https://github.com/CakeML/hardware>.

II. BACKGROUND: VPD AND TVD

This section serves two purposes: firstly, it introduces VPD and TVD in more detail, and, secondly, it establishes notation and terminology used in the rest of the paper.

A. Verified-program development (VPD)

We now give a more detailed description of VPD, following the exposition of Leroy [1]. In VPD, we start off with a source program P_S implemented in a source language S and a compiled program P_T implemented in a target language T produced by a compiler: $Comp P_S = \text{OK } P_T$. If the compiler is unable, for whatever reason, to compile P_S , then a compile-time error is reported: $Comp P_S = \text{ERROR}$. The source language S has a semantics L_S , and the target language T has a semantics L_T . The two semantics L_S and L_T associate sets of observable behaviors B to source and target programs. We write $P \Downarrow_L B$ to denote that a program P executes with observable behavior B under semantics L .

We say that a compiler $Comp$ is verified when we have proved $\forall P_S P_T, Comp P_S = \text{OK } P_T \implies P_S \approx P_T$ for some notion of semantic preservation \approx . The only notion of semantic preservation we use in this paper is backward simulation: $P_S \approx P_T \iff \forall B, P_T \Downarrow_{L_T} B \implies P_S \Downarrow_{L_S} B$; that is, any behavior of the target program must be a behavior allowed by the source semantics.

Compiler users, however, are not ultimately interested in the correctness of the compiler $Comp$ they are using; rather, when compiling a source program P_S with a compiler, users are ultimately interested in the correctness of the target program P_T produced by the compiler. This is, of course, also part of VPD. Since it is easier to prove the correctness of P_S and *transport* the result to P_T than it is to prove the correctness of P_T directly, VPD is as follows: Following Leroy’s exposition, users are asked to formalize what they mean by their program being correct by providing a predicate $Spec$ over observable behaviors. We write $P \models_L Spec$ for $\forall B, P \Downarrow_L B \implies Spec B$. Now, for a successful compiler run $Comp P_S = \text{OK } P_T$, if the user’s compiler $Comp$ has been verified (with backward simulation as the notion of semantic preservation), then the user can derive $P_T \models_{L_T} Spec$ (i.e., what the user is ultimately interested in) from $P_S \models_{L_S} Spec$ by simple composition.

B. Traditional Verilog development (TVD)

We now turn to TVD. As Weste and Harris [28, p. 699] put it, hardware description languages (HDLs) like Verilog are “better understood as shorthand for describing digital hardware” than programming languages. Continuing, Weste and Harris describe TVD as follows:

- 1) “[...] begin your design process by planning, on paper or in your mind, the hardware you want.”
- 2) “Then, write the HDL code that implies that hardware to a synthesis tool.”

In TVD, an important concept is *modeling idioms*, which enable the hardware designer to express not only the behavior of their design but *what kind of hardware they want*. Modeling idioms are what allow the hardware designer to write Verilog

code that “implies” the hardware design the hardware designer has formed “on paper or in [their] mind.”

Examples of modeling idioms include e.g. `always_ff` and `always_comb` blocks, allowing hardware designers to specify if sequential or combinational logic should be inferred by the synthesis tool. In general, what modeling idioms are available depends on what technology is targeted. E.g., the synthesis manual for Xilinx’s (unverified) synthesis suite Vivado [29, p. 111] contains modeling idioms and guidelines for modeling block RAMs (BRAMs), a type of memory available in Xilinx FPGAs. The modeling idioms related to BRAMs are presented as Verilog design fragments, instructing the hardware designer how to write their Verilog code such that the synthesis tool will infer features such as write enable inputs, byte-write-enable inputs, optional output registers, etc.

III. RECONCILING VPD AND TVD

Having introduced both VPD and TVD, we are now in a position to combine the best of two worlds: we want the methodology for circuit development and verification offered by Lutsig to provide the strengths of both VPD, i.e., theorem transportation, and TVD, i.e., synthesis-tool control by modeling idioms.

As a first step, as we want to apply the VPD methodology to Verilog hardware development, we must specialize $Comp, S, L_S, T,$ and L_T to appropriate hardware instances. Since we, in this paper, are working with Lutsig, we set: $Comp = \text{Lutsig}$, $S = \text{Verilog}$ (abbreviated “ver”), and $T = \text{technology-mapped netlists for (a class of) FPGAs}$ (abbreviated “nl”). For L_T , Lutsig uses a simple netlist language. What remains to specify is L_S – and this is where our problems begin.

The problems surrounding L_S arise from the fact that, traditionally conceived, Verilog has two semantics: one *simulation semantics* and one *synthesis semantics*. The reason for having two semantics, we will see, is TVD. This, however, does not fit cleanly into VPD since in VPD the source language S is supposed to have *one and only one* semantics L_S ; since otherwise theorem transportation cannot be carried out by simple composition.

We now discuss the two semantics in the context of synthesis tool design and how they relate and fit into VPD and TVD. We first introduce the two semantics, we then survey the state of the art, and then conclude by stating how our development methodology – combining VPD and TVD – as implemented in Lutsig contributes to the state of the art.

Simulation semantics. The simulation semantics is given by the (System)Verilog standard [30]. The semantics is large, complicated, and full of gotchas [31], but at the end of the day, is an informally specified event-based operational semantics.

Synthesis semantics. The situation for the synthesis semantics is less straightforward.

Firstly, one minor hurdle to overcome is that the authoritative source for the semantics is unclear. Since the Verilog standard does not provide a synthesis semantics and the Verilog synthesis standard [32] has been withdrawn, it is up to each synthesis tool to provide their own synthesis semantics. Current tool-specific

synthesis manuals, such as e.g. the synthesis manuals for Vivado [29] and Quartus [33], however, largely contain similar material as the withdrawn synthesis standard (similar modeling idioms, design and coding-style recommendations, etc.), except specified in a more detailed fashion since such manuals are both tool- and target-technology-specific. We therefore use the withdrawn Verilog synthesis standard as the basis for our discussion here.

Secondly – the major hurdle – the synthesis semantics, both as specified in the synthesis standard and the tool-specific synthesis manuals, is not a full semantics like the simulation semantics; rather, it is just a collection of modeling idioms and design recommendations built on top of the simulation semantics. This ends up causing problems since some of the modeling idioms prescribe semantics incompatible with the simulation semantics: specifically, some of the modeling idioms have not only *nonfunctional consequences* but also *functional consequences*; in other words, some modeling idioms have consequences for the (functional) behavior of synthesized circuits! In TVD, the problems this causes are known as *simulation-and-synthesis mismatches*. Some mismatches are highlighted in (the informative) App. B in the synthesis standard. E.g., we are warned that the following module¹ will cause a simulation-and-synthesis mismatch since the assignments to `y` and `tmp` are “mis-ordered” (since the block is supposed to describe combinational logic – that is, stateless logic – and `tmp` is read before being assigned):

```
module andor1b(output reg y, input a, b, c);
  reg tmp;

  always @* begin
    y = tmp | c;
    tmp = a & b;
  end
endmodule
```

State-of-the-art VPD. To some extent, VPD and TVD were reconciled already in the first version of Lutsig. However, except for `X` assignments, which, according to the synthesis standard, “tells the simulator to treat the signal as having an unknown value and tells the synthesis tool to treat the signal as a don’t care” [32, p. 106], not much attention was directed towards simulation-and-synthesis mismatches. This was because the supported subset of Verilog was sufficiently small and software-like that the parts of Verilog that risk causing simulation-and-synthesis mismatches were, in effect, avoided.²

Now, on the other hand, when adding support for `always_comb` to Lutsig, i.e., a feature that must be understood as a *hardware construct* rather than as a *software construct*, i.e., a feature that must be understood in terms of modeling idioms, further reconciliation between VPD and TVD is needed. At the same time, we should acknowledge that problems similar

¹Here presented verbatim, using an `always @*` block rather than an `always_comb` block since the synthesis standard was published before the first SystemVerilog standard – the synthesis standard based on the Verilog 2001 standard [34].

²Clearly, a discussion concluding “Lutsig takes Verilog’s simulation semantics as its synthesis semantics” [14, p. 50] is insufficient for handling `always_comb` blocks.

to our present problems can be found in software development as well. E.g., one aspect of what has happened is that we have ended up with *nonfunctional expectations* on our synthesis tool – and VPD, in its minimal incarnation, only covers *functional expectations*, specifically semantics preservation. Nonfunctional expectations are, of course, sometimes put on software compilers [35], since functional software-compiler guarantees say (most commonly) nothing about code size, memory usage, cache performance, overall performance, security, etc. Indeed, some of the software VPD work mentioned in the introduction provide examples of VPD work addressing nonfunctional properties, such as security [8] and space reasoning [9].

Another point of comparison is how so-called undefined behavior (UB) is handled in languages such as C [36], [37]. UB leaves some parts of the language in question left with unspecified semantics (to allow for compiler optimizations). UB forms a subset of the language to avoid. Simulation-and-synthesis mismatches are similar to UB in the sense that sources of such mismatches can be seen as parts of Verilog to avoid. However, the two are not equivalent since the concept that induces simulation-and-synthesis mismatches, modeling idioms, has no analog in UB-based approaches to language semantics.

Recall that we aim to keep the look-and-feel of TVD in Lutsig’s combination of VPD and TVD. We therefore must include modeling idioms in Lutsig’s synthesis methodology rather than try to formulate a synthesis story under a – potentially more familiar for software developers – UB framework.

State-of-the-art TVD. Today’s commercial (unverified) synthesis tools leave much to be desired; within the same tool, simulation-and-synthesis mismatches are handled along the whole spectrum of: silently miscompiling Verilog designs, issuing warnings, and aborting the compilation process entirely. In consequence, the result of a successful synthesis run is unclear for hardware developers: since an error-free synthesis run does not guarantee an actually successful synthesis run, some form of postsynthesis inspection, e.g. testing or manual visual inspection, is needed to ensure that the functional and nonfunctional properties we are interested in survived or were established during synthesis.

Lutsig’s methodology. The conclusion we draw from the above discussion is that, to handle both TVD and VPD, Lutsig must implement both Verilog’s semantics: the simulation semantics for VPD-style theorem transportation, and the synthesis semantics, in the form of synthesis idioms, for synthesis-idiom-based TVD.

In Lutsig, TVD is handled on an informal best-effort basis, since strict compliance prohibits too many optimizations, and VPD is handled, as it must, formally.

An interesting question is how much of TVD can be handled formally. For this paper, to illustrate that part of TVD can be treated formally, the feature of focus of this paper, `always_comb` blocks, diverges in Lutsig from the above general pattern of treating TVD informally: we prove that if the two semantics assign different behaviors to an `always_comb` block (e.g., because of “mis-ordered” writes) in a given input design, then Lutsig will abort – since Lutsig cannot abide

by both semantics if they point in different directions. It is Lutsig’s two top-level theorems (Sec. VIII and IX) that together formally show that Lutsig successfully handles both semantics for `always_comb` blocks. We leave the consideration of other synthesis idioms as future work.

Lutsig’s contribution to establishing functional properties.

Like for the first version of Lutsig, we have proved that Lutsig is semantics preserving (Sec. VIII). Specifically, after our discussion, it should now be clear that Lutsig must be semantics preserving with respect to Verilog’s simulation semantics. We call Lutsig’s formalization of the simulation semantics L_{ver} ; i.e., in terms of VPD, we have $L_S = L_{\text{ver}}$. The semantics is the same Verilog semantics used as in the first version of Lutsig, with the exception that we now have added support for `always_comb` blocks (as described in Sec. V).

Since Lutsig allows for VPD development, after the hardware designer has transported a source-level correctness theorem down to the netlist level, the designer can rest assured that the synthesis process has not introduced any functional bugs. For functional correctness, VPD effectively *forces Lutsig to adopt (in stark contrast to other Verilog synthesis tools) a uniform error handling mechanism*: if Lutsig cannot guarantee semantics preservation, it must abort. Like the first version of Lutsig, and other verified compilers and synthesis tools, silent miscompilation is guaranteed to never occur.

Lutsig’s contribution to establishing nonfunctional properties. We improve the state of the art in establishing nonfunctional hardware property by proving that Lutsig’s synthesis algorithm correctly implements the modeling idiom that `always_comb` must generate combinational logic (Sec. IX), i.e., enables proven-correct TVD for `always_comb` blocks. For other modeling idioms, Lutsig does not improve the state of the art with respect to establishing nonfunctional properties.

Other approaches to circuit correctness. The first Lutsig paper [14] compares VPD-style hardware development, as followed here, to other approaches to circuit correctness, such as translation validation (known as formal equivalence checking in the hardware world), so we do not repeat that discussion here.

IV. USING LUTSIG IN PRACTICE

The rest of the paper consists of putting the discussion up till now into practice by adding support for `always_comb` to Lutsig and surrounding components. But before heading into technical details, we show how all pieces of the development fit together by demonstrating how hardware designers can use Lutsig in combination with a proof-producing Verilog code generator, developed in conjunction with Lutsig, to transport correctness properties down to the netlist level.³

³We emphasize that what is demonstrated here is one of multiple potential use cases of Lutsig. Like any Verilog synthesis tool, Lutsig can be made part of different hardware-development flows. In particular, one can imagine many different front-ends capable of generating Lutsig Verilog ASTs and, in various ways, producing proofs of correctness for those ASTs. In this paper, the proof-producing code generator we use fits our purposes here. Someone wanting to verify and synthesize existing Verilog code will have other needs. For developers not interested in verification at all, there is a (unverified) Verilog-text-file front-end for Lutsig available such that Lutsig can be used like a conventional Verilog synthesis tool.

```

module avg(input logic clk,
           input logic[7:0] signal,
           output logic[7:0] avg);

logic[7:0] h0 = 0, h1 = 0, h2 = 0, h3 = 0;

always_ff @(posedge clk) begin
    h0 <= signal; h1 <= h0; h2 <= h1; h3 <= h2;
end

always_comb begin
    avg = h0 + h1 + h2 + h3;

    // Div by 4 by shifting
    avg[0] = avg[2]; avg[1] = avg[3]; avg[2] = avg[4];
    avg[3] = avg[5]; avg[4] = avg[6]; avg[5] = avg[7];
    avg[6] = 0; avg[7] = 0;
end

endmodule

```

Fig. 1. Example Verilog module

Example module. The Verilog module in Fig. 1, implementing a moving-average filter, serves as a running example in this section. The module utilizes Lutsig’s new support for `always_comb` blocks. Sec. V provides more details on Lutsig’s Verilog support.

Proving Verilog designs correct. Lutsig is accompanied by a proof-producing Verilog code generator. The code generator is explained in more detail in Sec. VI. In short, the code generator constructs a Verilog module P_{ver} given a HOL embedding P_{HOL} of a Verilog circuit. As the code generator is proof-producing, the code generator enables hardware designers to transport properties proved about the input HOL circuit P_{HOL} , e.g. $P_{\text{HOL}} \models_{L_{\text{HOL}}} \text{Spec}$, to the generated Verilog module P_{ver} , i.e. $P_{\text{ver}} \models_{L_{\text{ver}}} \text{Spec}$, by simple composition.

The Verilog module in Fig. 1 was in fact generated by the code generator from a HOL circuit. With the help of the code generator, we have proved that, if we by $s[n]$ mean the value of signal s at clock cycle n , the generated Verilog module satisfies the specification (in 8-bit modular arithmetic) $\text{avg}[n] = \frac{\sum_{i=1}^4 \text{signal}[n-i]}{4}$, i.e., the module is correct.

Going to the netlist level. Now having both a Verilog module (Fig. 1) and a correctness result for the module, we can synthesize a netlist implementation of the module, by invoking Lutsig, and transport the correctness result to the netlist implementation, by composing the Verilog-level correctness result with Lutsig’s correctness theorem (i.e., in general notation, derive $P_{\text{nl}} \models_{L_{\text{nl}}} \text{Spec}$ from $P_{\text{ver}} \models_{L_{\text{ver}}} \text{Spec}$). We discuss Lutsig in more detail in Sec. VII and the functional correctness of Lutsig in Sec. VIII. Since the behavior of the variable `avg` is specified using an `always_comb` block, no register should be generated for the variable; this is further discussed in Sec. IX in the context of the nonfunctional correctness property we have proved about Lutsig.

FPGAs. At this point, our formal development ends. To run the netlist implementation produced by Lutsig on an FPGA, the netlist needs to be placed and routed onto an FPGA chip and then encoded into a bitstream for the chip. In our experiments,

we used the unverified synthesis suite Vivado 2020.2 for these last steps. According to our manual testing, the netlist Lutsig synthesizes for the Verilog module in Fig. 1 runs correctly on top of the FPGA board we used for testing.

V. FORMAL SEMANTICS

In this section we first describe the updated source language of Lutsig (Sec. V-A); that is, we describe the subset of Verilog that Lutsig supports and Lutsig’s Verilog semantics L_{ver} for this subset. We then describe the updated target language of Lutsig (Sec. V-B), that is, Lutsig’s netlist language.

A. Lutsig’s Verilog semantics

In Lutsig, circuits are represented as Verilog modules. A Verilog module, in turn, in Lutsig, consists of:

- a set of input signals (including a clock signal `clk`),
- a set of variables, some marked externally visible,
- a set of `always_comb` blocks, and
- a set of `always_ff` @(posedge `clk`) blocks.

Lutsig’s Verilog semantics is a functional operational semantics that takes the following four inputs:

- a Verilog module m to execute,
- the number of clock cycles n to execute the module,
- a function $f_{\text{ext}} : \mathbb{N} \rightarrow \text{string} \rightarrow \text{value}$ modeling snapshots of the nondeterministic world outside the module, and
- a function $f_{\text{bits}} : \mathbb{N} \rightarrow \text{bool}$ modeling a stream of nondeterministic bits⁴.

Since Lutsig’s Verilog must be convenient to use in formal reasoning, Lutsig’s Verilog is not, in contrast to full Verilog, based on nondeterministic event processing. Since Lutsig targets synchronous designs, the complexities of an event-driven semantics can be fully avoided. Of particular interest is the process-level semantics of Lutsig’s Verilog semantics, since the expression-level and statement-level semantics have not been updated for this new version of Lutsig. In short, Lutsig’s Verilog semantics for executing one clock cycle is:

- For clock cycle zero, i.e. before the first clock tick, initialize all variables (for a variable without a specified initial value, assign a nondeterministic value) and then run all `always_comb` blocks in dependency order.
- For all other clock cycles, run all `always_ff` blocks in declaration order followed by all `always_comb` blocks in dependency order.

A module’s `always_ff` blocks are, in Lutsig’s Verilog, executed in declaration order since the order of execution does not affect the final result of execution as long as not more than one process writes to the same variable and all writes to variables that are read by processes other than the process making the writes are nonblocking (a type of assignment used for communication between processes in Verilog).

A module’s `always_comb` blocks are, in Lutsig’s Verilog, executed in dependency order since the order of execution

does matter since blocking writes are used even for variables shared between processes. All `always_comb` blocks are sorted before execution by their variable dependencies in the sense that no process writes to a variable that has been read by an earlier process. If the processes cannot be sorted in this way, the semantics aborts with an error. Sorting the processes complicates the semantics, since a sorting algorithm is embedded into the semantics. (We have, however, proved that the algorithm sorts correctly.) The sorting algorithm picks one particular permutation, but users of the semantics should think of it as an arbitrary permutation of the input `always_comb` blocks that satisfy the mentioned dependency-order criteria.⁵

Our intention is that Lutsig’s non-event-driven Verilog semantics should coincide with the event-driven simulation semantics of full Verilog, as defined by the Verilog standard, as long as good coding style is followed; e.g., as mentioned above, not writing blockingly in an `always_ff` block to a variable shared between processes. As part of future work, we plan to formally prove a correspondence between the two semantics to make the relationship between them more precise. Such future semantics work is important for Lutsig when arguing that Lutsig is a *Verilog* synthesis tool, but such work is simultaneously independent of Lutsig in the sense that it would not require Lutsig’s implementation and proofs to be updated, as long as the work does not unveil problems in the non-event-driven semantics (and hence requiring us to revisit the semantics).

B. Lutsig’s netlist semantics

For this version of Lutsig, to support the compilation of `always_comb` blocks, we split netlist registers into two groups: pseudoregisters and real registers. Pseudoregisters are only needed to represent intermediate compilation results – i.e., pseudoregisters are always compiled away before the compilation process is finished. We explain how pseudoregisters are used in the compilation process in Sec. VII. After adding pseudoregisters, a netlist in Lutsig consists of two lists of cells and two lists of registers: one list of cells for the real registers and one list of cells for the pseudoregisters.

There is a formal semantics in functional-operational style associated with Lutsig’s netlists. The semantics takes the same kind of arguments as Lutsig’s Verilog semantics except a netlist is given rather than a Verilog module. Netlist execution is similar to Lutsig’s Verilog execution. First, we define a netlist step to be running all pseudoregister cells, updating all pseudoregisters, and then running all remaining cells. Now, with this terminology in mind, we can describe the full semantics:

- For clock cycle zero, initialize all registers and then do a netlist step.
- For all other clock cycles, update all real registers and then do a netlist step.

⁴See Löw [14] for a discussion on how X values are treated in Lutsig. We do not repeat the discussion on X values here since such concerns are orthogonal to our current concerns.

⁵Picking one particular permutation rather than an arbitrary permutation simplifies some proofs in the development. But since picking an arbitrary permutation would simplify the user-facing presentation of the semantics, it might be worth revisiting this choice.

It is important that the netlist semantics is simple since the semantics is part of the trusted base of circuits produced with the help of Lutsig. In fact, for netlists without pseudoregisters, such as the final output netlists generated by Lutsig, it is easy to prove that the above semantics collapses into the following clean semantics L_{nl}' :

- For clock cycle zero, initialize all registers and then run all cells.
- For all other clock cycles, update all registers and then run all cells.

VI. THE PROOF-PRODUCING VERILOG CODE GENERATOR

For this paper, we have extended the proof-producing Verilog code generator bundled with Lutsig with support for translating `always_comb` blocks, such that we can prove circuits containing such blocks correct.⁶

The code generator can generate a deeply embedded Verilog circuit given a shallowly embedded Verilog circuit. To shallowly embed a Verilog circuit means to express it as a HOL function (i.e., a functional program). Shallowly embedded circuits are convenient to work with since HOL4 has well-developed infrastructure for reasoning about functional programs. The code generator is an SML function which is proof-producing in the sense that it, for every run, proves a HOL theorem (using the HOL4 API) ensuring that the input circuit and output circuit have the same behavior.

Since the input language to the code generator is Verilog, although shallowly embedded, there is no need to provide a new set of hardware-modeling idioms (i.e., a new synthesis semantics) for the input language. In other words, the input circuits should be seen as Verilog circuits, and, when shallowly embedding Verilog circuits, according to the style the code generator expects, the hardware developer should think of themselves as doing Verilog development.

The code generator assumes that circuits are embedded in the style we now describe. Verilog processes must be embedded as next-state functions over (module-specific) state records. For each process, the generated Verilog code closely mirrors the given input HOL function. E.g., recall that the `always_ff` block in the Verilog module in Fig. 1 is simply “`h0 <= signal; h1 <= h0; h2 <= h1; h3 <= h2;`”; the next-state function the block is generated from is:

$$\begin{aligned} \text{avg_ff } fext \ s \ s' &\stackrel{\text{def}}{=} \text{let} \\ s' &= s' \text{ with } h0 := fext.\text{signal}; \\ s' &= s' \text{ with } h1 := s.h0; \\ s' &= s' \text{ with } h2 := s.h1 \text{ in} \\ s' &\text{ with } h3 := s.h2 \end{aligned}$$

Note how field updates are translated to assignments in Verilog in a straightforward manner (the syntax r with $f := v$ means that field f of record r is updated to value v). Also note how two state records s and s' are passed around; these two state records are the basis of the nonblocking-assignments embedding style

⁶Unrelatedly, we have also changed how nonblocking assignments are shallowly embedded, such that a larger set of Verilog designs can be embedded.

used. The record s contains the values of all variables at the start of the current clock cycle, and the record s' contains the current values of all variables. To see why both records are needed, consider e.g. the assignments to `h0` and `h1` in the generated `always_ff` block: since the assignment to `h0` is nonblocking, the updated value of `h0` is not available until the next clock cycle, and the HOL embedding of the `h1` assignment must therefore read the value of `h0` from the s record (not the s' record) to model Verilog’s semantics correctly.

The rest of the HOL circuit embedding style closely mirrors Lutsig’s Verilog semantics. First, there is a function

$$\begin{aligned} \text{procs } [] \ fext \ s \ s' &\stackrel{\text{def}}{=} s' \\ \text{procs } (p::ps) \ fext \ s \ s' &\stackrel{\text{def}}{=} \text{procs } ps \ fext \ s \ (p \ fext \ s \ s') \end{aligned}$$

for combining a list of next-state functions into one single next-state function. The function allows for building one next-state function for all `always_ff` blocks in the module and one next-state function for all `always_comb` blocks. One important caveat is that the `always_comb` blocks must be provided in dependency order, otherwise the HOL circuit will not correctly mirror Lutsig’s Verilog semantics since Lutsig’s Verilog semantics sorts all `always_comb` blocks by dependency before execution. The resulting two next-state functions formed by composing all `always_ff` blocks and `always_comb` blocks, respectively, using `procs`, can then be given to the following function, also mirroring Lutsig’s Verilog semantics, to build a full circuit:

$$\begin{aligned} \text{mk_circuit } sstep \ cstep \ s \ fext \ 0 &\stackrel{\text{def}}{=} cstep \ (fext \ 0) \ s \ s \\ \text{mk_circuit } sstep \ cstep \ s \ fext \ (\text{Suc } n) &\stackrel{\text{def}}{=} \text{let} \\ s &= \text{mk_circuit } sstep \ cstep \ s \ fext \ n; \\ s &= sstep \ (fext \ n) \ s \ s \ \text{in} \\ cstep \ (fext \ (\text{Suc } n)) \ s \ s \end{aligned}$$

E.g., the HOL representation of the Verilog module in Fig. 1 is `mk_circuit (procs [avg_ff]) (procs [avg_comb])`.

Lastly, one more level of encoding is needed to handle variable initialization, which is simple and we do not detail here.

VII. LUTSIG

We now discuss Lutsig’s new support for `always_comb` blocks. To simultaneously honor both Verilog’s simulation semantics and Verilog’s synthesis semantics – in this paper, specifically, for the latter, the modeling idiom that `always_comb` blocks must always be mapped to combinational logic – Lutsig must take on the responsibility to abort if the two semantics differ in what semantics they assign to some `always_comb` block in a given design. In this section, we discuss how Lutsig implements this responsibility. In Sec. VIII, we show that Lutsig successfully achieves its responsibility towards Verilog’s simulation semantics, by presenting a theorem stating that Lutsig is semantics preserving with respect to Lutsig’s formalization of Verilog’s simulation semantics. In Sec. IX, we show that Lutsig successfully achieves its responsibility towards Verilog’s synthesis semantics (for `always_comb` blocks), by presenting a theorem stating that `always_comb` blocks are never be mapped to registers (or other stateful constructs).

Concretely, the above responsibility boils down to ensuring that there is no sequential logic inside any `always_comb` block. This is where pseudoregisters come in: all variables written to by an `always_comb` block are mapped to pseudoregisters, and all other variables are mapped to real registers. All pseudoregisters must then be compiled away before the synthesis process is over, otherwise Lutsig aborts with an error.

A. Variable-level and element-level analysis

To keep the implementation of Lutsig simple, the decision whether to map a variable to a pseudoregister or a real register is done on the level of variables. E.g., all elements of an array variable are either all mapped to pseudoregisters or to real registers. In full Verilog, the analysis is instead based on longest static prefixes [30, p. 282]. Such more fine-grained analysis allows for different parts of an array to be mapped to different kinds of logic, which could possibly be practically useful, but would clutter the solution presented here without providing additional insight.

Note, however, that some amount of element-level analysis is still needed. E.g., consider a module containing only one variable `a` with type `logic[1:0]` and the following block:

```
always_comb begin
  a[0] = inp0;
  a[1] = inp1;
end
```

The block represents combinational logic since all elements of the array are assigned. But if one of the assignments would have been left out, then the block would not represent combinational logic. Hence, an analysis on the element level cannot be fully avoided.

B. Lutsig’s synthesis passes

In Lutsig, pseudoregisters are removed at a late stage in the synthesis pipeline. The following pipeline passes in Lutsig are important for our discussion here:

- SYNT Synthesize the given Verilog design to a netlist
- REM Remove unused registers (variable-level analysis)
- DET Remove all nondeterminism from the netlist
- MAP Compile and technology-map away array cells
- REM Remove unused registers (element-level analysis)

Pseudoregisters are introduced in SYNT and not removed until MAP. Since MAP is done on the element level (rather than the variable level as the passes before it), it was natural to place the removal of pseudoregisters there. The downside of this approach is that we had to update *all* intermediate passes of Lutsig, such as REM and DET, to handle the more complex netlist semantics with pseudoregisters. (Note that REM is run twice, which we motivate in the next section.)

C. Problems in compiling combinational logic

We now highlight how Lutsig handles some of the problems related to compiling combinational logic. Our presentation is example driven and many of the examples relate to detecting simulation-and-synthesis mismatches. It is important to consider not only designs that are rejected by Lutsig but also designs

that are accepted, since compiler-correctness theorems like Lutsig’s (of the form $Comp P_S = OK P_T \implies P_S \approx P_T$) do not protect against compiler bugs that cause compilers to fail on valid input code (i.e., bugs causing the compiler to return `Error` when it should have returned `OK`). To exemplify, consider the extreme case of a compiler that always returns `Error`: such a compiler is vacuously correct, but, of course, not particularly useful.

1) *Combinational logic in `always_ff` blocks*: Code inside `always_comb` blocks must always represent combinational logic only, but code inside `always_ff` blocks can represent both combinational and sequential logic. E.g., consider a module consisting of three variables `a`, `b`, and `c` with type `logic[1:0]` with one single block:

```
always_ff @(posedge clk) begin
  a = inp0;
  b[0] = inp1;
  b[1] = inp2;
  c <= a + b;
end
```

Such code should not generate registers for `a` and `b` since those registers would never be read. REM makes sure the registers for `a` and `b` generated by SYNT are optimized away before the synthesis process is over. REM is run twice since we want to catch easy cases (such as `a` in the example) early but at the same time also make sure to catch cases requiring element-level analysis (such as `b` in the example).

2) *Sequential logic in `always_comb` blocks*: Lutsig must check that all `always_comb` blocks actually model combinational logic. E.g., Lutsig must reject the following block:

```
always_comb a = a + 1;
```

For this paper, we have extended MAP to handle this.

MAP handles the compilation of netlist-level array constructs such as array cells and array registers, by mapping them to array constructs natively available or to Boolean subcircuits. MAP is centered around a map σ from cell inputs to lists of “marked” cell inputs. MAP visits all netlist cells in order and the map σ is updated as the netlist is visited to keep track of mapped cells. For real registers, all inputs are marked legal from the start of compilation. For pseudoregisters, all inputs are initially marked as illegal inputs. If an illegal input is referenced during compilation (i.e. the (relevant part of the) σ entry for the cell input is marked illegal), the compilation is aborted.

We now consider two examples. First, note that the reference to `a` on the right-hand side in the above `always_comb` block will cause the compilation to abort. Now, instead consider the following Verilog code exemplifying code Lutsig accepts (although note that the illustration is done on the Verilog level rather than on the netlist level that MAP is actually run at):

```
always_comb begin
  // since b is a pseudoregister, we have:
  // sigma(b) = [illegal, illegal]

  b[0] = inp0; // sigma(b) = [illegal, inp0]
  b[1] = inp1; // sigma(b) = [inp1, inp0]

  // we can read the full b here since all
  // elements of b are legal
```

```

b = b + 1;
end

```

Note that since nonsynthesizable code is rejected by Lutsig, it is not important what semantics Lutsig’s Verilog semantics assigns to nonsynthesizable code. For some nonsynthesizable code, Lutsig’s semantics diverges from Verilog’s simulation semantics. E.g., recall that all blocks are unconditionally executed each clock cycle in Lutsig’s semantics. In contrast, in Verilog’s simulation semantics, `always_comb` blocks are only executed when something they depend on is updated. But since combinational logic is idempotent – that is, we can execute it multiple times without affecting the result – executing the same `always_comb` multiple times is harmless. However, if the `always_comb` block does not actually model combinational logic, this reasoning does not hold, and the two semantics might diverge.

3) *Intrablock order problems:* Recall the `andor1b` module with “mis-ordered” assignments discussed in Sec. III. The σ -based MAP pass also handles such code correctly. E.g., Lutsig rejects the following code with the same problem:

```

always_comb begin
  b = a + 1; // sigma(a) says a illegal here!
  a = inp;
end

```

4) *Interblock order problems:* Recall that Lutsig’s non-event-based Verilog semantics sorts `always_comb` blocks before execution (see Sec. V). E.g., to assign sensible semantics to the following code, the order of the blocks needs to be reversed before execution:

```

always_comb b = a + 1;
always_comb a = inp;

```

The same order problem occurs in compilation: To compile the above code correctly, Lutsig must first sort the `always_comb` blocks by their dependencies. To sort, Lutsig uses the same sorting algorithm as used in Lutsig’s Verilog semantics.

Not all processes can be ordered by their dependencies. Since combinational logic must not include combinational loops, the sorting algorithm used in Lutsig rejects code containing circular dependencies like the following:

```

always_comb a = b + 1;
always_comb b = a + 1;

```

5) *If statements:* Lutsig handles if statements correctly. E.g. the following code is rejected:

```

always_comb
  if (c)
    a = inp;
  //else
  // a = 'x;

```

If instead the else branch is uncommented, then Lutsig synthesizes the code successfully. The original block without an else branch gets stuck in the synthesis process since SYNT generates a mux with `inp` and the pseudoregister generated for `a` as inputs and MAP eventually detects that a pseudoregister is referenced and aborts the synthesis process.

6) *Case statements and nested if statements:* Compiling case statements is similar to compiling if statements: if a variable is assigned in one branch, then it must be assigned in all other

branches as well. Let the variable `c` have type `logic[1:0]` and consider the following code:

```

always_comb
  case (c)
    2'b00: a = 1;
    2'b01: a = 4;
    2'b10: a = 1;
    2'b11: a = 2;
  //default: a = 'x;
  endcase

```

A sufficiently smart synthesis tool would realize that `a` is assigned for all possible values of `c`. However, Lutsig’s synthesis algorithm is not smart and requires the commented-out `default` branch above to realize that all cases are covered. The same holds for the analogous situation with nested if statements. In fact, Lutsig handles case statements by expanding them to nested if statements, so Lutsig’s limited case statement handling is a consequence of Lutsig’s limited if statement handling.

VIII. FUNCTIONAL CORRECTNESS OF LUTSIG

We now state Lutsig’s functional-correctness theorem, thereby showing that Lutsig successfully abides by (its formalization of) Verilog’s simulation semantics. The theorem statement is the same as in the previous version of Lutsig; the HOL4 proof of the theorem, however, has been updated to take into account the new functionality added in this paper. If we let $P \Downarrow_L^{n,fbits} S$ denote that design P ’s externally visible state is S under the semantics L after n clock cycles with nondeterminism source $fbits$, then Lutsig’s correctness theorem is as follows:

$$\begin{aligned}
& \text{Lutsig } P_{\text{ver}} = \text{OK } P_{\text{nl}} \implies \\
& \exists S_{\text{nl}}, P_{\text{nl}} \Downarrow_{L_{\text{nl}}'}^{n,fbits} S_{\text{nl}} \wedge \\
& \exists fbits', P_{\text{ver}} \Downarrow_{L_{\text{ver}}}^{n,fbits'} S_{\text{ver}} \implies S_{\text{nl}} = S_{\text{ver}}
\end{aligned}$$

Per the usual convention, all free variables in the theorem are implicitly universality quantified. Note that since the netlist P_{nl} in the theorem statement never contains pseudoregisters, we can use the simplified netlist semantics L_{nl}' which does not handle pseudoregisters.

Although the theorem statement is more complex than straightforward backward simulation as presented in Sec. II-A, the theorem still allows for theorem transportation from the Verilog level down to the netlist level by simple composition (i.e., VPD): Given a circuit-correctness theorem stating that a Verilog module P_{ver} never crashes (regardless of what $fbits$ is supplied), say $\exists S_{\text{ver}}, P_{\text{ver}} \Downarrow_{L_{\text{ver}}}^{n,fbits} S_{\text{ver}} \wedge \text{Spec } S_{\text{ver}}$ for some specification Spec , if Lutsig successfully synthesizes P_{ver} to a netlist P_{nl} , then we can easily derive $\exists S_{\text{nl}}, P_{\text{nl}} \Downarrow_{L_{\text{nl}}'}^{n,fbits} S_{\text{nl}} \wedge \text{Spec } S_{\text{nl}}$.

IX. NONFUNCTIONAL CORRECTNESS OF LUTSIG

We now turn to the nonfunctional correctness of Lutsig. Recall that Verilog’s synthesis semantics enables hardware designers to express hardware design ideas to their synthesis tool through modeling idioms. The theorem presented in this section, which we have proved in HOL4, shows that Lutsig correctly handles `always_comb` blocks in the sense that the theorem captures the modeling idiom that `always_comb` blocks must be mapped to combinational logic [30, p. 207].

We formalize this modeling idiom as follows: for any run $Lutsig P_{ver} = OK P_{nl}$, if a variable is written to in an `always_comb` block in P_{ver} , then no register with the same name as the variable will be included in P_{nl} . Formally, the theorem is as follows:

$$\begin{aligned} Lutsig P_{ver} = OK P_{nl} &\implies \\ \forall var, var \in comb_vars P_{ver} &\implies var \notin regs P_{nl} \end{aligned}$$

Note that the theorem relates concepts in the input design P_{ver} (writes) to concepts in the final netlist P_{nl} (registers) – this means that we must, in our proofs, carry information from the very first compilation phase down to the very last.⁷

X. CONCLUSION

We now conclude. In our discussion on the relationships between Verilog’s simulation semantics, Verilog’s synthesis semantics, VPD, and TVD, we identify Verilog’s modeling idioms as the core cause of tensions between VPD and TVD. To put our discussion to test, we have added support for `always_comb` blocks to the verified synthesis tool Lutsig.

Our discussion on VPD and TVD paves the way for further Lutsig extensions that add support for Verilog constructs associated with simulation-and-synthesis mismatches, such as support for BRAM inference.

Another interesting direction for future work to explore is how a more detailed hardware semantics would affect the `always_comb` discussion. In this paper our Verilog semantics is at the level of cycle-by-cycle behavior – what are the alternatives for a more detailed hardware semantics that, while at the same time as keeping source-level reasoning feasible, allow us to turn the nonfunctional property we have proved in this paper into a part of the compiler’s functional correctness theorem?

Lastly, no approach to hardware development, regardless of hardware language used, completely shields the hardware designer from the synthesis aspects we have discussed in this paper. It would therefore be interesting to consider how much of our discussion on VPD and TVD translates into hardware development and synthesis-tool verification for other hardware languages. The questions we raise in this paper will reappear in similar form regardless of the hardware language used. After all, not even so-called high-level synthesis (HLS), i.e., generating hardware from software languages like C, can completely hide the synthesis process from hardware developers. E.g., the manual [38, p. 17] for Vitis, an unverified HLS tool for C, C++, and OpenCL, states that “arbitrary, off-the-shelf software cannot be efficiently converted into hardware” and that, moreover, “even if [a] software program can be automatically converted (or synthesized) into hardware, achieving acceptable quality of results, will require additional work such as rewriting the

⁷Before we started working on the proof, Lutsig did not actually satisfy our formalization of the `always_comb` modeling idiom. This was because the SYNT pass (see Sec. VII) used the presence of writes *in the design that was given to that pass* to decide which variables to map to real registers and which to pseudoregisters rather than the presence of writes *in the design as given by the user* (i.e., P_{ver} in the above theorem) – the former does not reliably track the latter since writes may be optimized away in the compilation process!

software to help the HLS tool achieve the desired performance goals.” The pessimism of the manual [38, p. 28] continues: “Software written for CPUs and software written for FPGAs is fundamentally different. You cannot write code that is portable between CPU and FPGA platforms without sacrificing performance.” To prepare its readers for hardware development using Vitis, the manual informs its readers what they need to know about the Vitis synthesis process to design efficient hardware; in other words, the HLS hardware designer, much like the Verilog hardware designer, must be aware of how to control their synthesis tool and how to communicate to their synthesis tool what kind of hardware they want. In total, the Vitis manual is 660 pages, reflecting the fact that not even HLS manages to abstract away the complexities of synthesis.

ACKNOWLEDGMENT

We thank Magnus Myreen, Adam Chlipala, David Greaves, Tom Melham, Warren Hunt, Koen Claessen, Wolfgang Ahrendt, Philippa Gardner, and Kashish Raimalani for comments on drafts of this paper. This work was supported by the Swedish Foundation for Strategic Research.

REFERENCES

- [1] X. Leroy, “A formally verified compiler back-end,” *Journal of Automated Reasoning*, vol. 43, no. 4, 2009.
- [2] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, “CakeML: A verified implementation of ML,” in *Principles of Programming Languages (POPL)*, 2014.
- [3] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, “CompCertTSO: A verified compiler for relaxed-memory concurrency,” *Journal of the ACM*, vol. 60, no. 3, 2013.
- [4] A. Barrière, S. Blazy, O. Flückiger, D. Pichardie, and J. Vitek, “Formally verified speculation and deoptimization in a JIT compiler,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, 2021.
- [5] M. O. Myreen, “Verified just-in-time compiler on x86,” in *Symposium on Principles of Programming Languages (POPL)*, 2010.
- [6] —, “A minimalistic verified bootstrapped compiler (proof pearl),” in *Conference on Certified Programs and Proofs (CPP)*, 2021.
- [7] D. Patterson and A. Ahmed, “The next 700 compiler correctness theorems (functional pearl),” *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, 2019.
- [8] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, “Formal verification of a constant-time preserving C compiler,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2019.
- [9] A. Gómez-Londoño, J. Åman Pohjola, H. T. Syeda, M. O. Myreen, and Y. K. Tan, “Do you have space for dessert? A verified space cost semantics for CakeML programs,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.
- [10] R. M. Amadio, N. Ayache, F. Bobot, J. P. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, and P. Tranquilli, “Certified complexity (CerCo),” in *Foundational and Practical Aspects of Resource Analysis (FOPARA)*, 2014.
- [11] Z. Paraskevopoulou and A. W. Appel, “Closure conversion is safe for space,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, 2019.
- [12] T. Braibant and A. Chlipala, “Formal verification of hardware synthesis,” in *Computer Aided Verification (CAV)*, 2013.
- [13] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, “The essence of Bluespec: A core language for rule-based hardware design,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [14] A. Lööv, “Lutsig: A verified Verilog compiler for verified circuit development,” in *Conference on Certified Programs and Proofs (CPP)*, 2021.

- [15] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, “Formal verification of high-level synthesis,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, 2021.
- [16] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “LLHD: A multi-level intermediate representation for hardware description languages,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [17] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, “Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations,” in *International Conference on Computer-Aided Design (ICCAD)*, 2017.
- [18] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanović, “Chisel: Constructing hardware in a Scala embedded language,” in *Annual Design Automation Conference (DAC)*, 2012.
- [19] R. Nikhil, “Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications,” in *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.
- [20] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, “Lava: Hardware design in Haskell,” in *International Conference on Functional Programming (ICFP)*, 1998.
- [21] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, “Clash: Structural descriptions of synchronous hardware using Haskell,” in *Euromicro Conference on Digital System Design*, 2010.
- [22] L. Vega, J. McMahan, A. Sampson, D. Grossman, and L. Ceze, “Reticle: A virtual machine for programming modern FPGAs,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2021.
- [23] J. P. Pizani Flor, W. Swierstra, and Y. Sijsling, “II-Ware: Hardware description and verification in Agda,” in *International Conference on Types for Proofs and Programs (TYPES 2015)*, 2018.
- [24] W. L. Harrison, I. Graves, A. Procter, M. Becchi, and G. Allwein, “A programming model for reconfigurable computing based in functional concurrency,” in *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2016.
- [25] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, “Predictable accelerator design with time-sensitive affine types,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [26] M. Christensen, T. Sherwood, J. Balkind, and B. Hardekopf, “Wire sorts: A language abstraction for safe hardware composition,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2021.
- [27] K. Slind and M. Norrish, “A brief overview of HOL4,” in *Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.
- [28] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Pearson, 2011.
- [29] *Vivado Design Suite User Guide: Synthesis (UG901, v2020.2)*, Xilinx, 2021.
- [30] “IEEE standard for SystemVerilog—unified hardware design, specification, and verification language,” *IEEE Std 1800-2017*, 2018.
- [31] S. Sutherland and D. Mills, *Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them*. Springer, 2007.
- [32] “Verilog register transfer level synthesis,” *IEEE Std 62142-2005*, 2005.
- [33] *Intel Quartus Prime Pro Edition User Guide: Design Recommendations (UG-20131, v21.1)*, Intel, 2021.
- [34] “IEEE standard for Verilog hardware description language,” *IEEE Std 1364-2001*, 2001.
- [35] L. Simon, D. Chisnall, and R. Anderson, “What you get is what you C: Controlling side effects in mainstream C compilers,” in *European Symposium on Security and Privacy (EuroS&P)*, 2018.
- [36] C. Hathhorn, C. Ellison, and G. Roşu, “Defining the undefinedness of C,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2015.
- [37] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, “Into the depths of C: Elaborating the de facto standards,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2016.
- [38] *Vitis High-Level Synthesis User Guide (UG1399, v2021.1)*, Xilinx, 2021.

Timed Causal Fanin Analysis for Symbolic Circuit Simulation

Roope Kaivola
Core and Client Development Group
Intel Corporation
Portland, OR, USA
roope.k.kaivola@intel.com

Neta Bar Kama
Core and Client Development Group
Intel Corporation
Haifa, Israel
neta.bar.kama@intel.com

Abstract—Symbolic circuit simulation has been the main vehicle for formal verification of Intel Core processor execution engines for over twenty years. It extends traditional simulation by allowing symbolic variables in the stimulus, covering the circuit behavior for all possible values simultaneously. A distinguishing feature of symbolic simulation is that it gives the human verifier clear visibility into the progress of the computation during the verification of an individual operation, and fine-grained control over the simulation to focus only on the datapath for that operation while abstracting away the rest of the circuit behavior.

In this paper we describe an automated simulation complexity reduction method called *timed causal fanin analysis* that can be used to carve out the minimal circuit logic needed for verification of an operation on a cycle-by-cycle basis. The method has been a key component of Intel’s large-scale execution engine verification efforts, enabling closed-box verification of most operations in the interface level.

As a specific application, we discuss the formal verification of Intel’s new half-precision floating-point FP16 micro-instruction set. Thanks to the ability of the timed causal fanin analysis to separate the half-precision datapaths from full-width ones, we were able to verify all these instructions closed box, including the most complex ones like fused multiply-add and division. This led to early detection of several deep datapath bugs.

Index Terms—Formal Verification, Symbolic Simulation, Complexity Reduction

I. INTRODUCTION

Comprehensive formal verification of execution engines has been standard practice in virtually all Intel® Core™ and Intel Atom® processor development projects in the last two decades, and extensive infrastructure has been built to support these efforts. Formal verification of Intel processor execution engines is primarily based on *symbolic circuit simulation*, a technology extending usual digital circuit simulation with symbolic values, representing sets of concrete values in a single simulation [1], [2], [3], [4], [5].

Full correctness of processor execution engines is indispensable for product quality, as errata in basic execution datapaths tend to be both customer visible and un-patchable. Due to the size of the data space and the difficulty of identifying and

Intel provides these materials as-is, with no express or implied warranties. Intel processors might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. Intel, Intel Core, Intel Atom, Pentium and Intel logo are trademarks of Intel Corporation. Other names and brands might be claimed as the property of others.

covering all internal corner cases with either pre-silicon or post-silicon testing, formal verification is the only approach that can ensure sufficient quality, especially for complex floating point datapaths.

Execution engines in industrial processor designs typically combine a set of different pipelined datapaths into a single design component. To minimize circuit size, each individual datapath multiplexes logic for a family of related operations, controlled by operation-specific control signals. The datapaths may support different latencies, with simpler operations executing in fewer pipestages than complex ones. Many datapaths are implemented as straight pipelines, however certain operations may use iterative algorithms with feedback loops. Designs also usually contain bypass networks that route data from the datapath outputs directly back to the inputs, avoiding the delay of going through a register file. The execution engine in a contemporary Intel processor has several million logic gates and hundreds of thousands of flip-flops, and the source code for it consists of hundreds of thousands of lines of code in a hardware description language.

Focusing on the verification of an individual operation implemented in an execution engine, we can conceptually distinguish two different sources of verification complexity:

- 1) the inherent complexity of the plain datapath for the operation, ignoring all other functionality of the execution engine, and
- 2) the complexity caused by the presence of the rest of the execution engine, and its possible effects on and interferences with the datapath of the operation.

As an example of the first, any datapath involving multiplication can be expected to pose a verification challenge, irrespective of any surrounding logic. For the second, the isolation of the result of an operation in a shared result bus depends on the control logic of all the datapaths sharing the bus. In a practical verification task, the verification engineer faces these two dimensions simultaneously, and the complexity caused by the surrounding logic may make the verification of even inherently trivial datapaths, such as bitwise OR, challenging or infeasible.

Considering the inherent datapath complexity, without surrounding environment, the large majority of operations implemented on an execution engine can be directly verified

by symbolic simulation in a closed-box fashion. This is the ideal scenario due to the many advantages of closed-box verification: a well-defined specification, no need of insight into implementation details, and low sensitivity to internal design changes. For the most complex operations, especially complex floating-point arithmetic such as multipliers, fused multiply-adders and dividers, this straightforward approach is computationally infeasible, and verification is done by means of decomposed reference models, requiring time and both design and verification expertise.

If the plain datapath for an individual operation were to be isolated from the surrounding logic, for most operations it would be amenable to verification by a variety of techniques besides symbolic simulation. However, in practice, the datapath is tightly enmeshed with the rest of the execution engine, and there is no straightforward way to isolate it. In this respect, symbolic simulation has a unique advantage over many competing verification approaches, such as formal equivalence verification or traditional model checking: it allows the verification engineer to understand the computational progress of an operation in the circuit in very concrete terms, to carve out a minimal amount of logic that needs to be simulated for the datapath of that specific operation, and to efficiently abstract away the rest. In other words, symbolic simulation provides an effective way to separate the two sources of verification complexity. The main technical ingredients enabling this ability are discussed in Section II.

Nevertheless, as execution engines typically implement thousands of individual operations, and for each operation the datapath controls are wired differently, the cost of the human effort to analyze and isolate each datapath becomes a limiting factor.

In this paper we describe an algorithmic technique called *timed causal fanin analysis* to derive a tight over-approximation of the circuit logic relevant for the simulation of the datapath of an individual operation (Section III). This method effectively automates the human process of determining the minimal circuit logic for a specific datapath. It is based on the use of information from an earlier, more abstract and less accurate symbolic simulation run to reduce the fanin cone of the logic of interest on a cycle-by-cycle basis. The method enables fully automated closed-box verification of most operations in an execution engine, not just for an isolated datapath, but in the context of the full design unit. It is meaningful only in the context of verification by symbolic simulation. The method has been a key technical enabler in Intel’s large-scale verification initiatives over the span of many years [3], [6]. However, the current paper is the first detailed exposition of the method in the public domain.

For a recent example illustrating the effects of timed causal fanin analysis, in Section V we discuss the verification of the new FP16 floating-point instruction set on a recent Intel Core processor design. Since the Intel 8087 floating-point co-processor was introduced in 1980, Intel processors have supported single, double, and extended precision floating point formats. The formal verification of complex operations such

as multiplication, division, etc., on these formats has always required decomposition, making such verification a time-consuming expert task. Recent Intel Core processor designs have added a new shorter half-precision floating-point format, also known as FP16 [7]. Because of the lower datapath width, the inherent verification complexity of FP16 datapaths is also lower, bringing them closer to the set of designs that one could hope to verify without decompositions.

As a practical result, we found out that *all* FP16 micro-operations could be verified closed box, including the complex multiplication, fused multiply-add, division and square root operations. This led to fast verification convergence and early detection of several high complexity datapath bugs. The timed causal fanin analysis technique was particularly crucial for datapaths shared between FP16 and higher precision operations. It allowed us to avoid simulating the higher-precision logic, the complexity of which would have otherwise made verification impossible.

II. SYMBOLIC CIRCUIT SIMULATION

Symbolic simulation extends traditional digital circuit simulation by allowing the input stimulus to contain *symbolic variables* in addition to the concrete values 0, 1 or X [1]. These symbolic variables are effectively names of values, denoting sets of possible actual concrete values. In the simulation, these symbolic values propagate alongside the concrete values, and in each logic gate, they may be combined with each other or one of the concrete values to result in either a concrete value or a logical expression on the symbolic variables, represented by an expression graph. In this paper, as in most of symbolic circuit simulation verification practice, we use the binary decision diagram (BDD) representation for symbolic expressions [8]. See Figure 1 for an example.

In a bit level symbolic simulator, a single symbolic variable a corresponds to the set of boolean values containing both 0 and 1. If stimulus to a symbolic simulation refers to the variables a , b and c , the internal signals might carry values like $a \wedge b$ or $a \vee (b \wedge \neg c)$. Usual logic rules apply: if the inputs to an AND-gate are a and 1, the output will be a , if the inputs to an AND-gate are a and b , the output is the logical expression $a \wedge b$, and if the input to a NOT-gate is b , the output will be $\neg b$. In symbolic simulation, a specific symbolic variable is associated with a specific signal and time in the stimulus. This does not fix the value, but instead gives a name that can be used to refer to the value.

The special value X is used in symbolic simulation to denote a universal undefined or unknown value, which propagates according to rules such as in Figure 2. The value X denotes lack of information: we do not know whether the value is 0 or 1. The propagation rules reflect this intuition. Symbolic simulation uses X ’s as an abstraction mechanism: unlike symbolic variables, X ’s are an over-approximation of Boolean circuit behavior. Both symbolic variables and X ’s allow us to verify a property over a single symbolic trace, and conclude that it is valid over every possible trace instantiating the X ’s and the symbolic variables with 0’s or 1’s. This ability of a

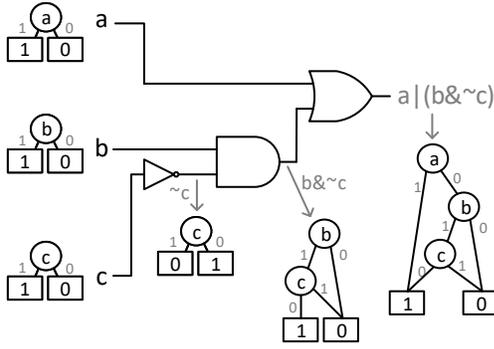


Fig. 1. Symbolic expressions in simulation

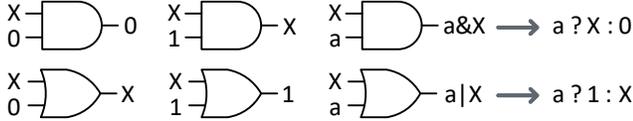


Fig. 2. Logic with the undefined value X

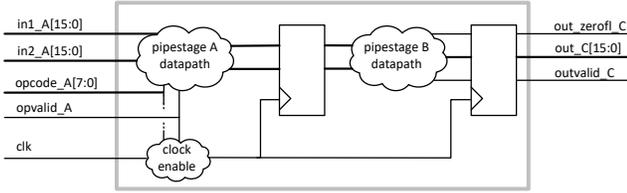


Fig. 3. Simplified ALU

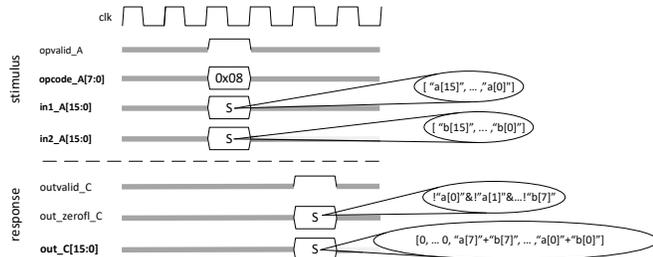


Fig. 4. Symbolic trace

single symbolic trace to cover all behaviors of a circuit allows us to use symbolic simulation as a formal verification method.

Figure 3 depicts a simplified pipelined ALU circuit with a 16-bit wide two-cycle datapath from inputs to outputs, and Figure 4 depicts a typical symbolic trace that might be used in the verification of this ALU, focusing on a single instance of an eight-bit wide bitwise OR operation. In the stimulus, the control signals are driven with concrete values corresponding to the operation, and the input data is driven with symbolic variables $a[15], \dots, a[0]$ and $b[15], \dots, b[0]$ in the one cycle in which the operation is issued. In all other cycles these signals have the undefined value X (gray waveform). In the simulation, the values of the output data and zero flag two cycles later are then expressions on the symbolic variables associated with the

input data, and in all other cycles they are X's.

The practice of verification by symbolic simulation has similarities to bounded model checking (BMC), however with two important differences. First, BMC considers instances of a property in a time window up to a given bound, whereas symbolic simulation focuses on one fixed instance of a property, and second, BMC starts from a properly initialized state of a system, and symbolic simulation from an unconstrained state. The focus on one fixed instance of a property can be seen as a distinguishing aspect of symbolic simulation.

The size of the symbolic expressions flowing in the signals of the circuit during the simulation is the most crucial complexity metric and the limiting factor determining what can and cannot be computed. We strive to minimize this symbolic complexity in several ways:

- 1) by choosing the properties to be verified so that they are as narrowly targeted as possible and by restricting the circuit simulation to only those scenarios that are relevant for the property under verification,
- 2) by limiting the number of symbolic variables and concrete 0/1 values used in the simulation stimulus to maximize the use of the default undefined value X,
- 3) by limiting the set of signals for which simulation values are computed, the times for which those values are computed, and the values that are computed, and
- 4) by choosing concise representations for the computed symbolic expressions.

For example, in execution engine verification we (1) focus on one operation instance at a time, (2) drive symbolic values on inputs only when the operation instance under verification samples them, (3) simulate only signals that are needed for the datapath of the operation and only at times relevant to the progression of its pipeline, and (4) use a BDD variable ordering that is a good match for the operation.

Symbolic simulation works best with targeted properties of fixed length pipelines, typically of the transactional form

trigger A at time t is followed by response B at time t + n

To restrict circuit behaviors to cover only cases where the trigger of the property under verification is true, we use the technique of *parametric substitutions* [9], [10]. The basic setup for the parametric substitution algorithm is that we want to verify an implication $C(\bar{v}) \Rightarrow D(\bar{v})$ between two symbolic expressions C and D over a set of symbolic variables \bar{v} , and the assumption C in some fashion makes it easier to compute the goal D. The algorithm creates a mapping $\bar{v} \mapsto \bar{p}$ from variables \bar{v} to symbolic expressions \bar{p} such that when the symbolic variables in \bar{p} range over all possible values, the values of the symbolic vector \bar{p} range exactly over the set of assignments to \bar{v} for which the condition C is true. Then, the implication can be verified by checking whether $D(\bar{p})$ holds.

In the context of symbolic simulation, the aim is to check an implication between the trigger and the goal of the property being verified over the traces of the circuit. This is done by computing a parametric substitution from the trigger, carrying

out the symbolic simulation with the parametrized expressions \bar{p} instead of the original variables \bar{v} in the stimulus, and by checking that the verification goal is true in the resulting trace. For a concrete example of parametric substitution for symbolic simulation triggers, please see Section III below, especially Figure 6 and the related discussion.

The techniques for limiting the sets of signals, times or values for which simulation is done are collectively called *weakening*. In weakening the user instructs the simulator to replace a value that would otherwise be computed with the undefined value X . We distinguish three kinds of weakening:

- *Universal weakening*, where the user instructs the simulator to replace the values of certain signals with X 's at all times in the simulation. It is equivalent to the concepts of 'free' or 'stop-at' present in many model checkers.
- *Cycle specific weakening*, where the user instructs the simulator to replace the values of certain signals with X 's, *but only at specified times*. This technique is unique to symbolic simulation, and the fact that it is even meaningful to talk about signals at specific times in the verification task is directly related to the fact that symbolic simulation focuses on just one fixed instance of the verification goal. Cycle specific weakening is an extremely versatile technique that allows users to apply their intuition about the usage of signals at times relative to the progress of the operation under verification in order to reduce the simulation cost.
- *Dynamic weakening*, where the user instructs the simulator to replace any symbolic value with X , if the size of the expression for the value would exceed a user-given threshold. Dynamic weakening is a robust technique that allows users to quickly resolve many symbolic complexity issues caused by the computation of unnecessary expressions in the simulation without detailed analysis.

Weakening is a safe complexity reduction technique: if we verify a property over a symbolic simulation trace with weakening, the same property also holds over a trace with the same stimulus and no weakening.

The computations in symbolic simulation are conceptually simple and concrete. Further, they can be naturally related to the progress of the operation under verification through its pipeline. This gives the verification engineer fine-grained visibility into the computations on the level of individual signals, enabling precise analysis and mitigation of computational complexity bottlenecks through weakening. In the context of execution engine verification, this visibility allows the verifier to identify the datapath of an individual operation and weaken the surrounding circuit logic. However, when pipelines for different operations are tightly enmeshed in a circuit, it is often time-consuming to determine which signals and simulation times are really needed for a specific operation.

III. TIMED CAUSAL FANIN ANALYSIS

As discussed above, the size of the symbolic expressions is the primary capacity barrier in a simulation, and consequently it is very important that we avoid the computation of symbolic

values unnecessarily, in contexts where they do not contribute meaningfully to the verification goal. In a forward simulator this is not trivial. When simulating a certain cycle, we do not know yet which signals in that cycle will matter to the verification goal in a later cycle.

One straightforward technique for reducing the set of signals for which simulation needs to compute values is the standard cone of influence (COI) reduction. The validity of a verification goal can only depend on the transitive fanin of signals referenced in it, and therefore signals outside of this set do not need to be simulated. However, for execution engines that contain bypass networks, the circuit forms in practice a nearly strongly connected graph, i.e. almost every signal is in the transitive fanin of almost every other signal, and the cone of influence reduction offers little help.

Another source of reduction comes from the simplifying effect of any global constants in the design. For example, an AND-gate with one input a constant zero does not actually depend on the value of its other input, and that other input can be removed from the fanin of the gate without changing the behavior of the circuit. As designers do not intentionally include dead logic in their designs, such global constants usually reflect circuit functionality, such as test or scan modes, that can be completely disabled for verification purposes. They usually offer only marginal help in reducing simulation scope around the main functionality of a design.

The *timed causal fanin analysis* algorithm is based on the idea of using constants to reduce the fanin cone of interest. However, this is done on a cycle-specific basis, relative to the cycle times in a fixed symbolic simulation, using the concrete 0/1 values present in that cycle only. As with cycle-specific weakening, the fact that we can meaningfully refer to a particular cycle relative to a verification task is specific to symbolic simulation. The three main steps of the method are:

- 1) Perform a preliminary symbolic simulation to determine *cycle-specific* concrete 0/1 values in the simulation.
- 2) Compute the transitive cone of influence of nodes and cycles in the verification goal *per cycle*, using the concrete 0/1 values from step 1 to reduce the fanins in each cycle.
- 3) Compute a cycle-specific weakening list, *per cycle*, that weakens every signal of the circuit except the signals in the transitive cone of influence for that cycle, as computed in step 2.

Step 1 consists of a symbolic simulation run for the circuit with the same stimulus that is used for the main verification run. However, for this initial simulation, the dynamic weakening threshold is low. As described in Section II, this means that any symbolic expressions above the threshold are discarded and replaced with X 's in the simulation. The size threshold is specified by the user. All relevant cycles of the resulting stimulation trace are then scoured for all concrete 0/1 values.

It is important to note that this preliminary simulation is much more than just timed constant propagation. First, the trigger of the property has already been factored into the stimulus with parametric substitution, and any concrete 0/1

values implied by the trigger are present in the trace, especially in the pipelined datapath control signals. Second, in addition to the concrete values that the trigger forces directly in the stimulus, also the concrete values that are implied indirectly by circuit logic together with the trigger restrictions are present in the trace, due to the canonicity of the BDD representations and the automatic simplification in BDD operations.

Step 2 consists of a backwards traversal over relevant simulation cycles, starting from the last cycle of interest and proceeding back in time. For each cycle, we compute the causal fanin at that cycle using the concrete 0/1 values present at the cycle to reduce the causal fanin cone.

For a combinational gate s of the circuit, we define the *combinational causal fanin set of s at simulation time t* to be the set of signals s_{in} such that s_{in} is an immediate fanin of s and either

- s_{in} has a concrete 0/1 value in cycle t in the simulation in Step 1, or
- the value of s_{in} may affect the value of s , given all the concrete 0/1 values in the fanins of s in cycle t in the simulation in Step 1.

In short, for each cycle the concrete 0/1 values computed in Step 1 for that cycle are used to reduce the fanin cone of combinational gates. For example, if selectors to a mux have concrete 0/1 values in a certain cycle, only the single mux input that is selected by those selectors is in the timed causal fanin in that cycle.

For a flip-flop (state element) s_{ff} of the circuit, with input s_{in} and clock c , we define the *flip-flop causal fanin set of s_{ff} at simulation time t* by the rules:

- If the clock c toggles in cycle t in the simulation in Step 1, then s_{in} belongs to the set.
- If the clock c does not toggle in cycle t in the simulation in Step 1, then s_{ff} belongs to the set.
- If the clock c is X in cycle t in the simulation in Step 1, then both s_{in} and s_{ff} belong to the set.

Conceptually, if we do not know whether the clock toggles or not, both the input and the held value of the flip-flop matter.

For each cycle t , we then define the *timed causal fanin set $cfan(t)$* as the minimal set of circuit signals satisfying the following rules:

- 1) If the verification goal directly refers to signal s in cycle t on the simulation, then $s \in cfan(t)$.
- 2) If signal s is in the flip-flop causal fanin set of a flip-flop s_{ff} at simulation time $(t + 1)$, and $s_{ff} \in cfan(t + 1)$, then $s \in cfan(t)$.
- 3) If signal s is in the combinational causal fanin set of a combinational gate s_{out} at time t , and $s_{out} \in cfan(t)$, then $s \in cfan(t)$.

For each cycle t , we compute $cfan(t)$ by starting from the set of signals determined by the rules (1) and (2) and by constructing the transitive closure of the set under rule (3), stopping at the flip-flop boundary.

Step 3 finally constructs a weakening list that for every cycle t replaces the value of every signal not in $cfan(t)$ with X .

This weakening list is then used in a full symbolic simulation for the original verification goal. As the computation of the timed causal fanin in Step 2 includes all signals and times that may affect the signal-time references in the property under verification, the weakening list never abstracts with X any values that could contribute to the property. As an optimization, we can alternatively weaken only the barrier of signals whose fanin intersects with $cfan(t)$ but which are not in $cfan(t)$ themselves.

As a point of comparison, consider the same verification task posed as a bounded model checking problem. If we look at just the timed constant propagation aspect of the preliminary simulation, and the concrete 0/1 values directly forced by the trigger, an analogous constant propagation process would take place at an early point inside the SAT call for the BMC problem, resulting in expression simplification similar to the fanin reduction above. As for the concrete 0/1 values indirectly implied by the trigger and the circuit logic, sooner or later they either might or might not be noticed and propagated by the SAT engine, depending on how hard the engine tries to determine constants. However, this whole process is completely hidden from the user, inside a SAT engine. In particular, if a potentially helpful simplification does not happen, either because the engine misses it or because the trigger does not capture the user intent accurately, the issue manifests to the user only through increased run time or the inability of the tool to resolve the verification goal, without actionable feedback that would enable the user to assist the tool.

However, when we use timed causal fanin analysis in the symbolic simulation flow, the results of the preliminary simulation and the concrete 0/1 values that are or are not present are visible and accessible to the user. The values can be queried, viewed as waveforms and root-caused through circuit gates. The user can understand what happens in the simulation and compare that to their intuition and expectations about what should happen. The concept of the timed causal fanin cone itself is based on a clear operational intuition, allowing the user to understand the computation in terms of circuit functionality. A commonly asked debug question is: “why is signal s in cycle t in the timed causal fanin cone of my property, when conceptually it should not matter, for example because it is in a different unit/datapath/pipestage?” This question can be concretely answered by showing a path of dependencies from the given signal and time through fanin relations to some signal and time relevant to the property being verified.

As an example, consider the simplified ALU circuit in Figure 5 with a one-cycle adder unit and a two-cycle multiplier unit. At the interface, the signal vld marks a valid operation and mul chooses between addition and multiplication. Further, suppose that we are focusing on adder correctness as expressed in the following property, where \mathbf{N} and \mathbf{P} are the next-time and previous-time temporal operators, respectively:

$$\underbrace{(vld \wedge \neg mul)}_{\text{ADD time } t} \wedge \underbrace{\mathbf{P}\neg(vld \wedge mul)}_{\text{NOT MUL time } (t-1)} \Rightarrow \mathbf{N}(\underbrace{is_ok(res)}_{\text{RESULT OK time } (t+1)})$$

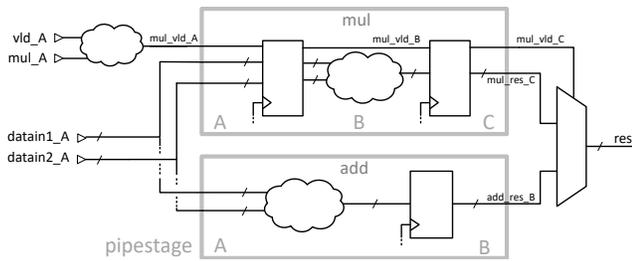


Fig. 5. Simplified ALU with adder and multiplier

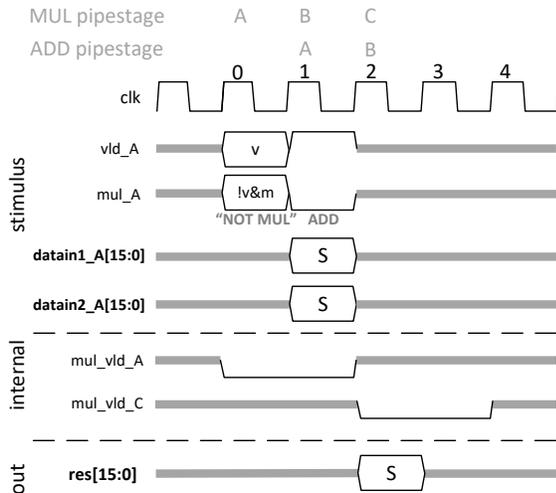


Fig. 6. Stimulus and preliminary simulation trace

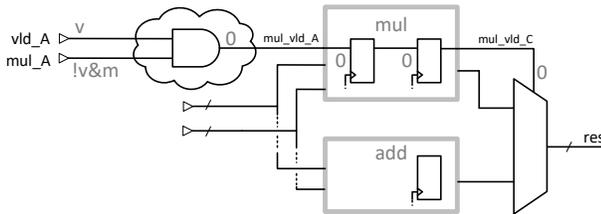


Fig. 7. Internal simplification in control logic

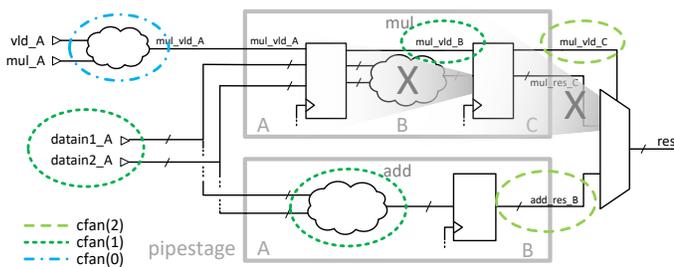


Fig. 8. Timed causal fanin cone computation

Conceptually this property says that if an addition operation is issued, and there is no pipeline hazard from a multiplication operation a cycle ago, then the circuit will produce functionally correct output in the next cycle (where we have omitted the details of ‘functionally correct output’ and its dependency on the data input signals).

Figure 6 depicts a stimulus and trace for the Step 1 preliminary simulation on the circuit, with an instance of the property above with time $t = 1$, starting in cycle 1 and producing output in cycle 2. The stimulus values for the control signals vld and mul in cycles 1 and 0 have been generated by parametric substitution from the triggers of the property:

- In cycle 1, the stimulus associates the concrete value 1 with the signal vld and the concrete value 0 with the signal mul , since this is the only possible assignment satisfying the trigger ‘ADD in cycle 1’, i.e. $vld \wedge \neg mul$.
- In cycle 0, the stimulus associates a symbolic variable v with the signal vld and the symbolic expression $\neg v \wedge m$ with the signal mul , reflecting the trigger ‘NOT MUL in cycle 0’. Note that the possible values of these two symbolic expressions range exactly over the set of assignments to vld and mul that make the trigger $\neg(vld \wedge \neg mul)$ true in cycle 0, a guarantee of parametric substitution. Note also that no concrete 0/1 assignment would capture the trigger fully, since there are three possible concrete value pairs satisfying the trigger.

Simplification on internal control signals, as depicted in Figure 7, then leads to the trace of Figure 6. Using the cycle-specific concrete 0/1 values from this trace, Step 2 of the timed causal fanin analysis method proceeds as in Figure 8. In Step 3, all signals and times outside the timed causal fanin of Figure 8 are weakened in the main simulation. Note, in particular, that all multiplier datapath logic is automatically weakened by the timed causal fanin algorithm.

From the perspective of the user applying the timed causal fanin method, the practical workflow can be divided into two stages. First, there is the computation of the causal fanin cone in Steps 1 and 2. In this stage the user may need to adjust a default dynamic weakening threshold for the preliminary simulation in Step 1 or the default depth of the fanin cone traversal in Step 2 to balance two needs. On the one hand, the threshold and the depth of the fanin cone need to be low enough that the steps can be computed in a reasonable time. On the other, the threshold and depth need to be high enough that as many concrete internal values as possible are computed to reduce the causal fanin cone. In this first stage of the work the user also may find out that the verification triggers are not strong enough to guarantee the satisfaction of the verification goals, by simply looking at the causal fanin cone and noticing unexpected causal dependencies. These may either reflect a design bug, or a need to strengthen the triggers to properly capture the intent of the property under verification.

In the second stage of the work the user then applies the weakening list computed in Step 3 in the main simulation, debugs any failures, and repeats the main simulation if necessary. In many instances the main simulation is less resource

intensive than the preliminary one, since although the symbolic expressions that need to be computed are larger, the number of signals for which they actually need to be computed is much lower, thanks to the timed causal fanin weakening list.

The timed causal fanin algorithm is helpful in most symbolic simulation verification tasks, and we use it as a routine step in our verification flow. Already on its own, symbolic simulation is at its strongest for narrowly targeted properties, and the timed causal fanin method accentuates this strength. When comparing the automated weakening provided by the method to manually crafted weakening lists, in our experience the automatically produced weakening is almost always superior, as user time and patience for fine-grained analysis of the design is often limited. As a weak point, the presence of data-qualified clocks in a design tends to reduce the efficacy of the method, as then the timed causal fanin cone will include same combinational logic over multiple cycles.

Two major building blocks underlying the timed causal fanin method are fundamentally BDD-based: first the parametric substitution algorithm, and secondly the automated simplification of symbolic expressions in the internal wires of the circuit, which results in the concrete 0/1 values that are used to contain the fanin cone. If we want to avoid BDD's and simulate with non-canonical expressions and use SAT instead, the same crucial process of identifying simplifying internal concrete 0/1 values could be achieved by speculative SAT queries checking for constants in the preliminary simulation under the trigger assumptions. The sheer number of internal signals in many circuits is a challenge in this approach, though. What works better in practice is a hybrid approach, where the preliminary simulation uses BDD's, with the resulting automated simplification, but the main simulation used for the verification of the goals is carried out with non-canonical expressions and SAT.

IV. EXECUTION ENGINE FORMAL VERIFICATION

At high level, a single Intel Core consists of a set of major design components called *clusters*. The front-end cluster fetches and decodes architectural instructions and translates them to micro-operations (abbreviated as uops), which the out-of-order cluster then schedules for execution. The execution engine, residing in the EXE cluster, carries out data computations for all micro-operations. The memory cluster handles memory accesses and may contain first level caches. Outside of an individual core is a system-on-chip layer including, for example, a graphics processing unit and a memory controller.

The execution engine for a typical Intel Core processor design implements over 5000 distinct uops in several different units: the integer execution unit (IEU) contains logic for plain integer and miscellaneous other operations, the single instruction multiple data (SIMD) integer unit (SIU) contains logic for packed integer operations, the floating-point unit (FPU) implements plain and packed floating-point operations such as FADD, FMUL, FDIV, etc., the address generation unit (AGU) performs address calculations and access checks for memory accesses, the jump execution unit (JEU) implements jump

operations and determines and signals branch mispredictions, and the memory interface unit (MIU) receives load data from and passes store data to memory cluster.

Formal verification of execution datapaths, especially for floating-point and other arithmetic operations has been a focus area at Intel ever since the Pentium[®] FDIV bug in 1994. The primary vehicle for this work is symbolic simulation, incorporated in Intel's in-house Forte/reFLect verification toolset under the name of Symbolic Trajectory Evaluation (STE) [2]. All Intel Core processor execution engine data-paths since 2005, as well as most Intel Atom processor and Gen Graphics arithmetic engines have been formally verified using symbolic simulation [3], [6].

In formal verification, every uop corresponds to a separate symbolic simulation task. In the verification setup for a single uop the control signals are set to fix the data-path controls to match a single instance of that uop, and symbolic variables on the data are used to exhaustively simulate the data-path instance. The simulation is connected to an abstract functional reference model for the uop through source and write-back mappings, and the output of the design and the reference model compared. These design-dependent mappings extract the intended source and result values for the uop at the relevant times relative to the instance we are verifying.

Formal verification of complex designs would ideally be done by closed-box verification for its many advantages: a well-defined specification, no need of insight into implementation details, and low sensitivity to internal design changes. For a large majority of uops in the execution engine, the data-path can be exhaustively symbolically simulated in one pass at the full cluster level.

However, for complex floating-point arithmetic, such as multipliers, fused multiply-adders and dividers, the computation of symbolic expressions for the datapaths is fundamentally technically infeasible. Instead, the verification of these complex uops is done through a decomposed reference model that splits an operation to several sequential stages, where each stage of the reference model is separately related to a stage of the implementation. With such decomposition cut-points, we reduce symbolic simulation complexity, as each stage on its own produces smaller symbolic expressions than a full input-to-output closed-box simulation. For years, this has been the technique used for all the floating-point types traditionally implemented on Intel designs, i.e., single, double, and extended precision floats.

Decomposed verification is technically much harder than closed-box verification, requiring both special verification expertise and detailed insight into implementation details to map the decomposition stage boundaries to the design. It is also much more sensitive to even small design changes, making the maintenance cost high. Generally, the more stages the decomposition has, the harder the verification task is. The hardest datapath verification tasks on current Intel processor designs are the dividers, which need a series of decomposition stages and advanced complexity management strategies in each individual stage.

V. HALF-PRECISION FLOATING-POINT ARITHMETIC

Floating-point numbers are a binary representation for a subset of real numbers as triples (s, e, m) , where the sign s is a single bit, and the exponent e and mantissa m are unsigned bit vectors of some fixed lengths. The IEEE standards on floating-point numbers define several different formats differing on details, as well as special encodings for zeros, infinities, denormal numbers (very small numbers that are below the main range of values representable in a format), and other exceptional values [11]. Since only a subset of the reals is representable as floating-point numbers, not all results of arithmetic operations on floating-point numbers can be expressed precisely as floating-point numbers themselves. Therefore, the IEEE standards define the concept of rounding, determining which sufficiently close representable number should be used, if the accurate result is not representable.

Intel designs have traditionally supported three formats of floating-point numbers: single, double, and extended precision. Recently, as a part of the AVX-512 extension set in the latest Intel Core processor designs, support was added for a new shorter floating-point format, the so-called half-precision or FP16, consisting of one sign bit, five exponent bits and ten mantissa bits [7]. While the new format offers a narrower range and less precision, it allows twice as many values to be packed into a vector than with single-precision floats, doubling the effective performance of vectorized algorithms for applications that do not need higher precision arithmetic.

The architectural and micro-architectural instruction sets of the latest Intel Core processor designs support most common arithmetic half-precision operations natively. Some half-precision uops are implemented in dedicated design units, some others in units shared with higher precision arithmetic. Half-precision division and square root uops are implemented by an iterative design shared with the similar higher precision uops. In contrast to some higher precision operations, denormal input and output values are handled natively for half-precision arithmetic, without microcode assistance.

As the basic datapath for a half-precision uop has only half as many input data bits than the corresponding single-precision uop, we know that the size of symbolic expressions in its simulation is always lower than for single precision. Without experimentation we do not know how much lower, as the symbolic expression sizes can be at best linear and at worst exponential in the number of input bits, depending on the operation. What we do know is that any verification recipes that work for single precision should easily work for half precision. Also, we can realistically hope that the reduction in size might be large enough to obviate the need for decomposition for some of the complex operations, pushing them to the domain of closed-box verification, or at least reduce the decomposition needed. On the negative side, experience shows that native denormal handling tends to materially increase symbolic complexity, as denormals break the separation of exponent and mantissa datapaths. Also, we know that special care will be needed for uops implemented

in units shared between half precision and higher precisions to avoid the prohibitive cost of simulating also the higher precision behavior.

From this starting point, we carried out verification of all half-precision arithmetic uops on an Intel Core processor design. The technical learnings from the initiative can be summarized as follows:

- Simple floating-point uops such as comparisons, conversions to and from integers, reciprocals, etc., that allow closed-box verification for higher precisions, were easily verifiable for half precision. As anticipated, floating-point addition (FADD) could also be directly verified, in contrast with higher precisions, where FADD needs an exponent difference-based case split. Timed causal fanin analysis was essential in the separation of the simple uop and FADD datapaths from the complex ones implemented in the same design units.
- As the first result for known high complexity uops, we were able to verify floating-point multiplication (FMUL) directly without a decomposition. This is in marked contrast with higher precisions where decomposition is unavoidable, as the symbolic expression sizes for multiplication are known to be exponential. However, the lower number of mantissa bits for half precision means that we are not too far up the exponential curve yet in the basic datapath for the operation. For FMUL, the datapath is shared with the more complex fused multiply-add (FMA) operation. Timed causal fanin analysis helps FMUL verification by removing FMA-specific parts of the shared datapath, in particular in the rounding logic where FMUL exhibits only a narrow range of possible behaviors compared to FMA.
- Somewhat surprisingly, we were also able to verify half-precision fused multiply-add (FMA) uops without decomposition. This required careful complexity management, and a large case split on addend mantissa values to reduce the symbolic complexity of the basic datapath, with a high total run time. As FMA is the most complex operation on its shared datapath, there is no circuit logic that timed causal fanin analysis could just directly cut out. However, for each case in the case split, the simulation of the basic datapath alone approaches the capacity limits of the tool. How timed causal fanin analysis helps is by removing logic that is on the basic datapath, but is not relevant to the specific case.
- Finally, with heavy use of simplifying case splits and timed causal fanin analysis, we were able to carry out closed-box verification for half-precision division (FDIV) and square root (FSQRT) operations, as well. For division and square root, timed causal fanin analysis was indispensable, as the datapaths are mixed with the higher precision ones, and the long-latency uops have ample potential for uncontrolled symbolic expression growth.

The most complex arithmetic datapath proofs showed that for FP16, verification of all uops can be done closed box. In most

of these tasks and all high complexity ones, the contribution of timed causal fanin analysis cannot be quantified by the computation time or memory usage with the method vs without, since without either automated or manual weakening the closed-box verification tasks are computationally infeasible. In our view, the best metric is the human effort required for the effort.

The largest positive impact was observed on the operations that are traditionally the most complicated and heavy to verify. For FMUL, the first higher-complexity operation, we implemented a new verification strategy that did not include the decomposition that the higher-precision proof requires. Note that FMUL is in fact FMA without an addend, which makes it a lighter task for verification, however any bug we would catch on FMUL, also exists on FMA. We continued with a new verification strategy for the FMA operations: closed-box input-to-output verification with a case-split on addend mantissa value. The effort of FMA verification bring-up was reduced from several quarters for a higher-precision ‘big-FMA’ in a standard Intel Core processor development project, to a couple of weeks.

For FDIV and FSQRT the effort reduction was also substantial. The proof was dramatically simplified, compared to the traditional multi-stage decomposed higher-precision proof. The FDIV and FSQRT proofs were completed in 6-8 weeks and provided confidence in design quality and arithmetic correctness. Like the FMA, effort for these verification tasks is usually measured in quarters of work.

Comparing then automated vs manual generation of weakening lists, the simple uop and FADD verification likely could have been carried out with manual analysis, as these tasks are not computationally challenging and a coarse analysis would suffice. On the other hand, a manual separation of the FMUL logic from the FMA, or the logic used vs not used by the different FMA cases, and especially the separation of the FP16 FDIV and FSQRT datapaths from the higher precision ones would likely have required an extraordinary human effort focusing on design minutiae.

The main advantages of the closed-box verification that enabled quick results were clear specification, ease of failure reproduction in dynamic validation with concrete source values, and the absence of any need to locate cut-points and define complicated side conditions. The first corner-case datapath bug was found in less than a week of work. Altogether, the FP16 verification initiative caught several extreme complexity bugs in just a few weeks of works at an early stage of the design project. This reduced the design cost of fixing the issues, and most importantly prevented them from escaping to the silicon implementation. Here are two examples:

- 1) An FMA16 uop multiplies two small positive normal numbers, produces a very small intermediate value, and adds the addend – the smallest normal negative. The mathematically accurate result is tiny, between the smallest normal negative and zero. Since Flush-To-Zero (FTZ) mode was set, the result ought to be zero, but the design returned the smallest normal negative.

- 2) FMA received three very specific normal numbers as inputs, and FTZ was set. We expected to produce the smallest normal number after rounding to nearest, but the result was flushed to zero. The specific inputs were:
 - a: $s = 0$; $e = 00010$; $m = 1.0110000000$
 - b: $s = 0$; $e = 01111$; $m = 1.0001011011$
 - c: $s = 1$; $e = 00010$; $m = 1.1111111101$

The intermediate result of the operation after it was normalized was: $s = 1$; $e = 0$; $m = 1.111111111111$ – one extra bit after the mantissa length, which is exactly at half-point for rounding, and therefore needs to round up. After rounding and normalizing we got a normal (non-tiny) number: $s = 1$; $e = 1$; $m = 1.0000000000$, that should not have been flushed to zero.

VI. SUMMARY

Empirical experience has consistently shown that the timed causal fanin reduction algorithm is a key complexity reduction technique for practical symbolic simulation. It has also proven to be robust in face of design changes and over different design styles.

Timed causal fanin analysis was the primary enabler allowing us to verify all FP16 uops, including the most complex arithmetic operations, without decompositions. Closed-box verification greatly reduced the development effort of complex proofs, leading to fast detection of deep corner-case bugs in early stages of the project. Avoiding the use of decomposition has lowered the sensitivity to design implementation and made the verification collateral easily reusable for future projects.

REFERENCES

- [1] C. H. Seger and R. E. Bryant, “Formal verification by symbolic evaluation of partially-ordered trajectories,” *Formal Methods Syst. Des.*, vol. 6, no. 2, pp. 147–189, 1995.
- [2] C.-J. Seger, R. Jones, J. O’Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme, “An industrially effective environment for formal hardware verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, 2005.
- [3] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, “Replacing testing with formal verification in Intel Core i7 processor execution engine validation,” in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), pp. 414–429, Springer, 2009.
- [4] T. Melham, “Symbolic trajectory evaluation,” in *Handbook of Model Checking* (E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds.), ch. 25, pp. 831–870, Springer International Publishing, 2018.
- [5] R. Kaivola and J. O’Leary, “Verification of arithmetic and datapath circuits with symbolic simulation,” in *Handbook of Computer Architecture* (A. Chattopadhyay, ed.), Springer, 2022.
- [6] A. Gupta, M. V. A. KiranKumar, and R. Ghughal, “Formally verifying graphics FPU,” in *FM 2014: Formal Methods* (C. Jones, P. Pihlajasaari, and J. Sun, eds.), pp. 673–687, Springer International Publishing, 2014.
- [7] “Intel AVX512-FP16 Architecture Specification, June 2021, Revision 1.0.” <https://software.intel.com/content/www/us/en/develop/download/intel-avx512-fp16-architecture-specification.html>, 2021.
- [8] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. on Computers*, vol. C-35, pp. 677–691, August 1986.
- [9] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, “Formal verification using parametric representations of Boolean constraints,” in *Proc. of 36th ACM/IEEE Design Automation Conference*, pp. 402–407, 1999.
- [10] R. B. Jones, *Symbolic Simulation Methods for Industrial Formal Verification*. Springer, 2002.
- [11] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, 1985. Note: Standard 754–1985.

Divider Verification Using Symbolic Computer Algebra and Delayed Don't Care Optimization

Alexander Konrad¹ Christoph Scholl¹ Alireza Mahzoon² Daniel Große³ Rolf Drechsler²

¹University of Freiburg, Germany ²University of Bremen, Germany ³Johannes Kepler University Linz, Austria

{konrada, scholl}@informatik.uni-freiburg.de, {mahzoon, drechsle}@informatik.uni-bremen.de, daniel.grosse@jku.at

Abstract—Recent methods based on Symbolic Computer Algebra (SCA) have shown great success in formal verification of multipliers and – more recently – of dividers as well. In this paper we enhance known approaches by the computation of *satisfiability don't cares* for so-called *Extended Atomic Blocks (EABs)* and by *Delayed Don't Care Optimization (DDCO)* for optimizing polynomials during backward rewriting. Using those novel methods we are able to extend the applicability of SCA-based methods to further divider architectures which could not be handled by previous approaches. We successfully apply the approach to the fully automatic formal verification of large dividers (with bit widths up to 512).

I. INTRODUCTION

Arithmetic circuits are important components in processor designs as well as in special-purpose hardware for computationally intensive applications like signal processing and cryptography. At the latest since the famous Pentium bug [1] in 1994, where a subtle design error in the divider had not been detected by Intel's design validation (leading to erroneous Pentium chips brought to the market), it has been widely recognized that incomplete simulation-based approaches are not sufficient for verification and formal methods should be used to verify the correctness of arithmetic circuits. Nowadays the design of circuits containing arithmetic is not only confined to the major processor vendors, but is also done by many different suppliers of special-purpose embedded hardware who cannot afford to employ large teams of specialized verification engineers being able to provide human-assisted theorem proofs. Therefore the interest in *fully automatic formal verification* of arithmetic circuits is growing more and more.

In particular the verification of multiplier and divider circuits formed a major problem for a long time. Both BDD-based methods [2], [3] and SAT-based methods [4], [5] for multiplier and divider verification do not scale to large bit widths. Nevertheless, there has been great progress during the last few years for the automatic formal verification of gate-level multipliers. Methods based on *Symbolic Computer Algebra (SCA)* were able to verify large, structurally complex, and highly optimized multipliers. In this context, finite field multipliers [6], integer multipliers [7]–[19], and modular multipliers [20] have been considered. Here the verification task has been reduced to an ideal membership test for the

specification polynomial based on so-called backward rewriting, proceeding from the outputs of the circuit in direction of the inputs. For integer multipliers, SCA-based methods are closely related to verification methods based on word-level decision diagrams like *BMDs [21]–[23], since polynomials can be seen as “flattened” *BMDs [24]. Moreover, rewriting based approaches [25], [26] have recently shown to be able to verify complex multipliers as well as arithmetic modules with embedded multipliers at the register transfer level.

Research approaches for divider verification were lagging behind for a long time. Attempts to use Decision Diagrams for proving the correctness of an SRT divider [27] were confined to a single stage of the divider (at the gate level) [28]. Methods based on word-level model checking [29] looked into SRT division as well, but considered only a special abstract and clean sequential (i.e., non-combinatorial) divider without gate-level optimizations. Other approaches like [30], [31], or [32] looked into fixed division algorithms and used semi-automatic theorem proving with ACL2, Analytica, or Forte to prove their correctness. Nevertheless, all those efforts did not lead to a fully automated verification method suitable for gate-level dividers.

A side remark in [23] (where actually multiplier verification with *BMDs was considered) seemed to provide an idea for a fully automated method to verify integer dividers as well. Hamaguchi et al. start with a *BMD representing $Q \times D + R$ (where Q is the quotient, D the divisor, and R the remainder of the division) and use a backward construction to replace the bits of Q and R step by step by *BMDs representing the gates of the divider. The goal is to finally obtain a *BMD representation for the dividend $R^{(0)}$ which proves the correctness of the divider circuit. Unfortunately, the approach has not been successful in practice: Experimental results showed exponential blow-ups of *BMDs during the backward construction.

Recently, there have been several approaches to fully automatic divider verification that had the goal to catch up with successful approaches to multiplier verification: Among those approaches, [33] is mainly confined to division by constants and cannot handle general dividers due to a memory explosion problem. [34] works at the gate level, but assumes that hierarchy information in a restoring divider is present. Using this hierarchy information it decomposes the proof obligation $R^{(0)} = Q \times D + R$ into separate proof obligations for each level of the restoring divider. Nevertheless, the approach scales

This work was supported by the German Research Foundation (DFG) within the project VerA (SCHO 894/5-1, GR 3104/6-1 and DR 297/37-1) and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

only to medium-sized bit widths (up to 21 as shown in the experimental results of [34]).

The approaches of [24], [35] work on the gate level as well, but they do not need any hierarchy information which may have been lost during logic optimization. They prove the correctness of non-restoring dividers by “backward rewriting” starting with the “specification polynomial” $Q \times D + R - R^{(0)}$ (similar to [23], with polynomials instead of *BMDs as internal data structure). Backward rewriting performs substitutions of gate output variables with the gates’ specification polynomials in reverse topological order. They try to prove dividers to be correct by finally obtaining the 0-polynomial. The main insight of [24], [35] is the following: The backward rewriting method definitely needs “forward information propagation” to be successful, otherwise it provably fails due to exponential sizes of intermediate polynomials. Forward information propagation relies on the fact that the divider needs to work only within a range of allowed divider inputs (leading to input constraints like $0 \leq R^{(0)} < D \cdot 2^{n-1}$). [24] uses SAT-based information propagation (SBIF) of the input constraint in order to derive information on equivalent and antivalent signals, whereas [35] uses BDDs to compute satisfiability don’t cares which result from the structure of the divider circuit as well as from the input constraint. (Satisfiability don’t cares [36] at the inputs of a subcircuit describe value combinations which cannot be produced at those inputs by allowed assignments to primary inputs.) The don’t cares are used to minimize the sizes of polynomials. In that way, exponential blowups in polynomial sizes which would occur without don’t care optimization could be effectively avoided. Since polynomials are only changed for input values which do not occur in the circuit if only inputs from the allowed range are applied, the verification with don’t care optimization remains correct. In [35] the computation of optimized polynomials is reduced to suitable *Integer Linear Programming* (ILP) problems.

In this paper we make two contributions to improve [24] and [35]: First, we modify the computation of don’t cares leading to increased degrees of flexibility for the optimization of polynomials. Instead of computing don’t cares at the inputs of “atomic blocks” like full adders, half adders etc., which were detected in the gate level netlist, we combine atomic blocks and surrounding gates into larger fanout-free cones, leading to so-called *Extended Atomic Blocks (EABs)*, prior to the don’t care computation. Second, we replace local don’t care optimization by *Delayed Don’t Care Optimization (DDCO)*. Whereas local don’t care optimization immediately optimizes polynomials wrt. a don’t care cube as soon as the polynomial contains the input variables of the cube, DDCO only adds don’t care terms to the polynomial, but delays the optimization until a later time. This method has two advantages: First, by looking at the polynomial later on, we can decide whether exploitation of certain don’t cares is needed *at all*, and secondly, the later (delayed) optimization will take the effect of following substitutions into account and thus uses a more global view for optimization. Using those novel methods we are able to extend the applicability of SCA-based methods

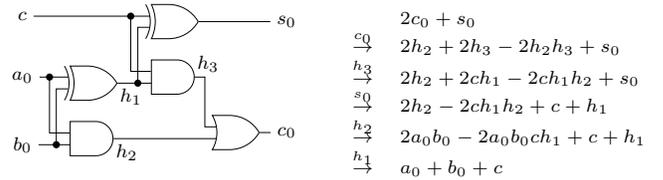


Fig. 1. Circuit with series of substitutions.

from [24], [35] to further optimized non-restoring dividers and restoring dividers which could not be handled by previous approaches.

The paper is structured as follows: In Sect. II we provide background on SCA and divider circuits. We motivate the need for novel optimizations by analyzing the existing approaches in Sect. III, and in Sect. IV we present the novel approach. The approach is evaluated in Sect. V and we conclude with final remarks in Sect. VI.

II. PRELIMINARIES

A. SCA for Verification

For the presentation of SCA we basically follow [24]. SCA based approaches work with polynomials and reduce the verification task to an ideal membership test using a Gröbner basis representation of the ideal. The ideal membership test is performed using polynomial division. While Gröbner basis theory is very general and, e.g., can be applied to finite field multipliers [6] and truncated multipliers [17] as well, for integer arithmetic it boils down to substitutions of variables for gate outputs by polynomials over the gate inputs (in reverse topological order), if we choose an appropriate “term order” (see [11] or [14], e.g.). Here we restrict ourselves to exactly this view.

For integer arithmetic we consider polynomials over binary variables (from a set $X = \{x_1, \dots, x_n\}$) with integer coefficients, i. e., a polynomial is a sum of terms, a term is a product of a monomial with an integer, and a monomial is a product of variables from X . Polynomials represent *pseudo-Boolean functions* $f : \{0, 1\}^n \mapsto \mathbb{Z}$.

As a simple example consider the full adder from Fig. 1. The full adder defines a pseudo-Boolean function $f_{FA} : \{0, 1\}^3 \mapsto \mathbb{Z}$ with $f_{FA}(a_0, b_0, c) = a_0 + b_0 + c$. We can compute a polynomial representation for f_{FA} by starting with a weighted sum $2c_0 + s_0$ (called the “output signature” in [10]) of the output variables. Step by step, we replace the variables in polynomials by the so-called “gate polynomials”. This replacement is performed in reverse topological order of the circuit, see Fig. 1. We start by replacing c_0 in $2c_0 + s_0$ by its gate polynomial $h_2 + h_3 - h_2h_3$ (which is derived from the Boolean function $c_0 = h_2 \vee h_3$). Finally, we arrive at the polynomial $a_0 + b_0 + c$ (called the “input signature” in [10]) representing the pseudo-Boolean function defined by the circuit. During this procedure (which is called *backward rewriting*) the polynomials are simplified by reducing powers v^k of variables v with $k > 1$ to v (since the variables are binary), by combining terms with identical monomials into one term, and by omitting terms with leading factor 0. We can

Algorithm 1 Restoring division.

```

1: for  $j = 1$  to  $n$  do
2:    $R^{(j)} := R^{(j-1)} - D \cdot 2^{n-j}$ ;
3:   if  $R^{(j)} < 0$  then
4:      $q_{n-j} := 0$ ;  $R^{(j)} := R^{(j)} + D \cdot 2^{n-j}$ ;
5:   else
6:      $q_{n-j} := 1$ ;
7:  $R := R^{(n)}$ ;

```

also consider $a_0 + b_0 + c = 2c_0 + s_0$ as the “specification” of the full adder. The circuit implements a full adder iff backward substitution, now starting with $2c_0 + s_0 - a_0 - b_0 - c$ instead of $2c_0 + s_0$, reduces the “specification polynomial” to 0 in the end. (This is the notion usually preferred in SCA-based verification.)

The correctness of the method relies on the fact that polynomials (with the above mentioned simplifications resp. normalizations) are canonical representations of pseudo-Boolean functions (up to reordering of the terms). (This is formulated as Lemma 1 in [35] and proven in [24], e.g..)

B. Divider Circuits

In the following we briefly review textbook knowledge on dividers. For more details, see [37], e.g.. We use $\langle a_n, \dots, a_0 \rangle := \sum_{i=0}^n a_i 2^i$ and $[a_n, \dots, a_0]_2 := (\sum_{i=0}^{n-1} a_i 2^i) - a_n 2^n$ for interpretations of bit vectors $(a_n, \dots, a_0) \in \{0, 1\}^{n+1}$ as unsigned binary numbers and two’s complement numbers, respectively. The leading bit a_n is called the sign bit. An unsigned integer divider is a circuit with the following property:

Definition 1. Let $(r_{2n-2}^{(0)} \dots r_0^{(0)})$ be the dividend with sign bit $r_{2n-2}^{(0)} = 0$ and value $R^{(0)} := \langle r_{2n-2}^{(0)} \dots r_0^{(0)} \rangle = [r_{2n-2}^{(0)} \dots r_0^{(0)}]_2$, $(d_{n-1} \dots d_0)$ be the divisor with sign bit $d_{n-1} = 0$ and value $D := \langle d_{n-1} \dots d_0 \rangle = [d_{n-1} \dots d_0]_2$, and let $0 \leq R^{(0)} < D \cdot 2^{n-1}$. Then $(q_{n-1} \dots q_0)$ with value $Q = \langle q_{n-1} \dots q_0 \rangle$ is the quotient of the division and $(r_{n-1} \dots r_0)$ with value $R = [r_{n-1} \dots r_0]_2$ is the remainder of the division, if $R^{(0)} = Q \cdot D + R$ (verification condition 1 = “vc1”) and $0 \leq R < D$ (verification condition 2 = “vc2”).

Note that we consider here the case that the dividend has twice as many bits as the divisor (without counting sign bits). This is similar to multipliers where the number of product bits is two times the number of bits of one factor. If both the dividend and the divisor are supposed to have the same lengths, we just set $r_{2n-2}^{(0)} = \dots = r_{n-1}^{(0)} = 0$ and require $D > 0$. Then $D > 0$ immediately implies $0 \leq R^{(0)} < D \cdot 2^{n-1}$.

The simplest algorithm to compute quotient and remainder is *restoring division* which is the “school method” to compute quotient bits and “partial remainders” $R^{(j)}$. Restoring division is shown in Alg. 1. In each step it subtracts a shifted version of D . If the result is less than 0, the corresponding quotient bit is 0 and the shifted version of D is “added back”, i. e., “restored”. Otherwise the quotient bit is 1 and the algorithm proceeds with the next smaller shifted version of D .

Non-restoring division optimizes restoring division by combining two steps of restoring division in case of a negative

Algorithm 2 Non-restoring division.

```

1:  $R^{(1)} := R^{(0)} - D \cdot 2^{n-1}$ ;
2: if  $R^{(1)} < 0$  then  $q_{n-1} := 0$  else  $q_{n-1} := 1$ ;
3: for  $j = 2$  to  $n$  do
4:   if  $R^{(j-1)} \geq 0$  then
5:      $R^{(j)} := R^{(j-1)} - D \cdot 2^{n-j}$ ;
6:   else
7:      $R^{(j)} := R^{(j-1)} + D \cdot 2^{n-j}$ ;
8:   if  $R^{(j)} < 0$  then  $q_{n-j} := 0$  else  $q_{n-j} := 1$ ;
9:  $R := R^{(n)} + (1 - q_0) \cdot D$ ;

```

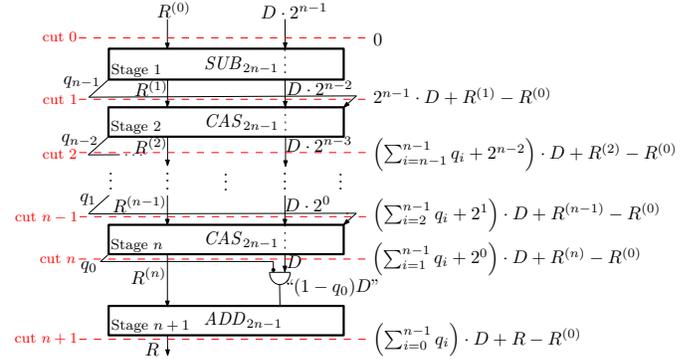


Fig. 2. Non-restoring divider.

partial remainder: adding the shifted D back and (tentatively) subtracting the next D shifted by one position less. These two steps are replaced by just adding D shifted by one position less (which obviously leads to the same result). More precisely, non-restoring division works according to Alg. 2.

SRT dividers are most closely related to non-restoring dividers, with the main differences of computing quotient bits by look-up tables (based on a constant number of partial remainder bits) and of using redundant number representations which allow to use constant-time adders. Other divider architectures like Newton and Goldschmidt dividers rely on iterative approximation. In this paper we restrict our attention to restoring and non-restoring dividers.

For dividers it is near at hand to start backward rewriting not with polynomials for the binary representations of the output words (which is basically done for multiplier verification), but with a polynomial for $Q \cdot D + R$. For a correct divider one would expect to obtain a polynomial for $R^{(0)}$ after backward rewriting. As an alternative one could also start with $Q \cdot D + R - R^{(0)}$ and one would expect that for a correct divider the result after backward rewriting is 0. This would be a proof for verification condition (vc1). (Then it remains to show that $0 \leq R < D$ (vc2) which we postpone until later.) This idea was already proposed by Hamaguchi in 1995 [23] in the context of verification using *BMDs [21]. As already mentioned in the introduction, Hamaguchi et al. observed exponential blow-ups of *BMDs in the backward construction and thus the approach did not provide an effective way for verifying large integer dividers.

However, this basic approach seems to be promising at first sight. As an example, Fig. 2 shows a *high level* view of a circuit for non-restoring division. Stage 1 implements a subtractor, stages j with $j \in \{2, \dots, n\}$ implement conditional

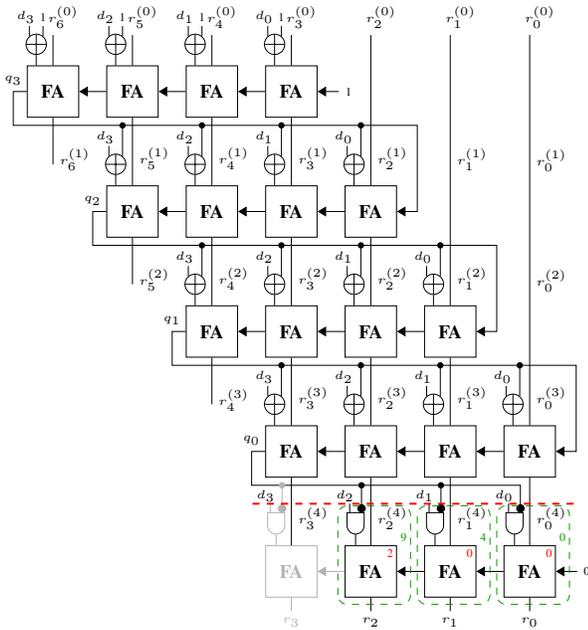


Fig. 3. Optimized non-restoring divider, $n = 4$.

adders / subtractors depending on the value of q_{n-j+1} , and stage $n+1$ implements an adder. If we start backward rewriting with the polynomial $Q \cdot D + R - R^{(0)}$ (which is quadratic in n) and if backward rewriting processes the gates in the circuit in a way that the stages shown in Fig. 2 are processed one after the other, then we would expect the following polynomials on the corresponding cuts (see also Fig. 2):

We would expect $(\sum_{i=1}^{n-1} q_i 2^i + 2^0) \cdot D + R^{(n)} - R^{(0)}$ for the polynomial at cut n which is obtained after processing stage $n+1$, since stage $n+1$ enforces $R = R^{(n)} + (1 - q_0) \cdot D$. For $j = n$ to 2 we would (by induction) expect $(\sum_{i=n-j+2}^{n-1} q_i 2^i + 2^{n-j+1}) \cdot D + R^{(j-1)} - R^{(0)}$ for the polynomial at cut $j-1$ after processing stage j , since stage j enforces $R^{(j)} = R^{(j-1)} - q_{n-j+1}(D \cdot 2^{n-j}) + (1 - q_{n-j+1})(D \cdot 2^{n-j}) = R^{(j-1)} + (1 - 2q_{n-j+1})(D \cdot 2^{n-j})$. Finally, the polynomial at cut 0 after processing stage 1 using the equation $R^{(1)} = R^{(0)} - D \cdot 2^{n-1}$ would reduce to 0 .

There may be two obvious reasons why backward rewriting might fail in practice all the same: (1) It could be the case that backward rewriting does not exactly hit the boundaries between the stages of the divider. (2) There may be significant peaks in polynomial sizes in between the mentioned cuts.

[24] and [35] show that there are additional obstacles apart from those obvious potential problems: In fact, with usual optimizations in implementations of non-restoring dividers the polynomials represented at the cuts between stages are different from this high-level derivation. The reason lies in the fact that the stages do not really implement signed addition / subtraction. In general, signed addition / subtraction of two $(2n - 1)$ -bit numbers leads to a $2n$ -bit number. The leading bit of the result can only be omitted, if “no overflow occurs”. The fact that no overflow occurs results from the input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ of the divider and from the way the results of the different stages are computed

[24]. Usual implementations even go one step further: By additional arguments using the input constraint and the circuit functionality it can be shown that it is not only possible to omit overflow bits of the adder / subtractor stages, but it is even possible to omit the computation of one further most significant bit. For a detailed analysis see [35]. These considerations lead to an optimized implementation shown in Fig. 3 for $n = 4$, e.g.. (For simplicity, we present the circuit before propagation of constants which is done however in the real implemented circuit.) In summary, it is important to note that (1) the stages in Fig. 3 cannot be seen as real adder / subtractor stages as shown in the high-level view from Fig. 2, (2) backward rewriting leads to polynomials at the cuts which are different from the ones shown in Fig. 2, and (3) unfortunately those polynomials have (provably) exponential sizes.

The conclusion drawn in [35] was that verification of (large) dividers using backward rewriting is infeasible, if there is no means to make use of “forward information” obtained by propagating the input constraint $0 \leq R^{(0)} < D \cdot 2^{n-1}$ in forward direction through the circuit. This idea indeed made it possible to verify large non-restoring dividers with bit widths up to 512 bits.

III. ANALYSIS OF EXISTING APPROACH

In this section we motivate our approach by analyzing weaknesses of the method from [35]. The algorithm from [35] starts with a gate level netlist and detects atomic blocks [16] like full adders and half adders. This results in a circuit with non-trivial atomic blocks (full adders, half adders etc.) and trivial atomic blocks (original gates not included in non-trivial atomic blocks). The method computes a topological order \prec_{top} on the atomic blocks with heuristics from [15], [16], computes satisfiability don’t cares [36] at the inputs of the atomic blocks, and performs backward rewriting starting with the specification polynomial $Q \cdot D + R - R^{(0)}$ by replacing atomic blocks in reverse topological order. During backward rewriting two optimization methods are used, if they are needed to keep polynomial sizes small: The first method uses information on equivalent and antivalent signals (which is derived by SAT-based information propagation (SBIF) using the input constraint and the don’t cares at the inputs of atomic blocks), the second method optimizes polynomials modulo don’t cares by reducing the problem to Integer Linear Programming (ILP).

A. Insufficient don’t care conditions

Let us start by considering stage $n+1$ of the non-restoring divider (see Figs. 2 and 3). Analyzing the method from [35] applied to optimized n -bit non-restoring dividers, we can observe that it does not make use of don’t cares at the inputs of atomic blocks corresponding to stage $n+1$ (although there exist some don’t cares), but it makes use of the (only existing) antivalence of q_0 and $r_{n-1}^{(n)}$ which is shown by SAT taking already proven satisfiability don’t cares into account (as already described above). If we only consider the circuit of stage $n+1$ (i.e., the circuit below the dashed

line in Fig. 3), replace $r_{n-1}^{(n)}$ by $\neg q_0$ (i.e. if we make use of the mentioned antivalence), and start backward rewriting with $(\sum_{i=0}^{n-1} q_i 2^i) \cdot (\sum_{i=0}^{n-1} d_i 2^i) + (\sum_{i=0}^{n-2} r_i 2^i - r_{n-1} 2^{n-1}) - (\sum_{i=0}^{2n-2} r_i^{(0)} 2^i)$, then we indeed obtain exactly the polynomial $(\sum_{i=1}^{n-1} q_i 2^i + 2^0) \cdot (\sum_{i=0}^{n-1} d_i 2^i) + (\sum_{i=0}^{n-2} r_i^{(n)} 2^i - (1 - q_0) 2^{n-1} - (\sum_{i=0}^{2n-2} r_i^{(0)} 2^i))$ which corresponds (with $(1 - q_0) = r_{n-1}^{(n)}$) to $(\sum_{i=1}^{n-1} q_i 2^i + 2^0) \cdot D + R^{(n)} - R^{(0)}$ as shown in Fig. 2, cut n . Fig. 4 shows the size of the final polynomial for stage $n + 1$ with increasing bit width n , with and without using the antivalence $r_{n-1}^{(n)} = \neg q_0$. Fig. 4 clearly shows that it is essential to make use of the mentioned antivalence.

Now we consider another version of the non-restoring divider which is slightly further optimized. It is clear that in a correct divider the final remainder is non-negative, i.e. $r_{n-1} = 0$. Therefore there is actually no need to compute r_{n-1} and the full adder shown in gray in Fig. 3 can be omitted. The verification condition `vc1` is then replaced by $R^{(0)} = Q \cdot D + \sum_{i=0}^{n-2} r_i 2^i$. Whereas in the original circuit making use of antivalences was essential for keeping the polynomial sizes small, in stage $n + 1$ of the further optimized version there are neither equivalent nor antivalent signals anymore. The only don't cares in the last stage (after constant propagation) are two value combinations at the inputs of the now leading full adder. However, making use of those don't cares does not help in avoiding an exponential blow up as Fig. 5 shows. Intuitively it is not really surprising that removing the full adder shown in gray potentially makes the verification problem harder, since the partial remainders $R, R^{(n)}, \dots, R^{(1)}$ in the high-level analysis of polynomials at cuts (see Fig. 2) represent signed numbers, but now R does not introduce a sign bit anymore.

Nevertheless, this raises the question whether the derivation of don't care conditions may be improved in a way that don't care optimization can avoid exponential blow ups like the one shown in Fig. 5.

B. Don't care optimization with backtracking

The method from [35] does not make use of don't care optimizations immediately, but stores a backtrack point after backward rewriting was applied to an atomic block which has don't cares at its inputs or has input signals with equivalent / antivalent signals. Whenever the polynomial grows too much, the method backtracks to a previously stored backtrack point and performs an optimization. Alg. 3 shows a simplified

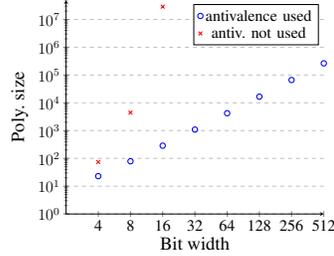


Fig. 4. Polynomial sizes, stage $n + 1$, optimized non-restoring divider.

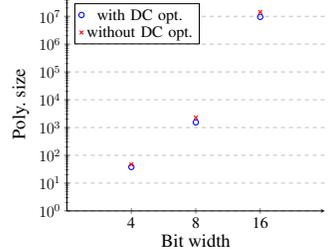


Fig. 5. Polynomial sizes, stage $n + 1$, further optimized non-restoring divider.

Algorithm 3 Backward rewriting with backtracking.

Input: Specification polynomial SP^{init} , Input constraint IC , Circuit CUV with atomic blocks $a_1 \prec_{top} \dots \prec_{top} a_m$ in topological order \prec_{top}
Output: 1 iff specification holds for all inputs satisfying IC
1: $SP_m := SP^{init}$; $oldsize := size(SP_m)$; $i := m$; $ST := \emptyset$;
2: $(dc(a_1), \dots, dc(a_m)) := \text{Compute_DC}(CUV, IC)$;
3: **while** $i > 0$ **do**
4: $SP_{i-1} := \text{Rewrite}(SP_i, a_i)$;
5: **if** $size(SP_{i-1}) > \text{threshold} \cdot oldsize$ **and** $ST \neq \emptyset$ **then**
6: $(SP, j) = \text{pop}(ST)$;
7: $i := j$; $SP_{i-1} := SP$;
8: $SP_{i-1} := \text{Opt_DC}(SP_{i-1}, dc(a_i))$;
9: **else**
10: **if** $dc(a_i) \neq \emptyset$ **then** $\text{push}(ST, (SP_{i-1}, i))$; $oldsize := size(SP_{i-1})$;
11: $i := i - 1$;
12: **return** $\text{evaluate}(SP_0)$;

overview of the approach.* For ease of exposition we omitted handling of equivalences / antivalences here.

As shown in [35], the approach works surprisingly well. It tries to restrict don't care optimizations (which are illustrated later on in Example 1, for more details see [35]) to situations where they are really needed. Only if the size threshold in line 5 is exceeded, backtracking is used and don't care optimization comes into play. A further analysis shows that the success of the approach in [35] is partly due to the following reasons: (1) In the non-restoring dividers used as benchmarks, atomic blocks that have any satisfiability don't cares grow only linearly with the bit width. (2) Only a linear amount of backtrackings is needed. (3) On the other hand, if backtrackings have to be used, don't care assignments have an essential effect in keeping the polynomials small (the size of the polynomials is quadratic in n just like the specification polynomial we start with).

Let us now consider a very simple example which does *not* have the mentioned characteristics.

Example 1. Consider a circuit which contains (among others) $2n + 1$ atomic blocks a_0, \dots, a_{2n} . Those blocks are the last atomic blocks in the topological order and $a_{2n} \prec_{top} \dots \prec_{top} a_0$. The initial polynomial is $SP^{init} = 8a + 4b + 2c + i_0$. a_0 has inputs x_1, i_1 , output i_0 , defines the function $i_0 = x_1 \vee i_1 = x_1 + i_1 - x_1 i_1$, and we assume that it has the satisfiability don't care $(x_1, i_1) = (0, 0)$. Correspondingly, for $j = 1, \dots, n$, a_j defines $i_j = x_{j+1} i_{j+1}$ with assumed satisfiability don't care $(x_{j+1}, i_{j+1}) = (0, 0)$, and for $j = n + 1, \dots, 2n$, a_j defines $i_j = x_{j+1} \vee i_{j+1} = x_{j+1} + i_{j+1} - x_{j+1} i_{j+1}$. We compute $size(p)$ as the number of terms in the polynomial p and assume $\text{threshold} = 1.5$ in line 5 of Alg. 3. Then Alg. 3 computes the following series of polynomials

$$\begin{aligned} SP_m &= 8a + 4b + 2c + i_0 \\ SP_{m-1} &= 8a + 4b + 2c + x_1 + i_1 - x_1 i_1 \\ SP_{m-2} &= 8a + 4b + 2c + x_1 + x_2 i_2 - x_1 x_2 i_2 \\ &\dots \end{aligned}$$

* SP_0 in Alg. 3 does not have to be 0 for correct dividers, it is sufficient that SP_0 evaluates to 0 for all inputs in the allowed input range $0 \leq R^{(0)} < D \cdot 2^{n-1}$. This can be checked by $\text{evaluate}(SP_0)$ in polynomial time [35].

$$\begin{aligned}
SP_{m-n-1} &= 8a + 4b + 2c \\
&\quad + x_1 + x_2 \dots x_{n+1}i_{n+1} - x_1x_2 \dots x_{n+1}i_{n+1} \\
SP_{m-n-2} &= 8a + 4b + 2c + x_1 + x_2 \dots x_{n+2} \\
&\quad + x_2 \dots x_{n+1}i_{n+2} - x_2 \dots x_{n+2}i_{n+2} \\
&\quad - x_1 \dots x_{n+2} - x_1 \dots x_{n+1}i_{n+2} + x_1 \dots x_{n+2}i_{n+2}
\end{aligned}$$

with sizes 4, 6, ..., 6, 10. SP_{m-n-2} is the first polynomial exceeding the size limit. For each of the $n + 1$ preceding atomic blocks there was a satisfiability don't care at the inputs, the size limit was not exceeded, and the corresponding polynomial has been pushed to the backtracking stack ST . Now backtracking to SP_{m-n-1} takes place. (Note that it is easy to see that without backtracking using don't care optimization the following $n - 1$ backwriting steps would quickly lead to a blowup in the polynomial sizes finally resulting in a polynomial with size $2^{n+2} + 2$.) SP_{m-n-1} is optimized with the don't care $(x_{n+1}, i_{n+1}) = (0, 0)$. Let us explain the idea of don't care optimization using this example: Don't care optimization adds $v \cdot (1 - x_{n+1}) \cdot (1 - i_{n+1})$ for the don't care $(x_{n+1}, i_{n+1}) = (0, 0)$ to SP_{m-n-1} with a fresh integer variable v . For all valuations $(x_{n+1}, i_{n+1}) \neq (0, 0)$, $v \cdot (1 - x_{n+1}) \cdot (1 - i_{n+1})$ evaluates to 0, thus we may choose an arbitrary integer value for v without changing the polynomial "inside the care space". The choice of v is made such that the size of SP_{m-n-1} is minimized. So the task is to choose v such that the size of $8a + 4b + 2c + x_1 + x_2 \dots x_{n+1}i_{n+1} - x_1x_2 \dots x_{n+1}i_{n+1} + v - vi_{n+1} - vx_{n+1} + vx_{n+1}i_{n+1}$ is minimal. We achieve this by using an ILP solver to get a solution for v which maximizes the number of terms with coefficients 0 and therefore minimizes the polynomial. It is easy to see that the best choice is $v = 0$ in this case. This means that we arrive at an unchanged polynomial SP_{m-n-1} and the don't care did not help. Then we do the replacement of a_{n+1} again, detect an exceeded size limit again, backtrack to SP_{m-n} and so on. Exactly as for SP_{m-n-1} , don't care assignment does not help for $SP_{m-n}, \dots, SP_{m-2}$. The first really interesting case occurs when backtracking arrives at SP_{m-1} . Adding $v \cdot (1 - x_1) \cdot (1 - i_1)$ with a fresh variable v to SP_{m-1} results in $8a + 4b + 2c + v + (1 - v)x_1 + (1 - v)i_1 + (v - 1)x_1i_1$ and choosing $v = 1$ leads to the minimal polynomial $8a + 4b + 2c + 1$ which is even independent from i_1 . Now replacing a_1, \dots, a_{2n} does not change the polynomial anymore and we finally arrive at $SP_{m-2n-1} = 8a + 4b + 2c + 1$ (without further don't care assignments).

The example shows that the backtracking method works in principle, but it comes at huge costs: Backtracking potentially explores all possible combinations of assigning or not assigning don't cares for atomic blocks with don't cares by storing backtrack points again in line 10 of Alg.3 after successful as well as unsuccessful don't care optimizations. In the example this leads to 2^{n+1} rewritings for atomic blocks and $2^{n+1} - 1$ unsuccessful don't care optimizations, before we finally backtrack to SP_{m-1} where we do the relevant don't care optimization.

Our goal is to come up with a don't care optimization

Algorithm 4 Computation of satisfiability don't cares.

Input: Input constraint IC , Circuit CUV with EABs $ea_1 \prec_{top} \dots \prec_{top} ea_l$ in topological order \prec_{top} , $dc_cand(ea_j) \forall j \in \{1, \dots, l\}$
Output: Satisfiability don't cares at inputs of EABs resulting from IC
1: $I = \{j \in \{1, \dots, l\} \mid dc_cand(ea_j) \neq \emptyset\}$; $i_{old} = 1$; $\chi = IC$;
2: $dc(ea_1) = \emptyset$; ...; $dc(ea_l) = \emptyset$;
3: **while** $I \neq \emptyset$ **do**
4: $i = \min(I)$; $slice = \{ea_{i_{old}}, \dots, ea_{i-1}\}$;
5: $\chi = compute_image(\chi, slice)$;
6: **for** $(\varepsilon_1, \dots, \varepsilon_n) \in dc_cand(ea_i)$ **do** $\triangleright x_1, \dots, x_n$: input signals of ea_i
7: **if** $\chi|_{x_1=\varepsilon_1, \dots, x_n=\varepsilon_n} = 0$ **then** $dc(ea_i) = dc(ea_i) \cup \{(\varepsilon_1, \dots, \varepsilon_n)\}$;
8: $I = I \setminus \{i\}$; $i_{old} = i$;
9: **return** $(dc(ea_1), \dots, dc(ea_l))$;

method which is robust against situations like the one illustrated in Example 1 where we have many blocks with don't cares, but only a few of those don't cares are really useful for minimizing the sizes of polynomials. As we will show in Sect. V, we run into such situations when we verify restoring dividers using the method from [35].

IV. DON'T CARE COMPUTATION AND OPTIMIZATION

A. Don't care computation for extended atomic blocks

This section is motivated by [8], [11] which combine several gates and atomic blocks into fanout-free cones, compute polynomials for the fanout-free cones first and use those precomputed polynomials for "macro-gates" formed by the fanout-free cones during backward rewriting. Whereas in [8], [11] the purpose of forming those fanout-free cones is avoiding peaks in polynomial sizes during backward rewriting without don't care optimization, the motivation here is different: Here we aim at detecting more and better don't cares.

First of all, we detect atomic blocks for fixed known functions like full adders and half adders as already mentioned in Sect. III. The result is a circuit with non-trivial atomic blocks and the remaining gates. Now we want to combine those atomic blocks and remaining gates into "extended atomic blocks (EABs)" which are fanout-free cones of atomic blocks and remaining gates. To do so, we compute a directed graph $G = (V, E)$ where the nodes correspond to the non-trivial atomic blocks, the remaining gates, and the outputs. There is an edge from a node v to a node w iff there is an output of the atomic block / gate corresponding to v which is connected to an input of the atomic block / gate / output node corresponding to w . We compute the coarsest partition $\{P_1, \dots, P_l\}$ of V such that for all sets P_i and all $v \in P_i$ with more than one successor it holds that all successors of v are not in P_i . We combine all gates / atomic blocks in P_i into an EAB ea_i .

The computation of satisfiability don't cares at the inputs of EABs that result from the input constraint IC (for dividers according to Def. 1 $IC = 0 \leq R^{(0)} < D \cdot 2^{n-1}$) is performed for EABs as described in [35] for atomic blocks. First of all, an intensive simulation (taking IC into account) excludes candidates for satisfiability don't cares. Value combinations at inputs of EABs that are seen in the simulation are excluded, finally resulting in a set $dc_cand(ea_j)$ for each EAB ea_j . Satisfiability don't cares at inputs of EABs are then computed by a series of BDD-based image computations [38] as shown

in Alg. 4, starting with IC . In the end we have classified all don't care candidates to be real don't cares or not.[†]

If we apply the method to the optimized divider in Fig. 3, the EABs below the dashed line are shown by dashed boxes. The number of satisfiability don't cares at the inputs of the dashed boxes (*after* constant propagation!) are shown at the right sides of the boxes just above the full adders. For the first EAB, the number of don't cares is 9, e.g., whereas for the atomic block (full adder) included in the EAB the number is only 2. At first sight, it is not clear that more don't cares really help during don't care based optimization, but we will show in Sect. V that this is definitely the case and that the use of *extended* atomic blocks is essential for a successful verification of large dividers.

B. Delayed Don't Care Optimization

In this section we introduce *Delayed Don't Care Optimization (DDCO)*. DDCO is based on the observation that don't care optimization as introduced in [35] is a *local* optimization that does not take its global effects into account. If backtracking goes back to a backtrack point with don't cares, then it backtracks to a situation where backward rewriting for an (extended) atomic block with don't cares at its inputs has taken place and the inputs of this block have been brought into the polynomial. The optimization locally minimizes the size of the polynomial using those don't cares immediately and the results of the optimization do not depend on rewriting steps which take place in the future. However, it is obvious that the future sizes of polynomials depend on the future substitutions during backward rewriting and therefore a local don't care optimization may go into the wrong direction. For that reason we propose a *delayed* don't care optimization taking future steps into account, which are performed after rewriting of the block for which the don't cares are defined. Before we will introduce DDCO, we illustrate the effect by an example.

Example 2. Consider the polynomial

$$p = x_1x_4x_5x_6 + x_2x_4x_5x_6 + x_3x_4x_5x_6 \\ - x_1x_2x_4x_5x_6 - x_1x_3x_4x_5x_6 - x_2x_3x_4x_5x_6 + x_1x_2x_3x_4x_5x_6$$

with size 7. Assume that the valuation $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$ is a don't care. By using the don't care optimization method from [35] which was already illustrated in Example 1, we arrive at a polynomial

$$q = p + vx_4x_5 - vx_1x_4x_5 - vx_2x_4x_5 - vx_3x_4x_5 + vx_1x_2x_4x_5 \\ + vx_1x_3x_4x_5 + vx_2x_3x_4x_5 - vx_1x_2x_3x_4x_5$$

with a new integer variable v . Since there is no pair of terms in q with the same monomials, $v = 0$ leads to the polynomial with the smallest number of terms. For all $v \neq 0$ q has the size 15 instead of 7. This shows that a local don't care optimization with don't care $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 1, 1)$

[†]It is easy to see that the don't care computation from Alg. 4 can be extended to a verification of vc2 (similar to [35]) just by adding a final step computing the image χ at the outputs. This way we obtain the image of the input constraint produced by the whole circuit. Then it has only to be checked whether χ implies $0 \leq R < D$.

Algorithm 5 Rewriting with DDCO.

Input: Specification polynomial SP^{init} ; Input constraint IC ; Circuit CUV with EABs $ea_1 \prec_{top} \dots \prec_{top} ea_m$ in topological order \prec_{top} ; EABs ea_i with input signals $x_1^{(i)}, \dots, x_{n_i}^{(i)}$; don't cares $dc(ea_i) = \{(\varepsilon_{1,1}^{(i)}, \dots, \varepsilon_{1,n_i}^{(i)}), \dots, (\varepsilon_{l_i,1}^{(i)}, \dots, \varepsilon_{l_i,n_i}^{(i)})\}$; "delay" d

Output: 1 iff specification holds for all inputs satisfying IC

```

1:  $SP_m := SP^{init}$ ;  $i := m + 1$ ;
2: while  $i - 1 > 0$  do
3:    $i := i - 1$ ;
4:    $SP_{i-1} := \text{Rewrite}(SP_i, ea_i)$ ;
5:   for  $j = 1$  to  $l_i$  do
6:      $SP_{i-1} := SP_{i-1} + v_j^{(i)} \cdot \prod_{\varepsilon_{j,k}^{(i)}=1} x_k^{(i)} \cdot \prod_{\varepsilon_{j,k}^{(i)}=0} (1 - x_k^{(i)})$ ;
7:   if  $i + d > m$  then continue;
8:    $SP_{i-1}^{tmp} := \text{assign\_dc}(SP_{i-1}, v_1^{(i+d-1)} = 0, \dots, v_{l_i}^{(i)} = 0)$ ;
9:    $dc0\_size := \text{size}(\text{assign\_dc}(SP_{i-1}^{tmp}, v_1^{(i+d)} = 0, \dots, v_{l_i+d}^{(i+d)} = 0))$ ;
10:  if  $dc0\_size \leq \text{increase}(\text{size}(SP_{i+d}))$  then
11:    for  $j = i - 1$  to  $i + d - 1$  do
12:       $SP_j := \text{assign\_dc}(SP_j, v_1^{(i+d)} = 0, \dots, v_{l_{i+d}}^{(i+d)} = 0)$ ;
13:  else
14:     $(z_1^{i+d}, \dots, z_{l_{i+d}}^{i+d}) := DC\_opt(SP_{i-1}^{tmp})$ ;
15:    for  $j = i - 1$  to  $i + d - 1$  do
16:       $SP_j := \text{assign\_dc}(SP_j, v_1^{(i+d)} = z_1^{i+d}, \dots, v_{l_{i+d}}^{(i+d)} = z_{l_{i+d}}^{i+d})$ ;
17:   $SP_0 := \text{assign\_dc}(SP_0, v_1^{(d)} = 0, \dots, v_{l_1}^{(1)} = 0)$ ;
18: return evaluate( $SP_0$ );

```

does not help in this example. Now assume that we perform a replacement of x_6 by $x_4 \cdot x_5$ in the polynomial q , resulting in

$$q' = vx_4x_5 + (1 - v)x_1x_4x_5 + (1 - v)x_2x_4x_5 + (1 - v)x_3x_4x_5 \\ + (v - 1)x_1x_2x_4x_5 + (v - 1)x_1x_3x_4x_5 + (v - 1)x_2x_3x_4x_5 \\ + (1 - v)x_1x_2x_3x_4x_5$$

Here it is easy to see that choosing $v = 1$ reduces q' to $q' = x_4x_5$. I.e., performing local don't care optimization before rewriting with $x_6 = x_4 \cdot x_5$ did not help and leads to a polynomial with 7 terms after the rewriting step, but don't care optimization after the rewriting step reduces the polynomial to a single term. By generalizing the example from 6 to an arbitrary number of n variables, we obtain $2^{n-3} - 1$ terms with don't care optimization before rewriting and one term with don't care optimization after rewriting, which shows that delayed don't care optimization can be exponentially better than local don't care optimization (even for a delay by one step only).

Alg. 5 shows an integration of DDCO into backward rewriting. In contrast to Alg. 3, it does not use backtracking and it always "delays" don't care optimization by d EAB rewriting steps. In the while loop from lines 2 to 16, don't care terms with fresh integer variables $v_j^{(i)}$ are immediately added to the polynomial SP_{i-1} for each don't care of the current EAB ea_i (line 6), but those don't cares may only be used with a delay of d EAB rewritings, i.e., in the iteration replacing ea_i only don't cares coming from ea_{i+d} may be used. Therefore, younger don't care variables are temporarily assigned to 0 in line 8, leading to a polynomial SP_{i-1}^{tmp} . Now the size of SP_{i+d} (which is the polynomial before rewriting with ea_{i+d}) is compared to the size $dc0_size$ of SP_{i-1} where the don't care variables from ea_{i+d} are assigned to 0 as well (i.e., they are not used). If $dc0_size$ did not increase too much compared to the size of SP_{i+d} ("too much" is specified by a monotonically increasing

function *increase*), then the don't care variables from ea_{i+d} are permanently assigned to 0 (lines 11 and 12) in the current as well as all previous polynomials containing those variables. Otherwise, the known ILP based don't care optimization is used and its results are inserted into SP_{i-1} and again also in all previous polynomials containing the don't care variables from ea_{i+d} (lines 14 to 16).

V. EXPERIMENTAL RESULTS

Our experiments have been carried out on one core of an Intel Xeon CPU E5-2643 with 3.3 GHz and 62 GiB of main memory. The run time of all experiments was limited to 24 CPU hours. All run times in Tables I, II and III are given in CPU seconds. We used the ILP solver Gurobi [39] for solving the ILP problems for don't care optimization of polynomials. For image computations we used the BDD package CUDD 3.0.0 [40]. For benchmarks and binaries see [41].

In our experiments we consider verification of three different types of divider benchmarks with different bit widths (Cols. 1 in Tabs. I to III). Tab. I shows results for non-restoring dividers “non-restoring₁” as seen in Fig. 3 (with the gray full adder included), which were also used in [35]. Table II contains results for further optimized non-restoring dividers “non-restoring₂” that omit the gray full adder shown in Fig. 3. Table III gives results for restoring dividers. All three tables share the same column labels. Note that we did not make use of any hierarchy information during verification, but only used the flat gate-level netlist (numbers of gates are shown in Cols. 2) and employed heuristics for detecting atomic blocks as well as for finding good substitution orders [15], [16].

We begin with three experiments for comparison where we check the equivalence of the divider circuits with a “golden specification”. In those experiments we restrict counterexamples to the allowed range $0 \leq R^{(0)} < D \cdot 2^{n-1}$ of inputs.

In the first experiment we used a SAT-solver (MiniSat 2.2.0 [42]) to solve the corresponding satisfiability problems. The results from Cols. 3 in Tabs. I, II, and III show that SAT-solving is hard for non-trivial arithmetic circuits and none of the benchmarks with bit widths larger than 8 could be solved in the specified time limit. In the second experiment we considered the combinational equivalence checking (CEC) approach of ABC [43], [44]. Since it is based on And-Inverter-Graph (AIG) rewriting via structural hashing, simulation, and SAT, the equivalence checking between two designs is reduced to finding equivalent internal AIG nodes. As for SAT-solving, ABC cannot verify the dividers with bit widths larger than 8, see Cols. 4 in Tabs. I, II, and III. In a third experiment we used a commercial verification tool. As Cols. 5 in Tabs. I, II, and III show, the commercial tool is able to verify also 16-bit dividers, for the restoring dividers it even verifies the 32-bit divider in about 15 CPU hours, but does not finish within the time limit for larger dividers.

From Col. 6 in Tab. I we can see that the method from [35] performs very well for the verification of the non-restoring₁ dividers. Col. 7 (“#bt”) shows how many backtrack operations were actually performed. For the non-restoring₂ benchmarks

considered in Tab. II the method exceeds the available memory for 16 bits and larger, for the restoring ones from Tab. III even already for 8 bits. As already shown by our analysis from Sect. III (see Fig. 5), equivalence/antivalence computation and don't care optimizations on atomic blocks as used in [35] are not strong enough to avoid exponential blowups of polynomials for the non-restoring₂ dividers. For restoring dividers the situation is similar.

In the next experiment we evaluate our new approach of using EABs for don't care computation instead of atomic blocks as used in [35] (at first without DDCO). For non-restoring₁ dividers (where the method from [35] already performed very well) this approach is somewhat slower than the original method, see Cols. 6 and 8 of Tab. I. The reason for this is that using EABs instead of atomic blocks as in [35] leads to more blocks where don't cares are applicable whereas the number of don't care optimizations which are really necessary stays the same. This can be seen in Cols. 7 and 9 of Tab. I which compare the number of performed backtracks. The version with EABs performs additional backtracks to backtrack points where optimization does not help and it has to store a larger amount of backtrack points. This even leads to running out of available memory for the 512-bit instance of non-restoring₁. But on the other hand already the usage of EABs enables to verify the non-restoring₂ dividers from Table II up to 256 bits in about 2 hours. Since don't care optimizations on atomic blocks as used in [35] are not strong enough to avoid exponential blowups for the non-restoring₂ dividers (as already mentioned above), using EABs is inevitable. However, the approach is not able to verify restoring dividers with bit widths larger than 64, see Col. 8 in Table III, due to increasing run times and memory consumption. This can be explained by the larger number of EABs with non-empty don't care sets for restoring dividers compared to non-restoring dividers. These numbers are given in Cols. 10 (“#EABs with DCs”) of Tabs. I and II for the non-restoring dividers and in Col. 10 of Tab. III for restoring dividers. The numbers grow only linearly for non-restoring dividers, but quadratically for restoring dividers. More EABs with non-empty don't care sets lead to an increased memory consumption by storing more backtrack points and to increased run times consumed by extensive backtracking. The effect occurring here has already been illustrated in Example 1 of Sect. III-B where we have to perform an exponential amount of unsuccessful backtracks before finally arriving at the relevant don't care optimization. For the 64-bit non-restoring₂ divider, e.g., the approach needs less than 50 seconds with 205 backtracks (Cols. 8, 9 of Tab. II) whereas the corresponding restoring divider only finishes in about 15 minutes with 3047 backtracks (Cols. 8, 9 of Tab. III).

Cols. 12 of Tabs. I, II, and III show that those difficulties can be overcome by using our novel DDCO method. It turned out that already the simplest possible parameter choice of $d = 1$ and $increase(size) = size + 1$ in Alg. 5 is successful. We were even able to verify the 256-bit restoring divider in less than 9.5 CPU hours and both 512-bit instances of non-restoring₁ and non-restoring₂ could be verified in about 7.5 hours. Comparing

TABLE I
VERIFYING DIVIDERS NON-RESTORING₁ FROM [35], TIMES IN CPU SECONDS.

n	#Gates	SAT time	ABC time	Com. time	[35]		[35]+EABs		Our method = [35]+EABs+DDCO			
					time	#bt	time	#bt	#EABs with DCs	#DC opt.	time	peak poly.
4	100	0.22	0.01	1.23	0.15	7	0.44	12	12	5	0.23	128
8	404	68.58	17.65	1.33	0.39	11	1.21	37	28	9	0.94	199
16	1,588	TO	TO	165.87	1.59	19	3.26	83	60	17	1.87	407
32	6,260	TO	TO	TO	5.06	35	12.10	166	124	33	6.78	1,207
64	24,820	TO	TO	TO	21.88	67	96.15	365	252	65	28.24	4,343
128	98,804	TO	TO	TO	114.73	131	1,434.11	909	508	129	153.71	16,759
256	394,228	TO	TO	TO	825.11	259	13,656.97	2,077	1,020	257	1,985.05	66,167
512	1,574,900	TO	TO	TO	9,183.28	515	MO	-	2,044	513	27,370.60	263,287

TABLE II
VERIFYING DIVIDERS NON-RESTORING₂, TIMES IN CPU SECONDS.

n	#Gates	SAT time	ABC time	Com. time	[35]		[35]+EABs		Our method = [35]+EABs+DDCO			
					time	#bt	time	#bt	#EABs with DCs	#DC opt.	time	peak poly.
4	96	0.23	0.01	1.21	0.17	8	0.26	17	11	5	0.23	61
8	400	31.83	16.78	1.86	2,486.89	31	0.99	21	27	9	0.95	117
16	1,584	TO	TO	108.23	MO	-	2.68	51	59	17	2.17	325
32	6,256	TO	TO	TO	MO	-	9.36	102	123	33	7.25	1,125
64	24,816	TO	TO	TO	MO	-	49.41	205	251	65	26.87	4,261
128	98,800	TO	TO	TO	MO	-	340.85	397	507	129	149.75	16,677
256	394,224	TO	TO	TO	MO	-	7,341.86	1,053	1,019	257	1,691.72	66,085
512	1,574,896	TO	TO	TO	MO	-	MO	-	2,043	513	27,351.10	263,205

TABLE III
VERIFYING RESTORING DIVIDERS, TIMES IN CPU SECONDS.

n	#Gates	SAT time	ABC time	Com. time	[35]		[35]+EABs		Our method = [35]+EABs+DDCO			
					time	#bt	time	#bt	#EABs with DCs	#DC opt.	time	peak poly.
4	140	0.27	0.01	1.21	2.59	17	0.47	35	16	8	0.38	61
8	700	14.88	14.27	1.49	MO	-	1.77	45	64	16	1.42	117
16	3,068	TO	TO	16.39	MO	-	8.41	171	256	32	6.63	325
32	12,796	TO	TO	53,277.73	MO	-	65.99	727	1,024	64	29.02	1,125
64	52,220	TO	TO	TO	MO	-	885.71	3,047	4,096	128	193.40	4,261
128	210,940	TO	TO	TO	MO	-	MO	-	16,384	256	2,244.24	16,677
256	847,868	TO	TO	TO	MO	-	MO	-	65,536	512	33,593.30	66,085
512	3,399,676	TO	TO	TO	MO	-	MO	-	262,144	-	TO	-

the numbers of EABs with non-empty don't care sets (Col. 10, "#EABs with DCs") with the actual numbers of don't care optimizations performed (Col. 11, "#DC opt.") in Tab. III, we observe that in particular for restoring dividers DDCO performs don't care optimizations only for a small fraction of the EABs with non-empty don't care sets. The effect is visible especially for larger instances. For the 256-bit divider this percentage is less than 1%, e.g..

Finally, Cols. 13 give the peak polynomial sizes during backward rewriting, counted in number of monomials. It can be observed that these peak sizes grow quadratically with the bit width. This shows that our methods are really successful in keeping the polynomial sizes small, since already the specification polynomial is quadratic in n .

In summary, the presented results show that our new method is able to successfully verify not only the divider benchmarks from [35], but also new divider architectures for which the previous approach fails.

VI. CONCLUSIONS AND FUTURE WORK

We analyzed weaknesses of previous approaches that enhanced backward rewriting in a SCA approach with forward information propagation and we presented two major contribu-

tions to overcome those weaknesses. The first contribution is the usage of Extended Atomic Blocks to enable stronger don't care computations. The second one is the new method of Delayed Don't Care Optimization which has two benefits: First, it performs don't care optimizations in a more global rewriting context instead of seeking for only local optimizations of polynomials, and second it is able to effectively minimize the number of don't care optimizations compared to considering all possible combinations of using / not using don't cares of EABs which can potentially occur in a backtracking approach. We showed that our new method is able to verify large divider designs as well as different divider architectures. For the future, we believe that the general approach of combining backward rewriting with forward information propagation will be a key concept to verify further divider architectures as well as other arithmetic circuits at the gate level.

REFERENCES

- [1] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobbs J.*, vol. 20, no. 4, pp. 129—135, 1995.
- [2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677—691, 1986.
- [3] J. R. Burch, "Using BDDs to verify multipliers," in *DAC*, 1991, pp. 408—412.

- [4] J. P. M. Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *DATE*. IEEE Computer Society / ACM, 1999, pp. 145–149.
- [5] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *DATE*. IEEE Computer Society, 2001, pp. 114–121.
- [6] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, 2013.
- [7] O. Wienand, M. Wedler, D. Stoffel, W. Kunz, and G. Greuel, "An algebraic approach for proving data correctness in arithmetic data paths," in *CAV*, 2008, pp. 473–486.
- [8] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.
- [9] M. Ciesielski, C. Yu, D. Liu, and W. Brown, "Verification of gate-level arithmetic circuits by function extraction," in *DAC*, 2015, pp. 52:1–52:6.
- [10] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.
- [11] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *DATE*, 2016, pp. 1048–1053.
- [12] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017, pp. 23–30.
- [13] C. Yu, M. Ciesielski, and A. Mishchenko, "Fast algebraic rewriting based on And-Inverter graphs," *TCAD*, vol. 37, no. 9, pp. 1907–1911, 2017.
- [14] D. Ritirc, A. Biere, and M. Kauers, "Improving and extending the algebraic approach for verifying gate-level multipliers," in *DATE*, 2018, pp. 1556–1561.
- [15] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.
- [16] —, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.
- [17] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019, pp. 28–36.
- [18] A. Mahzoon, D. Große, C. Scholl, and R. Drechsler, "Towards formal verification of optimized and industrial multipliers," in *DATE*, 2020, pp. 544–549.
- [19] D. Kaufmann, P. Beame, A. Biere, and J. Nordström, "Adding dual variables to algebraic reasoning for gate-level multiplier verification," in *DATE*. IEEE, 2022.
- [20] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, "Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability," in *DAC*, 2022, to appear.
- [21] R. E. Bryant and Y. A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *DAC*, 1995, pp. 535–541.
- [22] R. E. Bryant and Y. Chen, "Verification of arithmetic circuits using binary moment diagrams," *Int. J. Softw. Tools Technol. Transf.*, vol. 3, no. 2, pp. 137–155, 2001.
- [23] K. Hamaguchi, A. Morita, and S. Yajima, "Efficient construction of binary moment diagrams for verifying arithmetic circuits," in *ICCAD*, 1995, pp. 78–82.
- [24] C. Scholl and A. Konrad, "Symbolic computer algebra and sat based information forwarding for fully automatic divider verification," in *DAC*, 2020.
- [25] M. Temel, A. Slobodová, and W. A. Hunt, "Automated and scalable verification of integer multipliers," in *CAV*, 2020, pp. 485–507.
- [26] M. Temel and W. A. Hunt, "Sound and automated verification of real-world RTL multipliers," in *FMCAD*. IEEE, 2021, pp. 53–62.
- [27] J. E. Robertson, "A new class of digital division methods," *IRE Trans. Electronic Computers*, vol. 7, no. 3, pp. 218–222, 1958.
- [28] R. E. Bryant, "Bit-level analysis of an SRT divider circuit," in *DAC*, 1996, pp. 661–665.
- [29] E. M. Clarke, M. Khaira, and X. Zhao, "Word level model checking - avoiding the Pentium FDIV error," in *DAC*, 1996, pp. 645–648.
- [30] D. M. Russinoff, "A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor," *LMS Journal Comput. Math.*, vol. 1, pp. 148–200, 1998.
- [31] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," *Form Methods Syst. Des.*, vol. 14, no. 1, pp. 7–44, 1999.
- [32] J. O’Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating point hardware," *Intel Technology Journal*, vol. Q1, pp. 1–10, 1999.
- [33] A. Yasin, T. Su, S. Pillement, and M. J. Ciesielski, "Formal verification of integer dividers: Division by a constant," in *ISVLSI*, 2019, pp. 76–81.
- [34] —, "Functional verification of hardware dividers using algebraic model," in *VLSI-SoC*, 2019, pp. 257–262.
- [35] C. Scholl, A. Konrad, A. Mahzoon, D. Große, and R. Drechsler, "Verifying dividers using symbolic computer algebra and don’t care optimization," in *DATE*. IEEE, 2021, pp. 1110–1115.
- [36] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don’t cares for network optimization," in *ICCAD*, 1991, pp. 514–517.
- [37] I. Koren, *Computer arithmetic algorithms*. Prentice Hall, 1993.
- [38] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *ICCAD*, 1990, pp. 126–129.
- [39] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2020. [Online]. Available: <http://www.gurobi.com>
- [40] F. Somenzi, "Efficient manipulation of decision diagrams," *STTT*, vol. 3, no. 2, pp. 171–181, 2001.
- [41] A. Konrad, C. Scholl, A. Mahzoon, D. Große, and R. Drechsler, "Benchmarks and binaries," 2022. [Online]. Available: https://abs.informatik.uni-freiburg.de/src/projects_view.php?projectID=24
- [42] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518.
- [43] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.
- [44] "ABC: A system for sequential synthesis and verification," available at <https://people.eecs.berkeley.edu/~alanmi/abc/>, 2019.

Formally Verified Isolation of DMA

Jonas Haglund

dept. TCS

KTH Royal Institute of Technology

Stockholm, Sweden

jhagl@kth.se



Roberto Guanciale

dept. TCS

KTH Royal Institute of Technology

Stockholm, Sweden

robertog@kth.se



Abstract—Every computer having a network, USB or disk controller has a Direct Memory Access Controller (DMAC) which is configured by a driver to transfer data between the device and main memory. The DMAC, if wrongly configured, can therefore potentially leak sensitive data and overwrite critical memory to overtake the system. Since DMAC drivers tend to be buggy (due to their complexity), these attacks are a serious threat.

This paper presents a general formal framework for modeling DMACs and verifying under which conditions they are isolated. These conditions can be used as a specification for guaranteeing that a driver configures the DMAC correctly. The framework provides general isolation theorems that are common to all DMACs, leaving to the user only the task of verifying proof obligations that are DMAC specific. This provides a reusable verification infrastructure that reduces the verification effort of DMACs. Models and proofs have been developed in the HOL4 interactive theorem prover. To demonstrate the usefulness of the framework, we instantiate it with a DMAC of a USB.

Index Terms—formal verification, interactive theorem proving, DMA, I/O security, memory isolation

I. INTRODUCTION

Direct memory access controllers (DMACs) are hardware components transferring data between memory and I/O devices (e.g. memory-to-memory copies, and data transfers to and from network interface cards, USB, disks, and graphics accelerators). Without a DMAC, the CPU must perform these data transfers, spending time on data transfers rather than on applications, decreasing performance significantly [1]–[3], [44]. DMACs can also reduce power consumption since a CPU is more power demanding than a DMAC [4], [5], [44].

Since DMACs can access memory, where critical data and code are located, they can be used by attackers to overtake or crash the system. Examples include abusing a GPU DMAC to gain privilege escalation [9] and a network interface DMAC to crash Linux [10]. To prevent DMAC attacks, many formally verified high-security hypervisors and operating systems [23]–[30] either disable DMACs or rely on IOMMUs (memory management units [15]–[17] placed between the DMAC and memory). The use of IOMMUs have three significant disadvantages: not all hardware platforms have IOMMUs; it negatively impacts performance and further reduces time predictability (due to additional translation table walks [18],

[19]); and it requires additional non-trivial (potentially buggy [20]–[22]) software for configuring and protecting page tables and associated data structures.

Verifying memory safety in presence of DMACs and absence of IOMMUs require formal models of the DMAC hardware including the interface between DMAC, software and memory. Such models allow reasoning about the effects of software accessing DMAC registers, of DMAC memory accesses, and the interaction between of software and DMAC which share data structures in memory.

We present a general framework for modeling DMACs (Section III). The framework is implemented in the HOL4 interactive theorem prover [31] and includes a general DMAC model which can be instantiated to a given DMAC by defining 14 DMAC specific functions (the most significant ones are listed in Table II). This generalization allows us to identify and verify sufficient conditions to confine DMAC memory accesses to certain memory regions.

To achieve this general verification result, in Section IV we establish a refinement between an abstract DMAC model, which is easier to analyze, and identify sufficient conditions to preserve the refinement that must be satisfied by the DMAC instantiation and the DMAC driver. This strategy has three main benefits: (1) the refinement theorem can be reused to verify functional correctness of drivers using the abstract model; (2) the verification of the instantiation deals only with the identified sufficient conditions and do not have to deal with the entire transition system of the DMAC model; and (3) the software conditions can be verified using the abstract model.

In order for the framework to be as general as possible, we have reviewed numerous DMACs (Table I). In Section V we demonstrate our approach by instantiating the framework with the USB DMAC in an SoC from Texas Instruments [32]. We use our result to identify the conditions that must be satisfied by a driver or a security monitor. The use of the framework has largely reduced the time for analyzing the USB DMAC.

Finally, in Section VI we discuss the HOL4 implementation and the security analysis of the Linux USB DMAC driver.

II. BACKGROUND

DMACs perform memory accesses by operating on a queue of buffer descriptors (BDs), illustrated in Fig. 1, which are initialized by the driver. Each BD contains information about

Work partially supported by the TrustFull project financed by the Swedish Foundation for Strategic Research.

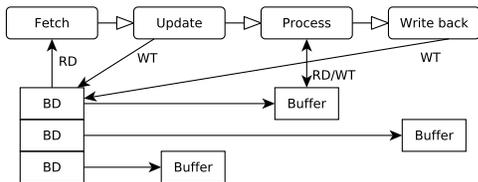


Fig. 1. DMAC

a memory transfer and the status of that transfer. The queue can be stored either in internal DMAC memory or in external main memory, either as a linked list (potentially cyclic), a ring, or as an array. Once the driver has initialized the BDs, the driver signals the DMAC to start operating on the BDs in the queue, which is done by a pipeline consisting of four stages. **(1) Fetch:** The DMAC fetches the BD into internal CPU-inaccessible memory. **(2) Update:** If the BD is operated on in multiple rounds, then the DMAC updates the BD to reflect the remaining transfers to perform for subsequent rounds. **(3) Process:** The DMAC performs the direct memory accesses (DMA transfers) to the buffers in main memory as specified by the BD. **(4) Write back:** If all memory accesses specified by the BD have been performed, the DMAC writes back the BD to signal the driver that the BD has been processed and can be reused for new transfers.

In the following we use $\mathcal{O} = \{f, u, p, w\}$ to refer to these four operations. The DMAC may also perform memory accesses due to **maintenance** operations, for example to store statistics or management data in memory. These operations are not atomic and may require multiple memory accesses. Furthermore, DMACs may be able to work on multiple queues of BDs concurrently, where each queue constitutes one DMA channel, and each channel may have more than one BD in each of its pipeline stages.

Both the driver and the DMAC can read and modify the queues: The driver reads the status of existing BDs and appends new BDs; the DMAC reads and updates BDs. For this reason verifying properties of this kind of system is challenging and similar to verifying concurrent threads sharing memory. In order to control the complexity caused by the interleaving of these the CPU/driver and the DMAC, the verification must exploit some sort of rely/guarantee [6], that enables verification of each component in isolation while assuming properties of the other component. Our verification approach follows this strategy, showing that there are sufficient conditions (rely) that if met by the driver allow to restrict (guarantee) the memory accesses of the DMAC.

A. DMAC Characteristics

In order to support a wide range of DMACs, our general model must accurately describe the memory accesses that may be performed by an arbitrary DMAC. To identify the common features of DMACs, we studied eight stand-alone DMACs, six embedded in USB controllers, and five embedded in Ethernet controllers, and the DMAC of IBM Cell, some characteristics of which are listed in Table I. The main difference among the

Stand-alone DMACs		
Chip	BD Organization	BD Location
Texas Instruments AM335x	Linked list	Internal memory
Microchip PIC32 Family	Linked list	Internal memory
Xilinx AXI DMA v7.1	Linked list	Main memory
NXP MPC5675/KMPC57xx	Linked list	Internal memory
Infineon GPDMA	Linked list	Main memory
Broadcom BCM2835	Linked list	Main memory
ST Microelectronics STR91xFA	Linked list	Main memory
Texas Instruments TMS320C5515	Linked list	Main memory
IBM Cell BE	Array/Ring	Main memory
USB DMACs		
Chip	BD Organization	BD Location
Cypress EZ-USB FX3	Linked list	Main memory
Xilinx Zynq-7000	Linked list	Main memory
Texas Instruments AM335x	Linked list	Main memory
NXP SAF1761 USB OTG	One BD	Internal memory
STM32F72xxx/STM32F73xxx	One BD per channel	Internal memory
Microchip PIC32 Family	Ring	Main memory
NIC DMACs		
Chip/Board	BD Organization	BD Location
Texas Instruments AM335x	Linked list	Internal memory
Broadcom NetXtreme/Netlink	Ring	Main memory
Realtek Ethernet RTL8100	Ring	Internal memory
3Com 3C90x/B	Linked list	Main memory
Intel e1000/e, X550, I350, I210	Ring	Main memory

TABLE I
STUDIED DMACs.

DMACs is the mechanism used to organize BD queues: 13 DMACs use linked lists; five use ring buffers; and two use queues of one single BD. Moreover, seven DMACs store the queues in internal memory and 13 store the queues in main memory; Furthermore, DMACs have different: internal states (e.g., address pointers, counters, and state machines); number of DMA channels; reactions to register accesses made by the CPU; scheduling of channels; BD format (e.g. fields for buffer start address and size); and behavior of the four pipeline stages (fetch, update, process, and write back).

B. Security Threat from DMACs

Without an IOMMU, a DMAC can access memory without restrictions. For instance, consider a microkernel (or a hypervisor), where a user-mode driver (or a guest) should not be able to directly access kernel memory. If the driver can directly configure a DMAC that can perform memory-memory transfers, then the driver could store a malicious program in its own memory, and configure the DMAC to transfer this buffer to the exception handling table of the kernel. This results in code injection, bypassing the normal protection provided by the MMU that prevents direct tampering from the driver. Similarly, the driver of an Ethernet controller may overwrite kernel data structures with an incoming network packet or to leak data in kernel memory.

In order to isolate a DMAC, its configuration must meet three sufficient conditions, which are all violated by the example of Fig. 2:

- 1) BDs specify DMA reads and writes to buffers that are considered “readable” and “writable”: BD1 can instruct the DMAC to violate isolation since part of the buffer is outside the allowed memory region.

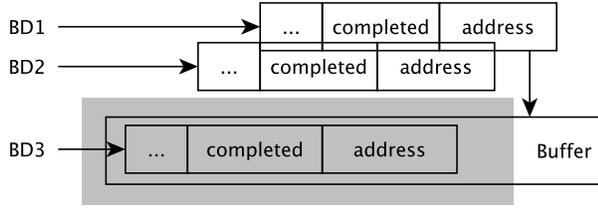


Fig. 2. DMAC isolation violations. Readable and writable region is colored in gray.

- 2) If BDs are stored in main memory, then the BDs must be located in “readable” and “writable” memory and must not specify DMA writes to BDs: The DMAC will violate memory isolation when fetching, updating and writing back BD1. Also, the DMAC can modify BD3 while processing BD1, since BD3 overlaps the buffer addressed by BD1.

Basically, these conditions guarantee that the BDs “instruct” the DMAC to access only “readable” and “writable” memory, and that the DMAC cannot change such BD “instructions”.

III. GENERAL DMAC MODEL

We assume a computer system to be the composition $c|m|d$, where each component represents the state of a CPU, a memory and a DMAC respectively. We use standard synchronous composition of the transition systems of the components (assuming that parallel composition is associative, symmetric, and commutative):

$$\frac{x \xrightarrow{\tau} x'}{x|y \xrightarrow{\tau} x'|y} \quad \frac{x \xrightarrow{l} x' \quad y \xrightarrow{\bar{l}} y'}{x|y \xrightarrow{\tau} x'|y'}$$

The labels of these transition systems are τ for internal operations, and $rd(as, bs)/wt(as, bs)$ for reading/writing the bytes bs at/to the locations with addresses as , where the latter two have co-labels $\bar{rd}(as, bs)$ and $\bar{wt}(as, bs)$.

We do not explicitly define the CPU model. This model could for instance be the formalization of an Instruction Set Architecture (ISA) or a more abstract model of a device driver. Memory is an array of bytes, where \mathcal{M} represents the addresses of the main memory:

$$\frac{as \subseteq \mathcal{M}}{m \xrightarrow{rd(as, m[as])} m} \quad \frac{as \subseteq \mathcal{M}}{m \xrightarrow{wt(as, bs)} m[as \mapsto bs]}$$

Notice that we use early semantics: the memory is always ready to receive a memory update non-deterministically selecting all possible bytes bs . This non-determinism is resolved when the the memory transitions system is composed with another transition system that performs a write.

A. DMAC Transition System

The DMAC state consists of three components, $d = (s, b, c)$: An internal state s , whose type depends on the specific DMAC; a message box b containing memory requests and replies; and a DMA channel c (the model supports multiple channels, but

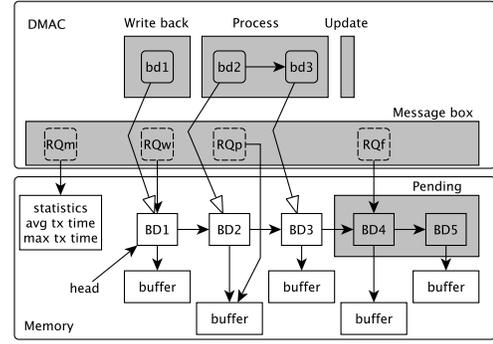


Fig. 3. DMAC model.

we omit them for simplicity). We will use Fig. 3 to illustrate the model, where a queue of five BDs has been configured in main memory and each BD points to a buffer.

The message box b allows the DMAC to operate asynchronously w.r.t. the memory. This box is a set of memory read and write requests and replies: $r_t^{op}[as]$, $w_t^{op}[as, bs]$ and $p_t^{op}[bs]$, where op , t , as and bs denote: The DMAC pipeline or maintenance operation $\mathcal{O} \cup \{m\}$ that issued the memory request or that shall have the reply; a memory request-reply identifier tag; addresses to read/write; and bytes read/written.

The component $c : \mathcal{O} \setminus \{f\} \leftrightarrow \mathcal{B}$ models the DMAC pipeline. In the following we use $c.op$ to denote $c(op)$. Hence, $c.u = [bd_1, \dots, bd_n]$ denotes the queue of BDs in the update stage, with n arbitrary and $n = 0$ denoting an empty queue; and similarly for p and w . We call these abstract BDs, since they are records whose type depends of the specific DMAC and contain the same information that is stored by the BDs in main memory. Independently of the DMAC instantiation, a BD bd always contains four mandatory fields specifying the addresses of the locations: where it is stored $bd.ra$, that are updated when it is written back $bd.wa$ (e.g., the address of its completion flag), and of the buffer that must be read and written via DMA, $bd.dra$ and $bd.dwa$. The BDs in c are the ones that have been fetched with each BD being in some DMAC pipeline stage. For instance, in Fig. 3 three BDs have been fetched and are therefore in the DMAC pipeline (bd2 and bd3 are being processed and bd1 is currently written back). We use “pending” BDs to refer to the BDs in the queue that are left to fetch (e.g. BD4 and BD5). Normally, the concatenation of the queues in c represents a sliding window of the queue in memory.

To account for the DMAC specifics the rules describing DMAC transitions are defined in terms of two records. The record Δ contains behavioral functions that model the specific actions of a DMAC. The record Π contains projection functions that extract information from the state and returns the proper data structures (e.g., BDs). These DMAC specific functions must be defined to obtain a concrete DMAC model. Table II summarizes the behavioral functions (except for a scheduler that resolves non-determinism) and the two most important projection functions.

Function	Modeled Operation/State Information
$\Delta.r_r$	(See rule $[rr]$) Given an internal state s_1 and the addresses as of the DMAC register to read, returns an updated internal state s_2 , the read bytes bs , and potential maintenance memory requests rs associated with the read.
$\Delta.w_r$	(See rule $[wr]$) Given an internal state s_1 , the addresses as of the DMAC register to write, the bytes bs to write, returns an updated internal state s_2 and potential maintenance memory requests rs associated with the write.
$\Pi.fas$	(See rule $[f_1]$) Given an internal state s , returns the memory read request $r_t^f[as]$ for fetching the next part of the BD being fetched at addresses as and with request identification tag t .
$\Delta.f$	(See rules $[f_2]$ and $[f_3]$) Given an internal state s_1 , and for external BDs a fetch reply $p_t^f[bs]$, (where the bytes bs constitutes a part of the currently fetched BD and with t being the request identification tag of the corresponding read request), but \perp for internal BDs; returns an updated internal state s_2 , and either a fetched BD bd and the stage $op \in \{u, p\}$ the BD shall be moved to, or \perp if additional external or internal memory reads are necessary to fetch the next BD.
$\Delta.p$	(See rule $[p_t]$) Given an internal state s_1 , the first BD bd in the process stage and whose memory transfers are currently being performed (i.e., the DMA transfers specified by bd), and the DMA read replies ps associated with the process stage; returns an updated internal state s_2 reflecting the processing of the given memory replies and the generation of potentially new memory requests rs , and a boolean flag indicating whether all requests/replies associated with bd have now been issued/processed and the BD shall be moved to the write back queue.
$\Delta.w$	(See rule $[w]$) Given an internal state s_1 , and the BDs in the write back queue $c.w$; returns an updated internal state s_2 , the memory write requests rs containing the bytes to write to memory associated with any given BD (not used for internal BDs), and the BDs bds that are now released due to the write back (removed from the write back queue).
$\Delta.m$	(See rule $[m]$) Given an internal state s_1 and memory read replies ps (to read requests issued by $[rr]$ and $[wr]$); returns an updated internal state s_2 and the processed replies pps that shall be removed from the message box.
$\Pi.cf$	(See rules $[w]$ and $[ma]$ in Subsection IV-A) Given internal state s and memory m , returns the pending BDs bds that remains to fetch ($bds = [BD4, BD5]$ in Fig. 3).

TABLE II
SUMMARY OF THE DMAC SPECIFIC FUNCTIONS.

In the following we use \mathcal{D} to represent the set of addresses of DMAC registers. The reaction of the DMAC when the CPU accesses such a register at addresses as is DMAC specific and must be described by the Read Register and Write Register functions: $\Delta.r_r$ and $\Delta.w_r$. Notice that these functions can affect the internal state of the DMAC and may return memory requests rs in case a register access makes it necessary for the DMAC to update maintenance data in main memory ($c = a + b$ denotes $c = a \cup \{b\} \wedge a \cap \{b\} = \emptyset$):

$$\frac{(s_2, bs, rs) = \Delta.r_r(s_1, as) \quad as \subseteq \mathcal{D}}{(s_1, b, c) \xrightarrow{rd(as, bs)} (s_2, b + rs, c)} [rr]$$

$$\frac{(s_2, rs) = \Delta.w_r(s_1, as, bs) \quad as \subseteq \mathcal{D}}{(s_1, b, c) \xrightarrow{wt(as, bs)} (s_2, b + rs, c)} [wr]$$

The message box acts as a buffer between the memory and the DMAC. The message box synchronizes with memory,

consuming a request (previously produced by operation op and with identifier t) and for reads adding a corresponding reply:

$$(s, b + r_t^{op}[as], c) \xrightarrow{rd(as, bs)} (s, b + p_t^{op}[bs], c) [rm]$$

$$(s, b + w_t^{op}[as, bs], c) \xrightarrow{wt(as, bs)} (s, b, c) [wm]$$

The other rules are for internal DMAC transitions. For fetching BDs ($op = f$) there are five cases: three if BDs are stored in main memory and two if BDs are stored in internal memory. $[f_1]$ describes the first step in fetching an external BD, that is applicable when there are no pending memory replies for BD fetches. In this case a memory request is added to the message box for fetching new BDs. In Fig. 3, the rule can produce the request RQf when starting to fetch BD4. The addresses and the tag are given by the function Fetch Addresses $\Pi.fas$:

$$\frac{\{p_t^f[bs] \in b\} = \emptyset \quad r_t^f[as] = \Pi.fas(s)}{(s, b, c) \xrightarrow{\tau} (s, b + r_t^f[as], c)} [f_1]$$

When a memory read request for fetching a BD is served, the corresponding reply is added to the message box. $[f_2]$ describes the behavior when such a reply exists but more reads are necessary to fetch the complete BD, in which case the function Fetch $\Delta.f$ returns \perp . $\Delta.f$ can update the internal state with the consumed reply, which contains a partial BD:

$$\frac{(s_2, \perp) = \Delta.f(s_1, p_t^f[bs])}{(s_1, b + p_t^f[bs], c) \xrightarrow{\tau} (s_2, b, c)} [f_2]$$

$[f_3]$ handles the case when a BD fetch reply $p_t^f[bs]$ exists and it contains the last chunk of bytes bs of the BD bd being fetched. In this case $\Delta.f$ returns a pair consisting of the abstract representation of the fetched BD bd and which pipeline stage queue $op \in \{u, p\}$ the BD shall be appended to (denoted by $\#$):

$$\frac{(s_2, (bd, op)) = \Delta.f(s_1, p_t^f[bs])}{(s_1, b + p_t^f[bs], c) \xrightarrow{\tau} (s_2, b, c[op \mapsto c.op \# bd])} [f_3]$$

The fetching BD rules for DMACs with internal BDs are similar to $[f_2]$ and $[f_3]$, but no memory requests and replies are involved, since BDs are obtained from the internal DMAC state.

Two rules model the process stage ($op = p$), depending on whether the currently processed BD is now completed or not. The following rule covers the case when a BD is completely processed (the other case when more DMA transfers remain of the BD is similar, but keeps the BD at the head of the process queue). In either case, the function Process $\Delta.p$ models the DMAC specific behavior of generating and processing memory requests and replies. It takes the currently processed BD bd at the head of $c.p$, and pending memory replies for the process stage; and returns an updated internal state, optional new memory requests rs , and a completion flag which specifies if the BD has now been processed and shall be moved to the write back stage. These requests represents DMA

reads and writes, while the replies are the results of previously issued read requests that have been served by memory. All replies are consumed and the new requests are added to the message box. In Fig. 3 the rule can produce the request RQp to write the buffer addressed by $bd2$.

$$\frac{c.p = bd :: bds \quad ps = \{\mathbf{p}_t^p[bs] \in b\} \quad (s_2, rs, \mathbf{true}) = \Delta.p(s_1, bd, ps)}{(s_1, b, c) \xrightarrow{\tau} (s_2, b - ps + rs, c[p \mapsto bds, w \mapsto c.w \# bd])} [p_t]$$

Updating and writing back BDs are similar and for this reason we only describe write back in detail. The main difference is that updating a BD moves the updated BD from the head of update queue to the tail of the process queue, while a write back may remove a (possibly empty) prefix of BDs from the write back queue $c.w$. If BDs are stored in main memory, the Write back function $\Delta.w$ returns the memory write requests rs for writing back the BDs, while internal BDs are written back by updating the internal state (in Fig. 3 the rule can produce the request RQw to update $bd1$ in main memory):

$$\frac{(s_2, rs, bds) = \Delta.w(s_1, c.w)}{(s_1, b, c) \xrightarrow{\tau} (s_2, b + rs, c[w \mapsto c.w - bds])} [w]$$

Finally, the DMAC can react to the replies ps to the read requests produced by the maintenance operations (i.e., requests issued by $[rr]$ and $[wr]$), removing the processed replies $pps \subseteq ps$ from the message box:

$$\frac{ps = \{\mathbf{p}_t^m[bs] \in b\} \quad (s_2, pps) = \Delta.m(s_1, ps)}{(s_1, b, c) \xrightarrow{\tau} (s_2, b - pps, c)} [m]$$

IV. VERIFICATION

Our goal is to verify general conditions that are sufficient to guarantee DMAC isolation (Theorem 1): The DMAC can only read “readable” and write “writable” memory regions, denoted by the sets of addresses \mathcal{R} and \mathcal{W} .

Our verification is based on refinement. Let M_3 be the DMAC model defined in Section III. We introduce two layered abstractions M_2 and M_1 . For each model M_{i+1} we introduce an invariant \mathcal{I}_{i+1} that allows us to prove bisimulation between M_{i+1} and M_i . We finally introduce an invariant \mathcal{I}_1 for M_1 that demonstrate DMAC isolation and use the bisimulation to transfer this property down to the M_3 DMAC model. This strategy has three benefits: (i) it allows us to solve one problem at a time via a single refinement step; (ii) it establishes a bisimulation between the concrete model and the more abstract one, which allows further properties (e.g., functional correctness of a device driver) to be verified using abstract models; (iii) it allows us to identify assumptions that all DMAC instantiations and drivers must satisfy in the form of proof obligations. The obligations must be proved for a given DMAC instantiation, but these proofs depend only on the instantiation (Δ and Π) in contrast to a complete DMAC model. The driver conditions can be proven relying only on the DMAC guarantee that are established by our verification.

A. Abstract DMAC Models

The lower abstraction M_2 is a virtual DMAC that cannot self-modify pending BDs. This property allows a driver to prepare, extend, and read the queue that must be fetched by the DMAC without being concerned that the DMAC may alter the queue. This is done by checking that pending BDs are not addressed by BD updates, write backs, and DMA writes. For instance, the rule for write back becomes (where $a \not\subseteq b$ means that sets a and b are disjoint: $a \cap b = \emptyset$):

$$\frac{(s_2, rs, bds) = \Delta.w(s_1, c.w) \quad \left(\bigcup_{bd \in bds} bd.wa \cup \bigcup_{\mathbf{w}_t^{op}[as, bs] \in rs} as \right) \not\subseteq \bigcup_{bd \in \Pi.cf(m, s_1)} bd.ra}{(s_1, b, c) \xrightarrow{\tau} (s_2, b + rs, c[w \mapsto c.w - bds])} [w]$$

The rule prevents write backs from modifying pending BDs, independently of whether the BDs are stored in internal or main memory. For internal BDs, the locations modified by $\Delta.w$ are identified from the list of released BDs bds . For external BDs, the addresses are in the requests rs produced by $\Delta.w$. $\Pi.cf$ returns the list of remaining (Concrete) pending BDs to Fetch, as identified by the internal state and memory (BD4 and BD5 in Fig. 3).

The upper abstraction M_1 guarantees that BDs cannot be changed by the CPU. The pending BDs to fetch are stored in an abstract queue $c.f$. By definition the CPU cannot modify or remove entries from this list, but it can append BDs by either: writing a DMAC register (e.g. by writing the tail pointer register or by writing the next pointer field of a BD in external memory). This makes it possible to prove properties of DMA transfers (e.g., memory isolation) without considering interleavings with CPU transitions which can potentially corrupt pending BDs. This abstract model alters the previous transition system by composing the abstract DMAC and memory in such a way that the abstract DMAC can “magically” extend the abstract queue of pending BDs with new BDs bds when the CPU writes memory m_1 at locations with addresses as and bytes bs resulting in memory m_2 (writing registers is similar but with the updated internal state considered instead of updated memory):

$$\frac{as \subseteq \mathcal{M} \quad m_2 = m_1[as \mapsto bs] \quad bds' = \Pi.cf(m_2, s) \quad \exists bds. bds' = c.f \# bds}{m_1[(s, b, c) \xrightarrow{wt(as, bs)} m_2[(s, b, c[f \mapsto bds'])]} [ma]$$

The internal operations of M_1 also differ. For $[f_3]$, the BD bd returned by $\Delta.f$ is ignored and instead the first BD of $c.f$ is moved to $c.u$ or $c.p$, depending on whether the BD shall be updated or not. The reason why main memory is still accessed to fetch BDs (even though they are not used) is to keep the transition systems synchronized: Internal states are updated identically in both M_1 and M_2 . In addition, the checks for updates/write backs and DMA writes in M_2 are also in M_1 .

B. Refinement Relations, Invariants, and Proof Obligations

We use $(m, d_{i+1}) \simeq_{i+1} (m, d_i)$ for the refinement relation between M_{i+1} and M_i . These relations require the common

state fields to be equal: $d_{i+1} = d_i$. Additionally $(m, d_2) \simeq_2 (m, d_1)$ requires that the abstract and concrete pending BDs are equal: $d_1.ch.f = \Pi.cf(m, d_2.s)$.

The refinement proofs depend on invariants that restrict the state of the lower layer. The invariant for M_2 requires that no DMA write request targets pending BDs $\mathcal{I}_2(m, s, b, c) :=$

$$\mathbf{w}_t^{op}[as, bs] \in b \wedge bd \in \Pi.cf(m, s) \implies as \not\subseteq bd.ra$$

This invariant simply propagates the checks of the internal abstract DMAC operations (e.g., $[w]$ of M_2).

In order to establish the bisimulation for the model of Section III, we also need an invariant that enforces the same constraints that are checked by the abstract models. The invariant \mathcal{I}_3 requires that every pending or fetched BD in the pipeline do not have update/write back addresses nor DMA writes to pending BDs (this includes that pending BDs do not overlap; in the definition of \mathcal{I}_3 , c denotes the concatenation of $c.u$, $c.p$ and $c.w$): $\mathcal{I}_3(m, s, b, c) :=$

$$\bigcup_{bd \in c \cup \Pi.cf(m, s)} (bd.wa \cup bd.dwa) \not\subseteq \bigcup_{bd \in \Pi.cf(m, s)} bd.ra$$

The last invariant restricts M_1 to force the DMAC to access only readable and writable memory (in the definition of \mathcal{I}_1 , c denotes the concatenation of $c.f$, $c.u$, $c.p$ and $c.w$): $\mathcal{I}_1(m, s, b, c) :=$

$$\bigcup_{\mathbf{r}_t^{op}[as] \in b} as \cup \bigcup_{bd \in c} bd.ra \cup \bigcup_{bd \in c.op, op \neq w} bd.dra \subseteq \mathcal{R} \wedge$$

$$\bigcup_{\mathbf{w}_t^{op}[as, bs] \in b} as \cup \bigcup_{bd \in c} bd.wa \cup \bigcup_{bd \in c.op, op \neq w} bd.dwa \subseteq \mathcal{W}$$

The instantiation of a given DMAC must satisfy some proof obligations, which mainly state that the behavioral and projection functions are consistent:

- 1) A fetched BD (by $[f_3]$) is the first pending BD: If $\mathbf{r}_t^f[as] = \Pi.fas(s_1)$, and $(s_2, (bd, op)) = \Delta.f(s_1, \mathbf{p}_t^f[m[as]])$, then there exist BDs bds such that $\Pi.cf(m, s_1) = bd :: bds$. Also, after fetching a BD, the projection function must reflect the removal of the BD from the pending queue: $\Pi.cf(m, s_2) = bds$.
- 2) The queue of pending BDs depends only on the locations of the BDs and the internal state: If $\forall a \in \bigcup_{bd \in \Pi.cf(m, s)} bd.ra. m_2[a] = m_1[a]$, then $\Pi.cf(m_1, s) = \Pi.cf(m_2, s)$.
- 3) The function associated with DMA transfers does not affect the queue of pending BDs: $(s_2, rs, cf) = \Delta.p(s_1, bd, ps)$ implies $\Pi.cf(m, s_2) = \Pi.cf(m, s_1)$

The proof obligations of the driver are that it only appends BDs and preserves the invariants \mathcal{I}_1 and \mathcal{I}_3 . This proof obligation is only relevant for non-internal CPU transitions, since the invariants do not depend on the CPU. For memory writes (other cases are similar) this means that if $\bigwedge_{i \in \{1, 3\}} \mathcal{I}_i(m, d)$, $cpu \xrightarrow{wt(as, bs)} cpu'$, and $as \subseteq \mathcal{M}$ then:

- 1) $\exists bds. \Pi.cf(m[as \mapsto bs], d.s) = \Pi.cf(m, d.s) \# bds$.

- 2) $(m|d) \xrightarrow{wt(as, bs)}_1 (m'|d')$ implies $\mathcal{I}_i(m', d')$, where \rightarrow_1 denotes the transition relation of M_1 .

That is, writes (updates, write backs and DMA writes) of appended BDs do not point to pending BDs or non-writable memory, appended BDs do not overlap, and reads (both fetches and DMA) of appended BDs do not point to non-readable memory. Notice that invariant preservation can be done by checking the state of the the more abstract DMAC model M_1 , disregarding the lower layers.

C. Refinement and Memory Isolation

Refinement is phrased as a bisimulation and assumes the invariant. For $i \in \{2, 3\}$ (\rightarrow_i denotes the transition relation of M_i):

Lemma 1. *If $\mathcal{I}_{i+1}(m, d)$, $(m, d) \simeq_{i+1} (m, e)$, and $(c, m, d) \xrightarrow{l}_i (c', m', d')$ then exists e' such that $(c, m, e) \xrightarrow{l}_i (c', m', e')$ and $(m', d') \simeq_{i+1} (m', e')$, and vice versa with transitions of \rightarrow_i .*

Proof. Consider $i = 1$. For the fetch rules the main difference between M_1 and M_2 is that M_1 fetches abstract BDs and M_2 fetches concrete BDs. \simeq_2 guarantees that these queues are equal. M_1 moves the first BD of $d_1.c.f$ to the tail of the update or process queue ($d_1.c.op$, $op \in \{u, p\}$). DMAC proof obligation 1) ensures that M_2 performs a corresponding operation by moving the first concrete BD of $\Pi.cf(m, d_2.s)$.

For updating, processing and writing back BDs, the abstract pending BDs of M_1 cannot change by definition. To show that the concrete pending BDs of M_2 are also unchanged we use the the update/write back checks in M_2 and DMAC proof obligation 2). Moreover, \mathcal{I}_2 and DMAC proof obligation 2) imply that memory writes do not change concrete pending BDs in M_2 , preserving equality between concrete and abstract BDs queues.

Finally, for CPU transitions, there are two cases depending on whether the pending BDs are modified. If not, then memory and register accesses have identical effects in M_1 and M_2 . Otherwise, Driver proof obligation 1) ensures that M_2 only appends BDs. This allows M_1 to produce the corresponding abstract queue of pending BDs by extending the existing one via the rule $[ma]$ (and similarly for register writes).

For $i = 3$, \mathcal{I}_3 is transferred by \simeq_3 to M_2 , implying that all checks in M_2 pass (e.g. $[w]$). Thus, M_2 and M_3 , perform identical operations. CPU and DMAC memory transitions are identical in M_2 and M_3 . \square

We then prove that invariants are preserved and transferred by the refinements:

Lemma 2. *If $\mathcal{I}_i(m, d)$ and $(c, m, d) \xrightarrow{l}_i (c', m', d')$ then $\mathcal{I}_i(m', d')$. Also if $j < i$ and $(m, d_j) \simeq_j (m, d_{j+1})$ then $\mathcal{I}_i(m, d_{j+1}) \Leftrightarrow \mathcal{I}_i(m, d_i)$.*

Finally we show that DMAC transitions modify and depends on only the right regions of memory (where $f|_A$ is the projection of a function over domain A and \bar{A} is set complement):

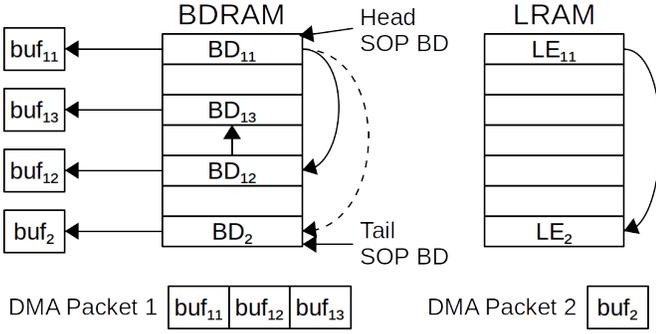


Fig. 4. Organization of BD queues of the USB DMAC.

Theorem 1. *If $\bigwedge_i \mathcal{I}_i(m, d)$ and $(c, m, d) \rightarrow_3 (c, m', d')$ then $m|_{\mathcal{W}} = m'|_{\mathcal{W}}$, and if $m|_{\mathcal{R}} = m_1|_{\mathcal{R}}$ then $(c_1, m_1, d) \rightarrow_3 (c_1, m'_1, d')$ and $m'|_{\mathcal{W}} = m'_1|_{\mathcal{W}}$*

The theorem follows from Lemmas 1, 2 and by establishing a further bisimulation with an even more abstract layer that is isolated by construction. This model have additional checks compared to M_1 that prevent adding memory requests to the message box that point outside \mathcal{R} and \mathcal{W} .

V. USB DMAC

We instantiate our framework with the DMAC of the USB controller of the AM3358 SoC by Texas Instruments [32], the SoC on the development board BeagleBone Black [7]. As Fig. 4 illustrates, the DMAC organizes BD queues by means of two memory regions, one storing BDs (BDRAM) and one storing linking information (LRAM), the base addresses of which are configurable. Both regions are organized as arrays with the same number of entries. To transmit a DMA packet, potentially scattered in memory in multiple buffers (e.g., DMA packet 1 is the concatenation of buf_{11} , buf_{12} and buf_{13}), the driver initializes in BDRAM one BD for each buffer (BD_{11} , BD_{12} and BD_{13}), linking them via the next descriptor pointer in the order the data buffers shall be transmitted to the USB device. The first BD of a packet is called Start Of Packet (SOP). The LRAM is used to link packets: if BD_{11} is a SOP then $\text{LRAM}[i]$ links the SOP BD of the next DMA packet (BD_{11} is linked to BD_2 via LRAM entry LE_{11} , in effect linking DMA packets 1 and 2). Both the driver and the DMAC read and write BDRAM, but only the DMAC uses LRAM.

To enqueue a DMA packet the driver writes the address of its SOP BD (e.g., BD_2 to enqueue packet 2) to the enqueue register Q . This write causes the DMAC to append the BDs of the new DMA packet to the pending queue: The LRAM entry of the previous tail SOP BD (e.g., LE_{11}) is updated with a link to the appended SOP BD. Once a BD has been fetched, it is processed, without being updated, and finally written back. A write back moves the head SOP BD of the transferred DMA packet from the pending queue to the tail of the completion queue (which is another queue whose links are also stored in LRAM). The completion queue is traversed by the driver to recycle BDs. The driver does this by reading the C register,

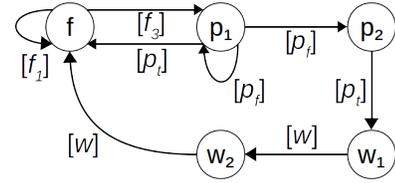


Fig. 5. State diagram of the USB DMAC instantiation. The transition labels denote the rules that cause the corresponding transition.

making the DMAC return the address of the first SOP BD in the completion queue, and read LRAM to find the next completed SOP BD which now becomes the first SOP BD in the completion queue.

We focus on the instantiation of the transmission channel, since reception is similar. The internal state is a record $s = (r, hp, tp, hc, tc, t)$ containing the registers r (except Q and C which are not physical registers), the head and tail pointers of the pending and completion queues hp, tp, hc, tc , and the state t of the automaton in Fig. 5 that keeps track of the state of the operation of the current DMA packet in transfer.

In state f , the rules $[f_1]$ and $[f_3]$ fetch the next BD and move it to the process queue $c.p$ (BDs are fetched atomically, making $[f_2]$ unnecessary; thus, $\Delta.f$ always returns a BD).

In state p_1 , $[p_f]$ repeatedly obtains memory read requests and handles replies until all data in the buffer has been read. If the BD in $c.p$ is not the last BD of the DMA packet (e.g. BD_{12}) then $[p_t]$ sets the next state to f to operate on the next BD of the DMA packet (e.g. BD_{13}). Otherwise, after processing the last byte of the buffer, a further application of $[p_f]$ is used to produce a DMA read request needed to read the LRAM entry of the SOP BD (LE_{11}) of the DMA packet in transfer. This data is needed later to update the linking ram in the write back stage and must be read by $[p_t]$, since $[w]$ cannot read memory. The state is set to p_2 , in which $[p_t]$ processes the reply containing the LRAM entry and sets the state to w_1 .

Write backs are performed in two steps. First, in state w_1 , $[w]$ updates the head pointer hp of the pending queue to the address of the next SOP BD (BD_2), which has been previously retrieved in p_2 . Second, in state w_2 , the tail pointer tc of the completion queue is set to the address of the completed SOP BD (BD_{11}); the LRAM entry of the previous tail SOP BD of the completion queue is now linked to the new tail (completed) SOP BD (e.g. BD_{11}); the next state is f to fetch the next SOP BD (BD_2); and all BDs accumulated in $c.w$ are released, meaning that the driver can reuse them.

Register accesses are performed by directly reading and writing $s.r$, except Q and C . When Q is written, tp is updated to the written address of the appended SOP BD. When C is read, the value of hc is returned with hc set to the address of the next SOP BD in the completion queue. These register accesses cause additional DMA management accesses to LRAM in order to reflect the queue updates (e.g., linking LE_{11} to LE_2 when BD_2 is written to Q).

The following is a description of $\Pi.cf$, and why $\Pi.cf$, $\Delta.f$ and $\Pi.fas$ satisfy DMAC proof obligation 1). $\Pi.cf(m, s)$ finds

Aspect	NIC DMAC w/o fw	USB DMAC w/ fw
LoC model	1500	2000
LoC verification	55000	2000
Modeling time	3 person-months	2 person-months
Verification time	9 person-months	1/2 person-month

TABLE III

EFFORT OF VERIFYING MEMORY ISOLATION OF A NIC [33] AND A USB DMAC WITH AND WITHOUT THE FRAMEWORK. THE HOL4 EXPERIENCE BEFORE THE NIC DMAC WORK WAS ABOUT FOUR MAN-MONTHS, AND ABOUT 30 MAN-MONTHS BEFORE THE USB DMAC WORK.

the un fetched BDs in four steps. (1) It retrieves the address of the *current SOP BD* of the current DMA packet in transfer. (2) The address of the next BD to fetch is obtained from *hp*. (3) If *hp* is zero, then the entire pending queue has been visited and the function returns the accumulated BDs so far. Otherwise, it collects the un fetched BDs of the current DMA packet, starting from the next BD to fetch and traversing the next descriptor pointer fields. (4) The next BD to fetch is the SOP BD of the next DMA packet, identified by reading the LRAM entry of the last visited SOP BD. The procedure continues with step 3. The first BD that is fetched by $\Delta.f$ is at the address given by $\Pi.fas$ obtained from *hp*, which is the address obtained by $\Pi.cf$ in step 2. Hence, the fetched BD is the first pending BD.

VI. APPLICATION AND EVALUATION

The framework consists of about 28000 lines of HOL4 code, including models and proofs. It was first described in pseudocode based on reviews of more than 20 DMACs, and then refined into HOL4 code [42]. The high-level design, definition, and proof took in total 18 person-months.

The instantiation of the USB DMAC consists of about 2000 lines for the model, and about 2000 lines for the proofs of the proof obligations. The model is based on the informal specification [32], which, as is common with informal specifications, contains undefined terms whose meaning must be derived from the (lacking) context, dispersed information, and typos. Similar to the framework, we started with high-level pseudocode that was gradually refined to remove ambiguities and to make it fit the framework, requiring seven person-weeks. Verifying the proof obligations took about two additional person-weeks. In previous work [33], we have modeled and verified memory isolation of a NIC DMAC without the support of the framework, taking about three months of modeling and nine months in proving that the invariant is preserved. Due to significantly less time in using the framework, we believe that the framework provides significant assistance in verifying memory isolation of DMACs, with the main benefit being the proof of that the invariant is preserved. Table III makes a comparison between the efforts invested into verifying the NIC and USB DMACs with and without the framework.

The benefit of our approach is that we can establish soundness of the verification conditions independently of the driver. Then one can independently analyze the driver. For instance, the Linux driver of the USB DMAC uses only a limited set of the features of the device: It allocates one single BD per channel, meaning that the DMA packets consist of only

one buffer, and it enqueues a new packet only after that the previous one has been completed. The driver allocates two memory regions for BDRAM and LRAM. These memory regions do not overlap, neither do the BDs, with each BD of each channel being allocated a fixed location. The Linux virtual memory manager allocates the BDRAM and LRAM regions, and likewise the DMA buffers for data transfers. Assuming that these memory regions are disjoint and located in “readable” and “writable” memory, this driver satisfies the two driver proof obligations as follows. First, the driver pops BDs from the completion queues by reading the C register, before reinitializing them and appending them by writing the Q register, thus only modifying the pending BD queues by appending BDs. LRAM is not accessed by the driver. Moreover, by assumption, BDs and DMA buffers are in readable and writable memory. The driver organizes the BDs in disjoint array slots in BDRAM, meaning that BDs do not overlap, and thus write back addresses do not coincide with read addresses of other BDs.

VII. RELATED WORK

Verification of Device Drivers without DMA Model checkers and interactive theorem provers have been used to verify various properties of drivers controlling devices without a DMAC: Reading from flash memory gives previously written data [34]; correct copying of data from memory to an ATAPI disk [35]; termination of a UART driver transferring data from memory to the external environment [36]; safety and liveness properties of a UART driver [37]; absence of data races and illegal memory accesses by a keyboard driver [38]; and equivalence between abstract and concrete models of an SPI driver and the SPI controller [39].

These devices do not have a DMAC, meaning that their memory isolation depends only on the memory accesses performed by the driver. For devices without a DMAC, methods have been investigated for synthesizing and (semi-) automatically generating device drivers that satisfy the interfaces of the OS and the I/O device [50], [51].

Hardware Verification Our work assumes that the hardware implementation of the device satisfies its hardware-software interface. Hardware verification is indeed an orthogonal problem to the driver verification problem.

A DMAC is reminiscent of a CPU in the sense that BDs corresponds to instructions, BD operations correspond to an instruction pipeline, and concurrent DMA channels correspond to multiple instruction streams (threads) with BDs from different channels. These aspects have been investigated by the CPU formal verification community [52]–[54].

Specifically for DMAC implementations, Clarke et al. [40] have used model checking to verify that DMAC transfers are eventually completed, that the DMAC is eventually ready for new transfers, and that memory operations terminate. The analyzed DMAC is relatively simple: The DMAC maintains no queues nor multiple channels; its configuration depends only on the DMAC registers; and the next transfer can be programmed only after the previous transfer is finished. The

same DMAC design was later used to verify relationships between signals, including clock cycle delays [41].

Verification of DMAC Drivers Monniaux [43] has verified a USB driver that controls a DMAC, using a static C code analyzer designed to detect memory access and arithmetic errors. The driver and the device are modeled in C, with interleaved execution. The C analyzer can automatically verify that the driver and the controller access only allowed memory.

Even if an existing C analyzer largely automates verification, the framework addresses some of the limitations of this work. First, to automate the analysis, the C model coarsely overapproximates all possible device actions. In order to check soundness of this overapproximation, one should refine the model and prove some sort of refinement (see Subsection IV-C), which can be difficult in C and is not supported by the tool. Second, the use of a general C verification tool requires the model to be defined in terms of C semantics. For example, the tool is designed for 32-bit atomic variable accesses, but some devices may use single byte granularity. Third, it is not clear if the tool can analyze models of DMACs that have complex BD queues. In fact, the analyzed model has a relatively simple structure, where BD queues consist of three static arrays. Finally, the overapproximation used to automate the analysis may prevent it from being used to verify functional properties (e.g., a buffer is actually copied from source to destination), which the tool has no support for.

Donaldson et al. [46] have used model checking to verify absence of data races to DMA buffers between the PPE (a general CPU) and SPEs (HW accelerators) of the IBM Cell BE processor, which have embedded DMACs in the SPEs to transfer data between main memory and their local memory. In their analysis, BD queues are not considered, only single atomic DMA commands. Hence, this work is limited to this specific hardware and does not consider memory isolation.

Schwarz et al. [47] have used Coq to model a DMAC and a hypervisor, which virtualizes the DMAC among two guests, and verified that the DMAC virtualization keeps the guest isolated. Also this work concerns a specific and simple DMAC, not dealing with complex organizations of BD queues.

In previous work [33] we modeled a DMAC of an Ethernet NIC in HOL4 and verified sufficient conditions for isolating packets in transfer. The BD queues are organized as linked lists stored in internal DMAC memory. The formalization and verification took about one person-year, the majority of which can be saved with the DMAC framework.

Techniques for Isolating DMACs The ability of isolating DMA accesses is fundamental for guaranteeing security of entire systems. For instance, the security of several verified systems [23]–[30], [48], [49] requires restricted DMA.

Hardware assisted DMAC isolation uses stand-alone IOMMUs [15] or IOMMU embedded in the DMAC [8] to prevent the DMAC from accessing critical memory due to untrusted configurations. In absence of dedicated hardware mechanisms, the common approach to enforce memory isolation is via a monitor in the OS [44], [55] or the hypervisor [45], that intercepts driver reconfigurations of the DMAC. Other meth-

ods analyze an aspect of the system in runtime and react to violations: Execution of device firmware follows a pre-determined pattern [14] (e.g. the stack pointer and program counters are in valid memory regions), memory bus activity follows a pre-determined pattern [13], execution traces recorded by hardware or binary instrumentation [12], and integrity of firmware and I/O configuration (the checks of which are triggered by interrupts and thresholds of hardware performance counters) [11].

Grisafi et al. [56] presents a mechanism to isolate memory for low-end embedded systems with DMACs. This is achieved by means of a hypervisor, and a compiler that inserts hypervisor calls in applications accessing DMAC registers. The software design has been verified, however the security of the system depends on the fact that the security policies enforced by the hypervisor prevent the DMAC to access critical region of memory. While this is simple to check for simple DMACs with single BDs and that are configured only via memory mapped registers, guaranteeing this property for complex DMACs requires to analyze the device model. Our work is complementary to the software verification, since it supports the identification of the verification of the security policies for teh devices.

VIII. CONCLUSION

We have implemented a framework in the interactive theorem prover HOL4 for modeling DMACs, and by means of refinement formally verified DMAC memory isolation. Comparing the efforts of the USB DMAC instantiation with previous verification of memory isolation of a NIC DMAC [33], strongly suggests that the framework can significantly reduce the cost of verification of isolation (i.e., proving that the invariant is preserved).

Our verification can be extended in two directions. Towards software, the proof obligations can be used to check that device drivers securely configure DMACs or to synthesize security monitors, and the abstract model can be used to check functional correctness (e.g., transmission of network packets). Towards hardware, the model M_3 can be used to either show that a formal hardware design respect the specification, or for model driven testing of closed source hardware.

We plan to implement and model a monitor that runs underneath the Linux USB DMAC driver for the USB DMAC on BeagleBone Black [7], [32], checking that the driver reconfigurations are secure; and then verify that the monitor satisfies the proof obligations. This fulfills two goals: The monitor preserves security even if the driver is buggy, and the monitor itself can be used to detect if the Linux driver has memory isolation bugs.

REFERENCES

- [1] Altera Corporation, “Increase System Performance & Efficiency Using Distributed Direct Memory,” 2004, p. 10. Accessed: April 14, 2022 [Online]. Available: http://xilinx.info/_exhibit/2004/altera/5_SOPC_04_DMA_RF_2_50min.pdf

- [2] J. Mangino, "Using DMA with High Performance Peripherals to Maximize System Performance," Texas Instruments Corporation, 2007, p. 13. Accessed: April 14, 2022. [Online]. Available: <https://www.ti.com/lit/wp/spna105/spna105.pdf>
- [3] F. Khunjush and N. J. Dimopoulos, "Extended Characterization of DMA Transfers on the Cell BE Processor," IEEE International Symposium on Parallel and Distributed Processing, 2008, Table 5.
- [4] K. Saether, "Using Event Systems and DMA to Cut Power Consumption," techbriefs.com, Table 2. Accessed: April 14, 2022. [Online]. Available: <https://www.techbriefs.com/component/content/article/tb/supplements/et/features/articles/6272>
- [5] T. Enami, K. Kawakami, and H. Yamazaki, "DMA-driven control method for low power sensor node," IEEE Topical Conference on Wireless Sensors and Sensor Networks, 2015.
- [6] W.-P. de Roever, et al., "Concurrency Verification: Introduction to Compositional and Non-Compositional Methods," USA: Cambridge University Press, 2012.
- [7] BeagleBone Black System Reference Manual. Accessed: April 26, 2022. [Online]. Available: <https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual#beaglebone-black-high-level-specification>
- [8] Z. D. Dittia, G. M. Parulkar, and J. R. Cox Jr, "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," Proceedings of the sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies, April 1997.
- [9] J. Danisevskis, M. Piekarska, and J.-P. Seifert, "Dark Side of the Shader: Mobile GPU-Aided Malware Delivery," Information Security and Cryptology, pp. 483-495, 2013.
- [10] A. Markuze, A. Morrison, and D. Tsafir, "True IOMMU Protection from DMA Attacks: When Copy Is Faster Than Zero Copy," Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 249-262, March 2016.
- [11] F. Zhang, H. Wang, K. Leach, and A. Stavrou, "A Framework to Secure Peripherals at Runtime," Computer Security - ESORICS, pp. 219-238, 2014.
- [12] O. Ruwase, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Guardrail: a high fidelity approach to protecting hardware devices from buggy drivers," ACM SIGARCH Computer Architecture News, vol. 42, no. 1, pp. 655-670, March 2014.
- [13] P. Stewin, "A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platform's Main Memory," Research in Attacks, Intrusions, and Defenses, pp. 1-20, 2013.
- [14] L. Duflot, Y.-A. Perez, and B. Morin, "What If You Can't Trust Your Network Card?," Recent Advances in Intrusion Detection, pp. 378-397, 2011.
- [15] AMD Corporation, "AMD I/O Virtualization Technology (IOMMU) Specification," 2021. Accessed: April 14, 2022 [Online]. Available: https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf
- [16] ARM Limited, "ARM System Memory Management Unit Architecture Specification SMMU architecture version 2.0," 2016. Accessed: April 14, 2022 [Online]. Available: <https://documentation-service.arm.com/static/5f900d34f86e16515cdc08fb>
- [17] J. Yao, V. J. Zimmer, and S. Zeng, "A Tour Beyond BIOS: Using IOMMU for DMA Protection in UEFI Firmware," Intel Corporation, 2017. Accessed: April 14, 2022 [Online]. Available: <https://www.intel.com/content/dam/develop/external/us/en/documents/intel-whitepaper-using-iommu-for-dma-protection-in-uefi-820238.pdf>
- [18] M. Ben-Yehuda et al., "The Price of Safety: Evaluating IOMMU Performance," The 2007 Ottawa Linux Symposium, 2007, p. 9, p. 13.
- [19] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "IOMMU: Strategies for Mitigating the IOTLB Bottleneck," Proceedings of the 2010 international conference on Computer Architecture, pp. 256-274, June 2010, Figure 2.
- [20] L. Lei, K. Cong, Z. Yang, and F. Xie, "Validating Direct Memory Access Interfaces with Conformance Checking," Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design, pp. 9-16, Nov. 2014.
- [21] A. A. Vasilyev, "Static verification for memory safety of Linux kernel drivers," Proceedings of ISP RAS, vol. 30, no. 6, pp. 143-160, 2018.
- [22] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, "Static Detection of Unsafe DMA Accesses in Device Drivers," 30th USENIX Security Symposium (USENIX Security 21), 2021.
- [23] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Formal Verification of a Microkernel Used in Dependable Software Systems," Computer Safety, Reliability, and Security, pp. 187-200, 2009.
- [24] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal verification of information flow security for a simple arm-based separation kernel," Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 223-234, Nov. 2013.
- [25] A. Vasudevan et al., "Design, Implementation and Verification of an eXTensible and Modular Hypervisor Framework," IEEE Symposium on Security and Privacy, May 2013.
- [26] G. Klein et al., "Comprehensive formal verification of an OS microkernel," ACM Transactions on Computer Systems, vol. 32, no. 1, pp. 1-70, Feb. 2014.
- [27] R. Gu et al., "Deep specifications and certified abstraction layers," Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 595-608, Jan. 2015.
- [28] C. Baumann, M. Näslund, C. Gehrman, O. Schwarz, and H. Thorsen, "A high assurance virtualization platform for ARMv8," European Conference on Networks and Communications, 2016.
- [29] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A Secure and Formally Verified Linux KVM Hypervisor," IEEE Symposium on Security and Privacy (SP), May 2021.
- [30] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor," Proceedings of the 30th USENIX Security Symposium, 2021.
- [31] "HOL Interactive Theorem Prover," <https://hol-theorem-prover.org> (accessed April 15, 2022).
- [32] Texas Instruments, "AM335x and AMIC110 Sitara Processors Technical Reference Manual," Rev. P. Accessed: April 15, 2022 [Online]. Available: <https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf>.
- [33] J. Haglund and R. Guanciale, "Trustworthy Isolation of DMA Enabled Devices," Proceedings of 15th International Conference on Information Systems Security, Hyderabad, India, pp. 35-55, Dec. 2019.
- [34] M. Kim, Y. Choi, Y. Kim, and H. Kim, "Pre-testing Flash Device Driver through Model Checking Techniques," 1st International Conference on Software Testing, Verification, and Validation, 2008.
- [35] E. Alkassar and M. Hillebrand, "Formal Functional Verification of Device Drivers," Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments, pp. 225 - 239, Oct. 2008.
- [36] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev, "Formal Device and Programming Model for a Serial Interface," Proceedings of the 4th International Verification Workshop, pp. 4-20, Bremen, Germany, 2007.
- [37] J. Duan, "Formal Verification of Device Drivers in Embedded Systems," Ph.D. dissertation, Dept. Computing, Univ. Utah, UT, USA, 2013.
- [38] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens, "Sound Formal Verification of Linux's USB BP Keyboard Driver," NASA Formal Methods, 2012.
- [39] N. Dong, R. Guanciale, and M. Dam, "Refinement-Based Verification of Device-to-Device Information Flow," Formal Methods in Computer Aided Design, 2021.
- [40] E.M. Clarke, S. Bose, M.C. Browne, and O. Grumberg, "The Design and Verification of Finite State Hardware Controllers," Technical Report CMU - CS - 87-145 , Carnegie Mellon Univ., July 1987.
- [41] H. Hiraishi, K. Hamaguchi, H. Fujii, and S. Yajima, "Regular Temporal Logic Expressively Equivalent to Finite Automata and Its application to Logic Design Verification," Journal of Information Processing, vol. 15, no. 1, pp. 130-138, 1992.
- [42] <https://github.com/kth-step/dma-controller-verification.git>
- [43] D. Monniaux, "Verification of Device Drivers and Intelligent Controllers: a Case Study," Proceedings of the 7th ACM & IEEE international conference on Embedded software, pp. 30-36, Sept. 2007.
- [44] A. Mera, Y. H. Chen, R. Sun, E. Kirda, and L. Lu "D-Box: DMA-enabled Compartmentalization for Embedded Applications," Network and Distributed Systems Security Symposium, San Diego, CA, USA, April 2022.
- [45] J. Haglund and R. Guanciale, "Trustworthy isolation of DMA devices," Journal of Banking and Financial Technology, pp. 75-94, 2020.
- [46] A. F. Donaldson, D. Kroening, and P. Rümmer, "Automatic analysis of DMA races using model checking and k-induction," Formal methods in system design, vol. 39, no. 1, pp. 83-113, 2011.
- [47] O. Schwarz and C. Gehrman, "Securing DMA through virtualization," Proceedings of Complexity in Engineering, 2012.

- [48] O. Schwarz and M. Dam, "Formal Verification of Secure User Mode Device Execution with DMA," *Hardware and Software: Verification and Testing*, pp. 236-251, Haifa, Isreal, 2014.
- [49] M. Yu, V. Gligor, and L. Jia, "An I/O Separation Model for Formal Verification of Kernel Implementations," *IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2021.
- [50] L. Ryzhyk, P. Chubb P, I. Kuz, E. Le Sueur, and G. Heiser, "Automatic device driver synthesis with termite," *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 73-86, 2009.
- [51] L. Ryzhyk et al., "User-guided device driver synthesis," *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pp. 661-676, 2014.
- [52] J. Sawada and W. A. Hunt Jr., "Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability," *Formal Methods in System Design* vol. 20, pp. 187-222, 2002.
- [53] M. N. Velev and P. Gao, "Automatic formal verification of multithreaded pipelined microprocessors," *IEEE/ACM International Conference on Computer-Aided Design*, 2011.
- [54] P.-M. Seidel, "Formal Verification of an Iterative Low-Power x86 Floating-Point Multiplier with Redundant Feedback", *10th International Workshop on the ACL2 Theorem Prover and its Applications*, pp. 70-83, 2011.
- [55] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneide, "Device Driver Safety Through a Reference Validation Mechanism," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, Dec. pp. 241-254, 2008.
- [56] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, "PISTIS: Trusted Computing Architecture for Low-end Embedded Systems," *31st USENIX Security Symposium*, 2022.

Foundations and Tools in HOL4 for Analysis of Microarchitectural Out-of-Order Execution

Karl Palmiskog , Xiaomo Yao , Ning Dong , Roberto Guanciale , and Mads Dam 

KTH Royal Institute of Technology, Stockholm, Sweden

Email: {palmiskog, xiaomoy, dongn, robertog, mfd}@kth.se

Abstract—Program analyses based on Instruction Set Architecture (ISA) abstractions can be circumvented using microarchitectural vulnerabilities, permitting unwanted program information flows even when proven ISA-level properties ostensibly rule them out. However, the low abstraction levels found below ISAs, e.g., in microarchitectures defined in hardware description languages, may obscure information flow and hinder analysis tool development. We present a machine-checked formalization in the HOL4 theorem prover of a language, MIL, that abstractly describes microarchitectural in-order and out-of-order program execution and enables reasoning about low-level program information flows. In particular, MIL programs can exhibit information flow side channels when executed out-of-order, as compared to a reference in-order execution. We prove memory consistency between MIL’s out-of-order and in-order dynamic semantics in HOL4, and define a notion of conditional noninterference for MIL programs which rules out trace-driven cache side channels. We then demonstrate how to establish conditional noninterference for programs via a novel semi-automated bisimulation based verification strategy inside HOL4 that we apply to several examples. Based on our results, we believe MIL is suitable as a translation target for ISA code to enable information flow analyses.

Index Terms—information flow, interactive theorem proving, HOL4, microarchitectures, out-of-order execution

I. INTRODUCTION

Vulnerabilities such as Spectre, Meltdown, and Fore-shadow [1]–[3] demonstrate that program analyses based on Instruction Set Architecture (ISA) abstractions cannot guarantee important program properties such as freedom from unwanted information flows. Consequently, microarchitectures (residing below the ISA level) are important to understand and take into account by developers of compilers and program analysis tools. However, the low abstraction level of most hardware description languages (HDLs) obscures important microarchitectural features such as out-of-order execution of program instructions. In particular, HDLs complicate reasoning about *low-level program information flows*.

To address this problem, Guanciale et al. [4] proposed the Machine Independent Language (MIL), which abstractly describes microarchitectures and permits analysis of information flows between microinstructions. In this paper, we present a deep embedding of MIL and an encoding of its out-of-order (OoO) and in-order (IO) dynamic semantics in the HOL4 theorem prover [5]. Using our embedding, we formalize two key aspects of the metatheory of MIL. Firstly, we provide,

This work has been partially supported by the KTH CERES Center and the Trustfull project funded by the Swedish Foundation for Strategic Research.

to our knowledge, the first general machine-checked proof of *memory consistency* between in-order and out-of-order execution of microinstructions. Secondly, we define a notion of *conditional noninterference* (CNI) capturing trace-driven cache based information flow [6]. To achieve this, we clarify the assumptions under which MIL programs (1) do not *go wrong* during runtime, and (2) *progress* as expected, which was previously left implicit.

We show that out-of-order execution can introduce information side channels, by exhibiting a violation of conditional noninterference. We then devise a semi-automated bisimulation based strategy to verify conditional noninterference, which we apply to several example MIL programs. To improve automation of conditional noninterference proofs, we developed functions and results for verified execution of MIL instructions inside HOL4 [7]. We also refined our functions to CakeML code [8], which, when compiled to native code, can execute instructions orders of magnitude faster than HOL4 and demonstrate side channels for concrete MIL programs.

In order to make our theory and tools applicable to a range of real-world ISAs such as ARMv8-A and RISC-V, we developed a translator from BIR, an architecture independent binary code representation from the HolBA binary analysis framework [9] that has proof-producing lifters. To validate the MIL formalization, we analyzed both hand-crafted programs and programs translated from BIR. Based on our results, we believe MIL is ready to be used as a form of abstract microcode language, e.g., as a target language for ISA instructions to enable low-level information flow analysis. From the hardware perspective, our memory consistency proof for MIL can be reused across different formalized microarchitectures.

In summary, we make the following contributions:

- **Foundations:** We define MIL and its dynamic OoO and IO semantics in HOL4, including notions of well-formedness and resource initialization for runtime states.
- **Metatheory:** We develop formal metatheory of MIL in HOL4, including a proof of *memory consistency* and a notion of *conditional noninterference* for the semantics.
- **Tools:** We verify functions for executing MIL programs and then refine them to CakeML, yielding trustworthy MIL analysis tools both inside and outside HOL4.
- **Applications:** We devise a semi-automated reasoning strategy for conditional noninterference, which we apply to verify confidentiality of several MIL programs.

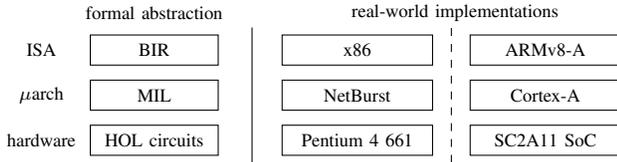


Fig. 1. Comparison of abstraction levels of BIR and MIL.

As supplementary material for the paper [10], we provide the HOL4 definitions and proofs, Standard ML code, CakeML programs, and a technical report that renders key MIL definitions and results into readable mathematical vernacular.

II. BACKGROUND

A. Instruction pipelining and OoO execution

Pipelined processors divide instruction execution into *stages*, such as fetching and decoding, which are carried out by distinct processing units working independently. However, a programmer or compiler developer typically assumes instructions are processed and completed in sequential order, which may cause pipelines to stall while a processing unit is waiting for instructions to complete the previous stage. By extracting *data and address dependencies* between instructions, microarchitectures can reorder instructions, leading to better pipeline utilization and performance [11], [12].

Instruction reordering, and thus OoO execution, is a fundamental microarchitectural mechanism that can be leveraged in isolation to increase performance of pipelined processors [13]. It is also a prerequisite of *speculative* execution, where instructions are fed into a pipeline even when they are not known to be necessary to execute. Our formalization of the foundations of OoO execution is therefore an important building block towards machine-checked analysis of speculation in MIL using the speculative MIL semantics by Guanciale et al. [4].

B. HolBA and BIR

HolBA is a binary analysis platform based on HOL4 with support for ISAs such as ARMv8-A and RISC-V [9]. HolBA provides proof-producing transformations of binaries to an intermediate HOL4 representation, called BIR. That is, HolBA generates a HOL4 theorem that the BIR representation of an input binary preserves its behavior, as given by a formalization of the corresponding ISA [14]. BIR is also the target language of Scam-V, a toolchain which finds discrepancies between abstract information side-channel models and real microarchitectures [15]. Figure 1 illustrates the intended abstraction levels of BIR and MIL compared to some real-world counterparts [16], [17]. However, MIL elides many microarchitectural features not relevant to information flow.

III. SYNTAX AND SEMANTICS OF MIL

In this section, we present the syntax of MIL and its OoO and IO dynamic semantics. The presentation largely follows Guanciale et al. [4], but we highlight key differences and additions due to the formalization in HOL4. Informally, MIL

is a single static assignment (SSA) language [18], where variables in an assignment are unique microinstruction names. Ultimately, a MIL program, if it terminates successfully, computes a set of assignments of 64-bit values to such names. We assume that names are totally ordered, which induces an order on instructions via their assigned names that we call the *program order*. A program can thus be given as a linear list of guarded assignments to variables.

Example 1: We use the small parameterized MIL program below as a running example. The program compares the content of the register *reg* to 1 and sets the program counter (PC) to the memory address *adr* if this is the case, or increments the current PC value by 4 otherwise. It thus implements a high-level conditional branch on equal (`beq`) instruction.

```
tb0 := true ? 0; // zeroed name for PC load/store
tb1 := true ? reg; // get register identifier
tb2 := true ? load(REG, tb1); // load register value
tb3 := true ? tb2 == 1; // is the register value 1?
tb4 := true ? load(PC, tb0); // load PC value
tb5 := true ? adr; // get memory address
tb6 := tb3 ? store(PC, tb0, tb5); // store to PC
tb7 := true ? tb4 + 4; // increment PC value by 4
tb8 := !tb3 ? store(PC, tb0, tb7); // store to PC
```

To obtain a fully defined (“ground”) MIL program, the assignment variable names (`tbX`) must be replaced by non-negative integers, and the parameters *reg* and *adr* must be replaced by 64-bit words. We usually use variable name suffixes to indicate desired integer ordering, e.g., `tb0 < tb6`. Subsequently, we will omit `true` guards, e.g., we will write `tb0 := 0`.

A. Abstract Syntax

In Figure 2(a), we define the abstract syntax of MIL.

Names. We use unbounded HOL4 natural numbers as microinstruction names *t*, and predicate sets [19] for collections of names *N*. This approach theoretically permits infinite sets which are not meaningful in our context, but allowed easy transcription of set-related definitions from the original definition of MIL.

Values. Values *v* (and *a*) are 64-bit words encoded in the usual way for HOL [20]. The constant values *false*, *true*, and 0 are defined according to conventions of the C language. Besides finiteness and distinctness of *false* and *true*, the MIL metatheory (in contrast to the tools and examples) does not rely on anything specific about the word size.

Expressions. Expressions *e* (and *c*) are side-effect free and are assumed to include at least names and values. However, as long as requirements on the semantics of expressions (outlined in Section III-B) are met, expressions can be arbitrarily added to MIL without affecting the metatheory. In our HOL4 encoding, we defined expression syntax and semantics to match the BIR language, streamlining the translation from BIR to MIL.

Resources and operations. MIL operations are defined on a *resource* τ , which is either the PC, memory, or a register. An operation *o* is either an expression, or a load or store on a resource. Since there is a single PC resource, PC loads and stores are intended to take a name as first argument that is assigned to the value 0; this is implicit for Guanciale et al.

Microinstructions. A MIL microinstruction ι , or *instruction* for short, is an assignment of a name t to (the result of) an operation o , guarded by an expression c . A single higher-level instruction, e.g., at the ISA level, will typically be represented by many MIL instructions, which is why MIL is parameterized on a *translation* function explained in Section III-B.

B. Runtime States and Semantic Definitions

To provide dynamic semantics for MIL, we define runtime states for programs; Figure 2(b) lists the basic syntax we use.

Programs. MIL programs I are predicate sets of instructions. Whenever convenient, we consider instructions in I in program order (using the assigned instruction names).

Stores. Stores s are finite maps from names to values, where $\text{dom}(s)$ is the set of names that are mapped by s . We write $s(t) \downarrow$ ($s(t) \uparrow$) for $t \in \text{dom}(s)$ (resp. $t \notin \text{dom}(s)$).

States. In addition to a program I and store s , a MIL runtime state σ contains two sets of names C and F that respectively track whether an associated instruction has been committed to memory or its successor instructions have been fetched.

Observations. An observation obs is either the silent observation ϵ , a data load dl , a data store ds , or an instruction load il . The three latter include a memory address value.

Actions. Actions α represent transitions. Instructions are first executed (EXE). Then, if an instruction is a memory store, it can be committed (CMT), or, if it is a PC store, it can cause the next instructions to be fetched (FTC).

Labels. In contrast to Guanciale et al., *transition labels* l contain not only observations, but also the action performed by the transition and the name of the instruction for which the action was performed.

In abstract syntax, the program in Example 1, which we abbreviate $I_{\text{beq}}(\text{reg}, \text{adr})$, is written:

$$\left\{ \begin{array}{l} t_{b0} \leftarrow 0, t_{b1} \leftarrow \text{reg}, t_{b2} \leftarrow ld \mathcal{R} t_{b1}, t_{b3} \leftarrow t_{b2} == 1, \\ t_{b4} \leftarrow ld \mathcal{PC} t_{b0}, t_{b5} \leftarrow \text{adr}, t_{b6} \leftarrow t_{b3} ? st \mathcal{PC} t_{b0} t_{b5}, \\ t_{b7} \leftarrow t_{b4} + 4, t_{b8} \leftarrow !t_{b3} ? st \mathcal{PC} t_{b0} t_{b5} \end{array} \right\}$$

Executing the last instruction in I_{beq} is represented by a label $(il(pc_0 + 4), \text{FTC}(I), t_{b8})$, where pc_0 is the original PC value and I is the translation of the program at $pc_0 + 4$.

Bound and free names. For an expression e , its set of names $n(e)$ is defined recursively on the structure in the obvious way. An instruction ι has a *bound name*, written $bn(\iota)$, and a set of *free names*, written $fn(\iota)$; the set of all names in ι is written $n(\iota)$. The set of all bound names of instructions in a program I is written $bn(I)$. In addition, $n(l)$ yields the name in the label l . For example, if $\iota = t_{b6} \leftarrow t_{b3} ? st \mathcal{PC} t_{b0} t_{b5}$, then we have $bn(\iota) = t_{b6}$ and $fn(\iota) = n(t_{b3}) \cup n(st \mathcal{PC} t_{b0} t_{b5}) = \{t_{b0}, t_{b3}, t_{b5}\}$, so $n(\iota) = \{t_{b0}, t_{b3}, t_{b5}, t_{b6}\}$.

Semantics of expressions. The semantics of an expression e in store s is given by a partial function returning a value v , which we write $[e]s = v$. If the function is (un-)defined, we write $[e]s \downarrow$ (resp. $[e]s \uparrow$). We do not define an explicit canonical function for the semantics of expressions, since it is microarchitecture dependent. However, in contrast to Guanciale et al., we impose requirements on such functions:

- 1) $[e]s \downarrow$ if and only if $n(e) \subseteq \text{dom}(s)$.
- 2) If $s(t) = s'(t)$ holds for all $t \in n(e)$, then $[e]s = [e]s'$.
- 3) For all v and s , $[v]s = v$.

For validation, we implemented a function consistent with BIR semantics, where for example $e + e'$ is evaluated using `word_add` from the HOL4 word theory. Given a store s , an expression c evaluates to a *true guard condition*, written $[c]s$, whenever there exists v such that $[c]s = v$ and $v \neq \text{false}$.

Address and resource of store or load. Given the name t of a store or load instruction in a program, we need to be able to obtain the resource and the name of the instruction that computes the *address* that t targets. We therefore define the partial function addr so that $\text{addr}(I, t) = (\tau, t')$ if $t \leftarrow c?ld \tau t' \in I$ or $t \leftarrow c?st \tau t' t'' \in I$. For instance, $\text{addr}(I_{\text{beq}}, t_{b2}) = (\mathcal{R}, t_{b1})$ for the example program.

Store may and store active. To handle store-to-load dependencies we define two auxiliary functions $\text{str-may}(\sigma, t)$ and $\text{str-act}(\sigma, t)$ that determine, for a given load instruction t and state σ , respectively, a) the set of stores $\iota = t' \leftarrow c'?st \tau t_1 t_2$ that *may* by further instantiation of names smaller than t assign to the (possibly as yet unknown) load address t_0 of t , and b) the set of stores ι in $\text{str-may}(\sigma, t)$ that cannot be eliminated due to another store $t'' : t' < t'' < t$ overwriting either the store address t_1 of t' or the load address t_0 of t . Formally:

$$\begin{aligned} \iota \in \text{str-may}(\sigma, t) \text{ iff } & t' < t \wedge ([c']s \vee [c']s \uparrow) \wedge \\ & (s(t_1) = s(t_0) \vee s(t_1) \uparrow \vee s(t_0) \uparrow) \\ \iota \in \text{str-act}(\sigma, t) \text{ iff } & \iota \in \text{str-may}(\sigma, t) \wedge \\ & t'' \leftarrow c''?st \tau t'_1 t'_2 \in \text{str-may}(\sigma, t) \wedge t'' > t' \wedge \\ & [c'']s \rightarrow s(t'_1) \neq s(t_0) \wedge s(t'_1) \neq s(t_1) \end{aligned}$$

Example 2: The MIL program below loads the register r_1 from the memory address b_1 , copies the value of r_1 into r_2 if the flag in register z is set, saves the result into the memory address b_2 , and then increments the PC by 4. At a high level, the program thus implements conditional copying of memory on equal, and we refer to it as $I_{\text{ceq}}(b_1, b_2)$.

```
tc00 := 0; tc01 := r1; tc02 := r2;
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM, tc04); // [1of2] r1 := *b1
tc12 := store(REG, tc01, tc11); // [2of2]
tc21 := load(REG, tc03); // [1of3] cmov z, r2, r1
tc22 := tc21 == 1 ? load(REG, tc01); // [2of3]
tc23 := tc21 == 1 ? store(REG, tc02, tc22); // [3of3]
tc31 := load(REG, tc02); // [1of2] *b2 := r2
tc32 := store(MEM, tc05, tc31); // [2of2]
tc41 := load(PC, tc00); // [1of3] pc := pc + 4
tc42 := tc41 + 4; // [2of3]
tc43 := store(PC, tc00, tc42); // [3of3]
```

We assume that $I_{\text{ceq}}(b_1, b_2)$ runs after another initialization program I_0 , i.e., that $\sigma = (I_0 \cup I_{\text{ceq}}(b_1, b_2), s, C, F)$ and all instruction names in I_0 are before t_{c00} .

Suppose that in the state σ , we have $s(t_{c00}) \uparrow, \dots, s(t_{c43}) \uparrow$. Then, $\text{str-may}(\sigma, t_{c31})$ contains all register stores coming before t_{c31} , since the load address of t_{c31} is undefined and any previous register store instruction can potentially affect the loaded value of t_{c31} .

$N, C, F ::= \{t_1, t_2, \dots\}$	names (set)	$I ::= \{\iota_1, \iota_2, \dots\}$	program (set)
$v, a ::= false \mid true \mid 0 \mid \dots$	value (word64)	$s ::= [t_1 \mapsto v_1, t_2 \mapsto v_2, \dots]$	store (fmap)
$e, c ::= v \mid t \mid !e \mid e + e' \mid \dots$	expression	$\sigma ::= (I, s, C, F)$	state
$\tau ::= \mathcal{PC} \mid \mathcal{R} \mid \mathcal{M}$	resource	$obs ::= \epsilon \mid dl\ a \mid ds\ a \mid il\ a$	observation
$o ::= e \mid ld\ \tau\ t \mid st\ \tau\ t\ t'$	operation	$\alpha ::= EXE \mid CMT(a, v) \mid FTC(I)$	action
$\iota ::= t \leftarrow c?o$	instruction	$l ::= (obs, \alpha, t)$	transition label
(a)		(b)	

Fig. 2. MIL abstract syntax (a), and syntax used for MIL runtime state and executions (b).

Suppose σ is the state after the execution of all instructions in I_0 and the instructions on the first line, i.e., $s(t_{c00}) = 0, \dots, s(t_{c05}) = b_2$. Then, $str\text{-}may(\sigma, t_{c31})$ contains all stores in I_0 that update r_2 , as well as the store t_{c23} . $str\text{-}may(\sigma, t_{c31})$ does not contain t_{c12} , since the destination register of t_{c12} (r_1) differs from the source register of t_{c31} (r_2).

Suppose σ is the state after the execution of all instructions until t_{c22} , and let $s(t_{c21}) = 0$. Then, $str\text{-}may(\sigma, t_{c31})$ does not contain t_{c23} , since the guard condition of t_{c23} is false, and therefore the store will not be executed. Hence, $str\text{-}act(\sigma, t_{c31})$ contains the last instruction in $str\text{-}may(\sigma, t_{c31})$ not overwritten by a subsequent store. However, if $s(t_{c21}) = 1$, then $str\text{-}may(\sigma, t_{c31})$ contains t_{c23} and $str\text{-}act(\sigma, t_{c31}) = \{t_{c23}\}$.

Semantics of instructions. The semantics of instructions is given by a partial function taking an instruction ι and state σ and returning a value and an observation, which we write as $[\iota]\sigma = (v, obs)$. We define the function by case analysis on ι .

- $[t \leftarrow c?e]\sigma = (v, \epsilon)$, if $[e]s = v$.
- $[t \leftarrow c?ld\ \tau\ t']\sigma = (v, dl\ a)$, if $bn(str\text{-}act(\sigma, t)) = \{t''\}$, $s(t') = a$, $s(t'') = v$, $\tau = \mathcal{M}$, and $t'' \in C$.
- $[t \leftarrow c?ld\ \tau\ t']\sigma = (v, \epsilon)$, if $bn(str\text{-}act(\sigma, t)) = \{t''\}$, $s(t') = a$, $s(t'') = v$, and either $\tau \neq \mathcal{M}$ or $t'' \notin C$.
- $[t \leftarrow c?st\ \tau\ t_1\ t_2]\sigma = (v, \epsilon)$, if $s(t_1) = v$ and $s(t_2) \downarrow$.

Completed microinstructions. To guarantee *progress* during execution of MIL programs, we provide a different criterion than Guanciale et al. for instructions to be completed. Specifically, we define ι as *completed* in a state $\sigma = (I, s, C, F)$, written $\mathcal{C}(\sigma, \iota)$, whenever

- $\iota = t \leftarrow c?st\ \mathcal{M}\ t_1\ t_2$ and either $[c]s = false$ or $t \in C$
- $\iota = t \leftarrow c?st\ \mathcal{PC}\ t_1\ t_2$ and either $[c]s = false$ or $t \in F$
- $\iota = t \leftarrow c?o$, and either $[c]s = false$ or $t \in dom(s)$.

For example, if the value of *reg* is 1 in Example 1, then after $t_{b3} \leftarrow t_{b2} == 1$ has been executed (mapping t_{b3} to *true*), instruction t_{b8} becomes completed, since its guard is *false*.

C. Transition Step Relations

We define two dynamic semantics of MIL in the structural operational semantics style: an OoO semantics and an IO semantics. Specifically, we define, by the rules in Figure 3, the labeled OoO transition step relation, $\sigma \xrightarrow{l} \sigma'$, and the labeled IO transition step relation, $\sigma \xrightarrow{l} \sigma'$.

OoO-Exe. This rule computes the value v of an instruction with bound name t and records the result in the store by adding

the mapping $[t \mapsto v]$. This uses the semantics of instructions, and therefore relies on most functions above, such as *str-act*.

OoO-Ftc. This rule fetches an already-executed PC store instruction, which potentially adds more instructions to the program in the state. Intuitively, the function $translate(a, t)$ used in the rule looks up the code at the data area address a and generates the corresponding MIL instructions using names greater than t . Fetches thus enable MIL programs to have iterative and possibly diverging behavior.

OoO-Cmt. This rule commits an already-executed memory store instruction to memory. Both the memory address a and the new value v are part of the label's action, while only the former is included in the observation.

IO-Step. This rule processes instructions using the OoO rules, but deterministically following the program order.

For instance, in an initial state for I_{beq} , OoO-Exe transitions are enabled for the instructions for t_{b0} and t_{b1} . However, if $t_{b0} < t_{b1}$ as expected, only t_{b0} is enabled for an IO-Step transition, i.e., the instruction for t_{b0} must be completed before the instruction for t_{b1} .

The OoO semantics can be viewed as abstracting the behavior of a pipelined single-core microarchitecture which receives CISC-like ordered program instructions, and then translates each such instruction into one or more RISC-like microinstructions which are nondeterministically executed and (possibly) completed. For instance, the OoO semantics is reminiscent of the NetBurst microarchitecture used in Pentium 4 processors [16]. In contrast, the IO semantics is more akin to abstract ISA behavior, where execution must always proceed according to an order specified by a programmer or compiler. Silver is an example where the microarchitecture itself behaves similarly to the MIL IO semantics [17].

IV. METATHEORY OF MIL

While a MIL program has no canonical initial state at runtime, we define in this section a notion of state well-formedness that, intuitively, ensures program execution does not *go wrong*. However, well-formedness does not by itself guarantee *progress*, e.g., that execution will end up in a state where all instructions are completed. For progress, we define *resource-initialized* states, which prevent instruction execution from getting stuck. By comparison, the MIL semantics of Guanciale et al. [4] did not explicitly account for progress and only ruled out some forms of malformed states.

$$\begin{array}{c}
\frac{t \leftarrow c?o \in I \quad s(t) \uparrow \quad [c]s}{[t \leftarrow c?o](I, s, C, F) = (v, obs)} \quad \text{OoO-EXE} \\
\frac{}{(I, s, C, F) \xrightarrow{(obs, EXE, t)} (I, s + [t \mapsto v], C, F)} \\
\\
\frac{t \leftarrow c?st \mathcal{PC} t_1 t_2 \in I \quad t \notin F \quad s(t) = a}{\begin{array}{l} translate(a, \max(bn(I))) = I' \\ bn(str\text{-}may((I, s, C, F), t)) \subseteq F \end{array}} \quad \text{OoO-FTC} \\
\frac{}{(I, s, C, F) \xrightarrow{(il\ a, FTC(I'), t)} (I \cup I', s, C, F \cup \{t\})} \\
\\
\frac{\sigma \xrightarrow{(obs, \alpha, t)} \sigma'}{\forall \iota \in \sigma. \text{if } bn(\iota) < t \text{ then } \mathcal{C}(\sigma, \iota)} \quad \text{IO-STEP} \\
\frac{}{\sigma \xrightarrow{(obs, \alpha, t)} \sigma'} \\
\\
\frac{t \leftarrow c?st \mathcal{M} t_1 t_2 \in I \quad t \notin C}{\begin{array}{l} s(t) \downarrow \quad s(t_1) = a \quad s(t_2) = v \\ bn(str\text{-}may((I, s, C, F), t)) \subseteq C \end{array}} \quad \text{OoO-CMT} \\
\frac{}{(I, s, C, F) \xrightarrow{(ds\ a, CMT(a, v), t)} (I, s, C \cup \{t\}, F)}
\end{array}$$

Fig. 3. OoO and IO labeled transition step relation rules.

Assuming well-formedness and resource initialization enabled us to prove in HOL4 the *memory consistency* of the OoO and IO semantics of MIL, fully elaborating the previous pen-and-paper reasoning and filling in all the gaps. We believe this puts our notion of *conditional noninterference* for MIL on firm ground. In turn, this notion allows us to reason about information flow in MIL programs, as described in Section VI.

A. Well-formedness of States

We now define the requirements for a state $\sigma = (I, s, C, F)$ to be *well formed*. Properties 1 to 8 below are the basic sanity properties that, e.g., express the absence of dangling instruction references, that instruction dependencies form a directed acyclic graph, and that instruction execution is properly recorded in the store and elsewhere. For instance, in states including I_{beq} , property 2 requires that $t_{b1} < t_{b2}$, and property 3 forbids having $t_{b0} = t_{b2}$.

- 1) I is a finite set such that $C \cup F \subseteq \text{dom}(s) \subseteq \text{bn}(I)$.
- 2) If $\iota \in I$, $t \in \text{fn}(\iota)$, then $t < \text{bn}(\iota)$ and $\exists \iota' \in I$ such that $\text{bn}(\iota') = t$.
- 3) If $\iota \in I$, $\iota' \in I$, and $\text{bn}(\iota) = \text{bn}(\iota')$, then $\iota = \iota'$.
- 4) If $t \in C$, then $\text{bn}(str\text{-}may(\sigma, t)) \subseteq C$ and $\exists \iota \in I$ such that $\iota = t \leftarrow c?st \mathcal{M} t_1 t_2$.
- 5) If $t \in F$, then $\text{bn}(str\text{-}may(\sigma, t)) \subseteq F$ and $\exists \iota \in I$ such that $\iota = t \leftarrow c?st \mathcal{PC} t_1 t_2$.
- 6) If $\iota \in I$ for $\iota = t \leftarrow c?st \mathcal{PC} t_1 t_2$ or $t \leftarrow c?ld \mathcal{PC} t_1$, then $t_1 \leftarrow true?0 \in I$.
- 7) If $\iota \in I$ for $\iota = t \leftarrow c?st \tau t_1 t_2$, and $s(t) = v$, then $s(t_1) \downarrow$ and $s(t_2) = v$.
- 8) If $t \leftarrow c?e \in I$, $s(t) = v$, then $[t \leftarrow c?e]\sigma = (v, \epsilon)$.

Properties 9 to 11 below next ensure that guards behave as expected and do not block execution. For instance, property 9 says that if t_{b6} in I_{beq} has a stored value, then t_{b3} has a value stored which is not equal to *false*.

- 9) If $t \leftarrow c?o \in I$ and $s(t) \downarrow$, then $[c]s$.
- 10) If $t \leftarrow c?o \in I$, $t' \leftarrow c'?o' \in I$ and $t' \in n(c)$, then $c' = \text{true}$.
- 11) If $t \leftarrow c?o \in I$, $t' \leftarrow c'?o' \in I$, $t' \in n(o)$, $[c]s'$, and $[c']s' = v'$, then $v' \neq \text{false}$.

Finally, we impose analogous properties for output from the *translate* function; motivation and details are in the supplementary material [10]. In lieu of subject reduction for an explicitly typed language, we then proved in HOL4 that well-formedness is preserved by all the OoO and IO transition

rules whenever *translate* returns output with the required properties. In particular, the proof relies on that $t < t'$ whenever $\iota \in \text{translate}(v, t)$ and $t' \in n(\iota)$. From now on, we always assume that states are well formed.

B. State Resource Initialization

Consider a load instruction in a state, e.g., the instruction for t_{b2} in a state whose program includes I_{beq} . Intuitively, during an **EXE** transition for t_{b2} , the previous value for the register *reg* is copied to the store, which is done by finding the last completed store instruction on *reg*. However, if there is no such store instruction, t_{b2} can never be completed.

To address this problem in the MIL metatheory of Guanciale et al. [4], we introduce a notion of resource initialization for states. Specifically, we say that the predicate *initialized-resource-set*(σ, τ, V) is true precisely when, for all $v \in V$, there exists a completed store instruction for v and τ in σ such that there is no earlier load instruction in σ for v and τ . We then say that, in *resource initialized* states, *initialized-resource-set* holds for *all* possible values when $\tau = \mathcal{R}$ or $\tau = \mathcal{M}$, and for 0 when $\tau = \mathcal{PC}$.

For example, in a well-formed resource initialized state $\sigma = (I, s, C, F)$ whose program includes I_{beq} , we know there exists an instruction $t \leftarrow c?st \mathcal{R} t' t''$ such that $t \in \text{dom}(s)$, $s(t') = z$, and $t < t_{b4}$, ensuring that we can complete the load instruction t_{b4} with an **EXE** transition.

C. Executions, Commits, and Traces

We define MIL *executions* formally as (bounded) lists π of state-label-state triples (σ, l, σ') . More specifically, for π to be an *OoO execution*, it must be non-empty and its triples must follow the OoO transition relation, which we write as $\pi = \sigma_1 \xrightarrow{l_1} \sigma_2 \xrightarrow{l_2} \sigma_3 \dots$. Analogously, when π is an *IO execution*, we write $\pi = \sigma_1 \xrightarrow{l_1} \sigma_2 \xrightarrow{l_2} \sigma_3 \dots$. We also write $\pi \# \pi'$ for the concatenation of two executions.

For an execution π and a memory address a , the function *commits*(π, a) returns a list with the history of values written (i.e., sent to the memory subsystem) in π to a . We define *commits* by case analysis on the first transition in π so that, e.g., $\text{commits}(\sigma_1 \xrightarrow{(obs, CMT(a, v), t)} \sigma_2 \# \pi', a) = v, \text{commits}(\pi', a)$. Finally, the function *trace*(π) returns the trace of the execution π , which is its (possibly empty) list of non-silent observations. As one example, we have $\text{trace}(\sigma \xrightarrow{(dl\ a, \tau, t)} \sigma' \# \pi') = dl\ a, \text{trace}(\pi')$.

D. Functional Correctness: Memory Consistency

Intuitively, two models of program execution are memory consistent when they yield the same sequence of memory updates for each memory address, which ensures that the final result of a program (if there is one) is the same in both models. More formally, *memory consistency* of the OoO and IO semantics requires that writes to the same memory location are always seen in the same order by an observer, which we state and prove as our main theorem.

Theorem 1: For all well-formed and resource initialized states σ_1 and OoO executions $\pi = \sigma_1 \xrightarrow{l_1} \sigma_2 \dots$, there exists an IO execution $\pi_i = \sigma_1 \xrightarrow{l'_1} \sigma'_2 \dots$ such that for all (address) values a , the list of commits for a in π_i is a prefix of the list of commits for a in π (and vice versa for IO and OoO execution).

Proof: The proof relies on a two-step reordering lemma which says that if $\sigma_k \xrightarrow{l} \sigma_{k+1} \xrightarrow{l'} \sigma_{k+2}$ and $n(l') < n(l)$, then there exists σ'_{k+1} and l'' such that $\sigma_k \xrightarrow{l''} \sigma'_{k+1} \xrightarrow{l} \sigma_{k+2}$, where l' and l'' have the same commits.

The key steps of the proof are illustrated in Figure 4, and can be divided into two parts. The first part establishes, by induction on execution length, that for every OoO execution there is a corresponding *ordered* OoO execution, with the same initial and final state, and the same order of commits per address, where transition labels respect the total order on names. In the following, we use $\hat{\cdot}$ to identify ordered OoO executions. Let $\pi = \pi_0 \dashv\vdash \sigma_{k+1} \xrightarrow{l'} \sigma_{k+2}$ be an OoO execution; then by induction there is an ordered OoO execution $\hat{\pi}_0 = \hat{\pi}_1 \dashv\vdash \sigma_k \xrightarrow{l} \sigma_{k+1}$ of π_0 . Hence, $\pi' = \hat{\pi}_1 \dashv\vdash \sigma_k \xrightarrow{l} \sigma_{k+1} \xrightarrow{l'} \sigma_{k+2}$ is also an OoO execution. If $n(l) \leq n(l')$, then π' is an ordered execution of π ; otherwise, the two-step reordering lemma guarantees that there exists an OoO execution $\pi'' = \hat{\pi}_1 \dashv\vdash \sigma_k \xrightarrow{l''} \sigma'_{k+1} \xrightarrow{l} \sigma_{k+2}$. We use induction again to show that there is an ordered OoO execution $\hat{\pi}_2$ of $\hat{\pi}_1 \dashv\vdash \sigma_k \xrightarrow{l''} \sigma'_{k+1}$. Clearly, $\pi''' = \hat{\pi}_2 \dashv\vdash \sigma'_{k+1} \xrightarrow{l} \sigma_{k+2}$ is also an OoO execution. Since the label names in $\hat{\pi}_2$ are the union of the label names in $\hat{\pi}_1$ and $n(l')$, the label names in $\hat{\pi}_1$ are less than or equal to $n(l)$, and if $n(l') < n(l)$ then π''' is an ordered execution of π .

In the second part of the proof, we establish that any ordered OoO partial execution π''' can be extended to an ordered OoO execution π_c where all instructions in the last state of π''' have been completed and no other instruction has been completed. We reuse the above reasoning to guarantee that there is an ordered OoO execution $\hat{\pi}_c$ of π_c , with last state σ_c . Finally, we show that if an OoO execution is ordered and its last state has an upper bound t such that all instructions with name smaller than t have been completed and no other instruction has been completed, then this execution is an IO execution. Therefore, for every address a , the commits for a in π are a prefix of the commits for a in the IO execution $\hat{\pi}_c$.

The vice versa case is trivial, since any IO execution is also an OoO execution. ■

In summary, Guanciale et al. proved a two-step reordering lemma in their paper [4], which we formalized in HOL4 with substantial required effort. However, to complete the memory consistency proof, we also provide novel formal proofs that

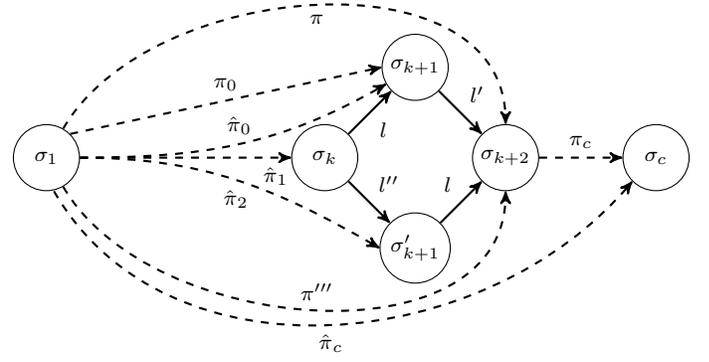


Fig. 4. Illustration of key steps in the memory consistency proof. π is the given OoO execution, σ_1 is the initial state in π , π_c is an extension to a state where all instructions in the last state of π (and no other) have been completed, and $\hat{\pi}_c$ is the IO execution with the commits from π as a prefix.

(1) the two-step reordering lemma implies the existence of an ordered OoO execution, (2) any OoO execution can be extended to complete all currently incomplete instructions, and (3) an ordered and completed OoO execution is an IO execution. These three properties were previously only hinted at and not formally stated or proved.

On one hand, memory consistency for the OoO semantics expresses that, subject to the conditions given by the semantics, executing instructions out-of-order is always correct. On the other hand, memory consistency provides a useful formal verification aid: to show that a real out-of-order processor pipeline satisfies memory consistency, it suffices to show that its design is simulated by the OoO semantics, without any need for dealing explicitly with instruction reordering. In practice, this requires demonstrating that processor scheduling is equally or more restrictive than MIL's conditions on resource loads and memory commits.

E. Confidentiality: Conditional Noninterference

In order to reason about information leaks via cache-based side channels transparently without an explicit cache model, we assume that the attacker can observe the address of a memory load ($dl\ a$), the address of a memory store ($ds\ a$), as well as the value of the program counter ($il\ a$). This approach makes the attacker more powerful than in many real-world scenarios, but is common in analysis of microarchitectural vulnerabilities [21] and for verifying constant time implementations [22]. In particular, the approach allows us to describe in a simple way, devoid of details on caches, when two states are indistinguishable by an attacker according to a given labeled state transition relation (for MIL, the OoO and IO relations).

Definition 1: States σ_1 and σ_2 are *trace-indistinguishable* for a labeled state transition relation T , written $\sigma_1 \simeq_T \sigma_2$, if for every T -execution π_1 starting in σ_1 , there exists a T -execution π_2 starting in σ_2 such that $trace(\pi_1) = trace(\pi_2)$, and \simeq_T is symmetric.

In the following, we assume a binary relation on states, \sim_ℓ , which we call the *security policy*. The security policy specifies the parts of the program state that contain sensitive/high information and the parts that contain public/low information; if states are related by \sim_ℓ , then this means they have the same public information and therefore cannot be distinguished by the attacker prior to execution. We usually assume that the attacker knows the executing program, which means that \sim_ℓ also constrains the current set of instructions to execute and future instruction fetches. Moreover, \sim_ℓ usually requires states to be initial for the program under analysis: i.e. no instruction of the program has been executed. We take the IO semantics as a *specification* of the permitted information flows, and consider a program secure if an OoO execution of the program does not leak more information than its IO execution. The following definition formalizes this intuition:

Definition 2: A system is *conditionally noninterferent* with respect to the security policy \sim_ℓ , written $CNI(\sim_\ell)$, if it holds that $\sim_\ell \cap \simeq_{IO} \subseteq \simeq_{OoO}$.

Unfortunately, conditional noninterference does not hold in general—execution of a program according to the OoO semantics can introduce new side channels. More specifically, there is a resource-initialized and well-formed state σ and policy \sim_ℓ such that $CNI(\sim_\ell)$ is false.

We demonstrate the CNI violation using a state with the program $I_{ceq}(b_1, b_2)$ from Example 2. IO execution of the state always produces the trace $[dl\ b_1, ds\ b_2, il\ (pc_0 + 4)]$. When the flag in the register z is 1, OoO execution produces one of three traces, due to the possibility of fetching t_{c42} independently of the memory operations and the fact that the memory store must follow the memory load to respect the data dependency introduced by t_{c23} , i.e., $[dl\ b_1, ds\ b_2, il\ (pc_0 + 4)]$, $[dl\ b_1, il\ (pc_0 + 4), ds\ b_2]$, and $[il\ (pc_0 + 4), dl\ b_1, ds\ b_2]$. On the other hand, the trace $[ds\ b_2, dl\ b_1, il\ (pc_0 + 4)]$ is only possible if the flag z is not 1, since then the memory store can be reordered ahead of the memory load. By observing such traces, the attacker learns the flag in z .

For this counterexample, the security policy \sim_ℓ requires the programs of the two initial states to have the shape $I_0 \cup I_{ceq}(b_1, b_2)$ and $I'_0 \cup I_{ceq}(b'_1, b'_2)$, where I_0 and I'_0 set the initial values of the resources accessed by I_{ceq} , and requires the instructions in I_{ceq} to be undefined in the initial stores. In this case, $CNI(\sim_\ell)$ can be proved only if \sim_ℓ also constrains I_0 and I'_0 to set the same initial value for z . Intuitively, this corresponds to considering z to be known by the attacker before program execution or to declassifying its value.

Due to the possibility of confidentiality violations, we develop a semi-automated strategy in Section VI to verify conditional noninterference of a given program.

V. TOOLS FOR ANALYSIS OF MIL PROGRAMS

A. Computing Executions and Traces Inside HOL4

Formalizing sets of instructions and names as HOL4 predicate sets was convenient for abstractly defining MIL and developing its metatheory. However, this encoding prevents many definitions from being computable, which is a prerequisite

for translation to CakeML. To obtain computable definitions, we introduced a refined runtime state (i, s, c, f) that replaces all sets with polymorphic lists. We then developed list-based analogues of the semantic definitions in Section III-B, such as *addr* and *str-act*, and proved that they preserve set-based behavior, assuming that names of instructions in i are unique.

Using our list-based semantic definitions, we developed a HOL4 function for running MIL refined runtime states and returning executions, dubbed *io-bounded-execution*. Besides the initial state, the function takes an *instruction offset* argument and a *fuel* argument. We found using fuel convenient since MIL program execution is not guaranteed to terminate. In *io-bounded-execution*, we proceed by looking up the instruction at the indicated offset, completing that instruction, and moving on to the next instruction in the list until fuel runs out. We proved the correctness of *io-bounded-execution* both in terms of IO and OoO transitions, but outline only the former and defer details to the supplementary material.

Soundness: If instructions in the initial state (i, s, c, f) are sorted by name and completed up to position p , and $io\text{-bounded-execution}((i, s, c, f), p, n) = \pi$, then π represents an IO execution starting in the initial state and ending in a state where all instructions up to position $p + n$ are completed.

Completeness: If the initial state is well-formed, resource initialized, and has instructions sorted by name and completed up to p , *io-bounded-execution* will indeed output an execution.

We used the same approach as for *io-bounded-execution* to develop a verified function dubbed *io-bounded-trace* that only outputs the corresponding trace of an execution from a given state, with some basic optimizations to handle large states and perform many transitions. These functions are useful not only for running concrete MIL programs—they also allow us to partially automate proofs [7].

B. Refinement of Computable Functions to CakeML

While feasible for states of small to moderate size, evaluating the functions *io-bounded-execution* and *io-bounded-trace* inside HOL4 can be slow and does not scale to large and long-running MIL programs. We therefore refined our datatypes and functions for MIL to be compatible with CakeML’s HOL4 *translation frontend* [23]. We then proved the refined functions equivalent to our previous list-based definitions. Once the CakeML translator accepted all our refined functions, we obtained a verified MIL evaluator as a native program.

C. Translation from BIR to MIL

To allow generating MIL programs from ISA level code, we developed an *unverified* translation in Standard ML (SML) from BIR to MIL, using the SML interfaces of each HOL4 theory. The main SML translation function takes a BIR program term and a function name g , and as a side effect defines a function in HOL4 with that name, mapping BIR block addresses (and other necessary parameters) to collections of MIL microinstructions. The function g then takes the place of *translate* in our MIL semantics; in particular, we can pass g to *io-bounded-trace* together with a (refined) MIL state.

Since MIL does not have a canonical expression semantics, we manually adapted the BIR expression semantics to MIL by introducing the corresponding expression abstract syntax and using the same HOL4 word theory operations as BIR in an executable MIL expression evaluation function.

VI. VERIFICATION OF CONDITIONAL NONINTERFERENCE

We develop a general verification strategy for conditional noninterference that follows the hypotheses of a lemma:

Lemma 1: If there exist (1) a relation \mathbf{L} such that $\sim_\ell \cap \simeq_{\text{IO}} \subseteq \mathbf{L}$ (i.e., \mathbf{L} underapproximates program information leakage during IO execution), (2) and a bisimulation \mathbf{R} for OoO semantics (i.e., \mathbf{R} overapproximates program information leakage during OoO execution), and (3) $\sim_\ell \cap \mathbf{L} \subseteq \mathbf{R}$ (i.e., the initial knowledge of the attacker and the IO information leakage “are not less than” the OoO leakage), then $\text{CNI}(\sim_\ell)$.

Below, we demonstrate our strategy using the MIL program $I_{\text{ceq}}(b_1, b_2)$ from Example 2. The supplementary material [10] contains applications of our strategy in HOL4 to verify CNI for the Example 1 program, the Example 2 program, and a program that moves values between two registers.

A. Computing the Relation \mathbf{L}

Our strategy uses the IO executor function *io-bounded-execution* described in Section V-A to analyze the information leakage relation symbolically, together with self composition [24]. Since the IO semantics is deterministic, we can compute the post-relation by limiting the analysis to maximal executions when programs terminate. In fact, a system is noninterferent for the IO semantics iff the traces of maximal executions of any two states in \sim_ℓ are indistinguishable.

For a state with I_{ceq} , the IO executor generates the trace $[dl\ b_1, dl\ b_2, il\ (pc_0 + 4)]$. By using self composition, we generate the relation $\mathbf{L} \triangleq pc_0 = pc'_0 \wedge b_1 = b'_1 \wedge b_2 = b'_2$, where primed variables are the parameters for the second state.

B. Identifying and Proving a Bisimulation Relation \mathbf{R}

Let $(I_1, s_1, C_1, F_1) = \sigma_1 \mathbf{R} \sigma_2 = (I_2, s_2, C_2, F_2)$. To guarantee that the two states can produce the same observations, i.e., lists of fetches and commits, we work under the assumption of control flow preservation, reflecting the no-branch-on-secrets condition common in cryptographic practice.

This condition leads to a number of constraints on \mathbf{R} that can be used in a proof search procedure:

- Preservation of executed, committed and fetched instructions, i.e., $\text{dom}(s_1) = \text{dom}(s_2)$, $C_1 = C_2$, and $F_1 = F_2$.
- Preservation of labels (addresses of PC stores and memory loads/stores), e.g., $s_1(t_{c43}) = s_2(t_{c43})$ for Example 2.
- Preservation of dependencies (including active stores for loads) and guards. For instance, for t_{c22} and t_{c23} in I_{ceq} , this leads to $s_1(t_{c21}) = s_2(t_{c21})$, since $t_{c21} == 1$ is used as the guard condition of t_{c22} and t_{c23} .

These constraints are then backpropagated to previous microinstructions, which for the example results in requiring that the initial value of the flag z (needed for t_{c22}) and pc are the same (needed for t_{c43}) in s_1 and s_2 .

The bisimulation proof is greatly simplified by control flow preservation. The main challenge is to prove preservation of the active stores. This is done by showing that an assignment to a name t will either have no effect on the active stores, or else the same instruction will be eliminated.

C. Proving the Entailment of the Bisimulation

The last verification step, $\sim_\ell \cap \mathbf{L} \subseteq \mathbf{R}$, is largely automated. For the initial states, each bisimulation constraint must be guaranteed by either \mathbf{L} (e.g., for Example 2 the equality of pc is implied by $pc_0 = pc'_0$ in \mathbf{L}), when the same information is leaked by both the OoO and IO semantics, or by \sim_ℓ (e.g., for Example 2, the equality of the flag z can be guaranteed only if we consider the initial value of z to be public, since it is not leaked by the IO execution), when the OoO execution introduces additional leakage.

VII. RELATED WORK

A. Theorem proving for hardware and its interfaces

Specifications of popular ISAs, e.g., ARMv8-A and RISC-V, are available for many theorem provers [14], [25], [26]. However, compilers and program analysis tools that only consider these specifications are unable to rule out illicit information flows due to microarchitectural vulnerabilities such as Spectre, Meltdown, and Foreshadow [1]–[3]. On the hardware side, theorem prover formalizations are available for HDLs and corresponding circuit synthesizers and compilers [27]–[32], but program analysis tools using such specifications have to target specific low-level microarchitectures and hardware, which may be unrelated to high-level languages or ISAs.

An alternative is to perform *end-to-end* specification and verification across high-level languages, ISAs, microarchitectures, and hardware. For instance, Löw et al. [17] connect the compiler for the CakeML language to the Silver ISA and single-core processor in HOL4, and Erbsen et al. [33] specify and verify in Coq the functional correctness (including instruction reordering) of a system based on a pipelined processor implementing the RISC-V ISA. However, these efforts focus only on functional correctness, and are tied to a particular stack of ISA and hardware. This makes proof reuse in other settings difficult. In particular, the instruction pipeline reordering proof by Erbsen et al. is specific to a processor defined in the Kami HDL. We believe that MIL, in contrast, can enable proof reuse across end-to-end verification efforts.

B. Formal models of low-level information flow

Several works have addressed the formalization of microarchitectural optimizations, such as different forms of speculation, to capture Spectre-like vulnerabilities [21], [34]–[37]. Similarly to MIL, these proposals model an attacker that can observe the program counter, memory load addresses, and memory store addresses. Their security conditions are defined as noninterference or a conditional hyperproperty, similarly to conditional noninterference, that compares information flows

of the same program in a speculative and a sequential semantics. The semantics by Barthe et al. [35] describes out-of-order execution, but memory commits have to be done in-order and consequently memory consistency is straightforward. Faidedeh et al. [37] consider a SAT-based register transfer level analysis of transient execution for concrete OoO processor designs, but they do not provide a general model. Other works only consider speculative in-order instruction processing.

Many of these works have inspired the implementation of tools, e.g., Spectector [38], to analyze program side channels using some form of (relational) symbolic execution. However, to our knowledge, the only mentioned work whose semantics and verification approach has reached an interactive theorem prover is that of Cheang et al. [36], which was formalized by Griffin and Dongol in Isabelle/HOL [39]. Using a translation from C-like programs to Isabelle theories similar to our BIR translation, Griffin and Dongol reason about information flow during speculative execution using Hoare-style triples, but they do not account for out-of-order execution. While our MIL semantics introduces nondeterminism relationally, i.e., by some states simply having several possible transitions according to the OoO step relation, the semantics used by Griffin and Dongol consults an abstract oracle to resolve nondeterminism [21].

C. Validation of hardware information flow models

Buiras et al. [15], [40] and Oleksenko et al. [41] developed tools (called Scam-V and Revizor, respectively) to validate hardware information flow models. The approaches are based on testing leakage models (e.g., the attacker observations of Section IV-E) using black box testing on actual CPUs. Both Scam-V and Revizor use a variation of conditional noninterference, where the goal is to establish that states that produce indistinguishable traces in a model produce indistinguishable cache footprints on the real hardware. We believe such tools can facilitate trustworthy connections between MIL-based information flow analyses and hardware behavior.

VIII. CONCLUSION

We presented a formalization in HOL4 of MIL, a language which captures key features of microarchitectures to allow reasoning about low-level program information flow. The formalization includes the in-order and out-of-order dynamic semantics of MIL, a proof of memory consistency between the two semantics, and a notion of conditional noninterference that rules out trace-driven cache based side channels. The formalization is around 34,000 lines of code with examples, and took around 24 person months to develop. The code [10] was tested on HOL4 kananaskis-14 and PolyML 5.9.

We envision that our MIL formalization and tools will be integrated into a trustworthy program information flow analysis workflow based on CakeML, where binaries for ISAs supported by HolBA are first represented in BIR and then translated to MIL to establish conditional noninterference or to demonstrate side channels. Our unverified BIR-to-MIL translation and verified example programs indicate that the

workflow is feasible, but the manual effort of conditional noninterference proofs is currently the main obstacle. In particular, bisimulation based reasoning can easily lead to unproductive exploration of the many possible transitions available due to nondeterminism. However, even without full automation of conditional noninterference proofs, we believe MIL and its metatheory and tools can improve productivity in formal verification of confidentiality properties of practical systems.

REFERENCES

- [1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Symposium on Security and Privacy*, 2019, pp. 1–19.
- [2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018, pp. 973–990.
- [3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium*, 2018, pp. 991–1008.
- [4] R. Guanciale, M. Balliu, and M. Dam, "InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Conference on Computer and Communications Security*, 2020, pp. 1853–1869.
- [5] HOL development team, "HOL interactive theorem prover," 2022. [Online]. Available: <https://hol-theorem-prover.org>
- [6] O. Acıçmez and Ç. K. Koç, "Trace-driven cache attacks on AES," in *Information and Communications Security*, 2006, pp. 112–121.
- [7] B. Barras, "Programming and computing in HOL," in *Theorem Proving in Higher Order Logics*, 2000, pp. 17–37.
- [8] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified CakeML compiler backend," *Journal of Functional Programming*, vol. 29, p. e2, 2019.
- [9] A. Lindner, R. Guanciale, and R. Metere, "TrABin: Trustworthy analyses of binaries," *Sci. Comput. Program.*, vol. 174, pp. 72–89, 2019.
- [10] K. Palmiskog, X. Yao, N. Dong, R. Guanciale, and M. Dam, "MIL formalization source code and documentation," 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.6997534>
- [11] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," *Proc. Computer Architecture*, pp. 34–44, 1985.
- [12] W.-M. Hwu and Y. N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," in *International Symposium on Computer Architecture*, 1986, pp. 297–306.
- [13] K.-A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Sjölander, and S. Kaxiras, "SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores," in *Conference on Programming Language Design and Implementation*, 2018, pp. 328–343.
- [14] A. Fox, "Improved tool support for machine-code decompilation in HOL4," in *Interactive Theorem Proving*, 2015, pp. 187–202.
- [15] H. Nemat, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, "Validation of abstract side-channel models for computer architectures," in *Computer Aided Verification*, 2020, pp. 225–248.
- [16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, vol. 5, 2001. [Online]. Available: <http://www.ecs.umass.edu/ece/koren/ece568/papers/Pentium4.pdf>
- [17] A. Löw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox, "Verified compilation on a verified processor," in *Conference on Programming Language Design and Implementation*, 2019, pp. 1041–1053.
- [18] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Symposium on Principles of Programming Languages*, New York, NY, USA, 1988, pp. 12–27.
- [19] J. Hurd, "Predicate subtyping with predicate sets," in *Theorem Proving in Higher Order Logics*, 2001, pp. 265–280.
- [20] J. Harrison, "The HOL Light theory of Euclidean space," *J. Autom. Reasoning*, vol. 50, pp. 173–190, 2012.
- [21] S. Cauligi, C. Disselkoben, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for software Spectre defenses," in *Symposium on Security and Privacy*, 2022, pp. 1517–1517.

- [22] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying constant-time implementations,” in *USENIX Security Symposium*, 2016, pp. 53–70.
- [23] M. O. Myreen and S. Owens, “Proof-producing translation of higher-order logic into pure and stateful ML,” *Journal of Functional Programming*, vol. 24, no. 2-3, pp. 284–315, 2014.
- [24] G. Barthe, P. R. D’Argenio, and T. Rezk, “Secure information flow by self-composition,” in *Computer Security Foundations Workshop*, 2004, pp. 100–114.
- [25] A. Fox and M. O. Myreen, “A trustworthy monadic formalization of the ARMv7 instruction set architecture,” in *Interactive Theorem Proving*, 2010, pp. 243–258.
- [26] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, “ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019.
- [27] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, “Modular deductive verification of multiprocessor hardware designs,” in *Computer Aided Verification*, 2015, pp. 109–127.
- [28] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, “Kami: A platform for high-level parametric hardware specification and its modular verification,” *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, 2017.
- [29] A. Lööw and M. O. Myreen, “A proof-producing translator for Verilog development in HOL,” in *International Conference on Formal Methods in Software Engineering*, 2019, pp. 99–108.
- [30] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, “The essence of Bluespec: A core language for rule-based hardware design,” in *Conference on Programming Language Design and Implementation*, 2020, pp. 243–257.
- [31] A. Lööw, “Lutsig: A verified Verilog compiler for verified circuit development,” in *Conference on Certified Programs and Proofs*, 2021, pp. 46–60.
- [32] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, “Formal verification of high-level synthesis,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, 2021.
- [33] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala, “Integration Verification Across Software and Hardware for a Simple Embedded System,” in *Conference on Programming Language Design and Implementation*, 2021.
- [34] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, “Hardware-software contracts for secure speculation,” in *Symposium on Security and Privacy*, 2021, pp. 1868–1883.
- [35] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, “High-assurance cryptography in the Spectre era,” in *Symposium on Security and Privacy*, 2021, pp. 788–805.
- [36] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, “A formal approach to secure speculation,” in *Computer Security Foundations Symposium*, 2019, pp. 288–303.
- [37] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, “A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors,” in *Design Automation Conference*, 2020, pp. 1–6.
- [38] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, “SPECTECTOR: Principled detection of speculative information flows,” in *Symposium on Security and Privacy*, 2020, pp. 1–19.
- [39] M. Griffin and B. Dongol, “Verifying secure speculation in Isabelle/HOL,” in *Formal Methods*, 2021, pp. 43–60.
- [40] P. Buiras, H. Nemat, A. Lindner, and R. Guanciale, “Validation of side-channel models via observation refinement,” in *International Symposium on Microarchitecture*, 2021, pp. 578–591.
- [41] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, “Revizor: Testing black-box CPUs against speculation contracts,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 226–239.

Synthesizing Instruction Selection Rewrite Rules from RTL using SMT

Ross Daly
Stanford University
Stanford, CA, USA
rdaly525@cs.stanford.edu



Jackson Melchert
Stanford University
Stanford, CA, USA
melchert@stanford.edu



Priyanka Raina
Stanford University
Stanford, CA, USA
praina@stanford.edu



Rajsekhar Setaluri
Stanford University
Stanford, CA, USA
setaluri@stanford.edu



Clark Barrett
Stanford University
Stanford, CA, USA
barrett@cs.stanford.edu



Caleb Donovanick
Stanford University
Stanford, CA, USA
donovick@cs.stanford.edu



Nestan Tsiskaridze Bullock
Stanford University
Stanford, CA, USA
nestan@stanford.edu



Pat Hanrahan
Stanford University
Stanford, CA, USA
hanrahan@cs.stanford.edu



Abstract—Creating a compiler for an instruction set architecture (ISA) requires a set of rewrite rules describing how to translate from the compiler’s intermediate representation (IR) to the ISA. We address this challenge by synthesizing rewrite rules from a register-transfer level (RTL) description of the target architecture (with minimal annotations about its state and the ISA format), together with formal IR semantics, by constructing SMT queries where solutions represent valid rewrite rules.

We evaluate our approach on multiple architectures, supporting both integer and floating-point operations. We synthesize both integer and floating-point rewrite rules from an intermediate representation to various reconfigurable array architectures in under 1.2 seconds per rule. We also synthesize integer rewrite rules from WebAssembly to RISC-V with both standard and custom extensions in under 4 seconds per rule, and we synthesize floating-point rewrite rules in under 8 seconds per rule.

I. INTRODUCTION

The end of Moore’s law and Dennard scaling means that processor performance will not continue to increase exponentially due to improvements in process technology. Future performance increases will instead be due to the increased efficiency of domain-specific architectures and accelerators. In their Turing Award lecture, John Hennessy and David Patterson envision such a future; they predict that these innovations will lead to a new golden age of computer architecture [33]. In order to realize this vision, there must be a corresponding golden age of software tools, programming models, and compilers to design and program specialized architectures [55].

Every new instruction set architecture (ISA) must be accompanied by a set of rewrite rules to be used in code generation. These rules describe how to transform a compiler’s intermediate representation (IR) to the ISA. Crafting these rules is a labor-intensive task and is often performed by

someone other than the ISA designer. Hence, the ISA must be carefully documented to support compiler writers—this too is a tedious, error-prone process. Moreover, changes to an ISA require new documentation and new rewrite rules.

This leads to a world where there are very few ISAs, and design space exploration is limited to microarchitectural details. To perform architectural design space exploration, a working compiler is critical to perform realistic benchmarking. The work in this paper arose in the context of the Agile Hardware Project, where one of the primary goals is to facilitate rapid design space exploration for a coarse-grained reconfigurable array (CGRA) [7]. We found that manually maintaining rewrite rules for a rapidly changing architecture was a constant pain point. This experience led us to develop a method for automatically synthesizing instruction selection rewrite rules, which is the primary contribution of this paper. Our method requires a register-transfer *level* (RTL)¹ description of the target architecture, a description of the architectural state, and a description of the instruction format. This method has made possible the efficient and algorithmic exploration of large design spaces [41], as generation of the rewrite rules can be efficiently performed without a human in the loop.

Even for established ISAs, it is easy to overlook nuances that are obvious to the ISA designers. This can lead to inefficiencies in compiled code. For example, the RISC-V ISA does not include equals or not-equals instructions but documents “pseudo operations” for performing them using a subtract and an unsigned less-than ($(x - y) < 1$ and $0 < (x - y)$ respectively) [2]. Similarly, there are no instructions for less-than-or-equals or greater-than-or-equals, each of which can

¹Not to be confused with Register Transfer *Language*

also be implemented as two instruction sequences using a less than and an xor ($(\bar{y} < x) \wedge 1$ and $(x < \bar{y}) \wedge 1$, respectively); however, these sequences are not documented.

Using the architecture’s RTL, we synthesize rewrite rules by constructing first-order logic queries whose solutions, obtained using a satisfiability modulo theories (SMT) solver, represent instruction selection rewrite rules. Additionally, we propose a methodology for abstracting complex operations, such as floating point operations, which proved too costly for previous SMT-based approaches [12]. While some prior work [18], [19], [49], [12] tackled similar problems, they used manually-defined ISA specifications in the form of enumerated lists of instructions with their parameters and semantics. Using RTL directly has the benefit of avoiding this manual specification step. This is particularly important when doing design space exploration, as it is difficult to maintain both the RTL and a corresponding formal specification for a rapidly changing design. ISA specifications also do not typically capture the instruction format or the instruction decode logic, both of which are needed for an end-to-end correctness argument. In addition to these benefits, using the RTL directly also presents unique challenges which we address. Our main contributions are as follows:

- Formalization of the correctness criteria for a general class of rewrite rules between arbitrary IRs and RTL-based architectures.
- A technique for supporting *parametric* rewrite rules.
- A method for abstracting operations whose semantics are either unknown or too complex to model efficiently (e.g., floating-point operations).
- A methodology for efficiently encoding and solving the rewrite rule synthesis problem using SMT.

In our evaluation, we synthesize rewrite rules from CoreIR (an IR designed for RTL) [16] to a family of CGRAs. We also synthesize rewrite rules from WebAssembly to various RISC-V architectures. We target both the base RISC-V ISA and a number of extensions, including extensions with floating-point operations. All of these tasks can be done in seconds. Additionally, we are able to synthesize short multi-instruction sequences for pseudo-operations such as those mentioned above (whether officially documented or not). These take at most 90 seconds to synthesize.

The rest of this paper is organized as follows. Section II provides background on compilers, instruction selection, rewrite rules, and SMT. Section III formalizes rewrite rules and describes our encoding of the problem into SMT. Section IV presents case studies highlighting the utility and performance of the tool. Section V covers related work and, finally, Section VI provides future steps to take towards a general, automatically-derived compiler.

II. BACKGROUND

A. Code Generation

Most compilers share a common structure: a front end which translates a high-level language into an IR, an optimizer

Notation	Meaning
$BV[n]$	Sort for bitvectors of length n
$+_{[n]}, -_{[n]}, \times_{[n]}, \div_{[n]}$	Arithmetic modulo 2^n
$+f[n]$	n -bit floating-point addition
$x \circ y$	Bitvector concatenation
$x[msb : lsb]$	Bitvector extraction
$\text{ite}(c, x, y)$	If-then-else: if c then x else y
$a[i]$	Read from array a at index i
$a[i] := v$	Result of updating array a at index i with value v
$T =$	Algebraic data type T with
$C_1(s_1 : \sigma_1) \mid$	constructors C_1, C_2 , testers is_C_1
$C_2(s_2 : \sigma_2, s_3 : \sigma_3)$	and is_C_2 , and selectors s_i of sort σ_i

TABLE I: Theory-specific notation.

which optimizes the IR, and a code-generator which translates the IR into a hardware-specific representation (which then may be further optimized for the target architecture). The code generation stage typically involves instruction selection, scheduling, resource assignment, and assembly.

There has been significant work devoted to developing instruction selection algorithms [29], [25], [26], [45], [20], [4], [24], [23], [10] that use a set of pre-defined rewrite rules to translate IR programs to architectural instructions. These rewrite rules are dependent on the target ISA and are usually constructed manually. In this paper, we automatically synthesize rewrite rules from the RTL of target architectures;

B. Logical Setting

We work in the setting of many-sorted logic (see e.g., [21], [54]). Let S be a set of *sort symbols* (sorts in this setting play a role similar to types in type theory). For every sort $\sigma \in S$, we assume an infinite set of variables of that sort. We assume the usual definitions of terms, literals, formulas, and interpretations, and use \models to denote the satisfiability relation between interpretations and formulas. We write $e\{x \mapsto t\}$ for the result of simultaneously replacing each occurrence of x in e by t . If \mathbf{x}_1 and \mathbf{x}_2 are two vectors of variables, we write $\mathbf{x}_1 :: \mathbf{x}_2$ to denote their concatenation. A term of the form $\text{ite}(\varphi, t_1, t_2)$ is an if-then-else operator, whose meaning is the same as t_1 in an interpretation I where $I \models \varphi$, and the same as t_2 otherwise.

A *theory* \mathcal{T} assigns meaning to certain theory-specific symbols by fixing a class of allowable interpretations (e.g., it may fix the meaning of the symbol ‘+’ to be the addition function). A formula φ is \mathcal{T} -*satisfiable* (resp., \mathcal{T} -*unsatisfiable*, \mathcal{T} -*valid*) if it is satisfied by some (resp., no, all) interpretations in \mathcal{T} . The satisfiability modulo theories (SMT) problem is simply the question of determining \mathcal{T} -satisfiability of a formula for some given theory \mathcal{T} . SMT solvers solve this problem for a standard set of useful theories (and their combinations).

Some examples of common theories supported by SMT solvers include fixed-width bit-vectors, arrays, integer and floating-point arithmetic, uninterpreted functions, and algebraic data types. Table I lists some notation from these theories that we will use in illustrative examples below. A more thorough introduction to SMT can be found in [9].

III. SYNTHESIZING REWRITE RULES

Rewrite rules are a key component in instruction selection, as they indicate the options for how to transform one or more IR instructions into one or more architecture-specific instructions. In this section, we show how to formalize and solve the rewrite rule synthesis problem using SMT.

A. Intermediate Representation Formalization

An intermediate representation (IR) includes a collection of instructions which can be composed together in various ways to represent programs. IR instructions can be represented in many ways, including as graphs or as functions. Here, we represent IR instructions as SMT formulas. The formulas encode how an instruction's inputs are transformed into a set of outputs. Formally, let $\mathbf{x} = (x_1 : \sigma_1, \dots, x_k : \sigma_k)$ be a vector of variables. Then, the tuple $\mathbf{IR}(\mathbf{x}) = (IR_1(\mathbf{x}), \dots, IR_l(\mathbf{x}))$ is an IR instruction with k inputs (each x_i is an input) and l outputs (represented by each IR_j). The value of output j for a given concrete input (c_1, \dots, c_k) is given by constructing the formula $IR_j(c_1, \dots, c_k)$ and then evaluating it using the semantics of the theory operations in the formula. For example, an 8-bit adder with two outputs, the sum and the carry-out, and inputs x_1 and x_2 of sort $BV_{[8]}$ could be represented as:

$$(x_1 +_{[8]} x_2, (0 \circ x_1 +_{[9]} 0 \circ x_2)[8 : 8]).$$

For the concrete input (11111111, 00000001), the outputs are 00000000 and 1, respectively.

A formula-tuple \mathbf{IR} need not represent only a single instruction. A complex operation or *pseudo-instruction* can be represented as a composition of other instructions. Our SMT representation can easily accommodate composition: if an output IR_j from \mathbf{IR}_1 is connected to an input x_i in \mathbf{IR}_2 , then the composition is simply the result of substituting IR_j for x_i in \mathbf{IR}_2 , i.e., $\mathbf{IR}_2 \{x_i \mapsto IR_j\}$. Below, we assume that \mathbf{IR} represents some IR program (comprising one or more IR instructions) that we wish to find a rewrite rule for.

B. Architecture Formalization

An architecture is a circuit that is parameterized by a single architectural instruction value (separate from and not to be confused with the IR instructions mentioned above), which indicates how other inputs and existing states are transformed into outputs and next states. As above, we represent an architecture as a tuple of SMT formulas. The instruction itself is an input to the architecture, which we assume can be modeled as a variable $inst$ of sort τ . We further let $\mathbf{y} = (y_1 : \tau_1, \dots, y_m : \tau_m)$ be a vector of variables with sorts in Σ , where τ_i is the sort of the architecture's i 'th input. The tuple $\mathbf{Arch}(inst, \mathbf{y}) = (Arch_1(inst, \mathbf{y}), \dots, Arch_n(inst, \mathbf{y}))$ is an architecture with $m + 1$ inputs and n outputs. As an example, consider an 8-bit ALU with 4 operations. An input $inst$ of sort $BV_{[2]}$ selects which operation to perform on two

other inputs, y_1 and y_2 , both of sort $BV_{[8]}$. Its single output is also of sort $BV_{[8]}$. For this example, \mathbf{Arch} could be:

$$\begin{aligned} &(\text{ite}(inst = 00, y_1 -_{[8]} y_2, \\ &\quad \text{ite}(inst = 01, y_1 +_{[8]} y_2, \\ &\quad \quad \text{ite}(inst = 10, y_1 *_{[8]} y_2, y_1 \div_{[8]} y_2))). \end{aligned}$$

States. Architectures with states can be modeled by including current state values as inputs and next state values as outputs. Suppose $\mathbf{z} = (z_1 : \omega_1, \dots, z_p : \omega_p)$ are variables representing the states. Then, we can represent the architecture as:

$$\begin{aligned} \mathbf{Arch}(inst, \mathbf{y}, \mathbf{z}) = \\ &(Arch_1(inst, \mathbf{y}, \mathbf{z}), \dots, Arch_n(inst, \mathbf{y}, \mathbf{z}), \\ &\quad Arch_{n+1}(inst, \mathbf{y}, \mathbf{z}), \dots, Arch_{n+p}(inst, \mathbf{y}, \mathbf{z})), \end{aligned}$$

where $Arch_{n+i}$ are formulas that encode the next-state function for the i^{th} state variable. An example with states appears in Section III-C, below.

Composing Architectures. A rewrite rule for an IR program might require more than one instruction at the architectural level. Fortunately, as was the case for IR programs, it is straightforward to compose multiple architectures using our SMT representation. Let $\mathbf{Arch}_1(inst_1, \mathbf{y}_1, \mathbf{z}_1)$ and $\mathbf{Arch}_2(inst_2, \mathbf{y}_2, \mathbf{z}_2)$ be two architectures with m_1 and m_2 inputs, p_1 and p_2 states, and n_1 and n_2 outputs, respectively, and suppose that output i of \mathbf{Arch}_1 is passed into input j of \mathbf{Arch}_2 . Let $inst = (inst_1, inst_2)$, $\mathbf{y} = \mathbf{y}_1 :: (y_{2,1}, \dots, y_{2,j-1}, y_{2,j+1}, \dots, y_{2,m_2})$, $\mathbf{z} = \mathbf{z}_1 :: \mathbf{z}_2$, and $\mathbf{y}'_2 = \mathbf{y}_2 \{y_{2,j} \mapsto Arch_{1,i}(inst_1, \mathbf{y}_1, \mathbf{z}_1)\}$. Then, the composition is:

$$\begin{aligned} \mathbf{Arch}(inst, \mathbf{y}, \mathbf{z}) = \\ &(Arch_{1,1}(inst_1, \mathbf{y}_1, \mathbf{z}_1), \dots, Arch_{1,n_1}(inst_1, \mathbf{y}_1, \mathbf{z}_1), \\ &\quad Arch_{2,1}(inst_2, \mathbf{y}'_2, \mathbf{z}_2), \dots, Arch_{2,n_2}(inst_2, \mathbf{y}'_2, \mathbf{z}_2), \\ &\quad Arch_{1,n_1+1}(inst_1, \mathbf{y}_1, \mathbf{z}_1), \dots, Arch_{1,n_1+p_1}(inst_1, \mathbf{y}_1, \mathbf{z}_1), \\ &\quad Arch_{2,n_2+1}(inst_2, \mathbf{y}'_2, \mathbf{z}_2), \dots, Arch_{2,n_2+p_2}(inst_2, \mathbf{y}'_2, \mathbf{z}_2)). \end{aligned}$$

C. Rewrite Rule Formalization

A rewrite rule defines how a specific IR program can be implemented using one or more instructions of a particular architecture. We start with a simple but incomplete definition of a rewrite rule and incrementally build up a definition with more generality and sophistication. The simplest rewrite rule is a tuple $(\mathbf{IR}, \mathbf{Arch}, inst_c)$, where \mathbf{IR} is an IR program, \mathbf{Arch} is an architecture (without states for now), and $inst_c$ is a concrete constant (i.e., a constant that maps to a particular domain value, like 0 or 1) of sort τ . We say such a tuple is a *valid* rewrite rule if the following formula is well-formed and T -valid:

$$\forall \mathbf{x}. \mathbf{Arch}(inst_c, \mathbf{x}) = \mathbf{IR}(\mathbf{x}) \quad (1)$$

Note that well-formedness requires that \mathbf{Arch} and \mathbf{IR} have the same number of inputs and outputs and that corresponding inputs and outputs have the same sort. As an example, take again the sum output of the IR program given in Sec. III-A,

that is, $\mathbf{IR} = (x_1 +_{[8]} x_2)$, and suppose \mathbf{Arch} is as given in Section III-B. Then, (1) holds when $inst_c = 01$, and so $(\mathbf{IR}, \mathbf{Arch}, 01)$ is a valid rewrite rule. In practice, things can be more complicated in several ways, which we address next.

Bindings. One problem with (1) is that the inputs and outputs of the IR rarely match those of the architecture. A more general rewrite rule is $(\mathbf{IR}, \mathbf{Arch}, inst_c, \mathbf{b}^{in}, \mathbf{b}^{out})$, where $(\mathbf{b}^{in}, \mathbf{b}^{out})$ is a pair of formula tuples, called a *binding*, that specifies how to map between the inputs and outputs of the two formulas. The rewrite rule is valid if the following formula is well-formed and valid:

$$\forall \mathbf{x}. \mathbf{b}^{out}(\mathbf{Arch}(inst_c, \mathbf{b}^{in}(\mathbf{x}))) = \mathbf{IR}(\mathbf{x}). \quad (2)$$

Here, well-formedness means $\mathbf{b}^{in}(\mathbf{x}) = (b_1^{in}(\mathbf{x}), \dots, b_m^{in}(\mathbf{x}))$, where each $b_i^{in}(\mathbf{x})$ has sort τ_i . We also require $\mathbf{b}^{out}(\mathbf{w}) = (b_1^{out}(\mathbf{w}), \dots, b_j^{out}(\mathbf{w}))$, where $\mathbf{w} = (w_1, \dots, w_n)$, the sort of each w_i matches $Arch_i$, and the sort of each b_j^{out} matches IR_j . As an example, consider $\mathbf{b}^{in} = (x_2, x_1)$ and $\mathbf{b}^{out} = (w_2)$. This binding swaps the two IR inputs and only uses the second architecture output.

Another complexity with bindings is that sometimes it is necessary to map the IR inputs to only a subset of the architecture inputs (for example, mapping a unary IR operation to an ISA supporting only binary operations). The extra inputs which do not correspond to any IR input must not have any effect on the output. To model this, we extend \mathbf{b}^{in} so that, in addition to \mathbf{x} , it also takes additional arguments $\mathbf{y} = (y_1 \dots y_m)$ with sorts (τ_1, \dots, τ_m) . The idea is that the binding can choose to simply map some variable y_i to an extra architecture input. With this extension, we can write the new rewrite rule formula as follows:

$$\forall \mathbf{x}, \mathbf{y}. \mathbf{b}^{out}(\mathbf{Arch}(inst_c, \mathbf{b}^{in}(\mathbf{x}, \mathbf{y}))) = \mathbf{IR}(\mathbf{x}). \quad (3)$$

Finally, we can handle the full generality of architectures with states by including these in the binding as well, where \mathbf{b}^{in} is extended to be a function of sort $(\sigma_1 \dots \sigma_k, \tau_1 \dots \tau_k, \omega_1 \dots \omega_p) \rightarrow (\tau_1 \dots \tau_m, \omega_1 \dots \omega_p)$, and \mathbf{b}^{out} also takes an additional p inputs of sort $\omega_1, \dots, \omega_p$.

$$\forall \mathbf{x}, \mathbf{y}, \mathbf{z}. \mathbf{b}^{out}(\mathbf{Arch}(inst_c, \mathbf{b}^{in}(\mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{x}). \quad (4)$$

As an example, consider a simple architecture which either multiplies its inputs and accumulates the result into a register file z (represented by an array variable) at index 0 while outputting the product or performs a subtraction, both outputting the result and storing it at index 1 of the register file. Assume the instruction is of sort $BV_{[1]}$, and the other inputs are of sort $BV_{[8]}$. All operators use 8-bit arithmetic (so we will omit the [8] subscript to ease readability). The formula for the architecture is then:

$$\mathbf{Arch}(inst, y_1, y_2, z) = (\text{ite}(inst = 0, y_1 * y_2, y_1 - y_2), \text{ite}(inst = 0, z[0] := z[0] + (y_1 * y_2), z[1] := y_1 - y_2))$$

Note that the first formula in the \mathbf{Arch} tuple represents the output of the architecture, while the second represents the next state of z . Now, suppose we are searching for a rewrite rule for $\mathbf{IR}(\mathbf{x}) = (x_3 * x_2) + x_1$. One valid rule is $inst_c = 0$,

$\mathbf{b}^{in}(\mathbf{x}, \mathbf{y}, z) = (x_3, x_2, z[0] := x_1)$, and $\mathbf{b}^{out}(\mathbf{w}) = w_2[0]$ (note that w_2 , represents the second input to \mathbf{b}^{out} , which corresponds to the register file state). This rule represents a solution using $inst_c = 0$ when x_1 is the value of $z[0]$, x_2 drives the y_2 input, and x_3 drives the y_1 input. The result is stored at index 0 of the (next state value of the) register file.

D. Rewrite Rule Synthesis

We next formalize the problem of synthesizing rewrite rules. We assume that we are given \mathbf{IR} and \mathbf{Arch} representing an IR program and an architecture, respectively. We must find $inst_c, \mathbf{b}^{in}$, and \mathbf{b}^{out} . Starting from (4), we can simply replace $inst_c, \mathbf{b}^{in}$, and \mathbf{b}^{out} with variables to get a (second-order) formula. It is also useful to make the bindings a function of the instruction, as we explain below. Thus, we have:

$$\begin{aligned} \exists inst, \mathbf{b}^{in}, \mathbf{b}^{out}. \forall \mathbf{x}, \mathbf{y}, \mathbf{z}. \\ \mathbf{b}^{out}(inst, \mathbf{Arch}(inst, \mathbf{b}^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{x}). \end{aligned} \quad (5)$$

If (5) holds, then there exists a valid rewrite rule.

In order to use (5) for a practical rewrite rule synthesis algorithm, we must additionally specify what kinds of functions are allowed for \mathbf{b}^{in} and \mathbf{b}^{out} . These functions should tell us how to map the inputs and outputs, but should not introduce extra functionality. For non-state inputs to the architecture, we simply require that the binding either pick a variable in \mathbf{x} or pass through the corresponding variable from \mathbf{y} .

For state inputs, there are two² cases. For programmable states (states with compile-time addresses that can be written and read by instructions, e.g., a register file), we allow the binding to *update* part of the state with a variable in \mathbf{x} . This corresponds to a previous instruction storing its result (the input for the current instruction) in the state. We do this by using array variables for these states and allowing the binding to write to the arrays. Other states, such as the accumulators or other non-programmable registers, are passed through unchanged by the binding. Formally, we require:

$$b_i^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}) = \begin{cases} y_i \text{ or } x_j (1 \leq j \leq k), & \text{if } i \leq m, \\ z_{i-m}, & \text{if } i > m \text{ (non-programmable)} \\ \text{update}(z_{i-m}, inst, \mathbf{x}), & \text{otherwise,} \end{cases}$$

where $\text{update}(z, inst, \mathbf{x})$ is one or more array writes to z at indexes specified by one or more fields in $inst$ and with values from the variables in \mathbf{x} . The output binding is similar:

$$b_j^{out}(inst, \mathbf{w}) = \begin{cases} w_i (1 \leq i \leq n + p), \text{ or} \\ \text{read}(w_i, inst) (n + 1 \leq i \leq n + p), \\ \text{where } w_i \text{ is programmable,} \end{cases}$$

where $\text{read}(w, inst)$ is a read from the array w at an index specified by some field of $inst$. Implicit in this formulation is the requirement that instructions must either directly output their result or write them to programmable state in a single a

²A third kind of state with computed addresses (like indirect loads and stores), can be handled in a way similar to [12], or by using the computed address from the architecture and the IR in the output bindings.

cycle. Pipeline registers and other micro-architectural state fall into the category of states which cannot be bound. We discuss possible approaches for handling pipelining in Section VI.

We next explain how to solve (5), subject to the constraints on bindings. But first, we introduce two useful generalizations.

Synthesizing Parametric Rewrite Rules. Sometimes, we are interested in finding a *parameterized* rewrite rule that works for a family of IR nodes (for instance, the family of IR instructions that multiply a constant parameter by some input). Rather than having to discover a different rewrite rule for each value of the parameter, we would like to solve the problem once and have it work for all possible values of the parameter. Formally, let \mathbf{c} be a vector of parameters, and let $\mathbf{IR}(\mathbf{c}, \mathbf{x})$ be a family of IR nodes parameterized by \mathbf{c} . Using equation (5) as a starting point, the new rewrite formula becomes:

$$\forall \mathbf{c}. \exists inst, \mathbf{b}^{in}, \mathbf{b}^{out}. \forall \mathbf{x}, \mathbf{y}, \mathbf{z}. \mathbf{b}^{out}(inst, \mathbf{Arch}(inst, \mathbf{b}^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{c}, \mathbf{x}). \quad (6)$$

In other words, we would like there to be an appropriate instruction encoding for each value of the parameter c . As it stands, this formulation is not very useful, as it does not tell us how to connect the instruction to the parameter. However, by Skolemizing (6), we get the following:

$$\exists inst, \mathbf{b}^{in}, \mathbf{b}^{out}. \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}. \mathbf{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}(inst(\mathbf{c}), \mathbf{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))) = \mathbf{IR}(\mathbf{c}, \mathbf{x}). \quad (7)$$

where now, $inst$ is a function from \mathbf{c} to instructions.³

Abstracting Complex Operations. Complex operations (e.g., floating-point arithmetic) can present a challenge. However, it is often the case that there are identical complex operations in the IR and in the architecture. We can handle such situations by replacing such complex operations with *uninterpreted functions* [13]. We must be careful about how this is done though. If we simply introduce new function symbols in the formulas for the IR and the architecture, they will be implicitly *existentially* quantified when checking for satisfiability, leading to spurious results as the solver can choose *any* interpretation. Hence, introduced function symbols must be *universally* quantified. Formally, let \mathbf{Arch}^{abs} and \mathbf{IR}^{abs} be the abstract versions of \mathbf{Arch} and \mathbf{IR} , respectively, where the complex operations are removed and replaced with a vector of function symbols \mathbf{f} . Then, building on (7), we get the following formulation for the fully general rewrite rule synthesis formula:

$$\exists inst, \mathbf{b}^{in}, \mathbf{b}^{out}. \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{f}. \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, \mathbf{f}) = \mathbf{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), \mathbf{f}, \mathbf{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (8)$$

³Technically, to maintain logical equivalence, \mathbf{b}^{in} and \mathbf{b}^{out} should also be functions of \mathbf{c} , but for simplicity, we omit this, keeping the restrictions on their form introduced above. We also did not find any additional dependency on \mathbf{c} to be needed in practice.

E. Rewrite Rule Synthesis Implementation

Here, we detail several additional considerations required to solve the rewrite rule synthesis problem formalized above in practice. Specifically, we discuss (i) removing second-order quantifiers; (ii) encoding instructions; (iii) formula optimizations; and (iv) solving algorithm optimizations.

Removing Second-Order Quantifiers. Note that $inst$, \mathbf{b}^{in} , \mathbf{b}^{out} , and \mathbf{f} are all quantified functions. In order to use an SMT solver, we first need to find an equivalent formulation using only first-order quantification. For the binding functions, this is straightforward. Given the restrictions outlined above, there are only a finite number of possible binding functions.⁴ Let \mathcal{B} be the set of all legal bindings $(\mathbf{b}^{in}, \mathbf{b}^{out})$. Then, formula (8) is equivalent to

$$\exists inst. \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{f}. \bigvee_{(\mathbf{b}^{in}, \mathbf{b}^{out}) \in \mathcal{B}} \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, \mathbf{f}) = \mathbf{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), \mathbf{f}, \mathbf{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (9)$$

Unfortunately, just satisfying this formula does not tell us which binding to use, so in practice, we also add an *indicator variable* i , whose value indicates which binding was used. Formally, we extend the notion of binding to a triple $(b, \mathbf{b}^{in}, \mathbf{b}^{out})$, where b is an integer unique to each binding. Then, our formula becomes:

$$\exists inst, i. \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{f}. \bigvee_{(b, \mathbf{b}^{in}, \mathbf{b}^{out}) \in \mathcal{B}} i = b \wedge \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, \mathbf{f}) = \mathbf{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), \mathbf{f}, \mathbf{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (10)$$

To remove the quantification on \mathbf{f} , we can use Ackermannization [3]. For each function symbol f , we replace each instance of f with a fresh variable of the same sort as the return sort of f and add constraints requiring that if the arguments to any two of those instances of f are equal, then the fresh variables representing those instances are equal too. Assume, for ease of presentation, that $\mathbf{f} = (f)$ and f appears only once in \mathbf{IR}^{abs} , with arguments \mathbf{s} , and once in \mathbf{Arch}^{abs} , with arguments \mathbf{t} . Then, (10) is equivalent to⁵

$$\exists inst, i. \forall \mathbf{c}, \mathbf{x}, \mathbf{y}, \mathbf{z}, f_1, f_2. \bigvee_{(b, \mathbf{b}^{in}, \mathbf{b}^{out}) \in \mathcal{B}} i = b \wedge (\mathbf{s} = \mathbf{t} \rightarrow f_1 = f_2) \rightarrow \mathbf{IR}^{abs}(\mathbf{c}, \mathbf{x}, f_1) = \mathbf{b}^{out}(inst(\mathbf{c}), \mathbf{Arch}^{abs}(inst(\mathbf{c}), f_2, \mathbf{b}^{in}(inst(\mathbf{c}), \mathbf{x}, \mathbf{y}, \mathbf{z}))). \quad (11)$$

Encoding Instructions. Above, we have assumed a simple instruction model, where instructions are taken from some sort τ . In practice, an architecture may have a variety of

⁴To ensure finiteness, we limit the *update* operation mentioned above to allow no more updates than there are IR inputs.

⁵With some abuse of notation, if $P(f)$ is a formula containing f , and f_1 is a variable whose sort matches the return sort of f , we write $P(f_1)$ to mean the result of replacing the application of f in P by f_1 .

instructions, each with different components. This can be modeled by letting τ be an *algebraic data type* (ADT), with different constructors for each type of instructions. This also solves the problem of how to handle *inst* as a function of \mathbf{c} . Some types of instructions allow *immediate* values to be encoded as part of the instruction. For those instructions, we allow a parameter from \mathbf{c} to appear as the immediate value. This is a very limited type of functional dependence on \mathbf{c} , but it is sufficient for modeling the kinds of parametric rewrite rules we are interested in.

To see how this works, consider as an example two formats from the RISC-V integer instruction set (RV32I): (i) R-type: register-register instructions; and (ii) I-type: register-immediate instructions. We can model these using the ADT:

$$\begin{aligned} INST = & RType(op : BV_{[7]}, rd : BV_{[5]}, func3 : BV_{[3]}, \\ & rs1 : BV_{[5]}, rs2 : BV_{[5]}, func7 : BV_{[7]}) \mid \\ & IType(op : BV_{[7]}, rd : BV_{[5]}, func3 : BV_{[3]}, \\ & rs1 : BV_{[5]}, imm : BV_{[12]}) \end{aligned}$$

This could be further refined by declaring *op*, *func3*, etc. as additional data types with limited sets of values. To handle the dependence on parametric values, we add a constraint stating that some immediate value is equal to a parameter. For example, if we want to encode the case where the immediate of *IType* is a constant c , we add the constraint $is_IType(inst) \wedge imm(inst) = c$. To consider many possible mappings of constants to immediates, we use a disjunction over a set of possibilities as we do with bindings.

Formula Optimizations.

For non-trivial designs, it is too expensive to repeat the architecture and IR formulas for every disjunct in the set of bindings. An alternative is to introduce additional variables for the inputs to and outputs from the architecture and to have the bindings operate only on those variables. For ease of presentation, let's go back to formula (5) and write it as:

$$\begin{aligned} \exists inst. \forall \mathbf{x}, \mathbf{y}, \mathbf{z}. \\ \bigvee_{(\mathbf{b}^{in}, \mathbf{b}^{out}) \in \mathcal{B}} \mathbf{b}^{out}(inst, \mathbf{Arch}(inst, \mathbf{b}^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}))) \\ = \mathbf{IR}(\mathbf{x}). \end{aligned} \quad (12)$$

This is equivalent to:

$$\begin{aligned} \exists inst. \forall \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{u}, \mathbf{v}, \mathbf{w}. \\ \left(\bigvee_{(\mathbf{b}^{in}, \mathbf{b}^{out}) \in \mathcal{B}} (\mathbf{b}^{out}(inst, \mathbf{u}) = \mathbf{v} \wedge \mathbf{b}^{in}(inst, \mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{w}) \right) \\ \rightarrow (\mathbf{Arch}(inst, \mathbf{w}) = \mathbf{u} \wedge \mathbf{v} = \mathbf{IR}(\mathbf{x})). \end{aligned} \quad (13)$$

In practice, it can also be inefficient to include memories and register files in the architecture. An alternative is to remove them and add an additional input for every read port and output for every write port. From the point of view of the rewrite rule synthesis, the problem is equivalent. This is the approach we take in our experiments. For example, the RISC-V register file,

which has the property that register 0 always holds 0, can be modeled with two formulae:

One for reads:

$$\begin{aligned} \text{let } r_1 = \begin{cases} 0 & \text{if } rs1 = 0 \\ v_1 & \text{otherwise} \end{cases} \quad r_2 = \begin{cases} 0 & \text{if } rs2 = 0 \\ v_1 & \text{if } rs1 = rs2 \neq 0 \\ v_2 & \text{otherwise} \end{cases} \\ \text{in } (r_1, r_2) \end{aligned}$$

and one for writes: $(ite(rd = 0, s, v))$

In the first formula, v_1 and v_2 are the values bound into the register file (or more precisely added as inputs to the architecture). r_1 and r_2 represent the values read from the register file. $rs1$ and $rs2$ are the read addresses calculated by the architecture from its instruction. Note that this is equivalent to having two reads on an array without an intervening update. However, it massively simplifies the task of generating \mathbf{b}^{in} , as we do not need to reason about how $rs1$ and $rs2$ will be derived from *inst*.

In the second formula, rd is the write address, s represents the previous state of the written register, and v is the value to be written. Similar to the abstraction of reads, this significantly simplifies the generation of \mathbf{b}^{out} . These simplifications are possible as we do not care about the full state of the register file. We only care about the two indices which are read and the one index that is written.

Solving Strategy. While some SMT solvers have support for quantified formulas, it is well-known that quantified formulas often lead to performance and robustness problems (and indeed, we observed this in preliminary experiments). We therefore adopt an external technique to solve the final quantified SMT queries, all of which are in *exists-forall* form:

$$\exists \mathbf{a}. \forall \mathbf{b}. \phi(\mathbf{a}, \mathbf{b}) \quad (14)$$

Our technique is inspired by the counter-example guided synthesis (CEGIS) [51] approach introduced in [15] and more formally described in [27]. The algorithm consists of alternating phases. The algorithm first suggests a solution for \mathbf{a} by simply checking the satisfiability of $\phi(\mathbf{a}, \mathbf{b})$. If \mathbf{a}_c is the value found, it then checks whether this works for all values of \mathbf{b} by checking the satisfiability of $\neg\phi(\mathbf{a}_c, \mathbf{b})$. If this is unsatisfiable, then \mathbf{a}_c is a solution for \mathbf{a} in (14). Otherwise, let \mathbf{b}_c be the satisfying value found. We simply update ϕ to be $\phi(\mathbf{a}, \mathbf{b}) \wedge \phi(\mathbf{a}, \mathbf{b}_c)$ and repeat. Essentially, we thus collect many *sample points*, \mathbf{b}_c with the hope that after enough are collected, it will drive the search to find a value for \mathbf{a} that satisfies (14). We found that in our setting of rewrite rule synthesis, this approach works well.

IV. EVALUATION

We evaluate the above approach for rewrite rule synthesis by showing the ability to efficiently synthesize rewrite rules in two settings. First, we synthesize rewrite rules from the CoreIR intermediate representation to different CGRA processing elements and, second, from the WebAssembly intermediate representation to RISC-V with extension.

We implement the architectures in the Magma hardware description language [1], [55]. We chose Magma as it has first class support for formal analysis through its associated “hwtypes” library [37], whose semantics match those of the SMT-LIB theory of bitvectors. We construct an SMT formula for the architecture by tracing the inputs of the circuit and the outputs of the architectural state to the outputs of the circuit and the inputs of its architectural state. While Magma is convenient, it is not essential; any HDL could be used to generate a formal model. We specify IRs directly in SMT using pysmt [27] and use Boolector [43] as the SMT solver. Additionally, we implement minimal compilers which apply the synthesized rules in order to compare to existing hand-coded tools. Details of our full experimental set up and more results can be found in the appendix.

A. Rewrite Rules for CGRAs

Our first case study targets CGRAs, style of spatial architecture similar to FPGAs which have been of increasing interest to both academia and industry. CGRAs differ from FPGAs by employing larger processing elements (PEs) instead of lookup tables (LUTs). Further, CGRAs typically have more restricted word-level routing networks rather than bit-level routing networks [39]. We evaluate our ability to synthesize rewrite rules for such architectures by synthesizing rewrite rules from CoreIR to four different PEs. We chose CoreIR as a source IR as it is formally specified [16], [40].

1) *CGRA Processing Element Implementation:* We use four versions (PE-A, PE-B, PE-C, PE-F) of an internally developed 16-bit processing element. PE-A contains a two-input ALU that can perform bit-wise operations, comparisons, shifts, addition, and multiplication, along with a lookup table for Boolean operations. Each ALU input can be driven by an external signal or a local immediate constant. PE-F adds 16-bit floating point (bfloat16) addition and multiplication to PE-A. We then extend PE-A with operations commonly occurring in image processing applications. PE-B extends PE-A with absolute difference ($|x-y|$), and PE-C extends PE-B with fused multiply-add with an immediate constant ($x*const + y$). Generating such a collection of similar architectures is a common practice when doing design space exploration. Our synthesis method, combined with a tool such as VTR [42] to perform place and route, could enable a designer to evaluate a large design space on real benchmarks.

2) *Rewrite Rule Synthesis:* We evaluate our ability to synthesize rewrite rules for CoreIR’s 16-bit integer instructions (`i16`), Boolean instructions (`i1`), and floating point instructions using Bfloat16 [17] (`bfloat16`).

The times to derive these rewrite rules are shown in Figure 1. Note that while most CoreIR operations can be mapped to the base PE, some can only be mapped to one or more of the variants. Each rule for the integer PEs can be found within 1.1 seconds. Additionally, the floating-point instructions can be found for PE-F within 1.2 seconds.

In Table II, we show the total time in seconds spent synthesizing rewrite rules (a SAT result) or proving that no

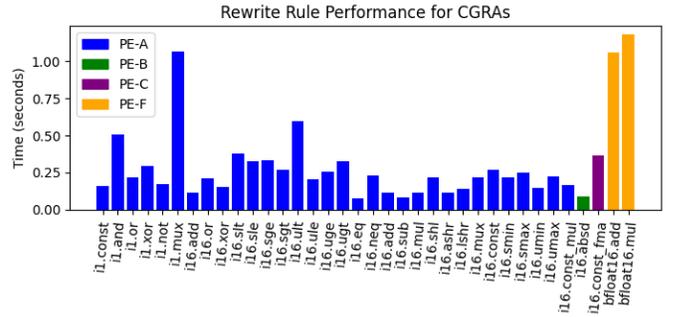


Fig. 1: The median time over 10 runs needed to derive a rewrite rule for various CoreIR operations to different PE architectures.

	PE-A	PE-B	PE-C	PE-F
UNSAT (s)	0.81	0.74	0.34	118.09
SAT (s)	8.63	10.15	11.06	91.49
Total (s)	9.44	10.88	11.40	209.58

TABLE II: Total time generating SAT results and UNSAT results, for each PE design.

rewrite exists (an UNSAT result, potentially due to the lack of a matching abstraction) for each PE design. Targeting the integer PEs is extremely fast, taking less than 12 seconds per design to generate a full set of rewrite rules. The process is "slow" for PE-F requiring about 3.5 minutes. However, this time is trivial compared to the time it would take to manually write these rules.

B. Rewrite Rules for RISC-V

Our second case studies shows how our technology can be used to synthesize rewrite rules from WebAssembly targeting RISC-V processors. WebAssembly is an intermediate representation designed to be a target for web applications. The IR itself has formally-defined semantics for each operation, making it suitable for our method.

We extract the post-instruction-fetch portion of the processor in order to give it the appearance of having an instruction input. Further, we replace the register file with the simplified model described in Section III-E. These transformations require only a handful of lines of boilerplate python for each architecture. Additionally, we construct specifications of instruction formats as ADTs and provide any necessary annotations for the register file (i.e., which registers have special semantics, like register 0 in RISC-V).

1) *RISC-V Implementation:* In addition to implementing a processor for the base RV32I ISA, we implement processors for the RV32IM and RV32IF standards. The "M" extension adds instructions for multiplication, division, and remainder. The "F" extension adds support for floating point operations. Full details can be found in the RISC-V manual [2]. In addition to these standard extensions, we define our own extension RV32X, which adds common bit-counting operations, which are defined in WebAssembly. Specifically: count-leading-zeros

Instruction	RV32I	RV32IM	RV32IX	RV32IF
<code>i20.const</code>	0.3	10.4	1.8	4.2
<code>i32.le_s</code>	2.2	27.3	3.7	80.1
<code>i32.ge_s</code>	1.6	30.8	4.5	71.7
<code>i32.le_u</code>	1.6	25.7	4.7	75.1
<code>i32.ge_u</code>	2.4	18.1	2.2	51.2
<code>i32.eq</code>	2.1	23.5	3.3	22.3
<code>i32.ne</code>	2.2	6.4	1.2	9.9

TABLE III: Median SMT performance in seconds for synthesizing two sequential instructions for `i20.const` and comparison instructions.

(`i32.clz`), count-trailing-zeros (`i32.ctz`), and population count (`i32.popcnt`).

2) *Rewrite Rule Synthesis*: We evaluate our ability to synthesize rewrite rules for WebAssembly’s 32-bit integer instructions (`i32`) and a subset of floating point instructions (`float`). The integer instructions also include pop-count, count-leading-zeros, and count-trailing-zeros.

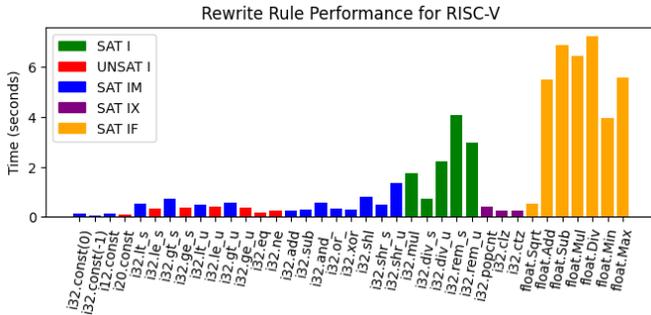


Fig. 2: Time needed to synthesize a single RISC-V instruction for each RISC-V Architecture. SAT means a rewrite rule was discovered. UNSAT means there is provably no single instruction rewrite that is possible. Reported times are the median result over 10 runs.

In Figure 2, either the time to synthesize a rewrite rule (SAT) or the time to prove that a rewrite rule does not exist (UNSAT) is shown for each IR instruction. Synthesis for RV32I succeeds in finding all instructions executable as a single instruction on the target architecture. For the integer processors, all rules are discovered within 4.1 seconds, with most only taking a few hundred milliseconds. Proving that rewrite rules do not exist is also possible within 4.1 seconds. For RV32IF, all the rules are found within 22 seconds, with most taking less than 8 seconds. Proving that particular rules like `i32.rem_s` are not possible takes up to 38 seconds. RV32IF contains many floating point instances, each requiring an expensive new universally quantified variable (explained in Section III-E). This can mostly explain the higher time compared to the other architectures.

Some comparison instructions are impossible to implement in a single instruction (a fact verified by our method), so we searched for sequences of two instructions, by composing two architectures as described in Section III-B. The times to find rewrites for these comparison operations for each of

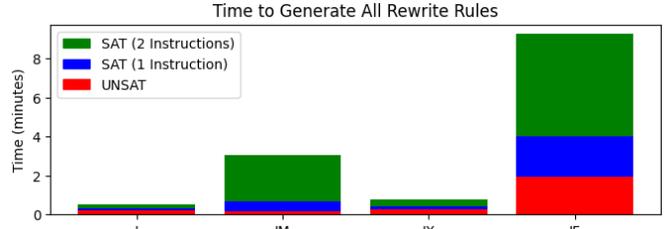


Fig. 3: Total time to generate single instruction SAT results, 2 instruction SAT results, and UNSAT results of 37 rewrite rules for each RISC-V architecture.

the RISC-V architectures are shown in Table III. We are able synthesize these rules for RV32I and RV32IX in a few seconds. For RV32IM and RV32IF, which are significantly more complex circuits, synthesis times are under 31 and 81 seconds, respectively. We note that verifying a rewrite rule can be done nearly instantaneously (well under a second for any rule we discovered). Therefore, given the knowledge that RV32I is a subset of RV32IF, one could simply verify that rules generated for RV32I work for RV32IF in order to avoid the longer synthesis times which arise from the complexity of floating point.

Similar to Table II, in Figure 3 we show the time spent synthesizing rewrite rules or proving no rewrite rule exists for each RISC-V architecture. This includes the time for proving that the instructions in Table III cannot be accomplished in one instruction, and the time for synthesizing each two-instruction rule. Results from targeting RV32I and RV32IX are fast, each taking less than a minute. Results targeting RV32IM and RV32IF are slower at around 3 minutes and 9 minutes, respectively, but this is still significantly faster than manually writing these rules.

V. RELATED WORK

In recent years, many new techniques and tools have been developed for synthesis based on SAT and SMT solving [30], [31], [34], [50], [52], [51], [53]s. In the SKETCH language, for example, a programmer provides a specification and a partial program with “holes” [52], [51]. SKETCH attempts to fill these holes so that the complete program matches the specification. However, due to the nuances of targeting RTL, we found that a direct encoding into SMT formulas was more flexible and convenient than using an existing program synthesis system. One promising approach is Syntax-guided synthesis (SyGuS) [5], [6], [48], in which a program must be synthesized within a given grammar to meet a given specification (the grammar and specification are given using a variant of the SMT-LIB language [8]). Exploring possible uses of SyGuS in this context is an interesting avenue for future work.

Perhaps more relevant is the work of Dias and Ramsey [18], [19], [49], who, in their 2006 work, propose a system to synthesize rewrite rules using an ISA specification where

each instruction is specified as a distinct formula. They use a pattern-matched syntax tree to synthesize these rules. In contrast, we use SMT to find all equivalences. Further, we use the RTL directly rather than a manually specified enumeration of instructions. This distinction is especially important during design space exploration, when automating as much as possible is crucial.

More recently, Buchwald, Fried, and Hack proposed a system which, like the work of Dias and Ramsey, synthesizes rewrite rules using an enumeration of an ISA's instructions [12]. However, instead of using pattern matching they leverage SMT to find rewrite rules for integer instructions. They notably lack support for floating-point, which we can handle efficiently. One interesting contribution of their work is the ability to synthesize control flow instructions by modeling them as a set of Boolean functions which indicate which branch target was taken. Applying a similar method in our approach is an interesting avenue for future work.

VI. DISCUSSION AND FUTURE WORK

Our technique for rewrite rule synthesis is a step towards automatically synthesizing a complete code generator from an RTL description of the target architecture. Future work includes two directions: synthesizing more kinds of rewrite rules, and targeting more expressive RTL. Pipelined architectures could leverage unpipelining [38] or unrolling [11] (with side conditions to ensure progress) to generate a model with the desired properties. Alternatively, if the RTL is derived from a high-level language, we could capture the synthesized design before micro-architectural details are added.

Architects often explore many alternatives when designing new hardware. This is often done incrementally. They propose a design change, implement it, then reevaluate the efficiency. A major impediment to design space exploration is implementing the software changes needed to compile the application to the new accelerator. The work in this paper enables automatically deriving part of the code generator and is one step towards the goal of eventually building a complete system for rapid and automated design space exploration.

REFERENCES

- [1] Magma. <https://github.com/phanrahan/magma>.
- [2] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2. RISC-V Foundation, 2017.
- [3] W. Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundation of Mathematics*, 1954.
- [4] Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.
- [5] Rajeev Alur, Rastislav Bodik, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emına Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. To appear in Marktoberdrof NATO proceedings, 2014. http://sygus.seas.upenn.edu/files/sygus_extended.pdf, retrieved 2015-02-06.
- [6] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emına Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
- [7] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovan, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, et al. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.
- [8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [9] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018.
- [10] Eli Bendersky. *A deeper look into the LLVM code generator, Part 1*, Feb 2013.
- [11] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.
- [12] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 300–313, 2018.
- [13] Jerry R Burch and David L Dill. Automatic verification of pipelined microprocessor control. In *International Conference on Computer Aided Verification*, pages 68–80. Springer, 1994.
- [14] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.
- [15] Chih-Hong Cheng, Natarajan Shankar, Harald Ruess, and Saddek Bensalem. EFSMT: A logical framework for cyber-physical systems. *CoRR*, abs/1306.3456, 2013.
- [16] Ross Daly and Lenny Truong. Invoking and linking generators from multiple hardware languages using coreir. In *Proceedings of the 1st Workshop on Open-Source EDA Technology*, 2018.
- [17] Jeffrey Dean, Greg S Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc Aurelio Ranzato, Andrew Senior, Paul Tucker, et al. Large scale distributed deep networks. 2012.
- [18] Joao Dias and Norman Ramsey. Converting intermediate code to assembly code using declarative machine descriptions. In *International Conference on Compiler Construction*, pages 217–231. Springer, 2006.
- [19] Joao Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. *ACM Sigplan Notices*, 45(1):403–416, 2010.
- [20] Helmut Emmelmann, F-W Schröer, and Rudolf Landwehr. Beg: a generator for efficient back ends. *ACM Sigplan Notices*, 24(7):227–237, 1989.
- [21] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.
- [22] Martin Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*.
- [23] Christopher W Fraser and David R Hanson. *A retargetable C compiler: design and implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [24] Christopher W Fraser, David R Hanson, and Todd A Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):213–226, 1992.
- [25] Mahadevan Ganapathi. *Retargetable Code Generation and Optimization Using Attribute Grammars*. PhD thesis, 1980. AAI8107834.
- [26] Mahadevan Ganapathi and Charles N. Fischer. Description-driven code generation using attribute grammars. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 108–119, New York, NY, USA, 1982. Association for Computing Machinery.
- [27] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT workshop*, 2015.
- [28] Philip B Gibbons and Steven S Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 11–16, 1986.
- [29] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 231–254, New York, NY, USA, 1978. Association for Computing Machinery.
- [30] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

- [31] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017.
- [32] Christopher G Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.
- [33] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.
- [34] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224, 2010.
- [35] Ron Kimmel. Demosaicing: image reconstruction from color ccd samples. *IEEE Transactions on image processing*, 8(9):1221–1228, 1999.
- [36] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [37] Caleb Donovan Leonard Truong. hwtypes. <https://github.com/leonardt/hwtypes>.
- [38] Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Proceedings of the 33rd annual Design Automation Conference*, pages 558–563, 1996.
- [39] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.
- [40] Cristian Mattarei, Makai Mann, Clark Barrett, Ross G Daly, Dillon Huff, and Pat Hanrahan. Cosa: Integrated verification for agile hardware design. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.
- [41] Jackson Melchert, Kathleen Feng, Caleb Donovan, Ross Daly, Clark Barrett, Mark Horowitz, Pat Hanrahan, and Priyanka Raina. Automated design space exploration of cgra processing element architectures using frequent subgraph analysis. *arXiv preprint arXiv:2104.14155*, 2021.
- [42] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai Min Wang, Mohamed ElDafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. Vtr 8: High performance cad and customizable fpga architecture modelling. *ACM Trans. Reconfigurable Technol. Syst.*, 2020.
- [43] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.
- [44] Mark Nixon and Alberto Aguado. *Feature extraction and image processing for computer vision*. Academic press, 2019.
- [45] Eduardo Pelegri-Llopert and Susan L Graham. Optimal code generation for expression trees: an application burs theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308, 1988.
- [46] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.
- [47] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.
- [48] Mukund Raghothaman and Abhishek Udupa. Language to Specify Syntax-Guided Synthesis Problems. *CoRR*, abs/1405.5590, 2014.
- [49] Norman Ramsey and Joao Dias. Resourceable, retargetable, modular instruction selection using a machine-independent, type-based tiling of low-level intermediate code. *ACM SIGPLAN Notices*, 46(1):575–586, 2011.
- [50] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. *SIGPLAN Not.*, 48(4):305–316, March 2013.
- [51] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, 2013.
- [52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.
- [53] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):497–518, 2013.
- [54] Cesare Tinelli and Calogero G. Zarba. Combining decision procedures for sorted theories. In José Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence*, pages 641–653, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [55] Lenny Truong and Pat Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In Benjamin S. Lerner, Rastislav Bodik, and Shriram Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA*, volume 136 of *LIPICs*, pages 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [56] Henry S Warren. *Hacker’s delight*. Pearson Education, 2013.
- [57] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. Interstellar: Using halide’s scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383, 2020.
- [58] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312, 2011.

APPENDIX

In addition to showcasing the efficiency of generating instruction rewrite rules, we wrote two compilers, one targeting the CGRA PEs, and one targeting the RISC-V processors, that showcase the synthesized rewrite rules can be used in real application compilation. The CGRA rewrite rule synthesis and compiler are actively being used in production in our lab’s efforts to design and run applications on our CGRA [7].

1) *CGRA Compilation Results:* We apply our synthesized rewrite rules to a number of image processing applications written in the domain-specific language Halide [47]. Halide is generally amenable to hardware acceleration [46], [57], making it a suitable source language to target our PE designs. The standard Halide compiler first lowers the program to its internal IR consisting of multiple computational kernels and structured for-loops [47]. Each kernel is further lowered to a dependency graph of CoreIR instructions. Our instruction selector then applies the synthesized rewrite rules to transform each kernel into a graph of PE instructions.

We selected four typical image processing applications: (1) Gaussian blur, an algorithm for blurring an image using convolution with a Gaussian kernel [44]; (2) A bfloat16 version of Gaussian blur; (3) Harris corner detection, which finds sharp corners of objects in images [32]; and (4) A complete camera pipeline, which is representative of end-to-end processing of raw sensor data to a final image [35]. The camera pipeline includes kernels for hot pixel suppression, demosaicing, and color correction. These applications have 3, 3, 10, and 14 distinct kernels, respectively, and the number of operations within a kernel range from just a single operation to almost 200.

We compare our synthesized rewrite rules to an existing hand-coded set for PE-F. Table IV shows the instruction counts for each application with each set of rewrite rules. The code sizes for the synthesized rewrite rules are the same or better than the sizes of those from the hand-coded rules. For Harris, which contains the `i16.umin` and `i16.smin` operations, the hand-coded result uses 2 instructions⁶ (`i16.lte` and `i16.mux`), but since we synthesize rewrite rules directly for `i16.umin` and `i16.smin`, the instruction selector produces more efficient code. Additionally, our instruction selector works for the new PE variants automatically. It uses the `i16.absd` instruction in both PE-B and PE-C, reducing the total instructions for Camera. Similarly, it leverages the `i16.const-fma` instruction in PE-C, greatly reducing the total instructions for Gaussian and slightly reducing them for Camera. This example demonstrates that it is easy to extend PEs with new instructions and automatically update the set of valid rewrite rules.

2) *RISC-V Compilation Results:* We also show that we can compile branch-free C programs using our synthesized rewrite rules. This approach has been used to evaluate other code

⁶We are unsure at this point whether this is a result of the architecture being updated without a corresponding update to the hand-coded rewrite rules or whether this rule was just overlooked by the original author of the tool. In any event this sort of mistake motivates this paper.

Application	Hand-Coded	Synthesized			
	PE-F	PE-F	PE-A	PE-B	PE-C
Gaussian i16	20	20	20	20	12
Gaussian bfloat16	20	20	N/A	N/A	N/A
Harris	116	109	109	108	108
Camera	343	338	338	309	308

TABLE IV: The number of PE instructions required for four Halide applications: Camera, Gaussian integer, Gaussian bfloat16, and Harris. The applications are compiled using both the hand-coded rewrite rules and the synthesized ones.

Benchmark	Synthesized	gcc -O0	gcc -O1
P1	3	16	3
P2	3	16	3
P3	3	16	3
P4	3	16	3
P5	3	16	3
P6	3	16	3
P7	5	19	4
P8	5	19	4
P9	4	20	4
P10	4	24	5
P11	4	22	4
P12	5	23	5
P13	5	22	4
P14	5	25	5
P15	5	25	5
P16	10	29	6
P17	6	23	5
P18	4*	36	7
P19	6	35	7
P20	9	35	8
P21	25	50	13
P22	26	39	11
P23	32	50	15
P24	18	50	12
P25	27	72	19

TABLE V: Number of RISC-V RV32IM instructions on 25 Hacker’s Delight programs (P1-P25). We show our system versus gcc with two levels of optimization. *The compilation of P18 to WebAssembly generated a `i32.popcnt` and hence could only be compiled to RV32IX.

generators [50], [30]. Specifically, we compile 25 Hacker’s Delight [56] programs. We use C implementations from Gulwani et al. [30].

We compile C to stack-machine WebAssembly byte code using Emscripten [58] (using `emcc -Os`). We then transform the resulting code into a basic block by abstract interpretation on a virtual stack [22], implemented with a modified WebAssembly interpreter.

We apply type legalization [36] to decompose `i32` constants into `i12` and `i20` constants. These bit-widths are chosen as they are the bit-width of immediate fields in the RISC-V ISA. Instruction selection is then applied using the synthesized rewrite rules. Next, we perform basic instruction scheduling and register allocation, and finally we assemble the instructions into RISC-V byte code [28], [14].

We compare the code we generate to that produced

by `gcc (riscv64-unknown-elf-gcc -march=rv32g -mabi=ilp32)`. The `gcc -O0` option uses the stack to store intermediates so our code size is better, while `gcc -O1` uses multiple basic blocks to decrease code size, which we do not support.

Table V shows a comparison of the number of instructions generated from our compiler versus a RISC-V `gcc` compiler for each Hacker's Delight program. For P21-P25, `gcc -O1` generates small code size by using branching code, an optimization we do not implement. P18 uses a `i32.popcnt` in the generated WebAssembly. When targeting RV32IX we can leverage the custom instructions to compile program P18 using only 4 instructions.

Error Correction Code Algorithm and Implementation Verification Using Symbolic Representations

Aarti Gupta
FVCTO¹
Intel Corporation
Santa Clara, CA, USA
aarti.gupta@intel.com

Roope Kaivola
Core and Client Dev. Group
Intel Corporation
Portland, OR, USA
roope.k.kaivola@intel.com

Mihir Parang Mehta
FVCTO¹
Intel Corporation
Santa Clara, CA, USA
mihir1.mehta@intel.com

Vaibhav Singh
FVCTO¹
Intel Corporation
Portland, OR, USA
vaibhav.singh@intel.com

Abstract—Error-correction codes (ECCs) are becoming a de rigueur feature in modern memory subsystems, as it becomes increasingly important to safeguard data against random bit corruption. ECC architecture constantly evolves towards designs that leverage complex mathematics to minimize check-bits and maximize the number of data bits protected, as a result of which subtle bugs may be introduced into the design. These algorithms traverse a vast data space and are subject to corner case bugs which are hard to catch through constraint-based randomized testing. This necessitates formal verification of ECC designs to assure correctness of the algorithm and its hardware implementation. In this paper we present a technique of representing various ECC algorithm outputs as Boolean equations in the form of Boolean Decision Diagrams (BDDs) to facilitate reasoning about the algorithms. We also discuss the counting and generation of examples from the BDD representations and how it aids in tuning ECC algorithms for performance and security. Additionally, we display the use of Symbolic Trajectory Evaluation (STE) to prove the correctness of register transfer level (RTL) implementations of these algorithms. We discuss the scaling up of this verification methodology, using different complexity and convergence techniques. We apply these techniques to a number of complex ECC designs at Intel and showcase their efficacy on several categories of bugs.

Index Terms—error correction codes, formal verification, symbolic simulation, binary decision diagrams

I. INTRODUCTION

With the ever-increasing capacity demands, memories are becoming denser and are more susceptible to soft errors. Error Correction Codes (ECCs) provide resiliency to the memory cell against errors due to cosmic rays, impurities during manufacturing, and other causes. Recent moves by chip manufacturers to extend ECC support to consumer processors, which was once limited to servers, emphasizes the universal necessity of ECCs. If the ECC fails, it will result in incorrect data getting read; in a safety-critical system, this can be catastrophic. ECC

¹ Formal Verification Central Technical Office

Intel provides these materials as-is, with no express or implied warranties. Intel processors might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. Intel and the Intel logo are trademarks of Intel Corporation. Other names and brands might be claimed as the property of others.

designs work by carefully adding data redundancy in the form of some check-bits to the data-stream while storing it. These check-bits and data-bits, which may have been corrupted during storage, are then used together to retrieve the original data. Though helpful in providing memory-protection, ECC designs are difficult to verify. ECC verification can be a challenge both for dynamic validation (DV) from the coverage perspective, and for formal verification (FV) from the convergence perspective. Consider the example of a Triple Error Correction Quadruple Error Detection (TECQED) design with 512 data-bits, 1-bit Metadata and 31 check-bits. Pre-silicon dynamic validation would require $4.87e163$ input patterns to fully validate the design, a nearly impossible task, and post-silicon issues are discovered very late in the design cycle, not providing enough time to determine a robust fix. Owing to the complex equations generally used in ECC logic, these designs are not tractable by different industry standard FV tools. Most commercial model-checking tools are better suited to solve control path challenges and falter in achieving convergence on big datapath designs. Commercial datapath FV tools tend to rely on structural similarities of the reference specification and the implementation. Such similarities are absent in the case of closed-box ECC verification, where the specification is just a property stating, “the resultant data equals the received data”.

This paper shows our results in verifying diverse ECC algorithms and designs, across a range of datacenter and consumer processors, using an Intel-internal datapath tool, Forte/rSTE [3], [12]. The complexity of these verification tasks varied from a 64-bit corruption on a Dynamic Random-Access Memory (DRAM) device in a memory controller to a 512b-sized TECQED ECC in a data cache. We analyze our results with respect to different verification parameters (complexity, coverage, runtimes etc.) and compare with commercial tools.

In the remainder of this paper, we briefly introduce error correction (section II), and the underlying proof methodology with the Forte tool (section III). We explain the verification setup, and the properties we prove (section IV) on ECCs. We evaluate the results of these verification activities (section V) and sum up our contributions (section VII).

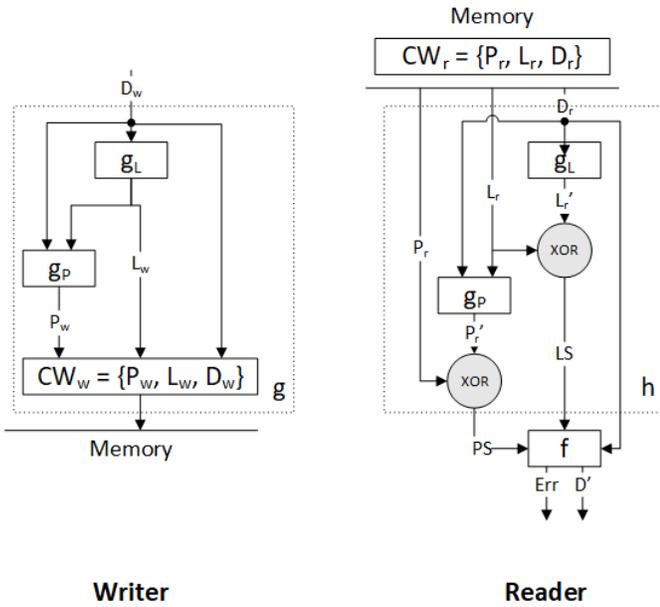


Fig. 1: ECC Writer and Reader

II. ERROR CORRECTION CODES

ECC functionality is usually implemented in hardware designs as two modules, a Writer and a Reader (Fig. 1). Using a generator function g , the Writer generates, from Data D_w , a codeword CW_w which consists of D_w appended with check-bits. There are two types of check-bits, locator bits L_w and parity bits P_w . Once CW_w is written to memory, it is subject to zero or more bits of corruption. Within the reader, the extractor function h computes the locator syndrome LS and parity syndrome PS . These syndromes are calculated by re-computing the locator bits L'_r and the parity bits P'_r and comparing them to their values P_r and L_r in the read codeword CW_r . Using these syndromes, the function f can determine the presence of the error and determine the location of the error, finally returning corrected data D' and error signal Err . Err can be:

- 1) No Error (NE): No corruption detected in CW_r .
- 2) Correctable Error (CE): Corruption detected in CW_r and fixed. Thus, output data $D' = D_w$.
- 3) Detectable but Uncorrectable Error (DUE): CW_r corruption detected; but correction outside algorithm capabilities. Thus, $D' \neq D_w$.

Reliability, Availability, and Serviceability (RAS), feature sets that are associated with system resiliency in the presence of hardware faults, impose requirements that vary across designs. For example, some SRAM (Static Random-Access Memory) cache designs may need protection from bit-flips that can randomly happen at any bit-position in the cache-line, while other designs, such as DRAM, may need protection on groups of neighboring bits, which we will refer to as bit-groups. RAS

requirements shape the choice of the ECC algorithm. These algorithms are based on the mathematical theories of Galois Extensions (Bose–Chaudhuri–Hocquenghem Codes) [4], [10], Lagrange Interpolation (Reed Solomon Codes) [15], and finite fields.

III. SYMBOLIC SIMULATION AND FORTE TOOLSET

Symbolic simulation extends standard digital circuit simulation with symbolic representations of values, covering behaviors of a circuit for all possible instantiations of the symbolic values in a single simulation. Used as a formal verification method, symbolic simulation is algorithmically simple and intuitive, which enables precise analysis and fine-grained mitigation of computational complexity, allowing the method to handle circuits that are above the capacity of standard formal model checking tools. Symbolic simulation excels in verification of deep targeted properties of fixed-length pipelines, in particular arithmetic and other datapath circuits. It has been the main vehicle for Intel arithmetic formal verification for over twenty years, and most arithmetic execution units of Intel processor designs have been exhaustively verified using it [3], [12]. It is the primary engine embedded in Intel's proprietary Forte/rSTE toolset. Symbolic simulation was first applied to ECC verification in 2005. Gradually, this application found its place in Server Memory Controller ECC (MC ECC) verification arsenal.

In a symbolic simulator the input stimulus may contain symbolic variables in addition to the concrete Boolean values 0, 1 and X. These symbolic variables are names of values, denoting sets of concrete values. The values of the internal signals computed in the simulation are then structural logical expressions on the symbolic variables on the inputs. For example, in a bit-level symbolic simulator, a single symbolic variable a corresponds to the set of Boolean values consisting of both 0 and 1, and if stimulus to a symbolic simulation trace contains the variables a , b , and c , the internal signals might carry values like $a \& b$ or $a + (b \& c)$. The symbolic expressions in a simulation are commonly encoded using Binary Decision Diagrams (BDDs) [5].

The limits of computational capacity are the limits between what can and cannot be verified in practice. When attempting to resolve a capacity challenge, the crucial difference between symbolic simulation and other formal verification methods is that in symbolic simulation a capacity problem is extremely concrete. It manifests itself as a symbolic expression (BDD) that is too large, associated with a particular node and time in the simulation. This concreteness allows a user to analyze, understand and resolve the problem with a greater degree of precision than other methods of verification. This amenability to precise performance analysis is a key differentiator enabling the success of symbolic simulation. Direct user-level access to BDDs also allows advanced complexity management techniques, such as parametric substitutions and symbolic indexing, as well as automated analysis of the logical contents of a computation, for example, counting the precise number of input vectors satisfying or violating a given property.

In the Forte/rSTE toolset the base symbolic simulator STE is embedded in a code layer called relational STE (rSTE) in the context of a full-fledged functional programming language. Common computational complexity reduction techniques, including weakening, parametric substitution, etc., are made easily accessible to the user through programmable options to the tool. The framework also provides sophisticated debug support, breakpoints, waveform and circuit visualization, etc., to enable users to quickly focus on usual verification problems. The full programmability of the tool allows users to write reusable verification recipes that automate and structure shared or repeated tasks.

An important aspect of the verification toolset is that it provides a general symbolic computation capacity for Booleans. Not only can circuits be simulated with symbolic values, but any user-written program operating on Boolean data can be symbolically computed. This feature is very useful for multiple purposes: *ad hoc* programmatic analysis of failures, breaking symbolic computations into parts to analyze complexity issues, and early algorithm experiments prior to the existence of hardware implementations of those algorithms.

IV. ECC FORMAL VERIFICATION

The verification setup for ECC FV involves connecting the Writer and the Reader, as shown in Fig. 3, abstracting out the storage component which usually sits in between these two blocks in real designs and replacing it with a corruption model. This model explicitly adds the effect of corruption on the codeword CW_w generated by the Writer before it is fed to the Reader.

In the setup described in Fig. 3, there are two inputs D_w and C . For symbolic analysis of the logic, we can assume these inputs to be symbolic variables instead of fixed stream of 0s and 1s, representing all values in the input space. Symbolic simulation then traverses the design, transforming input variables as BDDs in accordance with the design’s logic, and finally makes the transformed BDDs available at outputs D' and Err' . Correctness is then evaluated as a comparison between the output BDDs and the input BDDs under specific assumptions on the corruption.

For an ECC to guarantee correction of up to n bits/bit-groups and detection of up to $n + 1$ bits/bit-groups of corruption, the following must hold:

- Property 1: $(Countbits(C) = 0) \Rightarrow NE$ and $D' = D_w$
- Property 2: $0 < Countbits(C) \leq n \Rightarrow CE$ and $D' = D_w$
- Property 3: $(Countbits(C) = n + 1) \Rightarrow DUE$ and no guarantee on D'

If the number of corrupted bits/bit-groups exceeds $n + 1$, the algorithm makes no claims. For Single Error Correction Double Error Detection (SECDDED), $n = 1$; for Double Error Correction Triple Error Detection (DECTED); $n = 2$ and for TECQED $n = 3$. DRAM ECCs employ custom algorithms at the level of devices, groups of bits of size 32 or 64, on a DIMM (dual inline memory module). The levels of protection

provided by DRAM ECCs include full device protection, half device protection, and column protection.

In properties 1 to 3, it must be noted that the conditions NE, CE, and DUE are mutually exclusive and exhaustive. Different circuits implement this differently, but regardless it is necessary to prove mutual exclusiveness and exhaustivity. A circuit may encode 2 bits such that 00 is NE, 01 CE and 10 is DUE. In such a case we will need to show that 11 can not be computed. In other cases, each type of error is indicated by a separate signal, in which case we will need to show that these signals are muxed. Usually, though, circuits indicate whether data was corrected, or not, with just one signal. If this signal is 1, then it is DUE; if 0, it is CE or NE. We will need to show that none of these three conditions overlap.

A. ECC Implementation Verification

Using the symbolic simulator of Forte/rSTE toolset, the correctness of ECC designs can be ascertained without any reference to algorithms or design internals. This gives this technique a clear edge over other datapath FV tools which usually depend on a high-level model (HLM) against which an equivalence check is performed. Such HLMs are themselves prone to error and may incorporate an error which is also present in the design, in which circumstance a full equivalence check will nonetheless mask the bug. Moreover, such HLMs may need frequent remodeling in tandem with algorithm changes, which occur on a regular basis in the current landscape where ECC algorithms are continuously tuned in response to performance and security requirements.

To understand the nature of this verification process, let us take an example SECDDED design protecting 4 bits of data ($D[0]—D[3]$) using 4 check-bits. The corruption vector ($C[0]—C[7]$) represents corruption that can happen at any bit position of the 8 bit codeword (data and check-bits). After symbolic computation of BDDs at each relevant node and times of interest, the BDD at the output port ‘NE’, which indicates absence of corruption on read data, may look like the BDD in Fig. 2 (a). Importantly, this BDD only makes reference to corruption bits, although the symbolic simulation accounts for fully symbolic data bits. This suggests that the symbolic condition for ‘NE’ depends only on corruption bits and is independent of the data bits. It can also be noted that in this BDD there are several paths that lead to the terminal node ‘T’, while the naive expectation would be for a single path to reach this terminal i.e., the no-corruption path. This is due to the fact that ECC algorithms are constructed to guarantee error correction and detection up to a maximum bound of corruption, while the corruption vector that we considered allows corruption on every bit of codeword i.e., up to 8 bits of corruption. Therefore, to verify the algorithm’s properties, we must evaluate this BDD under the implication of the max-bound condition. Forte provides debug hooks that allows users to access the BDDs at different design nodes at various times, thus the ‘NE’ BDD can be extracted and evaluated for satisfiability using simple Forte commands when $Countbits(C) \leq 2$. Under this condition, property 1 is

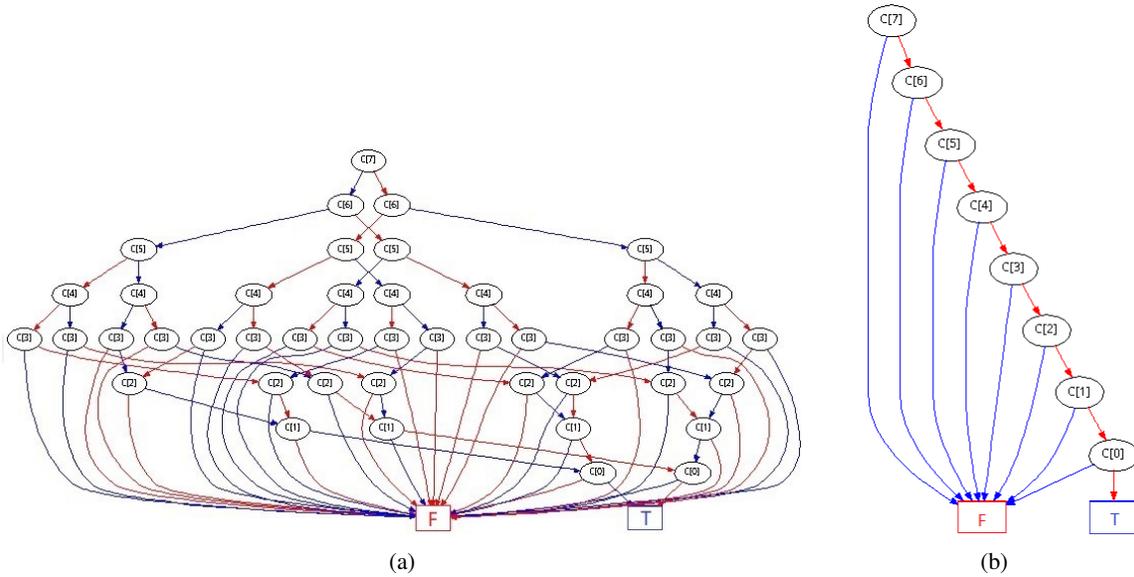


Fig. 2: BDD for NE in Example

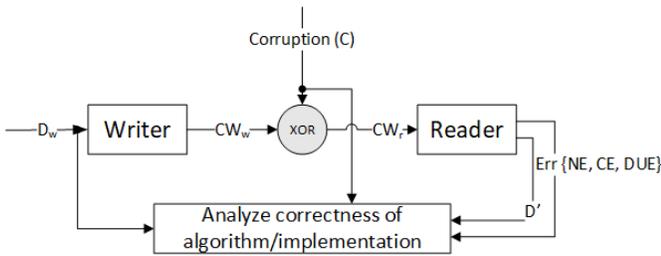


Fig. 3: ECC FV Setup

substantiated and the only satisfiable path is the one where $C[0]-C[7]$ are all false. Symbolic analysis can be done in a similar fashion on other properties.

We saw that a simple 4 bit SECDED could result in a 44-node BDD in Fig. 2 (a) for an error signal in the circuit. Computing and storing BDDs of this kind is a likely limiting factor as design complexity increases. By means of various techniques described below, we could limit the BDD sizes to smaller bounds and scale this technique to designs where commercial datapath tools failed to converge.

1) *Parametric Substitution*: In many circumstances, symbolically simulating for a subset of data, i.e., data under a specified condition, is more efficient than symbolically simulating with unconstrained data. In such circumstances, parametric substitution [2] is very effective. A generic correctness statement of a design can be represented as:

$$P(x) \rightarrow Q(x)$$

Where P is a constraint on the data space, x is a vector of BDD variables, and Q is a function that carries out symbolic simulation. Under parametric substitution, we use a

function $param$ to compute a parametrized functional vector representation of P and rewrite the correctness statement as:

$$Q(param(P(x)))$$

As an example, Fig. 2 (a) depicts the BDD for the No Error (NE) signal of the 4-bit SECDED design, when computed in a simulation with fully unconstrained values. This BDD captures the behavior of the design for any number of corruptions from zero to eight. However, the design is only expected to produce reasonable output when the number of corrupted bits is at most two, in other words when the condition $Countbits(C) \leq 2$ is true. We can compute a parametric substitution from this condition, and instead of simulating the system with fully unconstrained symbolic corruption bits, we can simulate it with small BDD's for the corruption bits, restricting the behavior only to the interesting cases. Conceptually, the parametric substitution produces BDD's for the corruption bits that allows the first two corruption bits to have any values, but any subsequent bits can only be high if at most one higher bit is already high. In the resulting simulation, the BDD for the No Error (NE) signal is as depicted in Fig. 2 (b), a considerable simplification when contrasted with the general case.

2) *Case-Splitting*: With case-splitting, we decompose the data space into a number of sets and separately verify the circuit for each set. This reduces the BDD complexity and search space for each case in a divide-and-conquer fashion. ECCs naturally lend themselves to a case-split on the number of bits of corruption that are allowed. For example, a SECDED design can be decomposed into 3 cases: no corruption, 1b corruption, and 2b corruption. Parametric substitution of the case constraint will lead to even smaller BDDs. In the case of the example illustrated in Fig. 2, it will lead to a zero-sized

BDD with only a terminal vertex “True” or “False” for the signal ‘NE’.

Further case-splitting can be done based on the locations of the (one or more) corruption bits. This has been essential in our verification of 512-bit TECQED designs, as it made convergence of the proof possible. In addition, case-splitting is useful towards reducing the runtimes of existing proofs by means of parallel processing.

3) *Symbolic Indexing*: Symbolic Indexing [1] is an efficient technique that can logarithmically scale down the number of variables a BDD is dependent on. Taking the example of 4-bit-SECDED, if we replace the 8-bit corruption vector (C[0]—C[7]) with two vectors (CI1[0]-CI1[2]) and (CI2[0]-CI2[2]), where the value CI1 gives the index of first bit that is corrupted and CI2 gives the index of second corrupted bit, then the same symbolic corruption information can be relayed to the simulator using 6 variables instead of original 8. Generally speaking, a symbolic corruption on an ECC design with codeword length n and up to k bits of corruption can be represented using $k \times \log_2(n)$ variables using symbolic indexing, which would otherwise require n variables. This state space reduction becomes all the more important as we move to larger designs such as 4096-bit-SECDED, where this technique allows use of two 13-bit corruption-index vectors instead of a 4110-bit corruption vector.

4) *Variable Ordering*: BDD size is very sensitive to its variable order [7]. Variable order of a BDD determines the order in which variables will appear for all its node-traversal paths. The optimal variable order is required to ease BDD computations on bigger circuits like memory controllers where one design may support multiple ECC schemes. In verification of such designs, it is advisable to put control variables before data variables. This is because the control variables may choose a completely different mode of operation in the circuit; and having them at the top of the BDD tree simplifies the branches by preventing a commingling of different ECC schemes. For example, variables on signals that select the ECC mode, or signals that are used for configuration settings such as error masking, should take precedence in ordering relative to variables for corruption and data.

5) *Dynamic Weakening*: Symbolic simulation on ECC designs may sometimes encounter a BDD blow-up. Forte assists in investigating and resolving such a bottleneck through dynamic weakening. The user can provide a maximum bound of BDD limit, and whenever BDD size at an internal node during the symbolic simulation exceeds the provided limit, tool automatically ‘weakens’ that node i.e., replaces that BDD with an ‘X’. This new value is then propagated through the circuit simulation. If the weakened node was irrelevant to the final output computation, then it saves unnecessary simulation on that path, else the X propagation reaches the output nodes. In these cases, the BDD representation at output node can be of form $BDD_A + X(BDD_B)$, where BDD_A represents

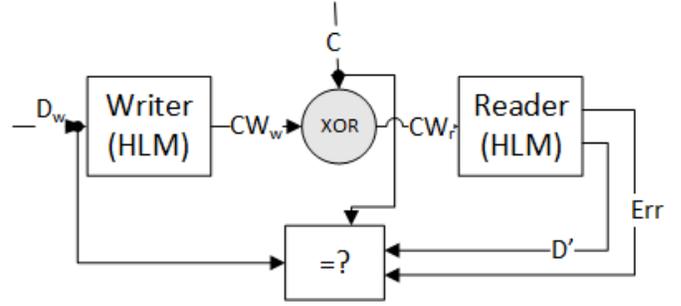


Fig. 4: Architectural ECC FV

the variable-assignments that give concrete values 1 and 0 to the output and BDD_B represents the variable assignments that can lead to X. Forte’s schematic viewer enables chasing this X and determining the cause of the divergence. The tool also facilitates substitution of variables with random example values. This makes sample cases more concrete and easier to debug.

B. ECC Architectural Verification

Forte can also be used to check algorithm architecture, in addition to its use in closed-box verification of design properties. This mode, however, does need algorithm understanding and modeling the Writer and Reader parts of algorithms as HLMs, but the goal remains the same i.e., checking overall correctness of algorithm by means of property checking. This is done by using a verification setup similar to the design verification, only replacing the Writer and Reader design blocks, as shown in Fig. 4, with their HLMs written in Forte’s functional language *reFlect* [9]. Verification tasks of this nature, instead of using the symbolic simulation capability of Forte, use its symbolic computation feature. In a manner akin to abstract interpretation [6], the input variables are propagated through the logical functions present in the HLM, undergoing BDD transformations at each function. Finally, BDDs are derived at the output of the HLM, which can then be used for reasoning about the correction and detection properties of the ECC algorithm. This architectural verification is independent of the design, and in practice it is often carried out before the algorithm is implemented in RTL. This shortens the feedback loop of design and verification, thus reducing time to market for such designs.

C. Counting and Enumerating Error Patterns

In modern server designs, some algorithms provide protection of a bit-group within specific published bounds. Design pressures to add metadata bits to the bit group lead to customizations which reduce the number of check-bits and result in such a lower bound being chosen over a guarantee of full correction. For example, a customization to include directory bits, poison bits, and tag bits (i.e., metadata) may lead to an algorithm which claims, “100% detection, and better than 99.999% correction.” This claim implies that fewer than

0.001% of all possible block corruption patterns can lead to a DUE. This performance-accuracy tradeoff makes verification of this claim complex. In contrast to the properties explained earlier in this section, which were of the nature “under the given conditions, BDDs on the outputs that indicate error must evaluate to True or False”, our claim now involves exact counting of the paths that lead to the terminal vertices. Additionally, an algorithm may make a conditional claim such as “Errors that fall on both right and left half of device are outside scope of ECC and are not corrected but detected for ~99.999% of error patterns.” Such a claim, in general, relates several design-outputs under a specific corruption condition. This claim bounds the number of memory failures that can go undetected, also known as SDCs (Silent Data Corruptions). To verify this, we need to count all corruption variable assignments under which output BDD for DUE error signal evaluates to False, but output data D' is not equal to write data D_w . Thus, the property to be checked becomes:

$$\text{satCount}(\text{Cond} \rightarrow \neg \text{DUE} \ \& \ (D' \neq D_w)) < x$$

Here, Cond is the corruption condition under which counting is performed, x is an upper bound on the number of expected SDCs, and satCount is a count of the number of satisfying assignments to a given formula.

The corruption condition and the DUE/SDC conditions can be composed together to form a new BDD, and we can count the number of satisfying instances through procedures written in *reFLECT*. We are also able to enumerate the corruption patterns that lead to SDC or DUE in addition to counting them. This data is sometimes needed by memory vendors and is also helpful during debugging to understand the frequency/location of failures.

One consideration while generating these counts is the avoidance of duplicates, which we illustrate for the example of a SECDDED algorithm. To count the SDC cases for 3 bit corruptions, we define symbolic indices $p1$, $p2$ and $p3$. Once we compute the SDC condition, there could be cases that are counted multiple times, such as $p1 = 0, p2 = 1, p3 = 2$ and $p1 = 0, p2 = 2, p3 = 1$. However, by assuming without loss of generality that $p1 > p2 > p3$ in the condition in the above expression, the counting of duplicate cases is avoided.

V. RESULTS

We discuss the impact seen from this verification effort on ECC designs of varying complexity. In the past 2 years, we have verified 14 ECC designs and their corresponding algorithms, resulting in the discovery of 48 bugs overall and proving the absence of bugs in customer releases. These ECCs are the state of the art for commercial designs. They represent a full range of Intel designs and were not cherry-picked for the case study.

Quantitatively, Table I lays out the results of ECC FV spanning multiple projects and design generations. Table I

compares ECC property checking using Forte against established industrial EDA (Electronics Design Automation) tools tuned for control-path and data-path FV. Since our BDD-based technique with Forte allows us to do a closed-box checking without reference to design internals, we explored the feasibility of similar testing with the EDA tools for a fair comparison. Tool #1 and Tool #2 in Table I can use the same verification setup as shown in Fig. 3 and allow the user to state the design properties by means of System Verilog Assertions (SVA). Both these tools use various engines that can run in parallel to achieve a concrete result and may give a bounded proof in case if they fail to converge. As seen from Table I, these tools are able to converge on small-sized designs based on simple ECC algorithms such as SECDDED, but as the design size or algorithm complexity increases, convergence is not seen. Our techniques, however, achieve convergence in a matter of minutes in all of the designs under consideration. Tool #2 is more tuned towards datapath verification, but no difference was observed between Tool #1 and Tool #2 with respect to convergence on these tasks. Typically, datapath FV commercial tools do better on arithmetic designs than standard model checkers due to their word-level engines. However, the arithmetic in ECC algorithms is primarily bit-level and, as seen from our results, word-level processing was not particularly useful here.

The size of ECC designs ranged from 3K gates (smallest) to over a million gates (largest). However, more than the design size the proof convergence depended on arithmetic complexity of the algorithm itself. For example, algorithm offering bit protection were more amenable to FV proofs compared to algorithms doing bit-group level protection. Also, the complexity increased as the number of bits under protection umbrella grew. For instance, the number of case-splits required to achieve proof convergence were 17K for a 512 bit TECQED and only 300 for a DECTED design of the same data-width, while none of the SECDDED designs verified needed a case-split. Within the same algorithm category, the complexity was directly proportional to the data-size. So, a 32 bit SECDDED is much easier to verify compared to a 4096 bit SECDDED.

Qualitatively, we consider it instructive to categorize the kinds of bugs we have found. This analysis is intended to help both design experts and verification experts identify common patterns that lead to design errors.

A. Architectural Bugs

Architectural FV allows early bug investigation, even before the implementation of an algorithm in RTL. As a result, bugs found in this process are prevented from ever entering the RTL design. This is a worthwhile exercise since the algorithms themselves are complex enough, owing to the interplay between different architectural features, to give rise to corner case bugs. For example, our recent investigation of single block corruption in a new ECC scheme in a memory controller found exactly 3 failure cases out of 18×2^{32} . Previously, some of our FV investigations have found corner case bugs

Algorithm	Protection level	Data width in bits	Engineering effort in person-days	Property Convergence		
				Forte	EDA tool #1	EDA tool #2
SECCDED	Bit	1-256	< 2	Yes	Yes	Yes
	Bit	4096	< 2	Yes	No	No
DECCDED	Bit	256	< 4	Yes	No	No
	Bit	512	< 4	Yes	No	No
TECCQED	Bit	512	< 15	Yes	No	No
Custom ECC schemes for DRAM device protection	Bit groups (16/32/64 bits)	512	Continuous engagement across design cycle	Yes	No	No

TABLE I: Comparison of Property Checking with Different Formal Tools. EDA Tool #1 is a Model Checking Tool and EDA Tool #2 is a Commercial Datapath FV Tool

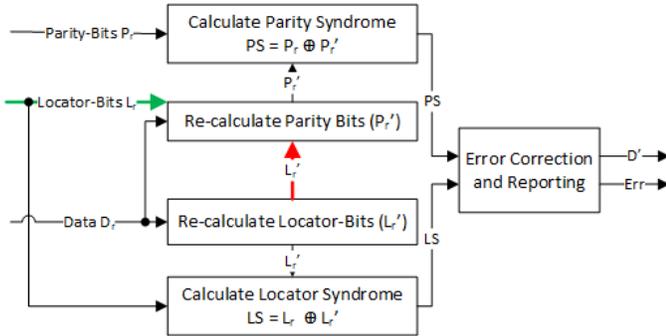


Fig. 5: Implementation Error Example

that escaped testing and subsequently led to the publication of customer errata [11].

B. Implementation Errors

Even with a correct algorithm, an implementation can be erroneous, due to a variety of reasons such as specification ambiguity. We encountered one such bug while reading parity from memory; while the architecture specified a column major order read, the RTL implementation was row major. In another example, a simple misconnection led to a breakdown of ECC functionality. This case is illustrated in Fig. 5 which shows the functionality of a generic ECC Reader. The Reader reads the codeword from memory which is comprised of Data D_r and check-bits (i.e., locator bits L_r and Parity bits P_r). The Reader uses the read data D_r and the Locator bits L_r to re-calculate the new check-bits (L'_r and P'_r). These recalculated values are then compared against the check-bits that were read from memory to compute syndromes that are then used to ascertain error presence and its correction. However, in the case presented in Fig. 5, instead of using original locator bit L_r , (green arrow indicated in Fig. 5) the recalculated version of L'_r was used (red arrow in Fig. 5) to re-compute Parity bits. Due to this seemingly innocuous issue, 60% of 1b corruption cases that specification claimed to be correctable were marked uncorrectable in the design, and around 25% of 2b corruption cases led to fatal SDCs. The timely verification of these designs prevented these critical bugs from making their way into the final products.

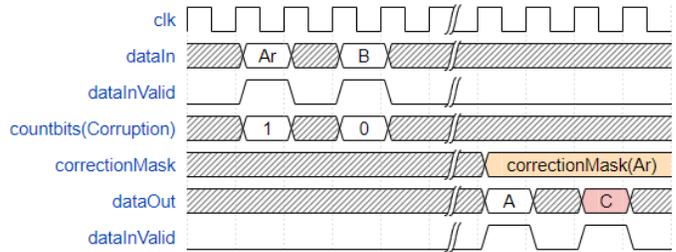


Fig. 6: Pipeline Bug Example

C. Pipeline Bugs

Frequently, bugs arise from pipelines where a signal was used at the wrong stage, or an incorrect clock-enable prevented the relevant values from propagating. One such failure is described in Fig. 6. Here 2 sets of data (A_r and B) enter the Reader in succession, where A_r has 1b corruption, and B is not corrupted. The design was expected to correct the corrupted A_r to its original value A and to leave B unchanged. However, B was changed. It was found that an internal signal, $correctionMask$, used for fixing the corruption was not updated while processing B due to an incorrect clock-enable, and its stale value resulted in a spurious correction. This behavior continued for a long time in the pipeline, until the next update of the clock enable signal. This shows that an algorithm, however carefully designed, can be rendered ineffective for a large number of corruption cases due to pipeline bugs. The fixing of this bug also shows the salutary effect of datapath FV on the surrounding control-path logic, as the closed-box verification approach focuses on the overall functioning of the design in addition to the correctness of the ECC algorithm.

D. Specification Bugs

The RAS capabilities of an ECC design need to be clearly documented for customers in an External Design Specification document. Thus, these specifications need to be accurate and must reflect exact ECC capabilities that exist in the silicon product. Many of the complex algorithms may not provide 100% correction on a block, but nonetheless specify $x\%$ correction, $y\%$ detection, and $z\%$ silent data corruption. These data percentages are critical to memory vendors and need to be verified, but this verification is complex as it is not a simple true or false claim but involves exact counting of each category

of results. Since the number of satisfying assignments can be counted using symbolic representations, it can be verified that both the ECC algorithms and their implementations deliver the claims that they make in the specification. We helped in fixing some of these results based on our calculations. In one such case, an anomaly was detected on the number of DUE counts where the actual counts offered by algorithm differed by the published claims by just $7.10e-13\%$.

E. Miscellaneous Bugs

Since we analyze each ECC design in depth, we sometimes encounter issues such as efficiency bugs, where the design uses more check-bits than required by the algorithm, or parametrization bugs, where some design parameters are not passed-down correctly in the design.

VI. RELATED WORK

Model-checking based FV techniques have been used for verifying ECC designs. For example, a 128-bit TECQED ECC was formally verified in [13], and a 256-bit Double Error Correction Triple Error Detect (DECTED) ECC design was formally verified in [8] using a commercial model checker. Both these proofs converged only after a lot of design interventions and rewriting the design to make the logic fully combinational. These interventions need special handling, and one needs to make sure the bridges between these abstract models are verified, maintaining overall coherence. In contrast, our approach does not need any reduction or abstraction of designs. Scaling up these approaches [8], [13] to bigger ECC designs will be difficult as model-checking tools get fatigued due to the inherent complexity of ECC designs and the vast input space. In [13], extreme convergence steps were taken to conclude the proof on a 128-bit TECQED with a proof runtime that is counted in days, while with our technique we could verify a $4\times$ data-width design (512-bit TECQED) in just 2 hours.

Lvov et al. [14] verified Reed-Solomon codes by computing Grobner bases, using the SINGULAR arithmetic engine. Their proofs are independent of data width and their runtimes are dependent only on the number of bits corrupted. However, their assumption of the insufficiency of BDD-based techniques for ECC verification has not been borne out in Forte, which is capable of crunching through Boolean equations of the required size. This is accomplished through variable ordering and parametric substitution techniques, as discussed further in section III. As a result, ECC verification in Forte becomes a much simpler matter of declarative specification of the desired ECC properties, without reference to the underlying algebraic structure.

VII. CONCLUSION

The results discussed in this paper show the efficacy of our BDD-based symbolic representation in verifying properties of ECC designs at both the algorithmic and RTL level, finding bugs which would have been infeasible to find through testing. These techniques are scalable to large ECCs by means

of parametric substitution and other complexity management techniques. The success of these techniques in discovering bugs on industrial designs allows the categorization of the most common kinds of ECC bugs, which in turn shapes the practice of ECC design towards avoiding these bugs from the very beginning.

These techniques are valuable because they allow for a closed-box approach that requires neither knowledge of the design nor an HLM for equivalence checking. Additionally, these Forte techniques outperform other closed-box tools. Forte differentiates itself here by allowing algorithmic verification, even in advance of the RTL being written, and by helping provide bounds on the incidence of certain kinds of errors. By facilitating efficient correctness proofs and supporting the development and tuning of ECC designs on multiple fronts, Forte-based ECC verification techniques position themselves to be useful well into the future.

ACKNOWLEDGMENT

Formal verification of Error Correction Codes in the paradigm discussed in the current paper has been practiced at Intel since 2005. We would like to express our gratitude to all of our former colleagues who have contributed to this effort either conceptually or through code. In particular, we would like to thank Scott Huddleston for his seminal work on the error probability counting methods discussed in section IV-C, and Levent Erkok, Flemming Andersen, John Matthews and John Erickson for advancing the methodology over a series of verification efforts on successive families of memory controllers. We also thank Jing Ling, Hsing-min Chen, Wei Wu, and Saurabh Kolambkar for architecture and design help on various ECC circuits. We thank Disha Puri for carrying out comparison experiments on commercial datapath formal verification tools. Finally, we would like to thank Gavriel Gavriellov and Achutha Kirankumar V. M. for the opportunity to carry out this work.

REFERENCES

- [1] S. Adams, M. Bjork, T. Melham, and C. H. Seger, "Automatic abstraction in symbolic trajectory evaluation," *Formal Methods in Computer Aided Design 2007*.
- [2] M. D. Aagaard, R. B. Jones, and C. H. Seger, "Formal verification using parametric representations of Boolean constraints," *Proceedings of the 36th annual ACM/IEEE Design Automation Conference 1999*.
- [3] Achutha Kirankumar V. M., A. Gupta, and R. Ghughal, "Symbolic Trajectory Evaluation. The Primary Validation Vehicle for Next Gen Intel® Processor Graphics FPU," *Formal Methods in Computer Aided Design 2012*.
- [4] R. C. Bose and D. K. Ray-Chaudhuri, "On A Class of Error Correcting Binary Group Codes," *Information and Control 1960*.
- [5] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, 100.8 (1986): 677-691.
- [6] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, January 1977. ACM Press. pp. 238-252.
- [7] D. Deharbe and J. Vidal, "Optimizing BDD-based verification analysing variable dependencies," *In XIV Symposium on Integrated Circuits and System Design (SBCCI'01)*, pp. 64-69. Computer Society Press, 2001.

- [8] K. Devarajegowda, V. Hiltl, T. Rabenalt, D. Stoffel, W. Kunz, and W. Ecker, "Formal Verification by The Book: Error Detection and Correction Codes," DVCon 2020.
- [9] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving", *Journal of Functional Programming*, 16(2):157-196, March 2006.
- [10] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres* 1959.
- [11] Intel Corporation, "Third Gen Intel® Xeon® Scalable Processors Specification Update", May 2022, <https://www.intel.com/content/www/us/en/design/resource-design-center.html>, Document ID 637780, Erratum ID ICX 66.
- [12] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber and A. Naik, "Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation," *Computer Aided Verification* 2009.
- [13] A. Kumar and K. Devarajegowda, "Verifying ECCs Used in Safety Critical Designs with Formal," Jasper User Group 2021.
- [14] A. Lvov, L. A. Lastras-Montano, V. Paruthi, R. Shadowen, and A. El-Zein, "Formal verification of error correcting circuits using computational algebraic geometry," *Formal Methods in Computer Aided Design* 2012.
- [15] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics* 1960.

First-Order Subsumption via SAT Solving

Jakob Rath* , Armin Biere† , Laura Kovács* 

*TU Wien, Vienna, Austria

{jakob.rath, laura.kovacs}@tuwien.ac.at

†University of Freiburg, Freiburg im Breisgau, Germany

biere@cs.uni-freiburg.de

Abstract—Automated reasoners, such as SAT/SMT solvers and first-order provers, are becoming the backbones of applications of formal methods, for example in automating deductive verification, program synthesis, and security analysis. Automation in these formal methods domains crucially depends on the efficiency of the underlying reasoners towards finding proofs and/or counterexamples of the task to be enforced. In order to gain efficiency, automated reasoners use dedicated proof rules to keep proof search tractable. To this end, subsumption is one of the most important proof rules used by automated reasoners, ranging from SAT solvers to first-order theorem provers and beyond. It is common that millions of subsumption checks are performed during proof search, necessitating efficient implementations. However, in contrast to propositional subsumption as used by SAT solvers and implemented using sophisticated polynomial algorithms, first-order subsumption in first-order theorem provers involves NP-complete search queries, turning the efficient use of first-order subsumption into a huge practical burden. In this paper we argue that integration of a dedicated SAT solver provides a remedy towards efficient implementation of first-order subsumption and related rules, and thus further increasing scalability of first-order theorem proving towards applications of formal methods. Our experimental results demonstrate that, by using a tailored SAT solver within first-order reasoning, we gain a large speed-up in state-of-the-art benchmarks.

Index Terms—first-order subsumption, multi-literal matching, automated theorem proving, satisfiability checking

I. INTRODUCTION

Most formal verification approaches use automated reasoners in their backend to, for example, discharge verification conditions [22], [10], [15], produce/block counter-examples [20], [29], [1], or enforce security and privacy properties [30], [25], [4], [32]. All these approaches crucially depend on the efficiency of the underlying reasoning procedures, ranging from SAT/SMT solving [6], [12], [3] to first-order proving [41], [21], [34], [11]. In this paper we focus on automated first-order theorem proving with the aim of improving efficiency towards proving first-order (program) properties.

The leading concept behind the proof-search algorithms used by state-of-the-art first-order theorem provers is saturation [34], [21]. While the concept of saturation is relatively unknown outside of the theorem proving community, similar algorithms that are used in other areas, such as Gröbner basis computation [9], can be considered examples of saturation algorithms. The key idea behind saturation-based proof search is to reduce the problem of proving validity of a first-order

formula A to the problem of establishing unsatisfiability of $\neg A$ by using a sound inference system, most commonly using the superposition inference system [28]. That is, instead of proving A , we refute $\neg A$, by selecting and applying inferences from the superposition calculus. In this paper, we focus on saturation algorithms using the superposition calculus.

Saturation with Redundancy. During saturation, the first-order prover keeps a set of *usable clauses* C_1, \dots, C_k , with $k \geq 0$. This is the set of clauses that the prover considers as possible premises for inferences. After applying an inference with one or more usable clauses as premises, the consequence C_{k+1} is added to the set of usable clauses. The number of usable clauses is an important factor for the efficiency of proof search. A naive saturation algorithm that keeps all derived clauses in the usable set would not scale in practice. One reason is that first-order formulas in general yield infinitely many consequences. For example, consider the clause

$$\neg \text{positive}(x) \vee \text{positive}(\text{reverse}(x)), \quad (1)$$

where x is a universally quantified variable ranging over the algebraic datatype `list`, where `list` elements are integers; *positive* is a unary predicate over `list` such that *positive*(x) is valid iff all elements of x are non-negative integers; and *reverse* is a unary function symbol reversing a list. As such, clause (1) asserts that the reverse of a list x of non-negative integers is also a list of non-negative integers (which is clearly valid). Note that, when having clause (1) as a usable clause during proof search, the clause $\neg \text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ can be derived for any $n \geq 1$ from clause (1). Adding $\neg \text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ to the set of usable clauses would however blow up the search space unnecessarily. This is because $\neg \text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ is a logical consequence of clause (1), and hence, if a formula A can be proved using $\neg \text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$, then A is also provable using clause (1). Yet, storing $\neg \text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ as usable formulas is highly inefficient as n can be arbitrarily large.

To avoid such and similar cases of unnecessarily increasing the set of usable formulas during proof search, first-order theorem provers implement the notion of *redundancy* [31], by extending the standard superposition calculus with term/clause ordering and literal selection functions. These orderings and selection functions are used to eliminate so-called redundant

clauses from the search space, where redundant clauses are logical consequences of smaller clauses w.r.t. the considered ordering. In our example above, the clause $\neg\text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ would be a redundant clause as it is a logical consequence of clause (1), with clause (1) being smaller (i.e., using fewer symbols) than $\neg\text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$. As such, if clause (1) is already a usable clause, saturation algorithms implementing redundancy should ideally not store $\neg\text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ as usable clauses. To detect and reason about redundant clauses, saturation algorithms with redundancy extend the superposition inference system with so-called *simplification rules*. Simplification rules do not add new formulas to the set of (usable) clauses in the search space, but instead simplify and/or delete redundant formulas from the search space, without destroying the refutational completeness of superposition: if a formula A is valid, then $\neg A$ can be refuted using the superposition calculus extended with simplification rules. In our example above, this means that if $\neg A$ can be refuted using $\neg\text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$, then $\neg A$ can be refuted in the superposition calculus extended with simplification rules, without using $\neg\text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ but using clause (1) instead.

Ensuring that simplification rules are applied efficiently for eliminating redundant clauses is, however, not trivial. In this paper, we show that *SAT-based approaches can be used to identify the application of simplification rules during saturation, improving thus the efficiency of saturation algorithms implementing the superposition calculus extended with simplification rules*, as discussed next.

Subsumption for Effective Saturation. While redundancy is a powerful criterion for keeping the set of clauses used in proof search as small as possible, establishing whether an arbitrary first-order formula is redundant is as hard as proving whether it is valid. For example, in order to derive that $\neg\text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ is redundant in our example above, the prover should establish (among other conditions) that it is a logical consequence of (1), which essentially requires proving based on superposition. To reduce the burden of proving redundancy, first-order provers implement sufficient conditions towards deriving redundancy, so that these conditions can be efficiently checked (ideally using only syntactic arguments, and no proofs). One such condition comes with the notion of *subsumption*, yielding one of the most impactful simplification rules in superposition-based theorem proving [2].

The intuition behind subsumption is that a (potentially large) instance of a clause C does not convey any additional information over C , and thus it should be avoided to have both C and its instance in the set of usable clauses; to this end, we say that the instance of C is subsumed by C . More formally, a clause C subsumes another clause D if there is a substitution σ such that $\sigma(C)$ is a submultiset of D ¹. In such a case, subsumption removes the subsumed clause D from the clause set. To continue our example above, a unit clause

$\text{positive}(\text{reverse}^m(x))$, with $m \geq 1$, would prevent us from deriving $\neg\text{positive}(x) \vee \text{positive}(\text{reverse}^n(x))$ for any $n \geq m$, and hence eliminate an infinite branch of clause derivations from the search space.

To detect possible inferences of subsumption and related rules, state-of-the-art provers use a two-step approach [35]: (i) retrieve a small set of candidate clauses, using literal filtering methods, and then (ii) check whether any of the candidate clauses represents an actual instance of the rule. Step (i) has been well-researched over the years, leading to highly efficient indexing solutions [27], [33], [35]. Interestingly, step (ii) has not received much attention, even though it is known that checking subsumption relations between multi-literal clauses is an NP-complete problem [19]. Although indexing in step (i) allows the first-order prover to skip step (ii) in many cases, the application of (ii) in the remaining cases may remain problematic (due to NP-hardness). For example, while profiling subsumption in the world-leading theorem prover VAMPIRE [21], we observed subsumption applications, and in particular calls to the literal-matching algorithm of step (ii), that consume more than 20 seconds of running time. Given that millions of such matchings are performed during a typical first-order proof attempt, we consider such cases highly inefficient, calling for improved solutions towards step (ii). In this paper we address this demand and show that *a tailored SAT-based encoding can significantly improve the literal matching, and thus subsumption, in first-order theorem proving*.

Our Contributions. In this paper, we bring the following main contributions.

(1) We propose a *SAT-based encoding for capturing potential applications of subsumption* in first-order theorem proving (Section III). A solution to our SAT-based encoding gives a concrete application of subsumption, allowing the first-order prover to apply that instance of subsumption as a simplification rule during saturation. Our encoding uses so-called substitution constraints to formalize matching of literals within the premises (i.e., subset relation among literals of premises). Our encoding can be extended to other simplification rules, in particular when applying simplifications using the combination of subsumption with binary resolution (i.e., subsumption resolution).

(2) We introduce a *lean SAT solving approach tailored to substitution constraints*, by adjusting unit propagation and conflict resolution towards efficient handling of such constraints. (Section IV). We introduce a tailored encoding of substitution constraints in SAT solving, advocating the direct use of our SAT solver for deciding application of subsumption within first-order proving.

(3) We implemented our SAT-based subsumption approach as a new SAT solver in the VAMPIRE theorem prover (Section V). We empirically evaluate our approach on the standard benchmark library TPTP (Section VI). Our experiments demonstrate that using SAT solving for deciding and applying subsumption brings clear improvements in the saturation process of first-order proving, for example improving the (time) performance of the prover by a factor of 2.

¹we consider a clause C as a multiset of its literals

II. PRELIMINARIES

Let \mathcal{V} denote a countably infinite set of *first-order* variables. We consider standard multi-sorted first-order logic with variables \mathcal{V} , and support all standard boolean connectives (see later) and quantifiers in the language. Throughout the paper, we write x, y, z for *first-order* variables, c, d for constants, f, g for function symbols, and p, q for predicates. The set of first-order terms \mathcal{T} consists of variables, constants, and function symbols applied to other terms; we denote terms by t . First-order *atoms*, or simply just atoms, are predicates applied to terms. Atoms and negated atoms are also called first-order *literals*, and denoted by L, M . First-order *clauses*, or simply just clauses, are disjunctions of literals, denoted by C, D . All our notation throughout this paper may possibly use indices. A clause that consists of a single literal is called a *unit clause*. Clauses are often viewed as multisets of literals; that is, a clause C being $L_1 \vee L_2 \vee \dots \vee L_n$ is considered to be the multiset $\{L_1, L_2, \dots, L_n\}$. For example, the clause $p \vee \neg q \vee p$ is the multiset $\{p, \neg q, p\}$.

An expression E is a term, literal, or clause. We denote the set of variables occurring in the expression E by $\mathcal{V}(E)$. A *substitution* is a function $\sigma: \mathcal{V} \rightarrow \mathcal{T}$ such that $\sigma(x) \neq x$ only for finitely many $x \in \mathcal{V}$. The function σ is extended to arbitrary expressions E by simultaneously replacing each variable x in E by $\sigma(x)$. We say an expression E_1 can be matched to expression E_2 if there exists a substitution σ such that $\sigma(E_1) = E_2$.

Saturation and Subsumption. Most first-order theorem provers, see e.g. [41], [21], [34], implement saturation with redundancy, using the superposition calculus [2]. A clause C subsumes a clause D iff there exists a substitution σ such that $\sigma(C) \subseteq D$, where C and D are treated as multisets of literals. *Subsumption* is a simplification rule that deletes subsumed clauses from the search space during saturation. Subsumption gives a powerful basis for other simplification rules. For example, subsumption resolution [21], [34], also known as contextual literal cutting or self-subsuming resolution, is the combination of subsumption with binary resolution; and subsumption demodulation [16] results from combining subsumption with demodulation/rewriting.

SAT Solving. Let \mathcal{B} be a countably infinite set of *boolean* variables. We denote boolean variables by b , possibly with indices. We use the standard boolean connectives $\wedge, \vee, \rightarrow, \neg$, and write \top for the boolean constant *true* as well as \perp for the boolean constant *false*. A boolean *literal*, denoted l , is a variable b or its negation $\neg b$. A boolean *clause* is a disjunction of literals. As before, we drop the qualifier *boolean* when it is clear from the context.

Modern SAT solvers are based on conflict-driven clause learning (CDCL) [24], with the core procedures *decide*, *unit-propagate*, and *resolve-conflict*. The solver maintains a partial assignment of truth values to the boolean variables. Unit propagation (also called boolean constraint propagation), that is *unit-propagate* in a SAT solver, propagates clauses w.r.t. the partial assignment. If exactly one literal l in a clause remains

unassigned in the current assignment while all other literals are false, the solver sets l to true to avoid a conflict. The two-watched-literals scheme [26] is the standard approach for efficient implementation of unit propagation.

If no propagation is possible, the solver may choose a currently unassigned variable b and set it to true or false; hence, *decide* in SAT solving. The number of variables in the current assignment that have been assigned by decision is called the *decision level*.

If all literals in a clause are false in the current assignment, the solver enters conflict resolution, via the *resolve-conflict* block of SAT solving. If the current decision level is 0, the conflict follows unconditionally from the input clauses and the solver returns “unsatisfiable” (UNSAT). Otherwise, by analyzing how the literals in the conflicting clause have been assigned, the solver may derive and learn a conflict lemma, undo some decisions, and continue solving.

III. SUBSTITUTION CONSTRAINTS AND SUBSUMPTION

Recall that a first-order clause C subsumes a clause D iff there exists a substitution σ such that $\sigma(C) \subseteq D$, where \subseteq is to be understood as multiset inclusion. In what follows, we refer by *clausal subsumption* between C and D to the case when clause C subsumes clause D . Similarly, *literal subsumption* between L and M refers to the case when literal L subsumes literal M . We note that deciding literal subsumption, that is whether a literal L subsumes a literal M , can be done in almost linear time, by constructing a substitution (if it exists) σ s.t. $\sigma(L) = M$; in this case, the value of $\sigma(x)$ is uniquely determined by L and M for each variable x occurring in L . However, when working with arbitrary, and not necessarily unit, clauses C, D , deciding clausal subsumption between C, D is NP-complete for the following reason: for each literal L_i of C , one of the literals M_{j_i} of D needs to be chosen in such a way that a substitution σ *simultaneously matches* each L_i with its respective M_{j_i} ; that is, $\sigma(L_i) = M_{j_i}$ for all i . Towards addressing NP-completeness of clausal subsumption, in this section we introduce *substitution constraints* (Section III-A), allowing us to formulate clausal subsumption as a SAT problem over substitution constraints (Section III-B). Based on this SAT-encoding of subsumption, we further present an effective approach towards using subsumption in saturation in Section IV.

A. Substitution Constraints

We first introduce *substitution constraints* to be further used in deciding clausal subsumption.

Definition 1 (Substitution Constraints): A *substitution constraint* Γ is a partial function from \mathcal{V} to \mathcal{T} , denoted as

$$(x_1, \dots, x_k) \triangleright (t_1, \dots, t_k),$$

where $k \geq 0$, $x_i \in \mathcal{V}$ are pairwise different, and $t_i \in \mathcal{T}$. The set $\text{dom}(\Gamma) := \{x_1, \dots, x_k\}$ is called the *domain* of Γ . We further write $\Gamma(x_i) = t_i$ for $i \in \{1, \dots, k\}$.

A substitution $\sigma: \mathcal{V} \rightarrow \mathcal{T}$ *satisfies the substitution constraint* Γ , written $\sigma \models \Gamma$, iff $\sigma(x_i) = t_i$ for all $i \in \{1, \dots, k\}$.

Two substitution constraints Γ_1, Γ_2 are *compatible* if there exists a substitution σ that satisfies both Γ_1 and Γ_2 , that is, if $\Gamma_1(x) = \Gamma_2(x)$ for all variables $x \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2)$.

As already discussed, literal subsumption between two literals L and M can easily be determined (as there is only one literal, i.e. L , that needs to be matched, i.e. to M). The substitution constraint corresponding to the literal subsumption between L and M is denoted by $\Gamma(L, M)$ and is defined below.

Definition 2 (Substitution Constraints for Literals): Let L and M be two literals. If there exists a substitution σ such that $\sigma(L) = M$, the *substitution constraint* $\Gamma(L, M)$ for literals L and M is

$$\Gamma(L, M) := (x_1, \dots, x_k) \triangleright (t_1, \dots, t_k),$$

where $\mathcal{V}(L) = \{x_1, \dots, x_k\}$ and $\sigma(x_i) = t_i$ for all $i \in \{1, \dots, k\}$. Otherwise, L cannot be matched to M and the *substitution constraint* $\Gamma(L, M)$ for literals L and M is

$$\Gamma(L, M) := \perp.$$

Example 1: Consider the following first-order literals:

$$\begin{aligned} L_1 &= p(x_1, x_2, x_3) & L_2 &= p(f(x_2), x_4, x_4) \\ M_1 &= p(f(c), d, y_1) & M_2 &= p(f(d), c, c) \end{aligned}$$

We obtain the following substitution constraints:

$$\begin{aligned} \Gamma(L_1, M_1) &= (x_1, x_2, x_3) \triangleright (f(c), d, y_1) \\ \Gamma(L_1, M_2) &= (x_1, x_2, x_3) \triangleright (f(d), c, c) \\ \Gamma(L_2, M_1) &= \perp \\ \Gamma(L_2, M_2) &= (x_2, x_4) \triangleright (d, c) \end{aligned}$$

The constraints $\Gamma(L_1, M_1)$ and $\Gamma(L_1, M_2)$ are incompatible, as these constraints map, for example, x_1 to different values. The constraints $\Gamma(L_1, M_1)$ and $\Gamma(L_2, M_2)$ are compatible, as both constraints require their only shared variable x_2 to be mapped to d .

To encode clausal subsumption, we need to combine substitution constraints using boolean connectives, and boolean variables. For this reason, we now define the semantics of boolean combinations of substitution constraints.

Definition 3 (Boolean Combination of Substitution Constraints): Let F be a formula using standard boolean connectives, whose atoms are boolean variables and substitution constraints. An interpretation $I = (\alpha, \sigma)$ for such a formula is a pair of a standard boolean assignment $\alpha: \mathcal{B} \rightarrow \{\top, \perp\}$ and a substitution $\sigma: \mathcal{V} \rightarrow \mathcal{T}$.

For a boolean variable b , we define $I \models b$ iff $\alpha(b) = \top$. For a substitution constraint Γ , we define $I \models \Gamma$ iff $\sigma \models \Gamma$. For formulas F with a top-level connective of $\wedge, \vee, \rightarrow$, or \neg , we define $I \models F$ inductively in the standard way. For boolean constants, $I \models \top$ and $I \not\models \perp$.

Remark 1: The formula F can also be translated into an SMT formula using the theory of equality and uninterpreted functions (EUF), where substitution constraints are replaced by conjunctions of equality literals. Let T denote the set of terms t appearing on the right-hand side of some substitution constraint

in F . We then introduce fresh constant symbols $\{c_t \mid t \in T\}$, and replace each substitution constraint $\Gamma = (x_1, \dots, x_k) \triangleright (t_1, \dots, t_k)$ in F by $x_1 = c_1 \wedge \dots \wedge x_k = c_k$. To obtain correct semantics of substitution compatibility, we also need to add

$$\bigwedge_{t, u \in T, t \neq u} c_t \neq c_u, \quad (2)$$

asserting that constants representing different terms in F cannot be equal.

However, for clausal subsumption in a first-order theorem prover, it is vital that the process of encoding subsumption in SAT, as well as the setting up of our SAT solver for handling this encoding are as lean as possible (see Section V). Hence, we did not employ a standard SMT solver with the EUF-based encoding discussed above, but instead opted to directly add support for substitution constraints to our SAT solver. The advantage of our SAT-based approach is that we use less boolean literals, and we avoid using all-different constraints for terms, such as (2).

B. SAT-Encoding of Clausal Subsumption

We now present our formalization to express clausal subsumption between clauses C and D as a SAT problem over substitution constraints. To this end, assume that clause C is $L_1 \vee L_2 \vee \dots \vee L_n$, whereas D is $M_1 \vee M_2 \vee \dots \vee M_m$. Recall that deciding whether C subsumes D reduces to the problem of deciding whether there exists a substitution σ such that $\sigma(C) \subseteq D$, where “ \subseteq ” denotes multiset inclusion (over multisets of literals).

For arbitrary literals L_i and M_j , deciding the existence of a substitution σ with $\sigma(L_i) = M_j$ can easily be done. Yet, for clausal subsumption we are left with the challenge of finding a substitution σ such that, for each L_i , we have one of the M_j such that $\sigma(L_i) = M_j$. To address this challenge, we introduce new boolean variables b_{ij} to encode possible matchings of L_i to M_j , given by $\sigma(L_i) = M_j$. Additionally, we use Definition 2 to derive the substitution constraints $\Gamma(L_i, M_j)$. Based on the boolean variables b_{ij} and substitution constraints $\Gamma(L_i, M_j)$, we formalize clausal subsumption between C and D by ensuring its three properties: (i) each literal L_i in C is matched to a literal M_j in D , (ii) the same substitution σ is used for each of these matchings, and (iii) $C\sigma \subseteq D$ is multiset inclusion. Our formalization of clausal subsumption between C and D is given as follows.

- (i) We first define the following clauses, capturing that each literal L_i from C must be matched to (at least one) literal M_j of D :

$$\bigwedge_{1 \leq i \leq n} b_{i1} \vee b_{i2} \vee \dots \vee b_{im} \quad (3)$$

- (ii) We connect the boolean variables b_{ij} to the substitution constraints $\Gamma(L_i, M_j)$ through the following clauses:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} b_{ij} \rightarrow \Gamma(L_i, M_j). \quad (4)$$

These clauses employ the substitution constraints $\Gamma(L_i, M_j)$ to ensure the same substitution σ is used for matching L_i and M_j simultaneously, for all i, j .

- (iii) As clausal subsumption uses *multiset* inclusion over the respective multisets of literals of C and D , we encode the requirement that each literal of D may only be matched at most once:

$$\bigwedge_{1 \leq j \leq m} \text{AtMostOne}(b_{1j}, \dots, b_{nj}), \quad (5)$$

where $\text{AtMostOne}(b_{1j}, \dots, b_{nj})$ is true iff zero or one of b_{1j}, \dots, b_{nj} are true.

Together, the constraints (3), (4), (5) fully capture clausal subsumption, yielding the following result.

Theorem 1 (Clausal Subsumption as SAT): Clausal subsumption between clauses C and D is given by the conjunction of (3), (4), and (5). That is, C subsumes D iff $(3) \wedge (4) \wedge (5)$ is satisfiable.

Note that for deciding clausal subsumption between C and D , we only need to *establish satisfiability* of $(3) \wedge (4) \wedge (5)$ in Theorem 1: one substitution σ such that $C\sigma \subseteq D$ is sufficient for deciding that C subsumes D , implying that D can be deleted from the set of usable clauses during saturation. Hence, while clausal subsumption $(3) \wedge (4) \wedge (5)$ captures all substitutions σ for which $C\sigma \subseteq D$, for deciding whether C subsumes D we are interested to find only one satisfying instance of $(3) \wedge (4) \wedge (5)$. As a result, application of clausal subsumption in saturation can be decided by solving the satisfiability of $(3) \wedge (4) \wedge (5)$.

Example 2: Consider the literals defined in Example 1 and clauses $C = L_1 \vee L_2$ and $D = M_1 \vee M_2$. The encoding of clausal subsumption between C and D resulting from Theorem 1 is the conjunction of the following clauses:

$$\begin{aligned} & b_{11} \vee b_{12} \\ & b_{21} \vee b_{22} \\ & b_{11} \rightarrow (x_1, x_2, x_3) \triangleright (f(c), d, y_1) \\ & b_{12} \rightarrow (x_1, x_2, x_3) \triangleright (f(d), c, c) \\ & b_{21} \rightarrow \perp \\ & b_{22} \rightarrow (x_2, x_4) \triangleright (d, c) \\ & \neg b_{11} \vee \neg b_{21} \\ & \neg b_{12} \vee \neg b_{22} \end{aligned}$$

This set of clauses is satisfiable, as witnessed by the model that assigns b_{11} and b_{22} to true, b_{12} and b_{21} to false, and $\sigma(x_1) = c$, $\sigma(x_2) = f(d)$, $\sigma(x_3) = y_1$, $\sigma(x_4) = c$. We conclude that the first-order clause C subsumes D .

Remark 2 (Subsumption Resolution): Our encoding of clausal subsumption can be adjusted to also decide the application of other simplification rules in saturation, when these rules implement variants of subsumption. To this end, we have extended the SAT encoding $(3) \wedge (4) \wedge (5)$ of clausal subsumption to the inference rule *subsumption resolution*. In addition to clausal subsumption, subsumption resolution also uses instances of binary resolution. Hence, for finding substitutions σ such that subsumption resolution between clauses C and D can be

applied (and D deleted from the set of usable clauses), we extended the clauses $(3) \wedge (4) \wedge (5)$ with additional constraints capturing application of resolution, while also adjusting the encoding of $(3) \wedge (4) \wedge (5)$ to set inclusion between literals of C and D (instead of multiset inclusion from subsumption).

Remark 3 (At-Most-One Constraints): We conclude this section by noting that a correct but naive solution to encode $\text{AtMostOne}(b_{1j}, \dots, b_{nj})$ in (5) would be the following:

$$\bigwedge_{1 \leq i_1 < i_2 \leq n} \neg b_{i_1 j} \vee \neg b_{i_2 j}. \quad (6)$$

More efficient encodings using at-most-one constraints (see, e.g., [13]) can be used instead of (6). In our work however, we opted to add direct support for at-most-one constraints when reasoning about (5) (see Section IV).

IV. EFFECTIVE SUBSUMPTION VIA LEAN SAT SOLVING

In Section III we showed that the application of subsumption, as an inference rule in saturation, can be reduced to the satisfiability problem of the formula $(3) \wedge (4) \wedge (5)$ using substitution constraints (Theorem 1). In this section we describe our approach for solving $(3) \wedge (4) \wedge (5)$.

A straightforward approach towards handling $(3) \wedge (4) \wedge (5)$ could come with translating $(3) \wedge (4) \wedge (5)$ into only propositional clauses; yet, such a translation would either require additional propositional variables to encode at-most-one constraints or would come with a quadratic number of propositional clauses [13]; similarly for substitution constraints.

Due to the particular distribution of subsumption instances (see Section V), the encoding must be lightweight to be practically feasible. To overcome the increase in propositional variables/clauses to be used for deciding clausal subsumption in an efficient manner, we support substitution constraints (4) and at-most-one constraints (5) directly in SAT solving, and introduce a lean SAT solving approach tailored to subsumption properties. In particular, we adjust unit propagation and conflict resolution in CDCL-based SAT solving for handling propositional formulas with substitution constraints. This way, we integrate our lean SAT solving methodology directly into the saturation process of first-order proving (Section V), instead of interfacing first-order proving with an existing off-the-shelf SAT solver. Such a direct integration allows us to efficiently identify and apply subsumption during proof search (see Section VI).

a) Using Substitution Constraints in SAT Solving: For handling substitution constraints in clausal subsumption, we attach a substitution constraint $\Gamma(L_i, M_j)$ to each freshly introduced boolean variable b_{ij} in (3), which is equivalent to adding the constraint $b_{ij} \rightarrow \Gamma(L_i, M_j)$ of (4).

b) Unit Propagation with Substitution Constraints: Consider now the clauses $b_{ij} \rightarrow \Gamma(L_i, M_j)$ using substitution constraints, with $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$, from

clausal subsumption (3) \wedge (4) \wedge (5). Semantically, these constraints are equivalent to the following set of binary clauses:

$$\begin{aligned} & \{-b_{ij} \vee \neg b_{i'j'} \mid i, i' \in \{1 \dots n\}, j, j' \in \{1 \dots m\}, \\ & \quad (i, j) \neq (i', j'), \\ & \quad \exists x \in \text{dom}(\Gamma(L_i, M_j)) \cap \text{dom}(\Gamma(L_{i'}, M_{j'})) \\ & \quad \text{s.t. } \Gamma(L_i, M_j)(x) \neq \Gamma(L_{i'}, M_{j'})(x)\}, \end{aligned} \quad (7)$$

which intuitively encodes that no two incompatible substitution constraints may be true at the same time.

In our work, instead of creating the binary clauses of (7) explicitly, we introduce support for substitution constraints as an *additional (unit) propagator* in SAT solving: whenever a boolean variable b_{ij} is assigned to true, our SAT solver processes the associated bindings for the first-order variables from $\text{dom}(\Gamma(L_i, M_j))$, and propagates all boolean variables $b_{i'j'}$ to false that are associated with conflicting bindings for variables $\text{dom}(\Gamma(L_i, M_j)) \cap \text{dom}(\Gamma(L_{i'}, M_{j'}))$; in other words, all $b_{i'j'}$ whose associated substitution constraints are incompatible with $\Gamma(L_i, M_j)$. This propagation is done exhaustively once b_{ij} is assigned to true and before standard unit propagation in SAT solving would be applied. Thus we ensure that no conflict can occur at this point: if there were a conflict, that would mean a $b_{i'j'}$ with conflicting bindings has already been assigned to true; in this case however, we would have already propagated b_{ij} to false when assigning $b_{i'j'}$. An exception in handling conflicts occurs with the initial propagation before starting the CDCL loop of SAT solving; in this case, we may get a conflict if two unit clauses with conflicting substitution constraints have been added, however, in that case the SAT solver is at decision level 0 and can terminate with reporting unsatisfiability (UNSAT) of (3) \wedge (4) \wedge (5).

c) Conflict Resolution with Substitution Constraints:

During conflict resolution in our SAT engine, we proceed as if the binary clauses (7) were part of the clause database, i.e., as if the binary clause $\neg b_{ij} \vee \neg b_{i'j'}$ were the reason for propagating $b_{i'j'}$. Therefore we only need to store the literal b_{ij} as the reason for unit propagation. Substitution constraints during conflict resolution thus do not need specialized treatment in our SAT solving approach.

d) At-Most-One Constraints:

During unit propagation and conflict resolution, our at-most-one constraints (5) are treated as if we had the corresponding binary clauses from (6), saving the overhead from creating additional clauses and variables.

Remark 4: While we presented our approach in the context of solving (3) \wedge (4) \wedge (5), our SAT solving approach naturally supports arbitrary boolean clauses and at-most-one constraints, as well as substitution constraints in the form $b \rightarrow \Gamma$ (where b is a boolean variable and Γ a substitution constraint).

V. SAT-BASED SUBSUMPTION IN FIRST-ORDER THEOREM PROVING

We implemented our lean SAT-based approach of Section IV as a new extension to the theorem prover VAMPIRE. While VAMPIRE already implements highly optimized algorithms for checking subsumption, these algorithms are built on a

standard, backtracking-based search procedure: using a static variable ordering and limited amount of unit propagation, without learning from conflicts. Hence, the full power of SAT-based reasoning with unit propagation and conflict resolution is not yet supported for subsumption. We overcome this limitation by integrating our SAT-based approach for clausal subsumption directly in VAMPIRE. Our implementation consists of about 5000 lines of C++ code and is available at <https://github.com/JakobR/vampire/tree/sat-subsumption>.

a) Implementing Subsumption: When establishing satisfiability of (3) \wedge (4) \wedge (5), we can observe two different types of subsumption instances:

- (i) easy subsumption instances, where not much SAT-based search is required (very few or even no decisions/conflicts), For such instances the overhead of setting up the clausal encoding of (3) \wedge (4) \wedge (5) largely determines the total running time of our SAT solver.
- (ii) hard subsumption instances, whose application is determined by a significant number of unit propagation and/or conflict resolution steps in SAT solving.

We recall that the overall goal of our work is to improve subsumption checking in first-order theorem proving. For this, we complemented VAMPIRE with a SAT-based approach to decide application of subsumption. Note that the majority of the subsumption instances encountered during a typical first-order proving attempt are of type (i), with instances of type (ii) appearing occasionally, depending on the input formula. Still, the total running time is often dominated by type (ii) instances, and these are the target of our SAT-based approach. We must however be careful to not become slower on type (i) instances, thus motivating our choice of a lean, dedicated SAT-solver embedded into VAMPIRE.

In many of the trivial instances of (3) \wedge (4) \wedge (5), the unsatisfiability (UNSAT) of these instances can be discovered already during the encoding of (3) \wedge (4) \wedge (5) (whenever an empty clause would be added). To save time on these instances, in our implementation we defer the construction of watch lists and other data structures until entering the solving loop of our SAT engine (if at all).

We note that the number of subsumption instances, especially easy ones of type (i), during first-order proving can become quite large, often in the order of millions of instances in a 60s run of a theorem prover. Allocating and deallocating a new SAT solver instance for each SAT-based subsumption query can thus become expensive (see Section VI); therefore, in our implementation we keep the same solver instance around, and re-use it for different queries. In particular, we keep the memory for data structures (such as clause storage, watch lists, trail, and others), instead of reallocating it for each query.

b) Unit Propagation: To achieve efficient unit propagation, our SAT solver for clausal subsumption watches two literals of each clause [26]. However, for at-most-one constraints the situation is different. Consider the constraint $\text{AtMostOne}(l_1, \dots, l_k)$ for some $k \geq 3$ (note that for $k \leq 2$ we either drop the constraint or add a binary clause instead). As soon as any l_i is assigned true, all l_j with $j \neq i$ must be false to

avoid violating the constraint, and are propagated thus. Hence, the solver watches *all* literals of at-most-one constraints.

VI. EXPERIMENTS

We evaluated our new SAT-based implementation for clausal subsumption in VAMPIRE (see Section V). In our experiments, we were interested (i) to measure the performance improvements we gain through our approach, as well as (ii) to assess the advantage of re-using our SAT solver objects, and thus having our SAT solver directly integrated the first-order proving process of VAMPIRE.

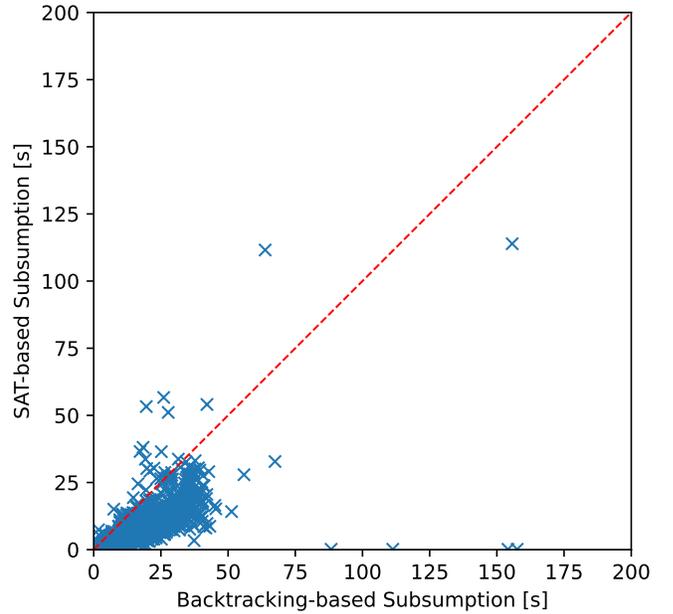
Benchmarks. The basis for our benchmarks is formed by the TPTP library [36] (version 7.5.0), which is a standard benchmark library in the theorem proving community. The TPTP library contains altogether 24,098 problems in various languages, out of which 16,312 problems have been included in our evaluation of SAT-based subsumption in VAMPIRE. The remaining TPTP problems that we did not use for our experiments either use features that VAMPIRE currently does not support (e.g., higher-order logic with theories), or did not involve subsumption checks.

Experimental Setup. All our experiments were carried out on a cluster at TU Wien, where the compute nodes contain two AMD Epyc 7502 processors, each of which has 32 CPU cores running at 2.5 GHz. Each compute node is equipped with 1008 GiB of physical memory that is split into eight memory nodes of 126 GiB each, with eight logical CPUs assigned to each node. We used the tool `runexec` from the benchmarking framework `BENCHEXEC` [5] to assign each benchmark process to a different CPU core and its corresponding memory node, while aiming to balance the load evenly across memory nodes. Further, we used `GNU PARALLEL` [38] to schedule 32 benchmark processes in parallel.

Experimental Results on Measuring Speed Improvements. We emphasize that using a SAT-based approach for deciding clausal subsumption will, in theory, not prove problems that were not provable before. If a problem is provable while using saturation with redundancy, and hence with subsumption, then it is also provable using saturation without redundancy, and vice versa. However, in practice, saturation with redundancy (hence with subsumption) will improve the prover’s performance in finding a proof. As such, the aim of our work is to speed up the application of subsumption in saturation. For this reason, we set up our first experiment to measure the cost of subsumption checks in isolation. A similar evaluation has previously been done for indexing techniques in first-order provers, see [27].

In preparation for this experiment, we ran VAMPIRE, using the original backtracking-based subsumption implementation, with a timeout of 60 seconds on each TPTP problem while logging each subsumption (and subsumption resolution) check into a file. Each of these files contains a sequence of subsumption (and subsumption resolution) checks, which we call the *subsumption log* for a problem. This preparatory step yielded a large number of benchmarks that are representative for the checks appearing during actual proof search. These benchmarks

Figure 1. Total running time (in seconds) of backtracking-based vs. SAT-based subsumption, with detailed information about outliers in Table I. For marks below the dashed line, our SAT-based approach was faster.



occupy 1.75 TiB of disk space in compressed form, and contain approximately 114 billion subsumption checks in total. About 0.5 % of these subsumption checks are satisfiable (561 million), while the rest are unsatisfiable.

In addition to generating these benchmarks, we have profiled the portion of time spent by subsumption in VAMPIRE. Over the TPTP problems used for our experiments and a time limit of 60 seconds, it ranges from 0 % (no subsumption checks) to more than 99 % (hard subsumption check), with a mean of 46 % and the median at 53 %.

Next, we executed the checks listed in each subsumption log and measured the total running times, once for the already existing subsumption algorithm of VAMPIRE using backtracking, and once for our SAT-based subsumption approach in VAMPIRE. The subsumption checks are benchmarked in a similar way as they would appear during a regular prover run, i.e., with the same caching of intermediate results. For increased reliability, each measurement was performed five times, and then taking the arithmetic mean.

The results of these experiments are given in Figure 1 and Table I. Each mark in Figure 1 represents one subsumption log from a TPTP problem, and compares the total running times of executing all subsumption checks contained in the log with the old backtracking-based algorithm vs. the new SAT-based algorithm. The dashed line indicates equal runtime, hence, our SAT-based approach was faster for marks below the line. In Table I, we give the cumulative times needed to set up the subsumption checks, to solve them, and the total time. Both the backtracking-based and our SAT-based subsumption algorithm can naturally be split up into a setup stage and a separate solving stage. The setup stage transforms the two

Table I
RUNNING TIME OF SUBSUMPTION CHECKS

Subsumption log for problem	Backtracking-based Subsumption			SAT-based Subsumption			Δ_{abs}	Δ_{rel}
	Setup	Solve	Total	Setup	Solve	Total		
GRP134-1.005	42.87 s	2.21 s	45.08 s	13.87 s	2.61 s	16.48 s	28.60 s	2.74 x
GRP396+1	67.05 s	88.65 s	155.70 s	15.90 s	98.01 s	113.91 s	41.79 s	1.37 x
HAL007+1	33.25 s	30.54 s	63.79 s	17.05 s	94.51 s	111.56 s	-47.78 s	0.57 x
HWV056+1	26.72 s	1.01 s	27.73 s	48.73 s	2.37 s	51.10 s	-23.37 s	0.54 x
HWV058-1	17.32 s	1.05 s	18.37 s	37.57 s	0.53 s	38.10 s	-19.73 s	0.48 x
HWV059-1	24.21 s	0.95 s	25.16 s	35.79 s	0.68 s	36.48 s	-11.31 s	0.69 x
HWV060+1	16.61 s	0.66 s	17.26 s	35.82 s	0.73 s	36.55 s	-19.28 s	0.47 x
HWV086+1	17.76 s	1.80 s	19.57 s	50.12 s	3.15 s	53.27 s	-33.71 s	0.37 x
LCL662+1.020	43.78 s	1.64 s	45.42 s	14.33 s	0.86 s	15.19 s	30.23 s	2.99 x
MGT038-1	13.15 s	12.88 s	26.04 s	15.35 s	41.33 s	56.67 s	-30.64 s	0.46 x
MGT066+1	3.45 s	63.99 s	67.44 s	1.95 s	30.87 s	32.82 s	34.63 s	2.06 x
NLP023+1	0.08 s	154.05 s	154.13 s	0.04 s	0.10 s	0.14 s	153.99 s	1082.84 x
NLP023-1	0.09 s	157.46 s	157.55 s	0.05 s	0.10 s	0.14 s	157.40 s	1087.59 x
NLP024+1	0.08 s	88.26 s	88.34 s	0.04 s	0.09 s	0.14 s	88.20 s	642.68 x
NLP024-1	0.09 s	111.20 s	111.28 s	0.05 s	0.10 s	0.15 s	111.13 s	748.52 x
PUZ073+1	24.69 s	26.60 s	51.29 s	14.02 s	0.14 s	14.17 s	37.12 s	3.62 x
SYN307-1	2.09 s	53.81 s	55.90 s	1.17 s	26.73 s	27.90 s	28.01 s	2.00 x
TOP003-2	41.71 s	0.43 s	42.13 s	48.92 s	5.13 s	54.05 s	-11.92 s	0.78 x
... (+16,294)
Total	16.31 h	2.39 h	18.70 h	7.21 h	1.23 h	8.44 h	10.27 h	2.22 x
Total (no reuse)	-	-	-	8.08 h	2.05 h	10.12 h	-	-
Total (VMTF)	-	-	-	7.62 h	1.40 h	9.02 h	-	-

input clauses into constraints while the solving stage searches for a solution to these constraints. Additionally the table gives detailed data for selected outliers (problems not in the bottom-left of Figure 1).

As shown in Figure 1 and Table I, our SAT-based algorithm for clausal subsumption gives a clear overall improvement of the running/proving time of VAMPIRE by a factor of 2.

Note that for some problems, the running time for the backtracking-based subsumption is higher than the original timeout of 60 s that has been used when collecting subsumption logs. The cause of this apparent discrepancy is that VAMPIRE was working on a hard subsumption instance when hitting the timeout, with the subsequent measurements in Table I showing the true cost. Problems such as NLP023+1 are getting stuck in the backtracking-based subsumption algorithm, while our SAT-based approach would allow proof search to continue much further within the same time limit.

We also evaluated the impact of our custom variable selection heuristic (see last paragraph of Section VII) compared to the variable-move-to-front (VMTF) heuristic of SAT solvers [8], as VMTF is conjectured to perform well for SAT problems that are unsatisfiable, being part of the “unstable phase” described in [7]. Given that almost all subsumption instances are unsatisfiable, we were interested to see how our SAT-based approach performs compared to a VMTF heuristic. Our results in this respect are listed in the last line of Table I. While our custom heuristic shows slightly better solving times than VMTF, the difference is rather small.

Experimental Results on the Advantage of Re-Using SAT Solver Objects. We also assessed the importance of re-using the SAT solver object instead of re-allocating the solver for every subsumption query. The result is given in the second-to-last line of Table I, confirming the significance of having

SAT-based subsumption directly integrated in VAMPIRE.

VII. RELATED WORK

Subsumption is one of the most important simplification rules in first-order theorem proving. While efficient literal- and clause-indexing techniques have been proposed [37], [33], optimizing the matching step among multisets of literals, and hence clauses, has so far not been addressed. In our work, we show that SAT solving methods can provide efficient solutions in this respect, further improving first-order theorem proving.

A related approach that integrates multi-literal matching into indexing is given in [35], using code trees. Code trees organize potentially subsuming clauses into a trie-like data structure with the aim of sharing some matching effort for similar clauses. However, the underlying matching algorithm uses a fixed branching order and does not learn from conflicts, and will thus run into the same issues on hard subsumption instances as the standard backtracking-based matching.

The specialized subsumption algorithm DC [18] is based on the idea of separating the clause C into variable-disjoint components and testing subsumption for each component separately. However, the notion of subsumption considered in that work is defined using subset inclusion, rather than multiset inclusion. For subsumption based on multiset inclusion, the subsumption test for one variable-disjoint component is no longer independent of the other components.

An improved version of that algorithm, called IDC [17], tests on each recursion level whether each literal of C by itself subsumes D under the current partial substitution, which is a necessary condition for subsumption. The backtracking-based subsumption algorithm of VAMPIRE uses this optimization as well, and our SAT-based approach also implements it as propagation over substitution constraints.

SAT- and SMT-based techniques have previously been applied to the setting of first-order saturation-based proof search, e.g., in form of the AVATAR architecture [39]. These techniques are however independent from our work, as they apply the SAT- or SMT-solver over an abstraction of the input problem, while in our work we use a SAT-solver to speed up certain inferences.

Some solvers, such as the pseudo-boolean solver Mini-Card [23] and the ASP solver Clasp [14], support cardinality constraints natively, in a similar way to our handling of AtMostOne constraints. Our encoding however requires only AtMostOne constraints instead of arbitrary cardinality constraints, thus simplifying the implementation.

We finally note that clausal subsumption can also be seen as a constraint satisfaction problem (CSP). In this view, the boolean variables b_{ij} in our subsumption encoding $(3) \wedge (4) \wedge (5)$ represent the different choices of a non-boolean CSP variable, corresponding to the so-called *direct encoding* of a CSP variable [40]. A well-known heuristic in CSP solving is the minimum remaining values heuristic: always assign the CSP variable that has the fewest possible choices remaining. We adapted this heuristic to our embedded SAT solver and use it to solve subsumption instances (see Section V).

VIII. CONCLUSION

We advocate the use of lean dedicated SAT solving to solve clausal subsumption in first-order theorem proving. We introduce substitution constraints to encode subsumption as a SAT instance. For solving such instances, we adjust unit propagation and conflict resolution in SAT solving towards a tailored treatment of substitution constraints. Crucially, our encoding together with our tailored solver enables efficient setup of subsumption instances. Our experimental results indicate that SAT-based subsumption significantly improves the performance of first-order proving. Extending our work towards equality reasoning, and hence addressing subsumption demodulation, is an interesting task for future work. For doing so, we believe our substitution constraints would need to encode matching also on the term level, and thus not only on the literal level, in order to find suitable terms to rewrite.

Acknowledgements. We thank Bernhard Gleiss for fruitful initial discussions about this work. This work was supported by the ERC Consolidator Grant ARTIST 101002685 and the Austrian FWF project W1255-N23.

REFERENCES

- [1] Sepideh Asadi, Martin Blicha, Antti E. J. Hyvärinen, Grigory Fedyukovich, and Natasha Sharygina. Incremental Verification by SMT-based Summary Repair. In *Proc. of FMCAD*, pages 77–82, 2020.
- [2] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.*, 4(3):217–247, 1994.
- [3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. CVC5: A Versatile and Industrial-Strength SMT Solver. In *Proc. of TACAS*, pages 415–442, 2022.

- [4] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying Relational Properties using Trace Logic. In *Proc. of FMCAD*, pages 170–178, 2019.
- [5] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable Benchmarking: Requirements and Solutions. *J. on Software Tools for Technology Transfer*, 21(1):1–29, 2017.
- [6] Armin Biere. PicoSAT Essentials. *J. Satisf. Boolean Model. Comput.*, 4(2-4):75–97, 2008.
- [7] Armin Biere. CaDiCaL at the SAT Race 2019. In *Proc. of SAT Race*, pages 8–9, 2019.
- [8] Armin Biere and Andreas Fröhlich. Evaluating CDCL Variable Scoring Schemes. In *Proc. of SAT*, pages 405–422, 2015.
- [9] Bruno Buchberger. Bruno Buchberger’s PhD thesis 1965: An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. *J. Symb. Comput.*, 41(3-4):475–511, 2006.
- [10] Martin Clochard, Claude Marché, and Andrei Paskevich. Deductive Verification with Ghost Monitors. *Proc. of POPL*, pages 2:1–2:26, 2020.
- [11] Simon Cruanes. Superposition with Structural Induction. In *Proc. of FroCoS*, pages 172–188, 2017.
- [12] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.
- [13] Alan M. Frisch and Paul A. Giannaros. SAT Encodings of the At-Most-k Constraint. Some Old, Some New, Some Fast, Some Slow. In *Proc. of WS on Constraint Modelling and Reformulation*, 2010.
- [14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers. In *Proc. of ICLP*, pages 250–264, 2009.
- [15] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace Logic for Inductive Loop Reasoning. In *Proc. of FMCAD*, pages 255–263, 2020.
- [16] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption Demodulation in First-Order Theorem Proving. In *Proc. of IJCAR*, pages 297–315, 2020.
- [17] Georg Gottlob and Alexander Leitsch. Fast Subsumption Algorithms. In *Proc. of EUROCAL '85*, pages 64–77, 1985.
- [18] Georg Gottlob and Alexander Leitsch. On the Efficiency of Subsumption Algorithms. *J. of the ACM*, 32(2):280–295, 1985.
- [19] Deepak Kapur and Paliath Narendran. NP-Completeness of the Set Unification and Matching Problems. In *Proc. of IJCAR*, pages 489–495, 1986.
- [20] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. *Formal Methods Syst. Des.*, 48(3):175–205, 2016.
- [21] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, pages 1–35, 2013.
- [22] K. Rustan M. Leino. Accessible Software Verification with Dafny. *IEEE Softw.*, 34(6):94–97, 2017.
- [23] Mark H. Liffiton and Jordyn C. Maglalang. A Cardinality Solver: More Expressive Constraints for Free. In *Proc. of SAT*, pages 485–486, 2012.
- [24] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 133–182. IOS Press, 2021.
- [25] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Gianarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Proc. of ESOP*, pages 30–59, 2019.
- [26] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of DAC*, pages 530–535, 2001.
- [27] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In *Proc. of IJCAR*, pages 257–271, 2001.
- [28] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
- [29] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proc. of PLDI*, pages 614–630, 2016.
- [30] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. Automating Modular Verification of Secure Information Flow. In *Proc. of FMCAD*, pages 158–168, 2020.

- [31] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.
- [32] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proc. of CCS*, pages 621–640, 2020.
- [33] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, pages 45–67, 2013.
- [34] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, Higher, Stronger: E 2.3. In *Proc. of CADE*, pages 495–507, 2019.
- [35] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term Indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001.
- [36] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *J. of Automated Reasoning*, 59(4):483–502, 2017.
- [37] Tanel Tammet. Towards Efficient Subsumption. In *Proc. of CADE*, pages 427–441, 1998.
- [38] Ole Tange. *GNU Parallel 2018*. Ole Tange, March 2018.
- [39] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In *Proc. of CAV*, pages 696–710, 2014.
- [40] Toby Walsh. SAT v CSP. In *Proc. of CP*, pages 441–456, 2000.
- [41] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischniewski. SPASS version 3.5. In *Proc. of CADE*, pages 140–145, 2009.

BAXMC: a CEGAR approach to Max#SAT

Thomas Vigouroux*[✉], Cristian Ene*[✉], David Monniaux*[✉], Laurent Mounier*, Marie-Laure Potet*[✉]

*Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

Firstname.Lastname@univ-grenoble-alpes.fr

Abstract—Max#SAT is an important problem with multiple applications in security and program synthesis that is proven hard to solve. It is defined as: given a parameterized quantifier-free propositional formula compute parameters such that the number of models of the formula is maximal. As an extension, the formula can include an existential prefix.

We propose a CEGAR-based algorithm and refinements thereof, based on either exact or approximate model counting, and prove its correctness in both cases. Our experiments show that this algorithm has much better effective complexity than the state of the art.

I. INTRODUCTION

#SAT is the problem of counting the solutions of a quantifier-free propositional formula, the counting version of the SAT problem. Max#SAT is the problem of optimizing, according to some propositional variables, the number of solutions according to the others. We generalize this problem to allow an existential prefix in the formula.

This problem has many practical applications in diverse areas of computer science such as *quantitative program analysis* and *program synthesis* [1]. Most approaches for quantitative information flow analysis use approximations, with fast yet imprecise solutions. Adaptive attacker synthesis [2] would also benefit from advances in Max#SAT efficiency, mainly by being able to avoid the use of imprecise heuristics.

Unfortunately, Max#SAT has high complexity [3], [4], and practical solving methods remain costly. At the time of writing, only one solver is publicly available off-the-shelf [1].

Earlier work on the Max#SAT problem proposed two approaches. The first is a probabilistic solving method [1], which unfortunately degrades to exhaustive search when seeking precise answers to the problem. The second approach [5] solves the problem exactly, but scales poorly.

We present in this paper a new approach to Max#SAT, leveraging ideas from CEGAR solvers, and show its effectiveness on various benchmarks used in previous publications on the subject. We also present improvements of our algorithm based on previous work about symmetry breaking in SAT solvers [6].

Our contributions are the following:

- An *effective algorithm* to compute maximal solutions for the projected model counting problem (Sections III and IV). This algorithm relies either on an *exact* projected model counter as a subprocedure, or on an *approximated* one, which should be the case most times in practice for

scalability reasons. A complete correctness proof of this algorithm is given for both cases.

- The *extension* of our algorithm with SAT *symmetry breaking* techniques (Section V) and *heuristics* (Section VI), to further improve its efficiency.
- The *implementation* of this algorithm in the tool BAXMC [7], together with a set of *experimental results* (Section VII), showing the accuracy and performances of our Max#SAT algorithm on various benchmarks, with respect to the only other available tool.

II. PRELIMINARIES

We set our problem in standard Boolean logic. Throughout the paper, Greek letters (ϕ, ψ, \dots) denote Boolean formulas, uppercase calligraphic Latin letters ($\mathcal{V}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}, \dots$) denote sets of variables, simple uppercase Latin letters (V, X, Y, Z, \dots) denote variables, lowercase variants of these letters (x, y, z, \dots) denote valuations for these sets of variables.

Let $\mathbb{B} = \{true, false\}$. A *literal* is a variable or its negation and the set of literals derived from a set of variables \mathcal{V} is denoted by $\bar{\mathcal{V}} = \mathcal{V} \cup \neg\mathcal{V}$. Let $\phi(\mathcal{V})$ be a Boolean formula over \mathcal{V} a set of variables. A valuation $v : \mathcal{V} \rightarrow \mathbb{B}$ is a *model* of ϕ if ϕ evaluates to *true* over v ; this is denoted by $v \models \phi$.

We say that a formula ϕ is *satisfiable* if there exists v such that $v \models \phi$. Otherwise, ϕ is deemed *unsatisfiable*. Determining whether a formula is satisfiable or unsatisfiable is called the *satisfiability* problem, also known as SAT.

The *restriction* of a valuation $v : \mathcal{V} \rightarrow \mathbb{B}$ to $\mathcal{E} \subseteq \mathcal{V}$ is denoted by $v|_{\mathcal{E}}$. We say that two valuations v_1 and v_2 *agree* on \mathcal{E} , denoted by $v_1 \sim_{\mathcal{E}} v_2$, if their restrictions to \mathcal{E} are equal.

A. Base definitions

Definition II.1 (Equivalence class). Given a valuation v and a set \mathcal{E} , we call *equivalence class* of v over \mathcal{E} the set of valuations that agree with v over \mathcal{E} , that is:

$$[v]_{\mathcal{E}} = \{v' \mid v' \sim_{\mathcal{E}} v\}$$

We call $v|_{\mathcal{E}}$ *partial* models, and v *complete* models. The elements of $[v]_{\mathcal{E}}$ are called the *extensions* or $v|_{\mathcal{E}}$.

Definition II.2. Given propositional formula $\phi(\mathcal{V})$ and $\mathcal{E} \subseteq \mathcal{V}$, $\mathcal{M}_{\mathcal{E}}(\phi) = \{v|_{\mathcal{E}} \mid v \models \phi\}$ denotes the *set of models projected over \mathcal{E}* .

Remark II.1. We omit the set \mathcal{E} when it contains all the variables of ϕ . That is $\mathcal{M}(\phi) = \mathcal{M}_{\mathcal{V}}(\phi)$.

This work was partially supported by the French ANR project TAVA (ANR-20-CE25-0009) and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissements d’avenir.

Definition II.3. Given a valuation v , we define the *update* of the variable $V \in \mathcal{V}$ to b as:

$$v[V \rightarrow b](X) = \begin{cases} v(X) & \text{if } X \neq V \\ b & \text{otherwise} \end{cases}$$

Definition II.4. Given two formulas $\phi(\mathcal{V})$ and $\psi(\mathcal{V})$, we say that ϕ *entails* ψ (denoted by $\phi \models \psi$) if $\mathcal{M}(\phi) \subseteq \mathcal{M}(\psi)$.

B. Domain-specific definitions

In the remaining of the paper we consider a *partition* of \mathcal{V} over three sets \mathcal{X} , \mathcal{Y} and \mathcal{Z} , respectively called *witness*, *counting* and *intermediate* variables. Given a Boolean formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, we define Max#SAT as an optimization problem stated as follows: find $x_m \in \mathcal{M}_{\mathcal{X}}(\phi)$ such that the projected model counting over \mathcal{Y} of the formula (which will be defined later) is maximal.

Definition II.5 (Induced set). Given a formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ and $x \in \mathcal{M}_{\mathcal{X}}(\phi)$, the set of models over \mathcal{Y} *induced* by x is:

$$I_{\phi}(x) = \{y \in \mathcal{M}_{\mathcal{Y}}(\phi) \mid \exists z. (x, y, z) \models \phi\}$$

We extend this definition to partial witnesses as follows: $I_{\phi}(x|_{\mathcal{E}}) = \bigcup_{x' \in [x]_{\mathcal{E}}} I_{\phi}(x')$.

Definition II.6 (Model counting). Given a formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, the *count* of a witness x is defined by the size of the set it induces $|\phi(x, \mathcal{Y}, \mathcal{Z})| = |I_{\phi}(x)|$.

We extend this definition to partial witnesses as follows $|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| = |I_{\phi}(x|_{\mathcal{E}})|$.

Definition II.7 (Max#SAT). Given formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, we can state the Max#SAT problem more formally as finding $x_m \in \mathcal{M}_{\mathcal{X}}(\phi)$ such that:

$$|\phi(x_m, \mathcal{Y}, \mathcal{Z})| = \max_{x \in \mathcal{M}_{\mathcal{X}}(\phi)} |\phi(x, \mathcal{Y}, \mathcal{Z})|$$

Property II.1. Given a formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, the count of a partial witness is an upper-bound of the count of its extensions:

$$\forall x' \in [x]_{\mathcal{E}}, |\phi(x', \mathcal{Y}, \mathcal{Z})| \leq |\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})|$$

Proof. This follows directly from Definition II.5 on induced sets. \square

Property II.2 (Monotony of model counting). Given a propositional formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, $\mathcal{A} \subseteq \mathcal{B} \subseteq \mathcal{X}$, and $x \in \mathcal{M}_{\mathcal{X}}(\phi)$, the count of partial solutions is monotonous:

$$|\phi(x|_{\mathcal{B}}, \mathcal{Y}, \mathcal{Z})| \leq |\phi(x|_{\mathcal{A}}, \mathcal{Y}, \mathcal{Z})|$$

Proof. First, following Definition II.1 we have:

$$[x]_{\mathcal{B}} \subseteq [x]_{\mathcal{A}}$$

Hence, following Definition II.5:

$$I_{\phi}(x|_{\mathcal{B}}) \subseteq I_{\phi}(x|_{\mathcal{A}})$$

\square

Property II.3. Given a Boolean formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ such that $x \in \mathcal{M}_{\mathcal{X}}(\phi)$, $\mathcal{E} \subseteq \mathcal{X}$ and $X_i \in \mathcal{X}$, we have:

$$\begin{aligned} |\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| &\leq |\phi(x|_{\mathcal{E}-\{X_i\}}, \mathcal{Y}, \mathcal{Z})| \leq \\ &|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| + |\phi(x|_{\mathcal{E}}[X_i \rightarrow \neg x(X_i)], \mathcal{Y}, \mathcal{Z})| \end{aligned}$$

Proof. The first inequality is a direct consequence of Property II.2. The last inequality follows from Definition II.5. \square

Property II.4. For a given $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ and $\phi'(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ such that $\phi \models \phi'$, a witness x , and $\mathcal{E} \subseteq \mathcal{X}$ we have:

$$|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| \leq |\phi'(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})|$$

Proof. For any $y \in I_{\phi}(x|_{\mathcal{E}})$, as $y \models \phi$, and $\phi \models \phi'$, we get $y \models \phi'$ and hence $I_{\phi}(x|_{\mathcal{E}}) \subseteq I_{\phi'}(x|_{\mathcal{E}})$. \square

Property II.5. Given $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, $\psi(\mathcal{X})$ and $x \models \psi$, we have:

$$|\phi(x, \mathcal{Y}, \mathcal{Z})| = |(\phi \wedge \psi)(x, \mathcal{Y}, \mathcal{Z})|$$

Proof. Since $x \models \psi$ and ψ does not depend on \mathcal{Y} and \mathcal{Z} , $(x, y, z) \models \phi$ if and only if $(x, y, z) \models \phi \wedge \psi$. \square

III. SOLVING MAX#SAT

This section presents the main algorithm we propose to solve the Max#SAT problem.

A. The main algorithm

Algorithm 1 takes as input a formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$ and computes a pair (x_m, n_m) such that x_m is a solution to Max#SAT for ϕ with model counting n_m . Together with the formula, the algorithm takes multiple precision parameters:

- (ϵ_i) that are called *tolerance* parameters [8];
- (δ_i) that are called *confidence* parameters [8];
- κ that is called the *persistence* parameter.

Further explanations about these parameters will be given later.

Roughly speaking, this algorithm consists in iterating over possible *witnesses* x of ϕ . If the model count for x is less than the current best solution, it *blocks generalizations* of x such that all extensions of these generalizations are *worse* than the current best solution (Lines 14 and 16), hence removing a chunk of the search space at each iteration. Otherwise, it saves the candidate, which is then the new maximum, and blocks it (Lines 10 and 16), removing only one candidate from the search space.

We use two kinds of oracles in this algorithm. At Line 7 we call a SAT solver. Calls to an existing #SAT oracle (Lines 5, 8 and 17) can be performed using either an exact or an approximate model counter. In the latter case the precision parameters taken as input of the algorithm are used to configure the oracle, and influence the correctness of the returned value (in the former case, simply assume that they are all 0).

Definition III.1. Given $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, $x \in \mathcal{M}_{\mathcal{X}}(\phi)$ we say that $\mathcal{E} \subseteq \mathcal{X}$ is n -*bounding* if $|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| \leq n$.

The GENERALIZE function used in Algorithm 1 at Line 14 is proved to return n_m -bounding sets in both the exact (Theorem IV.1) and the approximate case (with probability $1 - \delta$,

Algorithm 1 Pseudocode for the BAXMC algorithm

```
1: function BAXMCε0,ε1,δ0,δ1,δ2,κ(φ( $\mathcal{X}$ ,  $\mathcal{Y}$ ,  $\mathcal{Z}$ ))
2:   φs ← φ
3:   xm ← ⊥
4:   nm ← 0
5:   N ← MCε0,δ0(φ(∅,  $\mathcal{Y}$ ,  $\mathcal{Z}$ ))
6:   while nm <  $\frac{N}{1+\kappa}$  do
7:     x  $\stackrel{\$}{\leftarrow}$   $\mathcal{M}_{\mathcal{X}}(\phi_s)$       ▷ Pick a new candidate
8:     c ← MCε1,δ1(φs(x,  $\mathcal{Y}$ ,  $\mathcal{Z}$ ))
9:     if c > nm then          ▷ New maximum
10:       xm ← x
11:       nm ← c
12:        $\mathcal{E} \leftarrow \mathcal{X}$ 
13:     else                    ▷ Find generalization
14:        $\mathcal{E} \leftarrow \text{GENERALIZE}_{\delta_2}(x, \phi_s, n_m)$ 
15:     end if
16:     φs ← φs ∧ ¬(x| $\mathcal{E}$ )      ▷ Block
17:     N ← MCε0,δ0(φs(∅,  $\mathcal{Y}$ ,  $\mathcal{Z}$ ))
18:   end while
19:   return xm, nm
20: end function
```

Theorem IV.2). The GENERALIZE function is called Algorithm 2 in this paper and it will be presented in Section IV.

B. Termination and correctness with an exact #SAT oracle

In this subsection, each i -indexed variable of the algorithm denotes its value at the end of the i -th iteration of the main loop. In the exact version of the algorithm, all precision parameters are assumed to be equal to 0 and all calls to MC_{0,0}(φ(x, \mathcal{Y} , \mathcal{Z})) return |φ(x, \mathcal{Y} , \mathcal{Z})|.

Theorem III.1 (Termination with an exact #SAT oracle). *Algorithm 1 always terminates.*

Proof. By construction of (φ_{s_i)_i we have:}

$$\forall i > 0, 0 \leq |\mathcal{M}_{\mathcal{X}}(\phi_{s_{i+1}})| < |\mathcal{M}_{\mathcal{X}}(\phi_{s_i})|$$

The sequence (n_{m_i)_i is obviously increasing. From Property II.4, the sequence (N_{i) _i is decreasing and hence (N_i - n_{m_i)_i is decreasing.}}}

Putting all this together, (|\mathcal{M}_{\mathcal{X}}(\phi_{s_i})| + (N_i - n_{m_i)_i) is strictly decreasing.}

One can easily see that whenever |\mathcal{M}_{\mathcal{X}}(\phi_{s_i})| = 0 it follows that N_i = 0 and N_i - n_{m_i ≤ 0. Hence in all cases, after some iteration k , N_k - n_{m_k ≤ 0 and the termination follows. □}}

Remark III.1. The worst case complexity of Algorithm 1 is reached when it iterates over all the witnesses of the formula.

Let k be the number of iterations performed when Algorithm 1 terminates, then we have n_{m_k ≥ N_k.}

Lemma III.1. *At every iteration i of Algorithm 1, we have:*

$$\mathcal{M}_{\mathcal{X}}(\phi_{s_i}) = \mathcal{M}_{\mathcal{X}}(\phi) - \bigcup_{j < i} [x_j]_{\mathcal{E}_j}$$

Proof. This follows by construction of φ_{s_i. □}

Lemma III.2. *At every iteration i of Algorithm 1, and assuming GENERALIZE(x, φ, n) returns n-bounding generalizations of x (as defined in Definition III.1) we have:*

$$\forall x' \in \bigcup_{j \leq i} [x_j]_{\mathcal{E}_j}, |\phi(x', \mathcal{Y}, \mathcal{Z})| \leq n_{m_i}$$

Proof. Let $j \leq i$, and let $x \in [x_j]_{\mathcal{E}_j}$, following Definition III.1, we have |φ_{s_j}(x, \mathcal{Y} , \mathcal{Z})| ≤ n_{m_j.}

Then by construction of φ_{s_i and Property II.5 we have |φ(x, \mathcal{Y} , \mathcal{Z})| ≤ n_{m_j which, as (n_{m_i)_i is increasing, proves the lemma. □}}}

Theorem III.2 (Correctness with an exact #SAT oracle). *Algorithm 1 is correct, i.e., the returned tuple (x_m, n_m) satisfies the following relation:*

$$n_m = |\phi(x_m, \mathcal{Y}, \mathcal{Z})| = \max_{x \in \mathcal{M}_{\mathcal{X}}(\phi)} |\phi(x, \mathcal{Y}, \mathcal{Z})|$$

Proof. Following Property II.1 and since n_{m_k ≥ N_k we have:}

$$\forall x \in \mathcal{M}_{\mathcal{X}}(\phi_{s_k}) \cdot |\phi(x, \mathcal{Y}, \mathcal{Z})| \leq N_k \leq n_{m_k}$$

Then instantiating Lemma III.2 at iteration k we have:

$$\forall x \in \bigcup_{i \leq k} [x_i]_{\mathcal{E}_i}, |\phi(x, \mathcal{Y}, \mathcal{Z})| \leq n_{m_k}$$

Following Lemma III.1, at iteration k we have $\mathcal{M}_{\mathcal{X}}(\phi) = \mathcal{M}_{\mathcal{X}}(\phi_{s_k}) \cup \bigcup_{i \leq k} [x_i]_{\mathcal{E}_i}$, and the result follows. □

C. Correctness with a probabilistic #SAT oracle

Since the termination can be proven in the same way as in the exact case, we only prove the correctness.

Let us first recall the expected guarantees provided by an approximate model counter [8], where the ε parameter characterizes the precision of the result and the δ parameter determines its associated confidence.

Property III.1 (Correctness of the Model Counting). *The count MC_{ε,δ}(φ(x, \mathcal{Y} , \mathcal{Z})) returned by an approximate model counter satisfies the following:*

$$\mathbb{P} \left[\frac{1}{1+\epsilon} \leq \frac{\text{MC}_{\epsilon,\delta}(\phi(x, \mathcal{Y}, \mathcal{Z}))}{|\phi(x, \mathcal{Y}, \mathcal{Z})|} \leq 1+\epsilon \right] \geq 1-\delta$$

These guarantees extend to partial witnesses naturally, i.e., queries of the form MC_{ε,δ}(φ(x| \mathcal{E} , \mathcal{Y} , \mathcal{Z})).

The next theorem proves the correctness of Algorithm 1 in the approximate case and gives the associated *tight* bounds.

Theorem III.3. *Let (x_m, n_m) be the result returned by the call BAXMC_{ε₀,ε₁,δ₀,δ₁,δ₂,κ}(φ(\mathcal{X} , \mathcal{Y} , \mathcal{Z})), and let*

$$M = \max_{x \in \mathcal{M}_{\mathcal{X}}(\phi)} |\phi(x, \mathcal{Y}, \mathcal{Z})|$$

If δ₁ ≤ $\frac{\delta_2}{|\mathcal{X}|+1}$ then:

$$\mathbb{P} \left[\frac{1}{1+\epsilon_1} \leq \frac{n_m}{|\phi(x_m, \mathcal{Y}, \mathcal{Z})|} \leq 1+\epsilon_1 \right] \geq 1-\delta_1$$

and

$$\mathbb{P} \left[\begin{aligned} |\phi(x_m, \mathcal{Y}, \mathcal{Z})| &\geq \frac{M}{(1 + \epsilon_0) * (1 + \epsilon_1) * (1 + \kappa)} \\ &\geq (1 - \delta_1) * \min(1 - \delta_2, 1 - \delta_0) \end{aligned} \right]$$

Proof. Let ϕ_S be the final value of the variable ϕ_s after the last iteration of the **while** loop. We have the following guarantees from the approximate model counter (Property III.1):

$$\mathbb{P} \left[\frac{1}{1 + \epsilon_1} \leq \frac{n_m}{|\phi(x_m, \mathcal{Y}, \mathcal{Z})|} \leq 1 + \epsilon_1 \right] \geq 1 - \delta_1 \quad (1)$$

$$\mathbb{P} \left[\frac{1}{1 + \epsilon_0} \leq \frac{N}{|\mathcal{M}_{\mathcal{Y}}(\phi_S)|} \leq 1 + \epsilon_0 \right] \geq 1 - \delta_0 \quad (2)$$

From Theorem IV.2 regarding the GENERALIZE function (which will be proved in the next section), we also have that for any $x \in \mathcal{M}_{\mathcal{X}}(\phi \wedge \neg\phi_S)$ it holds (assuming that $\delta_1 \leq \frac{\delta_2}{|\mathcal{X}|+1}$):

$$\mathbb{P}[|\phi(x, \mathcal{Y}, \mathcal{Z})| \leq n_m] \geq 1 - \delta_2. \quad (3)$$

After the last iteration of the **while** loop we have that $n_m * (1 + \kappa) \geq N$. Using this and Equation (2) and Property II.5 we get that for any $x \in \mathcal{M}_{\mathcal{X}}(\phi_S)$ it holds

$$\begin{aligned} \mathbb{P}[|\phi(x, \mathcal{Y}, \mathcal{Z})| \leq n_m * (1 + \kappa) * (1 + \epsilon_0)] &\geq \\ \mathbb{P}[|\phi(x, \mathcal{Y}, \mathcal{Z})| \leq N * (1 + \epsilon_0)] &= \\ \mathbb{P}[|\phi_S(x, \mathcal{Y}, \mathcal{Z})| \leq N * (1 + \epsilon_0)] &\geq \\ \mathbb{P}[|\mathcal{M}_{\mathcal{Y}}(\phi_S)| \leq N * (1 + \epsilon_0)] &\geq 1 - \delta_0. \end{aligned}$$

From Equation (3), for any $x \in \mathcal{M}_{\mathcal{X}}(\phi \wedge \neg\phi_S)$ it holds

$$\begin{aligned} \mathbb{P}[|\phi(x, \mathcal{Y}, \mathcal{Z})| \leq n_m * (1 + \kappa) * (1 + \epsilon_0)] &\geq \\ \mathbb{P}[|\phi(x, \mathcal{Y}, \mathcal{Z})| \leq n_m] &\geq 1 - \delta_2. \end{aligned}$$

Hence, for any $x \in \mathcal{M}_{\mathcal{X}}(\phi)$ it holds

$$\begin{aligned} \mathbb{P}[|\phi(x, \mathcal{Y}, \mathcal{Z})| \leq n_m * (1 + \kappa) * (1 + \epsilon_0)] \\ \geq \min(1 - \delta_2, 1 - \delta_0) \end{aligned}$$

and hence

$$\mathbb{P} \left[\frac{n_m}{1 + \epsilon_1} \geq \frac{M}{(1 + \kappa) * (1 + \epsilon_0) * (1 + \epsilon_1)} \right] \geq \min(1 - \delta_2, 1 - \delta_0). \quad (4)$$

Combining this with the Equation (1), we obtain

$$\mathbb{P} \left[|\phi(x_m, \mathcal{Y}, \mathcal{Z})| \geq \frac{M}{(1 + \kappa) * (1 + \epsilon_0) * (1 + \epsilon_1)} \right] \geq (1 - \delta_1) * \min(1 - \delta_2, 1 - \delta_0). \quad \square$$

The following corollary instantiates Theorem III.3 in order to get the standard form (as in Property III.1).

Corollary III.1. *For any $0 < \epsilon, \delta < 1$, if in the call of the BAXMC function, we take as parameters $\epsilon_0 = \epsilon_1 = \kappa =$*

$\sqrt[3]{1 + \epsilon} - 1$, $\delta_0 = \delta_2 = \frac{\delta}{2}$ and $\delta_1 = \frac{\delta}{2 * (|\mathcal{X}|+1)}$, then the result (x_m, n_m) satisfies the following inequalities:

$$\begin{aligned} \mathbb{P} \left[\frac{1}{1 + \epsilon} \leq \frac{n_m}{|\phi(x_m, \mathcal{Y}, \mathcal{Z})|} \leq 1 + \epsilon \right] &\geq 1 - \delta \\ \mathbb{P} \left[|\phi(x_m, \mathcal{Y}, \mathcal{Z})| \geq \frac{M}{1 + \epsilon} \right] &\geq 1 - \delta \end{aligned}$$

where

$$M = \max_{x \in \mathcal{M}_{\mathcal{X}}(\phi)} |\phi(x, \mathcal{Y}, \mathcal{Z})|$$

Proof. It is easy to check that $\epsilon_1 = \sqrt[3]{1 + \epsilon} - 1 \leq \epsilon$, $\delta_1 = \frac{\delta}{2 * (|\mathcal{X}|+1)} < \delta_2 = \frac{\delta}{2} < \delta$, $(1 + \epsilon_0)^3 = 1 + \epsilon$ and $(1 - \delta_1) * (1 - \delta_0) = 1 - \delta_0 - \delta_1 + \delta_0 * \delta_1 > 1 - 2 * \delta_0 = 1 - \delta$. \square

IV. GENERALIZATION ALGORITHM

Algorithm 2 generalizes a single model x with insufficiently high count to a set of models with insufficiently high count. This is much the same that a CDCL loop blocks not only one assignment, but a whole set of assignments.

As shown in Property II.2, generalizing a witness is an instance of the MSMP problem (*Minimal Set subject to a Monotone Predicate*), which can be solved using generic algorithms such as QUICKXPLAIN [9]. Although in theory this should lead to a better algorithm, in practice we observed larger numbers of calls to the #SAT oracle, an issue already identified in other contexts [10].

Algorithm 2 is thus a specific solver of the MSMP problem in our setting, relying on a *linear sweep* over the variables that are part of the valuation.

For efficiency reasons, the steps mentioned in Algorithm 2 are in a precise order. The reason behind this is:

- 1) The first step relies on a consequence of Property II.3, allowing to relax variables with simple calls to a sat solver.
- 2) The log-based generalization is a heuristic allowing to do *big steps* in the generalization process by relaxing multiple variables at each loop turn.
- 3) The *linear sweep* pass generalizes x in such a way that the returned set is minimal, i.e. that none of the further generalizations of the returned value satisfies Definition III.1.

The returned \mathcal{E} is guaranteed only to be a *local minimum* and it may not be the *smallest* set such that Definition III.1 holds because of the order in which we consider variables of \mathcal{X} in Algorithm 2.

A. Correctness and complexity with an exact #SAT oracle

Property IV.1. If $\phi(x|_{\mathcal{E}}[X_i \rightarrow \neg x(X_i)], \mathcal{Y}, \mathcal{Z})$ is UNSAT then:

$$|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| = |\phi(x|_{\mathcal{E} - \{X_i\}}, \mathcal{Y}, \mathcal{Z})|$$

Proof. This follows directly from Property II.3. \square

Let us prove the correctness of Algorithm 2 in the context of an exact #SAT oracle. This will finish the correctness proof started in Section III-B.

Algorithm 2 Pseudocode for the generalization algorithm

```
1: function GENERALIZE $_{\delta}(x, \phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z}), n_m)$ 
2:    $\mathcal{E} \leftarrow \mathcal{X}$ 
3:   for all  $X_i \in \mathcal{X}$  do ▷ Redundancy elimination
4:     if  $\phi(x[X_i \rightarrow \neg x(X_i)], \mathcal{Y}, \mathcal{Z})$  UNSAT then
5:        $\mathcal{E} \leftarrow \mathcal{E} - \{X_i\}$ 
6:     end if
7:   end for
8:    $k \leftarrow \log n_m - \log \text{MC}_{\epsilon, \delta_1}(\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z}))$ 
9:   while  $k > 0 \wedge |\mathcal{E}| > 0$  do ▷ Log-elimination
10:     $\mathcal{A}_k \stackrel{\$}{\leftarrow} \{\mathcal{V} \subseteq \mathcal{E} \mid |\mathcal{V}| = k\}$ 
11:     $c \leftarrow \text{MC}_{\epsilon, \delta_1}(\phi(x|_{\mathcal{E}-\mathcal{A}_k}, \mathcal{Y}, \mathcal{Z}))$ 
12:    if  $c \leq \frac{n_m}{1+\epsilon}$  then
13:       $\mathcal{E} \leftarrow \mathcal{E} - \mathcal{A}_k$ 
14:       $k \leftarrow \log n_m - \log c$ 
15:    else
16:       $k \leftarrow k - 1$ 
17:    end if
18:  end while
19:  for all  $X_i \in \mathcal{X} - \mathcal{E}$  do ▷ Refinement
20:    if  $\text{MC}_{\epsilon, \delta_1}(\phi(x|_{\mathcal{E}-\{X_i\}}, \mathcal{Y}, \mathcal{Z})) \leq \frac{n_m}{1+\epsilon}$  then
21:       $\mathcal{E} \leftarrow \mathcal{E} - \{X_i\}$ 
22:    end if
23:  end for
24: end function
```

Theorem IV.1. *Algorithm 2 terminates and is correct: the returned set \mathcal{E} satisfies Definition III.1, i.e., $|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| \leq n$.*

Proof. In the **while** loop at Line 9 we can see that, at each iteration, either $|\mathcal{E}|$ or k decreases, thus ensuring the termination of the algorithm.

During any update of the temporary value \mathcal{E} (Lines 5, 13 and 21), we ensure that the new value of \mathcal{E} satisfies Definition III.1:

- 1) At Line 5, Property IV.1 keeps the model counting stable.
- 2) At Lines 13 and 21, the update is guarded by the explicit check of the property (in the **if** statement Lines 12 and 20).

Hence the correctness follows. \square

B. Bounds with an approximate #SAT oracle

Theorem IV.2. *Let $\mathcal{E} \subseteq \mathcal{X}$ be the set returned by the call $\text{GENERALIZE}_{\delta}(x, \phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z}), n)$, and assume that*

$$\mathbb{P}[|\phi(x, \mathcal{Y}, \mathcal{Z})| \leq n] \geq 1 - \frac{\delta}{|\mathcal{X}| + 1}$$

Then:

$$\mathbb{P}[|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| \leq n] \geq 1 - \delta$$

Proof. Using Property IV.1, the variable \mathcal{E} after the first loop within Algorithm 2 satisfies $|\phi(x, \mathcal{Y}, \mathcal{Z})| = |\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})|$.

We denote by $C_{x|_{\mathcal{V}}}$ the value returned by the call $\text{MC}_{\epsilon, \delta_1}(\phi(x|_{\mathcal{V}}, \mathcal{Y}, \mathcal{Z}))$. Since each time we update \mathcal{E} to a set \mathcal{V} we ensure $C_{x|_{\mathcal{V}}} \leq \frac{n}{1+\epsilon}$, we have the following probability:

$$\mathbb{P}[|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| \leq n] \geq 1 - \delta_1$$

Let \mathcal{E}_l denote the value obtained after l updates of variable \mathcal{E} during LOG-ELIMINATION and REFINEMENT steps within Algorithm 2 and let us denote by P_l the probability that the set \mathcal{E}_l is approximately n -bounding.

Using that we update \mathcal{E} to the value \mathcal{E}_l only if $C_{x|_{\mathcal{E}_l}} * (1 + \epsilon) \leq n$, we have the following recursive relation:

$$\begin{aligned} P_l &= \mathbb{P}[|\phi(x|_{\mathcal{E}_l}, \mathcal{Y}, \mathcal{Z})| \leq n] * P_{l-1} \\ &\geq (1 - \delta_1) * P_{l-1} \geq (1 - \delta_1)^l * P_0 \\ &\geq (1 - \delta_1)^l * \left(1 - \frac{\delta}{|\mathcal{X}| + 1}\right) \end{aligned}$$

Thus, as $l \leq |\mathcal{X}|$, if we take $\delta_1 = \frac{\delta}{|\mathcal{X}| + 1}$ and we call the #SAT oracle with parameters $(\epsilon, \frac{\delta}{|\mathcal{X}| + 1})$ we get:

$$\mathbb{P}[|\phi(x|_{\mathcal{E}}, \mathcal{Y}, \mathcal{Z})| \leq n] \geq (1 - \delta_1)^{|\mathcal{X}|} \geq 1 - (|\mathcal{X}| + 1) * \delta_1 \geq 1 - \delta$$

\square

Remark IV.1. The bound with respect to the number of updates is tight. The worst case is reached when the only valid subset of \mathcal{X} is \mathcal{X} itself, that is when the model cannot be generalized.

V. BREAKING SYMMETRIES IN MAX#SAT

Symmetries are a special kind of permutations of the input variables of a formula leaving it intact. Exploiting or breaking symmetries in SAT formulas has long been a topic of interest.

For instance, if a formula is left intact by such a permutation then for each blocking clause C , the solver may need to generate the full orbit of C by the group of permutations, leading to combinatorial explosion. Breaking the symmetry means selecting one solution per orbit by adding a predicate called *symmetry breaking predicate* to the formula, purposefully generated to break the symmetries. The resulting formula is equisatisfiable, but often simpler to solve.

A. Correctness in the presence of symmetries

In our context, handling symmetries within the witness set reduces the size of the search space, and leads to better complexity. We give in this section arguments about why this is true.

Definition V.1. Given a Boolean formula $\phi(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$, a *symmetry* of ϕ is a bijective function $\sigma : \overline{\mathcal{X}} \mapsto \overline{\mathcal{X}}$ that preserves negation, that is $\sigma(\neg X) = \neg \sigma(X)$, and such that, when σ is lifted to formulas, $\sigma(\phi) = \phi$ syntactically [11].

S_{ϕ} denotes the set of all symmetries of ϕ . We lift S_{ϕ} to models by defining the set of symmetries of a model x , $S_{\phi}(x) = \{x \circ \sigma \mid \sigma \in S_{\phi}\}$.

Theorem V.1. *In Algorithm 1, picking only one x per symmetry class of ϕ preserves the correctness of the algorithm both in the exact and approximate case.*

Proof. Whatever the method used to select only one member of each symmetry class, this corresponds to creating a symmetry breaking predicate $\psi(\mathcal{X})$ and solving the problem over $\phi \wedge \psi$, and thus the Property II.5 applies. \square

B. Implementing Max#SAT symmetry breaking

We detect symmetries in ϕ using the automorphisms of a colored graph representing the formula, defined as follows:

- For each variable, create two nodes: one for the positive literal, and one for the negative literal. Use color 0 if the variable is in X , otherwise use color 1. Add an edge (*Boolean consistency edge*) between the two nodes.
- For each clause, create a node, and assign to it the color 2. Add an edge between this clause node and every node corresponding to a literal present in the clause.

Many tools can be used in order to list the automorphisms of a graph. In our case, we used BLISS [12] because of its C++ interface, and its performance.

After detecting the symmetries, one can use any symmetry breaking technique available, either static [13] or dynamic [6]. In our implementation, we chose to use CDCLSYM [6] because of its ease of use, and because it avoids generating complex symmetry breaking predicates ahead of time.

VI. HEURISTICS AND OPTIMIZATIONS

We present in this section heuristics used in both Algorithms 1 and 2 in practice, and discuss their effectiveness.

A. Progressive construction of the candidate

A simple yet effective optimization is to gradually add literals to the candidate x in Algorithm 1 at Line 7. By stopping earlier, this allows to call GENERALIZE on a partial assignment instead of a complete one, and will decrease the number of calls to the #SAT oracle as it anticipates work that is done in Algorithm 2.

B. Leads

When performing the generalization in Algorithm 2, one can see that we can extract *hints* about promising parts of the search space when relaxing variables. Indeed, when relaxing parts of the solution (Lines 16 and 22), if the model count of the relaxation goes above n_m , then this part of the search space may contain an improvement over the current solution.

Following this intuition, one can hold a *sorted list*¹ of relaxations whose count is above the current best known maximum, and use it to favor parts of the search space that look promising. We call these promising relaxations *leads*. More formally, given $\tilde{x}|_{\mathcal{E}}$ a lead, when searching for a new solution in Algorithm 1 at Line 7, instead of searching in $\mathcal{M}_{\mathcal{X}}(\phi_s)$, one would search in $\mathcal{M}_{\mathcal{X}}(\phi_s) \cap [\tilde{x}]_{\mathcal{E}}$.

Let $L_n(\phi)$ denote the set of leads currently known to the solver with count lower than n . When the currently known

maximum is improved in Algorithm 1 at Line 10, we can block all leads whose count is below the new maximum:

$$\phi_{s_{i+1}} = \phi_{s_i} \wedge \bigwedge_{[\tilde{x}]_{\mathcal{E}} \in L_{n_m}(\phi)} \neg(\tilde{x}|_{\mathcal{E}})$$

C. Decision heuristic

As discussed in Section IV-A, the performances of the algorithm depend on the order with which variables are considered in various parts of the solving process (in the generalization and during the optimization presented in Section VI-A). One can see that this kind of problem, that we call *variable scheduling*, is actually predominant when solving SAT problems, and even #SAT problems.

One first heuristic arises from the leads described in Section VI-B. One can use the leads list as indications for literals leading to promising parts of the search space, by finding the literal which appears the most in the leads. We call this heuristic *leads*.

Another decision heuristic can be devised using VSIDS [14]. The idea is to assign a weight to each literal based on its last appearance in a blocking clause. The weight of each literal is increased by a constant amount every time the literal appears in a blocking clause, and is multiplicatively decreased at each blocking clause. This heuristic showed promising results in both SAT and #SAT [15]. We call this heuristic *vsids*.

One could also choose the next decision variable at random, which we call *rnd*. And finally, one could just pick the decision variables in the order they are provided to the tool, which we call *none*.

An experimental evaluation is done in Section VII-B.

D. Handling equivalent literals

Equivalent literals are a notorious property of Boolean formulas which, when exploited, results generally in better runtime performances [16].

Definition VI.1. Given a Boolean formula ϕ , we say that two literals $L_i \in \bar{\mathcal{V}}$ and $L_j \in \bar{\mathcal{V}}$ are *equivalent* if $\phi \models L_i \Leftrightarrow L_j$.

Equivalent literals allow to simplify formulas based on the following theorem.

Theorem VI.1. Let ϕ be a Boolean formula and two equivalent literals L_i and L_j . Then solving the Max#SAT problem for ϕ is reduced to solving the Max#SAT problem for the simpler formula ϕ' obtained by replacing all occurrences of L_j (resp. $\neg L_j$) by L_i (resp. $\neg L_i$) when:

- 1) either L_i and L_j are in the same literal class (either $\bar{\mathcal{X}}$, $\bar{\mathcal{Y}}$ or $\bar{\mathcal{Z}}$)
- 2) or $L_i \in \bar{\mathcal{X}}$ and $L_j \in \bar{\mathcal{Y}} \cup \bar{\mathcal{Z}}$
- 3) or $L_i \in \bar{\mathcal{Y}}$ and $L_j \in \bar{\mathcal{Z}}$.

Theorem VI.1 can be applied multiple times in order to further simplify the formula. Literal equivalence can be detected using binary implication graphs [17].

¹The order to use here is: first the count of the relaxation, then the size of the relaxation.

VII. EXPERIMENTAL EVALUATION

Algorithm 1 has been implemented in an open-source tool written in C++ called BAXMC [7], including dynamic symmetry breaking techniques (Section V) and all the heuristics discussed in Section VI. In this implementation, we only incorporated the approximate version of the algorithm using APPROXMC5 [18] as an approximated model counting oracle and CRYPTOMINISAT [19] as a SAT solver oracle. An exact solver is not implemented because we do not, at the time of writing, have another exact Max#SAT solver available as a comparison.

We use three sets of benchmarks, coming either from [20], or from MaxSat 2021 competition [21]. Benchmarks from this later class are transformed using the method from [1]. Table I shows more details about the benchmark set considered. Benchmarks annotated with a star indicate that a symmetry was found.

All experiments are run on a Dell R640 with 40 cores and 192 GB of RAM running Debian 11, with a 2-hour timeout, a 10 GB memory limit and with parameters $\delta = 0.2$, $\epsilon = 0.8$.

A. Comparison to MAXCOUNT

MAXCOUNT [1] is used as an off-the-shelf solver of the problem, with parameters corresponding to $\delta = 0.2$, $\epsilon = 0.8$. Note that these are not the parameters used in the experiments in [1] and that we reimplemented MAXCOUNT using newer oracles. We did this in order to see how MAXCOUNT and BAXMC behave when both are providing the same correctness guarantees and using the same oracles for fairness. All figures from Table II are obtained when BAXMC is used with the `(leads, rnd)` heuristic combination.

Table II shows the results obtained when running both tools on our three benchmarks. Bolded values are the *best* values on this line (i.e., smaller time or biggest answer). The *time* columns are the running times of the tools. The *model count* columns are the values returned by the candidate tools.

One can see that BAXMC outperforms MAXCOUNT in all benchmark timings. In cases where BAXMC did not find the best value, it terminates when the bounds on the possible maximum are *tight enough*. This yields a small error margin on the returned value of BAXMC, but is configurable through its κ argument.

B. Decision heuristic comparison

Table III shows a comparison between the heuristics that are currently available in BAXMC. Lines enumerate the decision heuristics from Section VI-C. Columns specify heuristics used by the underlying SAT oracle about literals polarities.

Each cell of this table contains, in sequence: the total running time, the number of time this combination ran the fastest compared to all others, and the number of times this combination timed out. For example combination `(leads, cache)` ran for a total time of 62968.38 seconds with 7 timeouts, and ran the fastest on 3 benchmarks over a total number of 26. In this setup, any time-out from BAXMC increases the total running time by 7200s.

The table shows that none of the heuristics stands out. We can only eliminate random decision as a bad heuristic. Nevertheless, the combination of heuristics allows to strongly reduce the overall number of timeouts.

VIII. RELATED WORKS

Previous works on Max#SAT solving may be classified into three categories, based respectively on probabilistic solving as in MAXCOUNT [1], exhaustive search [5] and knowledge compilation [22].

Probabilistic solving relies on “amplification” to build a new formula $\tilde{\phi}(\mathcal{X}, \mathcal{Y}, \mathcal{Z}) = \bigwedge_{i=1}^k \phi(\mathcal{X}, \mathcal{Y}_i, \mathcal{Z}_i)$, where the \mathcal{Y}_i and \mathcal{Z}_i are fresh copies of the initial \mathcal{Y} and \mathcal{Z} variables, and uniformly sampling among $\mathcal{M}_{\mathcal{X}}(\tilde{\phi})$. The higher the k , the more the sampling is attracted towards the \mathcal{X} with large projected model counting over $(\mathcal{Y}_i)_{i \leq k}$. Given parameters ϵ and δ , the guarantees provided about the returned tuple (\tilde{n}, \tilde{x}) are the same as in Corollary III.1 [1]. Unfortunately, when the size of the formula increases, uniform sampling may become quite expensive as shown in our benchmarks. Furthermore, this approach is not incremental: looking for a better solution involves re-running the search from scratch.

On the other side of the spectrum lie *exhaustive searches*. The idea here is to make incremental decisions among the variables in \mathcal{X} , propagating the decision in ϕ , and simplifying the formula in order to cache some results [5]. Such approaches are exact, but their exhaustive nature limits their scalability. Component caching [23] is a practical way to improve scalability [5] and it could be beneficial into our algorithm too.

Knowledge compilation consists in compiling the formula into a representation over which solving the problem (here, the optimal model counting) is expected to be much easier. Compilation times tend to dominate and the memory usage of the compiled form may be huge.

A possible approach could use a generalization of \mathcal{X} -constrained SDDs [22]. The idea here would be to build $(\mathcal{X}, \mathcal{Y})$ -constrained SDDs, that is SDDs that are \mathcal{X} -constrained, and for which each subtree that are not over \mathcal{X} are \mathcal{Y} -constrained. In this case, one can easily compute the count of every possible pair $x \in \mathcal{M}_{\mathcal{X}}(\phi)$ and then propagate the maximum to the root of the tree. To the best of our knowledge, this direction has not been explored yet.

IX. CONCLUSION AND FUTURE WORK

We proposed a CEGAR based algorithm allowing to solve medium-sized instances of the Max#SAT within reasonable time limits, as illustrated in our experiments. This algorithm allows either to compute exact solutions (when possible), or can be smoothly relaxed to produce approximated results, under well-defined probabilistic guarantees. Comparisons with an existing probabilistic tool showed the gains provided by our algorithm on concrete examples. Our implementation and all the related benchmarks are available on [7].

From an algorithmic point of view this work could be extended in several directions.

Table I
BENCHMARK LIST

Name	$ \mathcal{X} $	$ \mathcal{Y} $	$ \mathcal{Z} $	Nr. Clauses
backdoor-32-24*	32	32	83	76
backdoor-2x16-8*	32	32	136	272
pwd-backdoor	64	64	272	609
bin-search-16	16	16	1416	5825
CVE-2007-2875	32	32	720	1740
CVE-2009-3002	288	240	443	180
reverse	32	32	165	293
ActivityService	70	34	4063	15257
ActivityService2	70	34	4063	15257
ConcreteActivityService	71	37	4728	17856
GuidanceService	69	27	3167	11612
GuidanceService2	69	27	3167	11612
IssueServiceImpl	77	29	3519	13024
IterationService	70	34	4063	15257
LoginService	92	27	5110	21559
NotificationServiceImpl2	87	32	5223	22006
PhaseService	70	34	4063	15257
ProcessBean	166	39	9675	41444
ProjectService	134	48	6778	24944
sign	16	16	107	392
sign_correct	16	16	92	346
UserServiceImpl	87	31	3901	14653
drmx	1030	17	26	2094
keller4	43	15	62	2525
g2_n35e34_n58e61	34	7	954	38130

Table II
PERFORMANCE COMPARISONS BETWEEN BAXMC AND MAXCOUNT

Benchmark name	BAXMC			MAXCOUNT	
	Time (s)	Sym. Time (s)	Model count (log)	Time (s)	Model count (log)
backdoor-32-24*	611.12	34.50	32	231.87	32
backdoor-2x16-8*	60.02	61.07	16	6512.28	16
pwd-backdoor	236.87	240.63	64	TO	-
bin-search-16	1067.38	1048.43	16	1490.44	16
CVE-2007-2875	36.14	37.39	32	TO	-
CVE-2009-3002	TO	TO	-	MO	-
reverse	TO	TO	-	MO	-
ActivityService	3060.39	3064.60	33.95	TO	-
ActivityService2	3096.54	2999.72	33.95	TO	-
ConcreteActivityService	84.20	84.44	36.91	TO	-
GuidanceService	1468.39	1474.51	26.88	TO	-
GuidanceService2	1459.74	1474.76	26.88	TO	-
IssueServiceImpl	1603.21	1583.50	28.88	TO	-
IterationService	3081.86	3068.95	33.95	TO	-
LoginService	5275.25	5197.84	26.92	TO	-
NotificationServiceImpl2	1286.48	1287.94	31.91	TO	-
PhaseService	3071.86	3105.18	33.95	TO	-
ProcessBean	TO	TO	-	TO	-
ProjectService	5770.26	5544.02	47.92	TO	-
sign	73.56	73.43	15.90	819.58	16
sign_correct	74.58	73.78	15.89	819.56	16
UserServiceImpl	TO	TO	-	TO	-
drmx	24.39	24.07	16.99	TO	-
keller4	TO	TO	-	TO	-
g2_n35e34_n58e61	0.17	0.41	2.53	TO	-

Table III
PERFORMANCE COMPARISON BETWEEN HEURISTICS OF BAXMC

	cache	neg	pos	rnd
leads	62968.38 – 3 – 7	65542.94 – 2 – 6	65222.40 – 1 – 4	67233.96 – 0 – 5
rnd	144081.73 – 0 – 19	139002.15 – 0 – 18	140755.04 – 0 – 17	137407.70 – 0 – 17
none	60368.28 – 3 – 5	62729.76 – 2 – 4	61317.54 – 3 – 4	56860.50 – 3 – 4
vsids	69165.19 – 2 – 8	56189.26 – 1 – 5	54017.07 – 3 – 4	63865.03 – 2 – 6

First, we exploited some classes of symmetries when solving Max#SAT (Section V). This could be improved by detecting new kinds of symmetries [13], or exploiting them further using techniques such as symmetry propagation [24].

As discussed in Section IV, our relaxation algorithm (Algorithm 2) uses a *linear sweep* over the literals composing a witness. Instead of returning one possible minimal relaxation, MERGEXPLAIN [25] returns multiple ones, which may be helpful in our case by allowing the creation of multiple blocking clauses.

As expected, in some instances, our algorithm may degenerate into exhaustive search. While we do not know yet any characterization of all such instances, we believe that pre-processing and in-processing [26] techniques such as UNHIDING [17] should improve performances and limit the set of inefficient instances.

Finally, Algorithm 1 may be parallelized by correctly scheduling search spaces among threads, possibly using the leads described in Section VI-B. If we enforce the fact that all leads currently present in the lead list are disjoint, that is the $[\tilde{x}]_{\mathcal{E}}$ are pairwise disjoint (hence splitting the search space into parts), we expect a favorable parallelization setting.

REFERENCES

- [1] D. Fremont, M. Rabe, and S. Seshia, “Maximum model counting,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.
- [2] S. Saha, W. Eiers, I. B. Kadron, L. Bang, and T. Bultan, “Incremental attack synthesis,” *ACM SIGSOFT Software Engineering Notes*, vol. 44, no. 4, pp. 16–16, 2021.
- [3] D. Monniaux, “NP^{#P} = \exists PP and other remarks about maximized counting,” <https://hal.archives-ouvertes.fr/hal-03586193>, Feb. 2022.
- [4] J. Torán, “Complexity classes defined by counting quantifiers,” *J. ACM*, vol. 38, no. 3, pp. 753–774, 1991.
- [5] G. Audemard, J.-M. Lagniez, M. Miceli, and O. Roussel, “Identifying soft cores in propositional formulae,” 2022.
- [6] H. Metin, S. Baair, M. Colange, and F. Kordon, “Cdclsym: Introducing effective symmetry breaking in sat solving,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 99–114.
- [7] “Baxmc website.” [Online]. Available: <https://www-verimag.imag.fr/~vigourth/research/baxmc/>
- [8] S. Chakraborty, K. S. Meel, and M. Y. Vardi, “A scalable approximate model counter,” in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 200–216.
- [9] U. Junker, “QuickXplain: Conflict detection for arbitrary constraint propagation algorithms,” in *IJCAI’01 Workshop on Modelling and Solving problems with constraints*, vol. 4. Citeseer, 2001.
- [10] H. Zhang, “Combinatorial designs by sat solvers 1,” in *Handbook of Satisfiability*. IOS Press, 2021, pp. 819–858.
- [11] D. Monniaux, “Quantifier elimination by lazy model enumeration,” in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. B. Jackson, Eds., vol. 6174. Springer, 2010, pp. 585–599.
- [12] T. Junttila and P. Kaski, “Engineering an efficient canonical labeling tool for large and sparse graphs,” in *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007, pp. 135–149.
- [13] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker, “Improved static symmetry breaking for sat,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2016, pp. 104–122.
- [14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 530–535.
- [15] T. Sang, P. Beame, and H. Kautz, “Heuristics for fast exact model counting,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 226–240.
- [16] Y. Lai, K. S. Meel, and R. H. Yap, “The power of literal equivalence in model counting,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 5, 2021, pp. 3851–3859.
- [17] M. J. Heule, M. Järvisalo, and A. Biere, “Efficient cnf simplification based on binary implication graphs,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2011, pp. 201–215.
- [18] K. S. Meel and S. Akshay, “Sparse hashing for scalable approximate model counting: theory and practice,” in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2020, pp. 728–741.
- [19] M. Soos, K. Nohl, and C. Castelluccia, “Extending SAT solvers to cryptographic problems,” in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 244–257. [Online]. Available: https://doi.org/10.1007/978-3-642-02777-2_24
- [20] “Maxcount 1.0.0.” [Online]. Available: <https://github.com/dfremont/maxcount>
- [21] “Maxsat evaluation 2021.” [Online]. Available: <https://maxsat-evaluations.github.io/2021/index.html>
- [22] U. Oztok, A. Choi, and A. Darwiche, “Solving pp pp-complete problems using knowledge compilation,” in *Fifteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2016.
- [23] F. Bacchus, S. Dalmao, and T. Pitassi, “Dpll with caching: A new algorithm for #sat and bayesian inference,” in *Electronic Colloquium in Computation Complexity*. Citeseer, 2003.
- [24] H. Metin, S. Baair, and F. Kordon, “Composing symmetry propagation and effective symmetry breaking for sat solving,” in *NASA Formal Methods Symposium*. Springer, 2019, pp. 316–332.
- [25] K. Shchekotykhin, D. Jannach, and T. Schmitz, “Mergexplain: Fast computation of multiple conflicts for diagnosis,” in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
- [26] M. Järvisalo, M. J. Heule, and A. Biere, “Inprocessing rules,” in *International Joint Conference on Automated Reasoning*. Springer, 2012, pp. 355–370.

Compact Symmetry Breaking for Tournaments

Evan Lohn , Chris Lambert, and Marijn J.H. Heule 
 Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
 {evanlohn,chrislambert,marijn}@cmu.edu

Abstract—Isolators are a useful tool for reducing the computation needed to solve graph existence problems via SAT. We extend techniques for creating isolators for undirected graphs to the tournament (complete, directed) case, noting several parallels in properties of isolators for the two classes. We further present an algorithm for constructing n -vertex tournament isolators with $\Theta(n \log n)$ unit clauses. Finally, we show the utility of our new isolators in computations of tournament Ramsey numbers.

Index Terms—Satisfiability, Symmetry-breaking, Directed-graphs, Tournaments, Isolators.

I. INTRODUCTION

In recent years, SAT solvers have been used to solve several difficult combinatorial problems [1]–[3]. However, naive encodings of SAT problems often include undesired symmetries, i.e. certain matching subsets of variables that result in equivalent subproblems when given equivalent assignments. To prove the original formula unsatisfiable, in the worst case a solver must search through all possible symmetric parts of the problem space, which slows the generation of unsatisfiable proofs unnecessarily. Similarly, while the solver tries to find a satisfying assignment, symmetries in the input formula may cause the solver to effectively re-explore the same part of the search space even after proving the lack of a solution in a symmetric part of the problem.

The most common way of reducing the impact of symmetries in a given formula is by adding a set of new clauses called a Symmetry-Breaking Predicate (SBP) to the formula before solving [4]–[6]. The goal of a SBP is to preserve the satisfiability of the formula while removing from consideration any regions of the search space known to be symmetric to other regions. In this work we focus on SBP’s for *graph existence problems*, which are problems that can be solved by checking if a graph with a particular structure exists. Solving such problems is an active area of research [7]–[9]. A large class of problem symmetries in graph existence problems naturally results from the existence of isomorphic labeled graphs. These symmetries exist independent of any desired graph property related to graph structure. Rather, they occur because SAT solvers must search the space of labeled graphs in order to prove the (non-)existence of an unlabeled graph. A SBP that targets graph isomorphisms is known as an isolator. Isolators that break many symmetries with few clauses are most useful in practice, as SAT solvers generally take longer to solve formulas with more clauses. Such isolators are often described as “short”, “small”, or “compact.”

Prior work has shown that it is possible to generate small isolators for undirected graphs [10]. The present work instead handles the generation of short isolators for tournaments: complete, directed graphs. There are several mathematically interesting questions one can ask about tournaments that motivate the generation of tournament isolators. For example, Sumner’s conjecture and various election models in social choice theory rely on tournament properties [11], [12]. Tournament isolators can also aid in the search for doubly-regular tournaments. Doubly-regular tournaments are a class of tournaments that (among many other properties) can be efficiently transformed to skew-symmetric Hadamard matrices [13], which have a wide array of practical uses. However, the most well-known question about tournament structure is the Tournament Ramsey number problem, an analog to Ramsey numbers [14] that asks the question of “in what size tournament n must a transitive subtournament of size k exist.” A (sub)tournament is *transitive* if it contains no cycles. Calculating the tournament Ramsey number for $k = 7$ is likely the limit of currently known techniques, and doing so would be impactful for the mathematical community.

The first contribution of this work is the generation of compact tournament isolators that asymptotically match the search space reduction of a perfect isolator. Second, we present a methodology for the generation of compact isolators for small tournaments that extends prior work on undirected tournaments [10]. Finally, we demonstrate the practical usage of our small isolators for finding larger graphs relevant to the search for tournament Ramsey numbers.

II. PRELIMINARIES

We define the following common concepts from SAT literature: A *literal* is either a variable or a negated variable. We use \neg to denote negation. A *clause* is a disjunction of literals. A *unit clause* (sometimes referred to as simply a *unit*) is a clause containing exactly one literal. A *Conjunctive Normal Form (CNF) formula* is a conjunction of clauses. Unless otherwise specified, “formula” refers to “CNF formula.” An *assignment* α is a function from variables to truth values (True/False). α satisfies a formula F if the boolean function denoted by F returns True given the inputs specified by α .

We also define several graph-theoretic concepts. A tournament $G = (V, E)$ is a complete directed graph; more formally, $\forall (v_1, v_2) \in V \times V, v_1 \neq v_2 \rightarrow ((v_1, v_2) \in E) \oplus ((v_2, v_1) \in E)$ and $\forall v \in V, (v, v) \notin E$, where \oplus is the XOR operation. The phrase “ G is an n -vertex tournament” means $|V| = n$. Given

an n -vertex tournament $G = (V, E)$ and a permutation π on V , $\pi(G)$ is defined as $\pi(G) = (V, \{(\pi(v_1), \pi(v_2)) \mid (v_1, v_2) \in E\})$ and is colloquially referred to as applying π to G . Two n -vertex tournaments G_1, G_2 are *isomorphic* (written $G_1 \simeq G_2$) exactly when there exists a permutation π on the vertices of G_1 such that $\pi(G_1) = G_2$. When any such π exists, it is referred to as an *isomorphism* between G_1 and G_2 . The *isomorphism class* (also, *equivalence class*) I_G of a tournament G is defined as $I_G = \{G' \mid G' \simeq G\}$. An *automorphism* π on a tournament G is any permutation π such that $\pi(G) = G$. The set of automorphisms of G form a group under function composition. This group is referred to as $Aut(G)$.

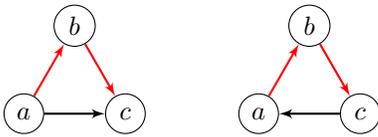
III. ISOLATOR NOTATION AND CONCEPTS

To search for a tournament G satisfying some structural property, we define variables with the semantics “edge e exists in graph G ” for use in a formula F . We say that F *admits* a graph G' exactly when there exists a satisfying assignment to the conjunction of F and the set of unit clauses semantically implied by the edges of G' . An *isolator* for n -vertex tournaments is a formula F that admits at least one tournament from each equivalence class on n -vertex tournaments. A *perfect* isolator is an isolator that admits *exactly* one tournament from each equivalence class. A perfect isolator F is *optimal* if there does not exist a perfect isolator with fewer non-unit clauses than F . A *compact* or *short* isolator is not rigorously defined. Rather, it describes an isolator with few enough non-unit clauses to be of practical use in solving SAT problems.

In this work, vertices will be denoted with lowercase letters a, b, c, \dots or with v_1, v_2, \dots, v_n when an ordering of the vertices is relevant. Arcs (directed edges) will be referred to with (u, v) , meaning “there is an arc from u to v .” In our construction of isolators, each variable is written in the form uv and has the semantics “arc (u, v) exists in the graph.” Note that the literal $\neg uv$ therefore means “arc (v, u) exists in the graph.”

A. Short Isolator Examples

Consider the following two labeled 3-vertex tournaments.



These tournaments represent the only two equivalence classes for $n = 3$ tournaments: a cycle and a transitive tournament. While any combination of a cycle and transitive tournament would suffice to represent both equivalence classes, the tournaments chosen above have the interesting property of sharing two edges ab and bc (colored red). This property allows us to produce a short formula that admits both graphs:

$$ab \wedge bc.$$

This formula admits exactly one of the two labeled cycles and one of the six labeled transitive tournaments on 3 vertices,

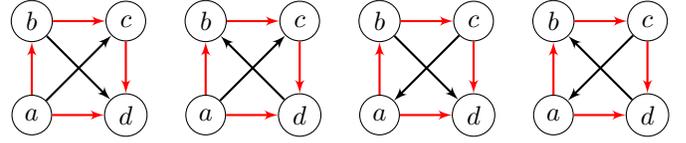


Fig. 1. All isomorphism class representatives admitted by a perfect, optimal isolator for 4-vertex tournaments. Red edges are edges fixed by unit clauses of the isolator, and the isolator has only unit clauses.

and does so with the fewest possible clauses. Therefore, $ab \wedge bc$ is a perfect, optimal isolator for $n = 3$ tournaments.

Figure 1 displays canonical representatives of all 4 isomorphism classes for $n = 4$ tournaments. We note that once again all highlighted edges have the same edge labels across graphs, and all permutations of non-highlighted edges are present. So, a short formula that admits exactly the set of graphs in the figure is

$$ab \wedge bc \wedge cd \wedge ad.$$

While the optimal isolators for $n = 3, 4$ are comprised entirely of unit clauses, this pattern does not hold for $n = 5$. Table I contains the number of unit and non-unit clauses for our isolators on $n \leq 8$ vertices.

B. Comparison of undirected graph and tournament isolators

Although the majority of this work focuses on tournament isolators, there are many interesting parallels between undirected and tournament isolators. In an undirected context, the *existence* of edge (u, v) is denoted by the literal uv , while its nonexistence is denoted by the literal’s negation $\neg uv$. Because edgeless and complete graphs are isomorphism classes for any n in the undirected case, every clause of an undirected graph isolator containing only arc literals must contain at least one positive and one negative literal. These two graphs do not exist in the case of tournaments; the closest parallel is transitive tournaments. Unlike the set of n -vertex undirected graphs which contains exactly one empty graph and one complete graph with $n!$ automorphisms each, there are $n!$ isomorphic transitive tournaments on n vertices. It is possible to select the particular transitive tournament TT that an isolator admits by ensuring that at least one edge from TT is present in each clause of the isolator. A simple way to do so is ensure each clause contains at least one edge uv s.t. $u < v$ in vertex numbering.

One consequence of undirected isolators requiring at least one positive and one negative literal per clause is that undirected isolators have no unit clauses. However, while negating all literals in an undirected isolator produces another undirected isolator (because the existence and non-existence of an edge is symmetric), there is no direct parallel to be found in tournaments as edge directionality does not have this property.

Another interesting difference between undirected graphs and tournaments is the low number of isomorphism classes for tournaments when n is small (see table I). Intuitively, this happens because it is “easier” for tournaments to be isomorphic. The two options for the edge between vertices

u and v in the undirected case are uv existing or not existing. Crucially, an undirected graph G will never be isomorphic to G' constructed by adding or removing an edge of G , which is an operation that can be seen as “flipping” an edge to its other possibility. However, “flipping” an edge of a tournament T by changing the edge’s direction will produce $T' \simeq T$ iff the two vertices u and v of the flipped edge had the same edges to the rest of the graph (the isomorphism is via the permutation that swaps u and v). Although this discrepancy exists for small n , the numbers of isomorphism classes for undirected graphs and tournaments are remarkably close for larger n (see OEIS A000088, A00056 [15]). Therefore, we expect that perfect, optimal isolators for undirected graphs and tournaments will have similar numbers of clauses for larger n .

C. Arc Literal Numbering

Each uv must be assigned a corresponding integer to conform to the commonly used DIMACS CNF format. To do so, we specify a function $idx_n(u, v)$ to map each possible arc (u, v) in an n -vertex tournament to a unique integer identifier. Because exactly one of (u, v) and (v, u) must exist for any two vertices u, v , idx must satisfy $idx_n(u, v) = -idx_n(v, u)$. To facilitate isolator comparisons across different n , idx also should satisfy the property that $idx_n(u, v) = idx_{n+1}(u, v)$. We therefore drop the subscript n when referring to $idx_n(u, v)$ in the future, as its value does not depend on n .

In particular, idx is inductively defined as follows for an $n + 1$ -vertex tournament with vertices $v_1, v_2, \dots, v_n, v_{n+1}$. Let $K = n(n - 1)/2$ be the largest output of idx for an n -vertex tournament (implying base case $idx(v_1, v_2) = 1$ when $n = 2$). Applying idx to each of the arcs $(v_1, v_{n+1}), (v_2, v_{n+1}), \dots, (v_n, v_{n+1})$ yields $K + 1, K + 2, \dots, K + n$, respectively. All arcs not included in this definition are of the form (v_w, v_u) where $w > u$, and are defined by the earlier mentioned constraint of $idx(u, v) = -idx(v, u)$.

IV. UNIT CLAUSES

In practice, SAT solvers immediately reduce formulas with unit clauses to shorter formulas without units via unit propagation. Additionally, each unit clause reduces the size of the search space by a factor of 2. Therefore, it is practically useful to create isolators with as many units as possible. The following sections detail and analyze our various methods for creating isolators with many unit clauses.

A. Provable Units

While constructing smaller isolators using the techniques above, we opted to manually inspect our results and see what patterns they shared. In doing so, we rediscovered a well-known fact from graph theory literature; every tournament contains a Hamiltonian path [16]. Proof sketch: inductively consider a length n Hamiltonian path v_1, v_2, \dots, v_n in an $n + 1$ -vertex tournament $G = (V, E)$. For the vertex v_{n+1} not part of the path, in the case that either (v_{n+1}, v_1) or (v_n, v_{n+1}) is in E , a length $n + 1$ Hamiltonian path is formed. Otherwise, (v_1, v_{n+1}) and (v_{n+1}, v_n) are in E and thus there must

exist consecutive vertices v_i, v_{i+1} in the Hamiltonian path such that arcs (v_i, v_{n+1}) and (v_{n+1}, v_{i+1}) are in E . In this case, the sequence $v_1, v_2, \dots, v_i, v_{n+1}, v_{i+1}, \dots, v_n$ forms a length $n + 1$ Hamiltonian path. As a result of this property, a set of unit clauses describing a Hamiltonian path on an n -vertex tournament is always a valid n -vertex isolator.

Given the utility of unit clauses in isolators, it is natural to ask how many units there can possibly be in an n -vertex isolator. As it turns out, there is a long-known result from graph theory that implies that asymptotically there are at most $O(n \log n)$ units possible. By the orbit-stabilizer theorem, the size of the equivalence class of a graph G on n vertices is $\frac{n!}{|Aut(G)|}$, where $Aut(G)$ is the set of distinct automorphisms of G . In 1963 Erdős and Rényi proved that as n approaches infinity, the proportion of undirected graphs of size n with nontrivial automorphisms approaches 0 [17]. The same result for tournaments directly follows. Therefore, a proportion of tournaments approaching 1 has equivalence classes of size $n!$, so the asymptotic number of equivalence classes is

$$\frac{2^{\binom{n}{2}}}{n!} \in \Theta\left(\frac{2^{\binom{n}{2}}}{2^{n \log n}}\right) = \Theta(2^{\binom{n}{2} - n \log n}).$$

An isolator with k unit clauses for n -vertex graphs admits at most $2^{\binom{n}{2} - k}$ equivalence class representatives, so in order to admit at least one member of each equivalence class (by the definition of an isolator), the number of units in an isolator must also be asymptotically upper-bounded by $n \log n$.

In the next section, we provide a procedure that achieves this bound.

B. TT-fixing

In situations where we know that every member of the class of n -vertex tournaments contains a TT_k (a transitive tournament of size k), we also know that every equivalence class must contain a member with the tournament fixed in some arbitrary position and orientation (i.e. vertices 1 through k in ascending order). Therefore, any formula that *fixes* (i.e. asserts the existence of) a TT_k on the class of n -vertex tournaments is a valid isolator. Because the remaining subset of $n - k$ non-fixed vertices also forms a tournament, further knowledge about the existence of a transitive tournament within the remaining $n - k$ vertices can be used to fix (via units) another transitive tournament within the $n - k$ vertex subtournament. This procedure can be repeated until all vertices of the original tournament are part of some fixed transitive subtournament. Tournament Ramsey numbers provide exactly the required information about the existence of a transitive subtournament. In fact, tournament Ramsey numbers $R(k)$ (when known) provide the *largest* TT_k guaranteed to exist in a tournament of size at least $R(k)$. Therefore, tournament Ramsey numbers (as well as upper bounds, which exist for arbitrarily large n) can be used to iteratively construct large sets of unit clauses for tournament isolators: we will refer to this process as TT-fixing.

TT-fixing is best understood via a small example like figure 2. For an arbitrary 16-vertex tournament G , $R(5) = 14$ implies that G must contain a TT_5 as a subtournament. Therefore, the

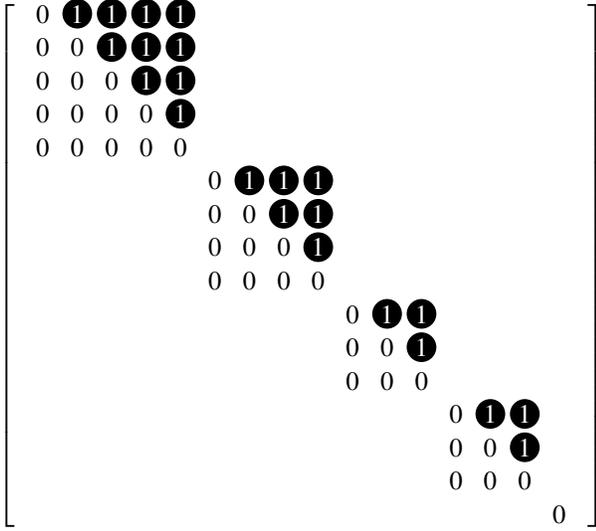


Fig. 2. A visual depiction of the unit clauses provided TT-fixing in the adjacency matrix of an $n = 16$ tournament. For entries with value 1 at row i and column j , the unit clause corresponding to (v_i, v_j) is added by TT-fixing. Equivalently, the 1's and 0's shown will exist in the adjacency matrix of any graph admitted by a TT-fixing isolator.

arcs between vertices $v_1 \dots v_5$ can be “fixed” to be a transitive tournament by generating all unit clauses corresponding to those arcs. However, the remaining $16 - 5 = 11$ vertices of G also form an arbitrary subtournament G' on 11 vertices. Because $R(4) = 8$, a TT_4 is guaranteed to exist in G' , so we can add unit clauses corresponding to the specific location of that TT_4 's existence in vertices $v_6 \dots v_9$. The repetition of this procedure down to 1 or 0 remaining vertices is TT-fixing.

C. TT-fixing gives $\Theta(n \log n)$ units

Let $units(n)$ be the function that returns the number of units that can be added to an isolator when using the TT-fixing method on n -vertex tournaments. Our goal is to prove a lower bound on $units(n)$. Unfortunately, exact tournament Ramsey numbers are non-trivial to calculate (only up to $R(6) = 28$ is known). However, from Erdős and Moser we have that $R(k) \leq 2^{k-1}$ [18], i.e. that a TT_k must exist when considering any tournament on 2^{k-1} or more vertices. Erdős and Moser's bound can thus be used with TT-fixing to lower-bound $units(n)$.

We claim that $units(n) \geq \sum_{i=1}^n \frac{1}{2} \lfloor \log_2(i) \rfloor$. We proceed via induction, with step n depending on step $n - k$, with $k = \lfloor \log_2(n) \rfloor + 1$. The proposition is true for $n = 1$ because $0 \geq 0$, $n = 2$ because $1 \geq 0.5$. By definition of TT-fixing, for a graph with n vertices we have

$$units(n) = \frac{k(k-1)}{2} + units(n-k). \quad (1)$$

By the inductive hypothesis,

$$units(n-k) \geq \sum_{i=1}^{n-k} \lfloor \log_2(i) \rfloor / 2. \quad (2)$$

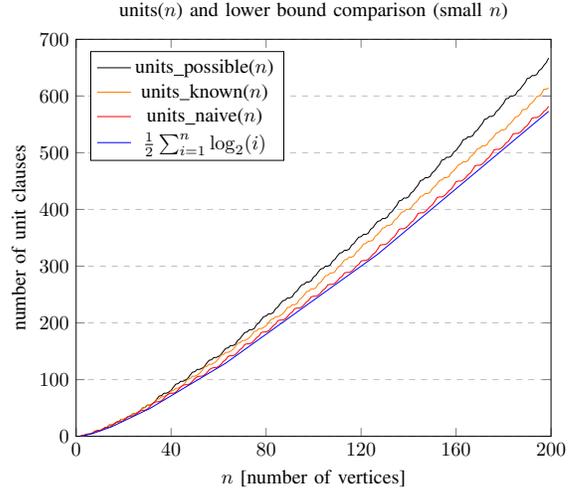


Fig. 3. A visual depiction of how the number of unit clauses produced by TT-fixing grows under different assumptions about Ramsey numbers.

Next, we have that

$$\begin{aligned} k \frac{k-1}{2} &= k \lfloor \log_2(n) \rfloor / 2 \\ &= \sum_{i=n-k+1}^n \lfloor \log_2(n) \rfloor / 2 \\ &\geq \sum_{i=n-k+1}^n \lfloor \log_2(i) \rfloor / 2. \end{aligned} \quad (3)$$

Combining lower bounds (2) and (3) for the terms of eq. (1) completes the proof:

$$\begin{aligned} units(n) &= \frac{k(k-1)}{2} + units(n-k) \\ &\geq \sum_{i=n-k+1}^n \lfloor \log_2(i) \rfloor / 2 + \sum_{i=1}^{n-k} \lfloor \log_2(i) \rfloor / 2 \\ &= \sum_{i=1}^n \lfloor \log_2(i) \rfloor / 2. \end{aligned} \quad (4) \quad (5)$$

This inequality result directly implies the asymptotic $n \log n$ bound, because $\log_2(n!) \in \Theta(n \log n)$.

D. Practical vs Theoretical TT-fixing units

We first note a useful recurrence relation on tournament Ramsey numbers: $R(k) \leq 2R(k-1)$.

Proof. Consider an arbitrary vertex v in an arbitrary tournament G on $2R(k-1)$ vertices. v must have either an out-degree or an in-degree of at least $R(k-1)$. In either case, consider the subset of at least $R(k-1)$ vertices pointed to/at by v . This subset must contain some TT_{k-1} as a subgraph by definition of $R(k-1)$. However, v points to or at all vertices in this TT_{k-1} , which demonstrates that a TT_k comprised of the TT_{k-1} vertices and v exists in G . \square

In Figure 3 the bottom two lines depict the strict lower bound used in the $n \log n$ units proof (blue), as well as the

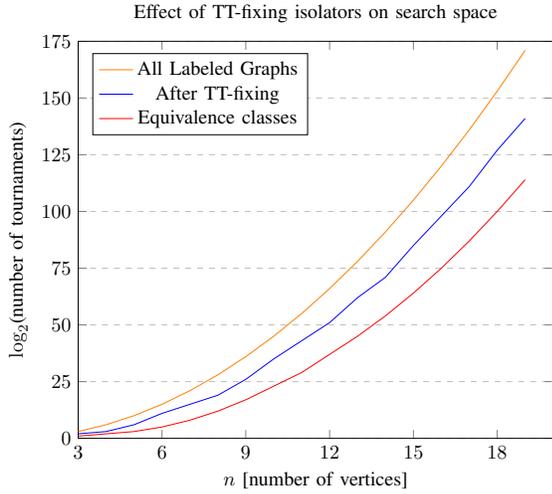


Fig. 4. A depiction of the search space reduction provided by TT-fixing using best-known Ramsey number bounds up to $n = 18$ in \log_2 space.

actual number of units TT-fixing would provide if we only used the $R(k) \leq 2^{k-1}$ bound from the proof (red). Above that (orange) is the number of units TT-fixing provides given the best currently known Ramsey number bounds. The best known bound on $R(7)$ is $34 \leq R(7) \leq 47$ [19], so the black line describes the best case for how many unit clauses TT-fixing could provide if $R(7) = 34$ was proven. The recurrence relation $R(k) \leq 2R(k-1)$ is what allows even improvements to small Ramsey number bounds to impact the efficacy of TT-fixing for large n .

In Figure 4, the top (orange) line is the total number of graphs a SAT solver must search in a tournament existence problem in the absence of an isolator. The bottom (red) line is the number of unlabeled tournaments on n vertices; this is the minimum number of graphs that any brute-force solver must search to solve a tournament existence problem. This data was taken from OEIS sequence A000568 [15], which limits the size of n for which we can make this comparison to $n = 19$. The middle (blue) line shows how many graphs are admitted by a TT-fixing isolator using the best known bounds on tournament Ramsey numbers. As n grows large, the gap between the bottom two lines should grow small as per the $n \log n$ units upper bound proof.

E. Undirected Isolators: Clique-fixing

As mentioned earlier, undirected isolators cannot have unit clauses. Therefore, undirected isolators cannot directly benefit from units via TT-fixing. However, a crossover result for undirected graphs does exist for binary clauses that uses the same ideas as TT-fixing; we term this process *clique-fixing*. Undirected Ramsey number guarantee the existence of a red or blue colored k -clique for graphs with more than $R_u(k)$ vertices (R_u used here for undirected Ramsey numbers). Clique-fixing uses the same iterative process as TT-fixing, but generates the following clauses instead of TT_k units:

$$\{r \vee e, \neg r \vee \neg e \mid e \in \text{Edges}(K_k)\}$$

where r is an auxiliary variable representing the concept “the k -clique is red” and $\text{Edges}(K_k)$ is the set of edge literals for the complete graph on k vertices. We note that these clauses are “almost” units in the sense that after a solver makes a decision about whether to set r to true or false, $\binom{k}{2}$ edges are set by unit propagation. Therefore, clique-fixing steps reduce the search space by half as much as TT-fixing steps do. Although not the focus of this work, it is plausible that a similar asymptotic optimality analysis could be done for clique-fixing given this small discrepancy. However, undirected Ramsey numbers (necessary for clique-fixing) empirically grow much faster than tournament Ramsey numbers (and also theoretically: $R_u(k) \leq 4R_u(k-1)$), so clique-fixing may not be as practically useful as TT-fixing.

V. PERFECT, OPTIMAL ISOLATOR SAT ENCODING

Unit-based techniques scale to arbitrary n , and TT-fixing is “asymptotically perfect” in the sense that for large tournaments, no isolator generation technique can provide more than a non-constant factor of search space reduction over TT-fixing. However, no known perfect isolators for $n > 4$ consist solely of unit clauses. Additionally, it can be practically useful to have an *optimal* perfect isolator for small tournaments to allow searching via SAT solver for only non-isomorphic (sub-)graphs as efficiently as possible. The practical utility of compact perfect isolators is demonstrated in our own experiments in the later “Tournament Ramsey Graphs” section. In the following sections, we describe our technique for creating perfect, optimal isolators for $n \leq 6$.

A. Basic SAT encoding

We re-implemented and modified the perfect isolator encoding for undirected graphs [10] to be used for tournaments. Formally, we encoded the question “Is there a set of k clauses C_1, C_2, \dots, C_k that is a perfect isolator for n -vertex tournaments.” Decoding a solution to this formula allowed us to produce an n -vertex isolator with k clauses.

For the i th isolator clause C_i and arc literal l , we defined the variable $In(C_i, l)$ to represent “ l is in C_i ”. Then, for each tournament G on n vertices, we define variables $Excludes(G, C_i)$ for $1 \leq i \leq k$ to mean “clause C_i does not admit G .” This specification is implemented as follows with a Tseitin encoding [20] to handle the equality and conjunctions:

$$\text{Excludes}(G, C_i) \leftrightarrow \bigwedge_{l \in A_G} \neg In(C_i, l). \quad (6)$$

Here A_G is the set of arc literals corresponding to the arcs present in graph G . We also define the variable $\text{Canon}(G)$ for all graphs G , meaning “Graph G is the canonical representative of its isomorphism class I_G .” We implement this as follows (again using Tseitin):

$$\text{Canon}(G) \leftrightarrow \bigwedge_{i=1}^k \neg \text{Excludes}(G, C_i). \quad (7)$$

Finally, for each isomorphism class I , we add the following clauses representing “exactly one graph in I is canonical” to the formula for each isomorphism class I :

$$\textit{ExactlyOne}(\{\textit{Canon}(G) \mid G \in I\}) \quad (8)$$

Here *ExactlyOne* is implemented with an At Most One operation via Sinz encoding [21] and an At Least One via disjunction. Therefore, a satisfying assignment to this formula corresponds to a perfect isolator on k clauses. If the formula is unsatisfiable for k and satisfiable for $k + 1$, then the perfect isolator with $k + 1$ clauses is optimal for the n in question.

B. Symmetry Breaking

One symmetry in the above encoding is the order of the isolator clauses, as reordering clauses of an expression in CNF does not affect its satisfying assignments. To break this symmetry, we added clauses that ensured a lexicographic ordering of the clauses in the resulting isolator. For every adjacent pair of clauses C_i and C_{i+1} , we fixed some ordering of every literal that may appear in them l_1, l_2, \dots, l_n , and then created variables e_0, e_1, \dots, e_n where e_j represents that clauses C_i and C_{i+1} are equivalent when considering only the first j literals. e_0 is always true, and to maintain the semantics of the other e_j we added the clauses

$$e_j \leftrightarrow (e_{j-1} \wedge (\textit{In}(C_i, l_j) \leftrightarrow \textit{In}(C_{i+1}, l_j)))$$

via the Tseitin transformation for every $1 \leq i < k$ and $1 \leq j \leq n$. Then, we enforced a lexicographic ordering by requiring that for every j such that C_i and C_{i+1} were equal up to j , that if clause C_i contained l_j then C_{i+1} must also contain l_j . Explicitly, we added the following requirement via the Tseitin transformation for every $1 \leq i < k$ and $1 \leq j \leq n$:

$$e_{j-1} \wedge \textit{In}(C_i, l_j) \implies \textit{In}(C_{i+1}, l_j)$$

and furthermore we required that e_n is false to ensure a strict ordering. When searching for an isolator with k clauses, this reduces the search space by a factor of $k!$ as only one of the $k!$ permutations of a given distinct set of clauses will be considered.

There is another symmetry in the vertex labeling. For a given isolator, for each literal l corresponding to arc index I_{ab} , we can change l to correspond to arc index $I_{\pi(a), \pi(b)}$ where π is a permutation of vertex labels. The resulting isolator accepts the same graphs that the original did, but under vertex permutation π . To break this symmetry, note that any tournament isolator must admit exactly one transitive tournament. So, we choose to admit only the canonical transitive graph with edges of the form $(v_i, v_j), i < j$. Note that because every edge in this graph goes from a lower numbered vertex to a higher numbered vertex, the corresponding literals in our encoding are all positive. As such, we know that for any isolator, there is a permuted isolator such that every clause has at least one positive literal in each clause. We may add this to our encoding by requiring for all clauses C

$$\bigvee_{l \in A_p} \textit{In}(C, l)$$

with A_p being the set of all positive literals. When trying to find an isolator for n vertices, this reduces the search space by a factor of $n!$ since the solver is guaranteed to only consider isolators for which the canonical transitive graph is the one described above.

C. Encoding Unit Propagation

Under the encoding described above, our solver finds isolators with many large clauses. However, by applying unit propagation it was often possible to reduce clause sizes. This indicated that not only was the solver generating solutions that needed postprocessing, but candidate isolators that were equivalent under unit propagation were being considered multiple times — a sort of symmetry in this problem. To resolve this, we added variables $\textit{Unit}(l)$ representing “literal l is a unit clause.” We then required that the isolator be already unit-propagated with respect to these literals by adding the requirement

$$\neg \textit{In}(C, l) \vee \neg \textit{Unit}(l)$$

for all clauses C and literals l . We also had to account for these units excluding graphs in the *Canon* clauses, which were updated to

$$\textit{Canon}(G) \leftrightarrow \bigwedge_{i=1}^k \neg \textit{Excludes}(G, C_i) \wedge \bigvee_{l \in A_G} \neg \textit{Unit}(\neg l)$$

Finally, we considered whether to count these special unit literals towards the clause count in determining isolator optimality. As mentioned in the preliminaries, we chose not to do so. When an isolator with units is used in a SAT solver, the units will be instantly eliminated through unit propagation and thus will reduce the complexity of the resulting problem. Therefore, we consider an optimal isolator to not just have the minimal number of clauses, but the minimal number of non-unit clauses. Since units cannot exist in undirected graph isolators (because an undirected graph isolator must admit both the complete and empty graph), this definition of optimality is consistent with the prior work on the undirected case [10]. Note that we only needed to consider positive unit literals as per the vertex-labeling symmetry breaking, which drastically reduced the search space.

VI. ADDITIONAL ISOLATOR GENERATION TECHNIQUES

The following sections describe several miscellaneous techniques, ranging from practical ways to gain slight improvements on prior techniques to possible directions for future research.

A. Incremental Isolators

Prior work has already shown that any isolator for n -vertex tournaments is also an isolator for $n+k$ -vertex tournaments for any positive k , and that combining an isolator on m vertices with an isolator on n vertices by applying each isolator to a disjoint subset of vertices creates a new isolator on $m+n$ vertices [22]. Therefore, it is possible to construct perfect isolators for $n+k$ -vertex tournaments by adding clauses to

any isolator for n -vertex tournaments. In particular, our SAT encoding pipeline had the option to ignore graphs that are not admitted by a given set of units. Including the maximal set of units from an n -vertex isolator when generating an encoding for $n + 1$ -vertex isolators reduces the number of tournaments to generate *Canon* clauses for by a factor of at least 2^n because each isolator has at least the units corresponding to a Hamiltonian path. It is worth noting that we do not have any proofs that any of our non-perfect or non-optimal isolators can be extended to an *optimal* isolator, even when the isolator being extended from is comprised of only unit clauses. However, extending an isolator from an initial set of units can make searching for compact isolators much more efficient.

The technique of combining isolators is useful for creating compact isolators for large n . Although TT-fixing guarantees asymptotic optimality, it does not always add the optimal number of units for small n . For example, TT-fixing will generate 9 units when processing 8-vertex (sub)tournaments, while an isolator for $n = 8$ with 11 units is possible.

B. Probing

In addition to the SAT encoding approach to isolator generation, we also generated isolators using a method from prior work called “random probes” [10]. On a high level, this approach starts with an empty set of clauses and adds randomly generated clauses that preserve at least one member of each equivalence class until the isolator is perfect. There were only two non-superficial changes needed to adapt the prior work on random probes for undirected graphs to the directed case; allowing unit clauses and allowing clauses with only positive literals. While not guaranteed to generate optimal isolators, the strength of this approach is the relative speed with which isolators are generated. This approach also benefited in efficiency from the technique of disallowing clauses with all negative literals and extending isolators from the unit clauses of smaller isolators.

VII. RESULTS

Our experimental results include the sizes of known perfect isolators for small n , as well as experiments showing the practical utility of small $n = 6, 7$ perfect isolators for solving a tournament existence problem. All results and code are available at https://github.com/evanlohn/digraph_isolators.

A. Experimental Setup

Our SAT-based approach to generating isolators rely on the creation of “map” files: text files associating each tournament of size n with a label representing that graph’s isomorphism class. In order to generate a map file for tournaments on n vertices, we began by enumerating all $2^{n(n-1)/2}$ graphs of size n . We converted each graph into an adjacency matrix and then into the “.d6” format specified in the NAUTY handbook, then fed the resulting graphs into the labelg script bundled with the NAUTY tool for graph isomorphisms [23]. labelg produced a file where each graph was converted to the canonical form

used by nauty. We gave each canonical form a unique label and outputted the arc (directed edge) indices of each original graph alongside its canonical form.

B. Small Optimal Isolators

Our SAT encoding allowed us to compute optimal isolators up to $n = 6$. The SAT solver CaDiCaL [24] solves the two instances required to prove optimality ($k = 6, 7$ non-unit clauses) within 24 hours. Figures 1 and 6 graphically display optimal, perfect isolators for $n = 4, 5$ by displaying a graph from each isomorphism class. Figure 5 presents the same image for one of the 56 isomorphism classes for $n = 6$. Most of the structure of these isolators can be seen from their unit clauses, which are depicted via red edges in the figures.

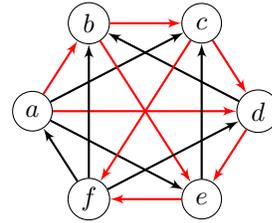


Fig. 5. One of the 56 isomorphism class representatives admitted by a particular isolator for 6-vertex tournaments. Red edges are edges fixed by unit clauses of the isolator.

For $n = 7$, solving the SAT instance directly became clearly infeasible (taking several days without any signs of progress). However, random probing allowed us to find a perfect isolator for $n = 7, 8$. Each probe ran in around 10 seconds when restricted to force a positive literal in each clause with the map file reduced by the unit clauses from the next largest isolator. Table I describes the best (fewest non-unit clauses) isolator found for $1 \leq n \leq 8$. Several thousand probes were required to find our best known isolator for $n = 7$, while 2 probes were used to find our $n = 8$ isolator (each $n = 8$ probe required about 2 days to finish). We note that $n = 8$ isolators can have up to 11 unit clauses; the $n = 8$ isolator in Table I was the shortest *perfect* isolator we generated via probing.

TABLE I
SHORTEST PERFECT ISOLATORS FOUND FOR $n \leq 8$

Vertices	Isomorphism classes	Best units	Best non-units
1	1	0	0
2	1	1	0
3	2	2	0
4	4	4	0
5	12	6	2
6	56	8	6
7	456	9	47
8	6880	10	665

C. Tournament Ramsey Graphs

The known tournament Ramsey numbers are $R(2) = 2$, $R(3) = 4$, $R(4) = 8$, $R(5) = 14$, and $R(6) = 28$ [25]. Note that in most cases, the next number is two times its predecessor. Recently, the lower and upper bounds for $R(7)$ have been

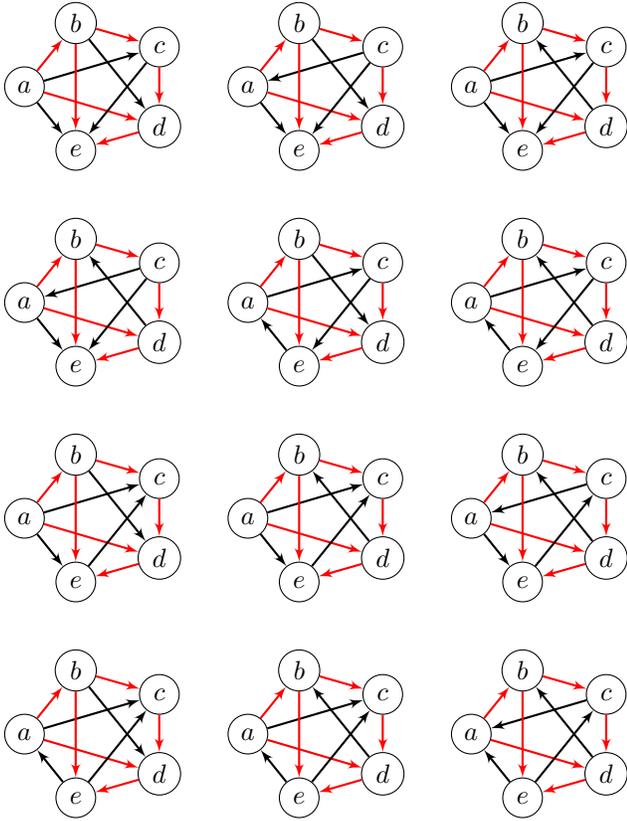


Fig. 6. All isomorphism class representatives admitted by a particular isolator for 5-vertex tournaments. Red edges are edges fixed by unit clauses of the isolator. The two non-unit clauses in the isolator are $ac \vee \neg bd \vee ce$ and $ac \vee ae \vee \neg ce$.

improved from $32 \leq R(7) \leq 54$ to $34 \leq R(7) \leq 47$ [19]. The improved lower bound is due to dozens of TT_7 -free tournament on 33-vertices found using SAT.

McKay [26] extended this set of 33-vertex TT_7 -free tournaments to 5303 using the following method: generate all 29-vertex subtournaments of known 33-vertex TT_7 -free tournaments and extend them in all possible ways to 33-vertex TT_7 -free tournaments. Repeat this procedure until no new 33-vertex TT_7 -free tournaments are found. Also note that if a tournament has no TT_k , then its complement (reversing all arcs) also doesn't. This can be used to find additional TT_k -free graphs as well.

Looking for neighbors and complement graphs is a well-known technique to compute more graphs with a certain property. McKay and Radziszowski used it to compute all known 42-vertex graphs that have no clique of size 5 nor a co-clique of size 5 [27]. They conjecture that this method generated all possible graphs of this type.

For all known Ramsey numbers $R(k)$, there are unique tournaments without a TT_k of size $R(k) - 1$ and $R(k) - 2$. Generalizing this property, if for some n there exists a k with a unique TT_k -free tournament on n vertices, then that graph is known as ST_n . For example, the unique TT_6 -free tournaments on 26 and 27 vertices are referred to as ST_{26}

and ST_{27} respectively.

Prior to our work, there were 5303 known TT_7 -free tournaments on 33 vertices, implying that $R(7) \geq 34$. So, either $k = 7$ breaks the pattern of existence of ST_n tournaments, or $R(7) > 34$. We studied the 5303 33-vertex TT_7 -free tournaments and found that they all have ST_{26} as a subtournament. Moreover, 4952 of them have ST_{27} as a subtournament.

It is the case that any TT_7 -free tournament on 34 vertices contains at least 1 (up to isomorphism) TT_7 -free subtournament on 33 vertices. Therefore, enumerating further TT_7 -free tournaments on 33 vertices is a step towards either finding a TT_7 -free 34-vertex tournament or proving that no such tournament exists. With this motivation, we explored whether the suite of 5303 was complete or whether there are any other 33-vertex TT_7 -free tournaments. Our main experimental setup involved finding new members containing ST_{26} but not ST_{27} by solving a CNF formula with a SAT solver, which uses our isolator on 7 vertices. The formula can be described as the union of the following sets of clauses:

- 1) $\binom{26}{2} = 325$ unit clauses requiring that ST_{26} be present in vertices v_1 through v_{26} ;
- 2) The perfect isolator for $n = 7$ on the seven remaining vertices v_{27} through v_{33} ;
- 3) A clause blocking each of the 5303 known solutions for each vertex permutation that caused the solution to have ST_{26} in vertices v_1 through v_{26} and a graph admitted by the $n = 7$ isolator in vertices v_{27} through v_{33} ; and
- 4) clauses enforcing the “no TT_7 ” condition from [19].

While this formula does not disallow all ST_{27} s (i.e. a solution might include an extension from ST_{26} that was not present in the original solution set), it disallows all currently known extensions, including the most common by far 1-vertex extension from ST_{26} to ST_{27} . Additionally, the $n = 7$ perfect isolator plays a crucial role for finding new solutions in that without it, the SAT solver could find any tournament equivalent to one of the previously known 33-vertex TT_7 -free tournaments except for some non-automorphic permutation of the last 7 vertices (which would thus be isomorphic to the previously known solution). All solutions to our formula are non-isomorphic to the original 5303 tournaments.

On the Pittsburgh Supercomputing Center [28], we ran 640 shuffled (clause permuted) versions of the above formula on 640 cores for 6 hours using the Kissat solver [29]. We found three different satisfying assignments. These three solutions represented a single new 33-vertex TT_7 -free tournament, which is shown in Figure 7. This tournament is special as it is self-complementary: reversing all arcs result in an isomorphic graph. Only a small fraction of tournaments has this self-complementary property [30]. Note that all ST_n graphs have this property by definition. After finding this new tournament, we updated the formula to include the blocking clauses for the new tournaments and its isomorphisms. Kissat did not produce further solutions when using 640 shuffled (clause permuted) version of the updated formula on 640 cores in a day, so it is possible that the formula is simply unsatisfiable.

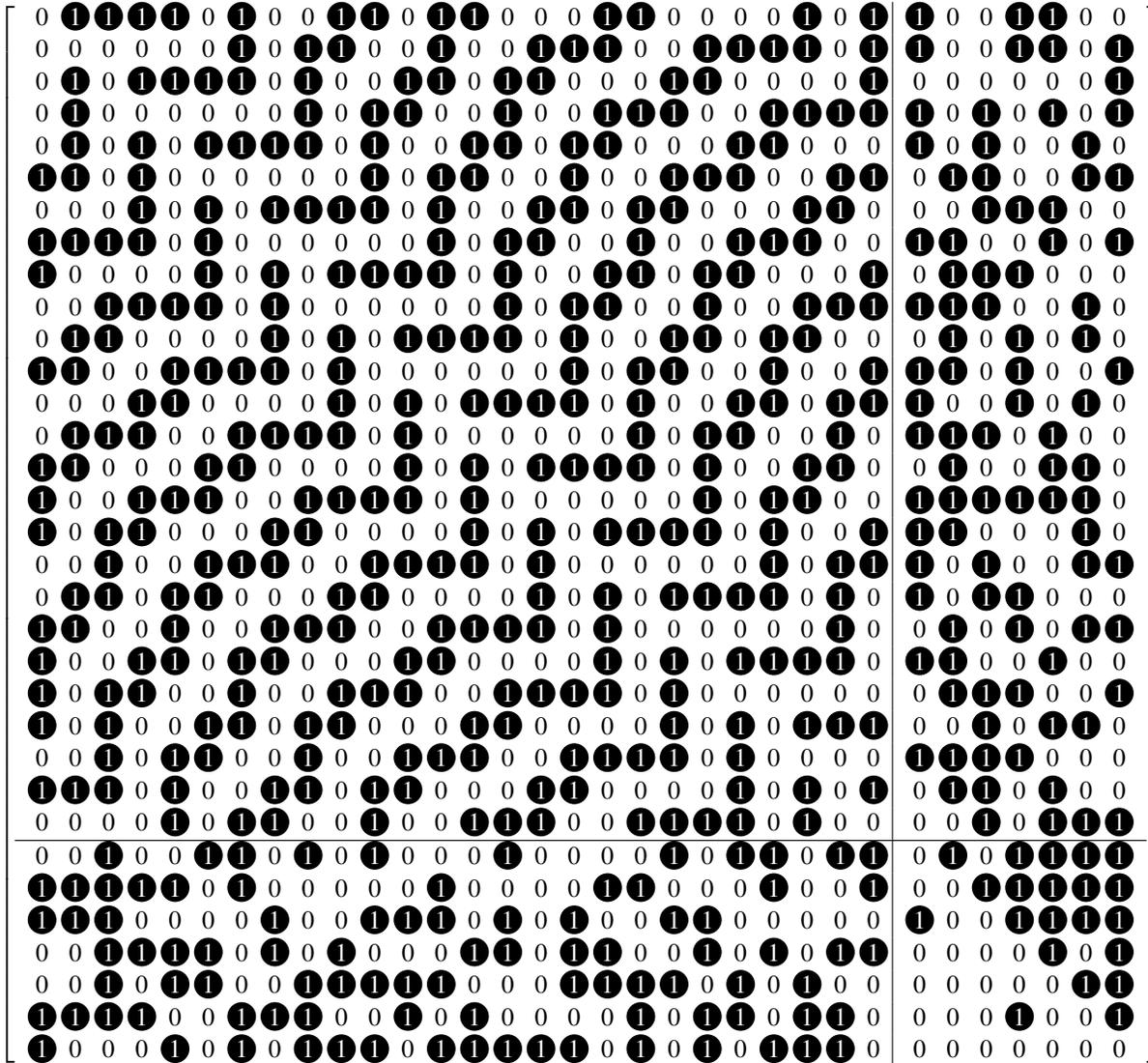


Fig. 7. The new 33-vertex TT_7 -free tournament found using our perfect isolator for $n = 7$. The upper-left section of the matrix is ST_{26} , while the bottom-right section is a graph admitted by our $n = 7$ isolator.

VIII. CONCLUSIONS

Our techniques allow the generation of isolators with asymptotically optimal numbers of unit clauses, as well as perfect, optimal isolators for $n \leq 6$ and compact isolators for $n = 7, 8$ found by random probing. We further demonstrate how small isolators can be effectively used in the search for much larger graphs relevant to tournament existence problems. Future work using our results may lead to further improvements on bounds for the tournament Ramsey number problem.

ACKNOWLEDGEMENTS

This work was partially supported by the Hoskinson Center for Formal Mathematics and the National Science Foundation under grant CCF-2015445. We thank Jeremy Avigad for his comments on earlier drafts and John Mackey for his advice on the graph theoretical parts of the paper.

REFERENCES

- [1] J. Brakensiek, M. Heule, J. Mackey, and D. Narváez, “The resolution of Keller’s conjecture,” in *Automated Reasoning*, N. Peltier and V. Sofronie-Stokkermans, Eds. Cham: Springer International Publishing, 2020, pp. 48–65.
- [2] M. J. H. Heule, “Schur number five,” in *AAAI*, 2018.
- [3] M. J. H. Heule, O. Kullmann, and V. W. Marek, “Solving and verifying the boolean pythagorean triples problem via cube-and-conquer,” in *Theory and Applications of Satisfiability Testing – SAT 2016*, N. Creignou and D. Le Berre, Eds. Cham: Springer International Publishing, 2016, pp. 228–245.
- [4] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy, “Symmetry-breaking predicates for search problems,” in *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, ser. KR’96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, p. 148–159.
- [5] I. Shlyakhter, “Generating effective symmetry-breaking predicates for search problems,” *Discrete Applied Mathematics*, vol. 155, pp. 1539–1548, 06 2007.
- [6] W. Wang, M. Usman, A. Almaawi, K. Wang, K. S. Meel, and S. Khurshid, “A study of symmetry breaking predicates and model counting,”

- in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 115–134.
- [7] T. Blankenship, J. Cummings, and V. Taranchuk, “A new lower bound for van der Waerden numbers,” *European Journal of Combinatorics*, vol. 69, pp. 163–168, 2018.
- [8] M. Codish, M. Frank, A. Itzhakov, and A. Miller, “Computing the Ramsey number $R(4,3,3)$ using abstraction and symmetry breaking,” *Constraints*, vol. 21, no. 3, p. 375–393, jul 2016.
- [9] N. Komarov and J. Mackey, “On the number of 5-cycles in a tournament,” *Journal of Graph Theory*, vol. 86, 2017.
- [10] M. J. Heule, “Optimal symmetry breaking for graph problems,” *Mathematics in Computer Science*, vol. 13, no. 4, pp. 533–548, 2019.
- [11] D. Kühn, R. Mycroft, and D. Osthus, “A proof of Sumner’s universal tournament conjecture for large tournaments,” *Proceedings of the London Mathematical Society*, vol. 102, no. 4, pp. 731–766, 2011.
- [12] W. Suksompong, “Tournaments in computational social choice: Recent developments,” in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2021, pp. 4611–4618, survey Track.
- [13] A. Hanaki, “Skew-symmetric hadamard matrices and association schemes,” *SUT Journal of Mathematics*, vol. 36, 06 2000.
- [14] S. P. Radziszowski, “Small Ramsey numbers,” *Electronic Journal of Combinatorics*, vol. 1000, 2011.
- [15] N. J. A. Sloane and T. O. F. Inc., “The on-line encyclopedia of integer sequences,” 2020. [Online]. Available: <http://oeis.org/>
- [16] J. W. Moon, *Topics on Tournaments*. Holt, Rinehart and Winston, 1968, p. 28.
- [17] P. L. Erdős and A. Rényi, “Asymmetric graphs,” *Acta Mathematica Academiae Scientiarum Hungarica*, vol. 14, pp. 295–315, 1963.
- [18] P. Erdős and L. Moser, “On the representation of directed graphs as unions of orderings,” in *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, vol. 9, 1964, pp. 125–132.
- [19] D. Neiman, J. Mackey, and M. Heule, “Tighter bounds on directed Ramsey number $R(7)$,” *arXiv preprint arXiv:2011.00683*, 2020.
- [20] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” in *Automation of reasoning*. Springer, 1983, pp. 466–483.
- [21] C. Sinz, “Towards an optimal CNF encoding of boolean cardinality constraints,” in *International conference on principles and practice of constraint programming*. Springer, 2005, pp. 827–831.
- [22] M. Codish, A. Miller, P. Prosser, and P. Stuckey, “Breaking symmetries in graph representation,” in *International Joint Conference on Artificial Intelligence*, 08 2013, pp. 510–516.
- [23] B. D. McKay and A. Piperno, “Practical graph isomorphism, ii,” *Journal of symbolic computation*, vol. 60, pp. 94–112, 2014.
- [24] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, “CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020,” in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Jarvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.
- [25] A. Sanchez-Flores, “On tournaments free of large transitive subtournaments,” *Graphs and Combinatorics*, vol. 14, no. 2, pp. 181–200, 1998.
- [26] B. McKay, “Digraphs,” <http://users.cecs.anu.edu.au/~bdm/data/digraphs.html>, accessed: 2022-05-10.
- [27] B. D. McKay and S. P. Radziszowski, “Subgraph counting identities and ramsey numbers,” *Journal of Combinatorial Theory, Series B*, vol. 69, no. 2, pp. 193–209, 1997.
- [28] S. T. Brown, P. Buitrago, E. Hanna, S. Sanielevici, R. Scibek, and N. A. Nystrom, *Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–4.
- [29] A. Biere, M. Fleury, and M. Heisinger, “Cadical, kissat, paracooba entering the sat competition 2021,” 2021.
- [30] W. J. R. Epllett, “Self-converse tournaments,” *Canadian Mathematical Bulletin*, vol. 22, no. 1, pp. 23–27, 1979.

Enumerative Data Types with Constraints

Andrew T. Walter 

Khoury College of Computer Sciences
Northeastern University
 Boston, MA, USA
 walter.a@northeastern.edu

David Greve

Collins Aerospace
 Cedar Rapids, IA, USA
 david.greve@collins.com

Panagiotis Manolios 

Khoury College of Computer Sciences
Northeastern University
 Boston, MA, USA
 pete@ccs.neu.edu

Abstract—Many verification and validation activities involve reasoning about constraints over complex, hierarchical data types. For example, distributed protocols are often defined using state machines that govern the behavior of processes communicating with messages which are hierarchical data types with state-dependent constraints and dependencies between component fields. Fuzzing, analyzing and evaluating implementations of such protocols requires solving complex queries that pose challenges to current SMT solvers. Generating fields that satisfy type constraints is one of the challenges and this can be tackled using enumerative data types: types that come with an enumerator, an efficiently computable function from natural numbers to elements of the type. Enumerative data types were introduced in ACL2s as a key component of counterexample generation, but they do not handle constraints such as dependencies between types. We extend enumerative data types with constraints and show how this extension enables applications such as hardware-in-the-loop fuzzing of complex distributed protocols.

Index Terms—verification, data types, distributed systems, fuzzing, counterexample generation, ACL2s

I. INTRODUCTION

The motivation for this paper stems from a project to analyze the IEEE 802.11 Wi-Fi protocol. Since the introduction of the first IEEE 802.11 standard in 1997 [1], the Wi-Fi family of protocols have become a key part of many user’s ability to access the Internet. In 2019, Cisco predicted that over half of global Internet traffic will be transmitted over Wi-Fi and over 20% of global Internet traffic will be transmitted over a mobile network by 2022 [2]. Therefore, securing wireless networks and their underlying hardware is of critical importance. One method that researchers have used to demonstrate vulnerabilities in the Wi-Fi protocol is *fuzzing*, a form of testing in which generated data (possibly invalid) is input to a system, which is monitored for crashes, nonconforming responses, or other undesired behavior. Fuzzing has historically been successful in testing software systems, but bringing it into the realm of hardware raises several challenges.

Consider the general problem of validating the conformance of a given hardware device to a wireless protocol using hardware-in-the-loop fuzzing, where we have no internal knowledge of the device under test (DUT). In order to obtain good coverage of such protocols, we have to force the DUT into a variety of protocol states. Interesting protocols are nondeterministic, so we cannot easily precompute a set of messages to send; instead we must generate messages dynamically, in response to actual messages received from the

DUT. Another complication is that protocols typically contain complex constraints on the format and contents of messages, making it infeasible to generate well-formed messages using standard fuzzing techniques. Finally, we note that such hardware devices are fast and associated protocols often involve short timeouts, on the order of hundreds of microseconds. Therefore, to effectively validate the protocol conformance of such devices, we must generate well-formed messages at high speeds.

The prevailing approach for message generation in scenarios like the above has been the development of custom software like Wifuzzit [3] and owfuzz [4]. Developing such software takes a significant amount of highly specialized engineering effort. A more general and powerful approach is to use formal methods to model the protocol under which the DUT is being tested and to then automatically generate protocol messages from that model, using formal methods tools. Unfortunately, current formal methods are not powerful enough to generate messages of the required complexity and at the required rate, as explained in detail later.

To address the above problem, we present *enumerative data types with constraints*, an idea that enables the fast generation of elements of hierarchical data types with constraints and inter-field dependencies. Our work is a natural extension of *enumerative data types* [5]: types that have *enumerators*, functions from natural numbers to elements of that type. We implemented the idea in the context of ACL2s [6], [7] and performed an evaluation by generating certain messages described in the 802.11 Wi-Fi protocol. Our evaluation shows that we are able to generate messages for a wide variety of sizes, something that neither SMT solvers nor pure enumerative data types can do. For the classes of messages that can also be generated by SMT or enumerative data types, our approach is at least two orders of magnitude faster.

Our contributions are as follows. (1) The idea of *enumerative data types with constraints*, which allows for the efficient generation of elements of dependent types with constraints and field interdependencies. (2) Extensions to the existing enumerative data type framework in ACL2s to support lists with length and ordering constraints, as well as improved support of numeric ranges. (3) The evaluation of our ideas with a case study on fuzzing Wi-Fi access points. All tools, models and artifacts developed for the case study, including sets of SMTLIB2-formatted constraints that may be useful

```

(definec foo (x :int) :bool
  (!= x (expt 2 63)))
(property (x :int) (foo x))
$>...
We falsified the conjecture. Here are
counterexamples:
--(X 9223372036854775808))

```

Fig. 1. A definition of a function and a property that ACL2s can find a counterexample to, but QuickCheck cannot in an equivalent Haskell formulation without the use of a custom generator.

for benchmarking SMT solvers will be publicly available [8]. (4) The idea of FM/hardware-in-the-loop for protocol conformance testing, where formal methods are used in the loop of a hardware-in-the-loop approach to protocol conformance testing.

The paper is organized as follows. Section II discusses related work in the areas of property testing, constraint-solver aided test data generation, and Wi-Fi fuzzing. Section III describes our extensions to enumerative data types and Section IV describes the idea of enumerative data types with constraints. A full, formal description is beyond the scope of the paper, due to the complexity of the data definition framework, but we have endeavored to present the ideas in a way that experts will be able to adapt them to other languages, type systems and tools. Section V discusses aspects of the implementation relevant for our Wi-Fi fuzzing case study, described in Section VI. Conclusions are presented in Section VIII.

II. RELATED WORK

ACL2s (the ACL2 Sedan) [6], [7], is an extension of the ACL2 [9], [10] automated theorem prover that includes a powerful data definition framework (*defdata*) [5], a counterexample generation framework (*cgen*) for finding counterexamples to conjectures [11]–[13], a power termination analysis based on calling-context graphs [14] and ordinals [15]–[17] and IDE support in the form of an Eclipse plug-in.

QuickCheck [18] is a tool for performing property-based testing. It is emblematic of a family of tools that perform property-based testing of program without considering the formal semantics of those programs. Such tools are capable of finding many bugs, but there are many incorrect properties that they are highly unlikely to find counterexamples to without specific direction from the user. The *cgen* framework of ACL2s was inspired by QuickCheck and builds on it by combining random generation with theorem proving. Fig. 1 highlights an example of a function and property that ACL2s can find a counterexample to, but QuickCheck cannot in an equivalent Haskell formulation.

ACL2s is able to find a counterexample in the Fig. 1 example by making use of reasoning capabilities provided by ACL2. Note that *cgen* was able to produce this result without any property-specific configuration. *cgen* is successful because it is able to benefit from ACL2’s process of transforming and splitting up the property being tested into smaller pieces.

cgen also makes use of random testing during counterexample search. This random testing is deeply entwined with ACL2s’ *defdata* data definition system for defining types [5]. *cgen* will be discussed in more detail in Section III.

Constraint Solvers and Test Data Generation: Outside of ACL2, many systems have been developed that allow the combination constraint solvers with models or specifications for the purpose of test data generation. The Alloy modeling language and its analyzer [19] constitute one such system: see Sullivan *et al.*’s framework for automated test generation in Alloy [20] as well as Abdul Khalek *et al.*’s use of Alloy to generate database management systems tests [21]. The Alloy analyzer’s model-finding system differs substantially in approach from *cgen*—in particular, Alloy only supports bounded verification, meaning that it considers only a finite subset of all possible models, those with sizes in a user-provided bound, when verifying or searching for a counterexample to a property. Chamarthi *et al.* provide a detailed discussion of the differences between *cgen* and Alloy in [12], including that Alloy does not in general support recursive function definitions.

Other purpose-built systems include PLEDGE [22] and TAF [23]. Some of these systems attempt to generate test data that satisfies some coverage criterion of the given model; this is an interesting goal that is not described in this paper.

FuzzM [24] uses the JKind SMT-based model checker [25] to generate test data for fuzzing systems modeled in the Lustre programming language [26]. Depending on the complexity of the model provided, FuzzM may make queries to JKind that take a significant amount of time to solve. For this reason, FuzzM provides a generalization technique known as trapezoidal generalization [27] that can be used to generate many test data from a single datum produced by a query to JKind. Using trapezoidal generation with FuzzM can result in a data generation rate increase of several orders of magnitude.

Wi-Fi and Fuzzing: The Wi-Fi family of protocols is extensively used to provide local-area internet connections in a wide variety of settings including homes, businesses, and universities. Therefore, bugs and vulnerabilities in Wi-Fi protocols and implementations thereof can have a wide reach. For example, the 2017 KRACK attack [28] exposed a vulnerability in the 4-way handshake described by the 802.11 standard, affecting nearly every Wi-Fi device on the market at that time. The Wi-Fi protocols are based on the IEEE 802.11 standard [1], which describes the MAC (medium access control) and PHY (physical) layers of a network. We concern ourselves here with the MAC layer. The 802.11 standard describes the binary format of MAC frames, a generic overview of which is shown in Fig. 2.

Due to their prevalence, Wi-Fi protocols have previously been subjected to hardware-in-the-loop fuzz testing by several groups. In 2007, Laurent Butti and Julien Tinnés presented a hardware-in-the-loop approach [29] fuzzing Wi-Fi client drivers; this work resulted in the discovery of multiple bugs. Butti’s 2007 system did not model the 802.11 MAC frame specification, and it instead focused on generating fuzzed

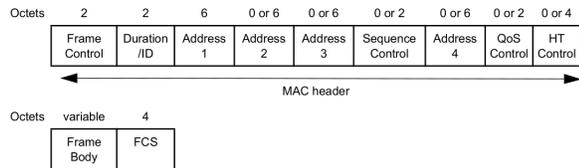


Figure 9-2—MAC frame format

Fig. 2. The binary layout of a generic 802.11 MAC frame. Figure taken from the IEEE 802.11-2020 standard [1].

frame elements and using the Scapy library [30] to generate packets with the appropriate structure that contain the fuzzed elements. More recently, Vanhoef *et al.* [31] described an approach for fuzzing access points’ implementation of the 802.11 Wi-Fi handshake in which an abstract model of the Wi-Fi handshake is combined with test generation rules to produce test cases. These test cases consist of a sequence of abstract messages which are concretized into appropriate MAC frames when executed. This approach was able to find several vulnerabilities and quirks in the tested systems. In 2019, Garbelini *et al.* described their Greyhound system [32], which uses a model of the 802.11 protocol to generate frames that should drive the 802.11 client device into a particular protocol state before sending a fuzzed frame. Using a protocol model also allows Greyhound to analyze responses from the client device to determine if the client’s responses comply to the 802.11 protocol. None of the aforementioned works regarding Wi-Fi fuzzing describe using theorem provers or constraint solvers to generate test data from protocol models. Based on our experience, we believe there would be a benefit to using constraint solvers in Wi-Fi protocol fuzzing, but the performance of existing approaches using constraint solvers is insufficient for use in the context. We will touch on this topic more in Section VI.

III. ENUMERATIVE DATA TYPES

The idea of enumerative data types was introduced by Chamarithi *et al.* in the context of ACL2s and its `defdata` framework [5], a rich data definition framework that allows one to specify and reason about user-defined types. All `defdata` types have predicative characterizations in the form of *recognizers*, functions that recognize exactly the elements of the type, as well as enumerative characterizations in the form of *enumerators*, functions that, given a natural number, return an element of the data type. Enumerators in ACL2s are efficient, in part because they do not involve any theorem proving. In this section, we provide a short overview of `defdata` and present extensions to `defdata` that were added to support our application. These extensions are publicly available and formally verified using ACL2s.

The introduction of enumerative data types was partially motivated by counterexample generation and satisfiability solving. ACL2s automatically generates counterexamples to function definitions and conjectures using a synergistic combina-

tion of theorem proving and enumerative data types. Theorem proving is used to decompose and simplify conjectures, at which point counterexample generation algorithms use type inference and enumerators to randomly generate elements based on the types of the variables appearing in the conjecture. In fact, counterexample generation in ACL2s uses enumerators and theorem proving in a recursive fashion, *e.g.*, after assigning a value to a variable, theorem proving is used to propagate consequences of the assignment, which may lead to further decompositions and simplifications as well as stronger type inferences, which are then exploited in further rounds of enumeration and theorem proving [11]–[13]. Satisfiability solving of ACL2s queries is performed similarly. This will be discussed in more detail in Section IV.

The `defdata` framework includes a large collection of built-in types. These types include basic types such as atoms, symbols, characters, strings, numbers and Booleans. Subtypes are supported and used extensively. Examples of subtypes include standard, non-special characters, keywords, symbols corresponding to variable names, and numeric types such as rationals, complex rationals, non-zero rationals, positive rationals, negative rationals, non-positive rationals, non-negative rationals, ratios (rationals that are not integers), positive ratios, negative ratios, integers, non-zero integers, natural numbers, positive integers, negative integers, non-positive integers, odd integers, even integers and zero. List and association list (`alist`) types, as well as non-empty versions, are also supported and are included for built-in types. There is also a universal type that includes all other types.

The `defdata` framework allows one to easily define new types by providing support for singleton types, enumeration types and range types (numeric ranges), as well as types built out of existing types, such as product types, union types, alias types, record types, list types, `alist` types, recursive types, mutually recursive types and map types (finite partial functions). The framework also allows one to define custom types, *e.g.*, to define the primes as a type, a user only needs to define a recognizer and an enumerator and then register the type. Custom types can then be used as if they were built-in to construct new types.

Polymorphic functions are also supported by `defdata`, *e.g.*, the form

```
(sig nth (nat (listof :a)) =>:a
:satisfies (< x1 (len x2)))
```

states that `nth` is a function that given a natural number and a list of some type `:a` returns a list of type `:a`, as long as the first argument (`x1`) is less than the length of the list (`x2`).

The `defdata` framework automatically generates theorems in the form of various rules that ACL2s can use to reason about types using techniques such as rewriting, forward chaining, type reasoning, linear and non-linear arithmetic, as well as various decision procedures; see [9] for an in-depth discussion of the types of rules supported by ACL2. The framework includes support for specifying and reasoning about subtypes, *e.g.*, it includes and generates subtype theorems for built-in

and user-defined types. It also generates auxiliary functions, such as constructors and destructors, as appropriate.

Finally, the `defdata` framework includes numerous advanced features, *e.g.*, it allows users to select different randomization schemes, to define custom enumerators and to switch between enumerators dynamically.

We extended `defdata` by adding two libraries. The first library, `deflist`, provides support for defining list types with certain length and ordering constraints. The second library, `definrange`, provides improved support for numeric range types over integers. The libraries are formally verified using ACL2 and are publicly available.

The `deflist` library provides the `defdata-list`, `defdata-ordered-list`, and `defdata-list-rng` forms, which are used to define `defdata` lists whose length is between two natural numbers, ordered lists with length constraints and lists with irregular length constraints, respectively. Consider the following example, derived from our Wi-Fi application:

```
(defdata-list SR8 SRType 1 8)
```

This defines the type `SR8`, which corresponds to lists whose length is between 1 and 8 (inclusive) of elements of type `SRType`, where `SRType` is a previous defined type recognizing 39 numbers between 2 and 236 that correspond to certain supported rates, as specified by the Wi-Fi protocol. The above form defines a recognizer and an enumerator for such lists. A type corresponding to lists of `SRType` with no length constraints is generated, if it does not already exist. Various tables keeping track of data types are updated. Rules for reasoning about lists of this type are also generated, *e.g.*, forward-chaining, type-prescription, compound-recognizer and rewrite rules that characterize the type and relate it to other types are automatically generated. Rules for reasoning about polymorphic functions and for controlling how the theorem prover uses these rules are also generated. This form generates a collection of forms totaling 7,944 lines and consisting of 434K bytes, all of which is formally verified by the ACL2 theorem prover.

The `defdata-ordered-list` form provides a similar capability but also imposes the constraint that the list is ordered. Consider the following example, derived from our Wi-Fi application:

```
(defdata-ordered-list BO255 uint8 0 255)
```

This defines the type `BO255`, which corresponds to lists of bytes (`uint8`) whose length is between 0 and 255 and whose elements are in increasing order. This form generates all of the forms that `defdata-list` generates, as well as rules for reasoning about the sorted lists. Finally, the `defdata-list-rng` form is similar to the `defdata-list` form, but allows one to specify irregular length constraints. Consider the following example, derived from our Wi-Fi application:

```
(defdata-list-rng BTS uint8 (gen-skip 22 254 2))
```

This defines the type `BTS`, which corresponds to lists of bytes (`uint8`) whose length is contained in the list of numbers

generated by the form `(gen-skip 22 254 2)`, which includes the numbers 22, 24, ..., 254. This form generates all of the forms that `defdata-list` generates, specialized to the irregular lengths.

The enumerators generated by the `deflist` library work by selecting a length in the appropriate range and then generating that many elements of the element type. This can be done very efficiently. If there are ordering constraints, then the generated list is sorted, using a verified sorting library we developed that includes an efficient sorting algorithm and supports sorting and potentially removing duplicates in the output. If duplicates are not allowed by the type, then they are removed, but this can result in lists whose length is shorter than desired. We experimented with a version of the library that generated lists of the appropriate length and where each such list had the same probability of being selected (*i.e.*, a uniform distribution), but that turned out to be computationally expensive for long lists. Therefore, once we sort the list and remove duplicates, we add a pass where we add elements not already in the list until we reach the target length. This turns out to be almost as fast as the non-ordered case.

The second library, `definrange`, provides `definrange` and `defnatrange` forms, which improved support for numeric range types over integers and natural numbers. Consider the following example, derived from our Wi-Fi application:

```
(defnatrange uint48 (expt 2 48))
```

This defines the type `uint48` which corresponds to the natural numbers less than 2^{48} . As was the case with `deflist`, we generate enumerators and rules for reasoning about the type, subtypes and polymorphic functions.

IV. ENUMERATIVE DATA TYPES WITH CONSTRAINTS

Complex data types often include type dependencies between fields. For example, consider a stack type which contains a field corresponding to the length of the stack with the type invariant that the value of this field is equal to the length of the stack. Sometimes there are dependencies between types, *e.g.*, a function may require that it is provided with two arguments, both of which are ordered lists of equal length. In this section, we show how to extend enumerative data types to support such constraints. The idea is relatively simple, but very powerful. As we show in this paper, this extension enables applications such as hardware-in-the-loop and theorem-prover-in-the-loop fuzzing of distributed protocols.

As a simple motivational example, consider a record consisting of n fields, f_1, \dots, f_n , each of which is a list whose length is between 1 and 10 (inclusive). Before our work, an enumerator for f_i would generate a list of length l , with $1 \leq l \leq 10$ with probability $\frac{1}{10}$. However, suppose that we had a constraint that the size of the record, defined as the sum of the lengths of the fields, is $10n$. The probability of that happening, using the `defdata`-generated enumerator, is $\frac{1}{10^n}$, which for large n is essentially 0. Or, suppose that we have a dependent type where the lengths of the fields are required to be equal. The probability of that happening is $\frac{1}{10^{n-1}}$, which is also essentially 0 for large n .

The idea of enumerative data types with constraints is that we allow users to define types with parameters. These parameters are associated with functions over the data types and we require that, given values for these parameters, efficient enumerators for the types can be defined. For example, consider a list type with a parameter corresponding to the length of the list; the associated function is just the length function. Given a particular length, it is easy to generate a list of that length by generating the required number of elements using the enumerator for the element type. The next idea is to allow users to define constraints over the parameters and associated functions of types. If these constraints are over a decidable fragment of logic, then enumeration winds up becoming a two-stage process by which we find satisfying assignments to the constraints, providing values for the parameters, which are then used by the corresponding enumerators. Consider the motivating example where we had fields f_1, \dots, f_n with parameters p_1, \dots, p_n , corresponding to the field lengths. The constraint that the size of the record is $10n$ gets turned into a constraint that the sum of the lengths of the fields, is $10n$ and this can be given to an SMT/IMT solver [33]–[35]. This is a simple constraint, which in terms of the parameters is $p_1 + \dots + p_n = 10n$, and which only has one solution, namely $p_i = 10$. With the appropriate values for the parameters, we can now call the enumerators for the fields of the record, which will generate lists of the appropriate length, with probability 1. In general, enumerators require solving a set of constraints and then calling enumerators of component types, which may also require solving a set of constraints, and so on, recursively. As an optimization, recursive constraints associated with an enumerator can be packaged into single queries during the enumerator generation process, thereby minimizing the number of constraint-solving queries required by enumerators.

In our Wi-Fi application, and more generally in other verification efforts, we want to determine the satisfiability of a set of ACL2s constraints which include not only various data types, but also other constraints arising from a variety of sources, including coverage criteria, responses to messages from the DUT, well-formedness constraints, protocol constraints and modeling constraints. Queries to the underlying solver consist of the maximal subsets of these ACL2s constraints that can be expressed in the theory supported by the solver. If such a query is unsatisfiable, so is the corresponding ACL2s query; if the query is satisfiable, then we have values for the data type parameters which can be used to efficiently (without constraint solving) generate satisfying assignments to the datatype variables. If there are any remaining constraints, they are handled by the ACL2s counterexample generation process.

As we show later, we can formalize complex protocol interactions using types. These types include fields that are ordered lists over certain numbers, that have variable length and optional fields and that include other complex dependencies. Finding satisfying assignments to such types is difficult for current SMT solvers, but easy when using enumerative data types with constraints because we use constraint solving only for the true dependencies; we then we use the enumerative

```
(solver-init)
(z3-assert (x :bool y :int z (:seq (:bv 3)))
  (and x (>= y 5) (= (seq.len z) y)))
(check-sat)
$> ;; This is SAT, so we get a model:
((X T) (Y 5) (Z (0 0 0 0 0)))
```

Fig. 3. An example showing the use of our Common Lisp-Z3 interface.

characterization of `defdata` to generate assignments using computation alone (*i.e.*, no constraint solving).

V. IMPLEMENTATION

We implemented enumerative data types with constraints in ACL2s, which provides support for defining tools on top of ACL2s via “ACL2s systems programming” [36]. We used Z3 as the constraint solver, which required that we integrate Z3 with ACL2s. To this end, we developed a library allowing one to easily call Z3 from Common Lisp. In this section, we will describe both the Common Lisp-Z3 interface library, and how we interacted with ACL2s.

Common Lisp-Z3 Interfacing: We decided to implement a close integration of ACL2 and Z3, using the CFFI Common Lisp library [37] to directly load Z3 into an ACL2s process and interact with it using Z3’s C API. Such a close integration brings several benefits, including a low overhead when interacting with Z3 and the ability to support Z3 features like incremental solving. We developed our own Common Lisp library that provides both a low-level interface with Z3’s C API and a high-level interface that allows the user to add assertions to Z3 using a syntax similar to that of ACL2s’ `property` macro. See Fig. 3 for an example showing the use of our library. Our interface supports a broad swathe of Z3’s features, including many of its built-in functions and types, several kinds of user-generated types and incremental solving.

ACL2s Interfacing: Since our system is implemented using the ACL2s systems programming paradigm, we are able to write Common Lisp code that calls into ACL2s. Our system starts inside the ACL2 read-eval-print loop (REPL), where we load in the ACL2s model that we will pull enumerators from. We then are able to exit from the ACL2 REPL into the underlying Common Lisp REPL that our copy of ACL2 is built on top of, where we can load any Common Lisp code that we might want, including our Common Lisp-Z3 library. To evaluate a function inside of ACL2—for example, an enumerator for a `defdata` type—we first generate an S-expression corresponding to the function call, and then pass that S-expression to the appropriate function provided by Walter *et al.*’s `acl2s-interface` library [38].

For our application, after running Z3 and getting back a length for each element of the structure being generated, we need to then generate elements with those lengths. Since each variable-length element has a list type corresponding to the set of bodies that it may have, we can make use of a special kind of enumerator that ACL2s produces for list types. This enumerator takes two arguments: the number of elements to generate, and the random seed to use. To generate an element

of a list type with a particular length, we simply call the enumerator with the desired length and an appropriate random seed. We can then construct our structure from its constituent parts by performing an appropriate ACL2s call.

VI. WI-FI MODEL CASE STUDY AND EVALUATION

We present an application of enumerative data types with constraints to hardware-in-the-loop 802.11 wireless router fuzzing. We focus on the problem of generating a particular kind of 802.11 MAC frame, the *probe request* frame, as this is already sufficiently complex to present the challenges in modeling and frame generation. We first describe some challenges that come with hardware-in-the-loop fuzzing before discussing the probe request frame in more depth. We then discuss two models of the probe request frame that we developed, the first using Lustre and the second using ACL2s. We highlight the key challenges that arose when developing the Lustre model, and how we were able to use ACL2s to surmount these challenges and produce a more concise model. We then describe a system that implements enumerative data types with constraints alongside the ACL2s model, and conclude with experiments showing that our enumerative data type approach is able to generate probe request frames at a significantly greater rate and for a wider range of frame sizes than either a pure constraint solving approach or a pure enumerative data type approach.

Hardware-in-the-loop Fuzzing for Protocol Conformance

Fuzzing a hardware system like a wireless router brings with it certain requirements on the fuzzer and fuzzing infrastructure. The device under test (DUT) needs to be monitored, an interface must be formed between the DUT and the fuzzer, and in the case of protocol fuzzing, the fuzzer may be required to adhere to timing constraints imposed by the DUT. The latter constraint means that the performance of a fuzzer may not just affect how long it may take to find a particular vulnerability, but it may entirely preclude a fuzzer from use if it cannot generate a fuzzed response to a message sent by the DUT quickly enough.

The systems described below are intended to be one part of a larger hardware-in-the-loop fuzzing system, an architecture of which can be seen in Fig. 4. Each approach that we describe contains two parts: a model describing the probe request frame, and a fuzzer that uses the model to generate descriptions of concrete 802.11 probe request frames given some additional constraints on the size of the frame.

The Probe Request Frame

When a wireless device aims to connect to a 802.11 Wi-Fi access point, it must first gather information on the capabilities of wireless access points that are within range. To do this, the wireless device first sends out a *probe request* message with some basic information on its capabilities. Any Wi-Fi access point that is within range and supports at least one of the capabilities advertised by the wireless device will then respond with a *probe response* message containing information about

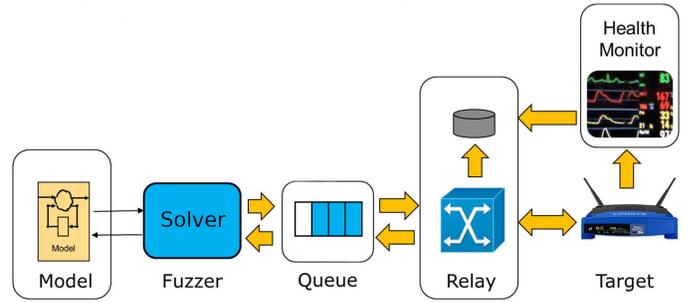


Fig. 4. An overview of a hardware-in-the-loop fuzzing architecture

itself. The wireless device will then select an access point to connect to and continue exchanging messages. The details of this process are described in the IEEE 802.11 specification [1]. Here we concern ourselves with the MAC frame corresponding to the probe request message.

The 802.11 specification states that every MAC frame consists of three parts: a header, a body, and a frame check sequence (FCS), which is a checksum for the previous two parts. We will not discuss the header and FCS parts, as the hardware-in-the-loop testing system can take care of setting the header and FCS as appropriate.

A probe request frame body consists of a variable-length sequence of elements, some of which are optional. Any elements that appear must appear in a specified order relative to each other. Elements typically contain a 1-byte “Element ID” field that has a constant value for all elements of a particular type, a 1-byte “Length” field that indicates the number of bytes remaining in the element after the end of the “Length” field, an optional 1-byte “Element ID Extension” field, and a variable-length set of element-specific fields. In this paper, we will consider the element-specific fields to all be concatenated into one “Body” field. The size of a probe request frame is the sum of the size of the MAC header (32 bytes) and the sizes of all elements appearing in the frame body. The 802.11 specification enumerates 33 element types for the probe request frame body, and the constraints on valid values for each element type vary widely. For example, the “DSSS Parameter Set” element’s body is 1 byte long and should specify the “Current Channel” that the device is using; the set of valid values depends on the PHY implementation being used as well as some other settings. The “Request” element’s body has a more complicated constraint: it is a variable-length list of bytes corresponding to “Element ID”s, and the bytes must be listed in increasing order. As we will see, such constraints are difficult to express in Lustre, and lead to a lengthy specification.

The Lustre Model

Our first model of the 802.11 probe request frame was developed using the Lustre programming language. When modeling the probe request frame body specification, we chose to abstract away some details of the specification in the interest of focusing on aspects of the specification that are interesting

```

type RequestElementType = struct {
  ElementID : byte      ;
  Len       : byte      ;
  Body      : byte[255] };
--Each element of the Body field is a byte.
node RequestElementTypeAssertions
  (e: RequestElementType) returns (r: bool);
let
r = ...
(0<=e.Len      ) and (e.Len      <=255) and
(0<=e.Body[0] ) and (e.Body[0]<=255) and ...
(0<=e.Body[254]) and (e.Body[254]<=255);
tel
--The first Length elements of Body are
--sorted.
node RequestElementOrderedElementIDConstraint
  (e: RequestElementType) returns (r: bool);
let
r =
((e.Len<1) or (e.Body[0]<e.Body[1])) and ...
((e.Len<254) or (e.Body[253]<e.Body[254]));

```

Fig. 5. A code snippet highlighting how an element containing a variable-length sorted array of bytes is modeled in Lustre.

and representative. For example, we simply modeled the body of the DSSS parameter set element as a byte. In general, the Lustre model constrains the *shapes* of elements but not their body values, which we believe is reasonable considering the model is intended for use for fuzzing. That is, the Lustre model specifies probe request frame bodies that are of valid lengths and that have elements in the correct locations, but does not constrain the exact values that the body of each element may take to only those that are valid based on the 802.11 specification.

Lustre does not provide built-in support for bounded integer types, which means that specifying that a field is a byte is done by declaring that the field is an integer and that its value is between 0 and 255 inclusive. This becomes even more problematic when modeling variable-length arrays: to model an array of bytes of length between 0 and 255, the Lustre model specifies an array of length 255, specifies a variable representing the length of the array and adds a constraint for every element of the array stating that its value should be between 0 and 255 inclusive. This means that 255 array elements are always generated, and the system consuming values generated from the Lustre model simply omits any array elements that occur past the generated length value. Specifying the “Request” element is even more verbose, since in addition to the aforementioned constraints, 254 constraints are generated to specify that if the length of the array is greater than i , the element at index $i - 1$ in the array is strictly less than the element at index i . See Fig. 5 for a snippet of the Lustre model that defines a frame element with a variable-length sorted array of bytes.

The Lustre model was used in conjunction with FuzzM to generate probe request frames. FuzzM was not able to generate assignments for certain frame sizes, as the SMT queries did not produce results even given a timeout of many minutes.

```

;; A natural number less than 256
(defnatrang uint8 (expt 2 8))
;; a list of uint8s with a length in [0,255)
(defdata-list byte255 uint8 0 255)
;; a byte255 that is also strictly ordered
(defdata-ordered-list byte255-increasing uint8
 0 255)
;; Sanity check: should always be able to find
;; a byte255 that is not a byte255-increasing
(must-fail (property (x :byte255)
  (byte255-increasingp x)))
;; A type for the constant 10
(defdata exact10 10)
;; We model elements using records
(defdata RequestElementType
  (record (ElementID . exact10)
          (Body      . byte255-increasing)))

```

Fig. 6. A snippet of the ACL2s model showing how an element containing a variable-length sorted array of bytes is modeled. Also included are sanity checks that do not appear in the Lustre model.

The ACL2s Model

We developed an ACL2s model based on the Lustre model. The ACL2s model makes heavy use of `defdata`, which has a much more powerful notion of types than Lustre. The expressiveness of ACL2s allows us to more succinctly encode the constraints imposed by the 802.11 standard. `defdata` has built-in support for bounded integer types, making redundant many of the constraints that had to be stated explicitly in the Lustre model. We also used the extensions described in Section III to define list types with length bounds and ordering constraints. Fig. 6 shows all of the definitions necessary to model the “Request” element in ACL2s with our extensions.

Another benefit of developing our model in ACL2s is that we can include sanity checks inline with the model. ACL2s will evaluate the checks when the model is loaded during development, helping catch mistakes in the model specification that may otherwise go undetected. These checks can include validating that ACL2s can find a counterexample to a property (as seen in Fig. 6) but also may include proofs or code execution. If proofs are included, they may be used by ACL2s to prove or generate counterexamples to future conjectures. Even with sanity checks, the ACL2s version of the model has roughly a quarter of the lines of code present in the Lustre model.

Evaluation

We performed experiments to compare the performance of three approaches to probe request frame generation: enumerative data types using the ACL2s model and `cgen` (ACL2s-ET below), enumerative data types with constraints using the ACL2s model and an application-specific prototype of the approach described in Section IV (ACL2s-ETC below), and a pure constraint solving approach using a Z3-only version of the Lustre model and Z3 (Z3 below).

We measured the performance of each approach when queried for probe request frame bodies of various sizes, including sizes for which no probe request frame body exists.

Z3 and ACL2s were both configured to timeout after 20 seconds. ACL2s was set to use the `:uniform-random cgen` sampling method and was configured to terminate once it found a single counterexample rather than the default three; this brings its behavior more into line with Z3’s. All other Z3 and ACL2s settings were left in their default state. We provide code for reproducing these experiments along with this paper.

Fig. 7 shows the number of query responses per minute for each approach across a range of probe request frame body sizes from 0 to 5000 bytes, sampled every 10 bytes. Five trials were performed for each frame size for all approaches. The number of query responses per minute for a particular approach and probe request frame body size was calculated by dividing the total number of queries made for that size that resulted in definitive responses (*e.g.* not timeouts) by the total amount of time in minutes spent on all queries for that size.

There are three regimes of frame size to discuss:

Small invalid probe request frame sizes (0-170 bytes): We expected all of the approaches to perform well in this regime. ACL2s-ETC consistently was able to determine UNSAT across this range of sizes, and the Z3-only approach performed well up to sizes of 150 bytes. ACL2s-ET was only able to determine sizes up to 30 bytes were UNSAT; all of the other queries in this regime resulted in timeouts. Note that Z3’s performance begins to fall exponentially for frame sizes of 160 or greater.

Valid probe request frame sizes (180-2740 bytes): ACL2s-ETC is consistently able to generate frames at a rate greater than 1000 per minute, while ACL2s-ET is only able to generate frames for a subset of the frame sizes at a rate of at most 22 per minute and the Z3 approach is unable to generate any frames with a size greater than 300 bytes. The distribution of ACL2s-ET’s response rate (approximately normally distributed around the average valid frame size of 1456 bytes) suggests that ACL2s is falling back on random generation of frame bodies; that is, generating a frame body by independently and randomly generating each element without consideration of the frame size constraint. The exponential drop in the Z3 approach’s performance suggests that Z3’s search space grows exponentially with frame size.

Large invalid probe request frame sizes (2750-5000 bytes): ACL2s-ETC is consistently able to quickly determine these sizes are UNSAT, while ACL2s-ET can do so slowly but consistently. The Z3 approach is always able to determine UNSAT, though it was only able to do so in all of the experimental trials in 100 of the 226 probe request frame sizes sampled between 2750 and 5000 bytes. This highlights inconsistency in Z3’s ability to determine UNSAT for large frame sizes.

These results highlight the weaknesses of the Z3-only and ACL2s-ET approaches. The Z3 approach was able to quickly determine that small frame sizes are impossible and was consistently able to generate frames with sizes up to 210 bytes. However, the proportion of trials that resulted in SAT responses began to quickly drop after that point, and no SAT responses were received for trials with valid packet sizes of 290 bytes or greater. The Z3 approach’s performance was

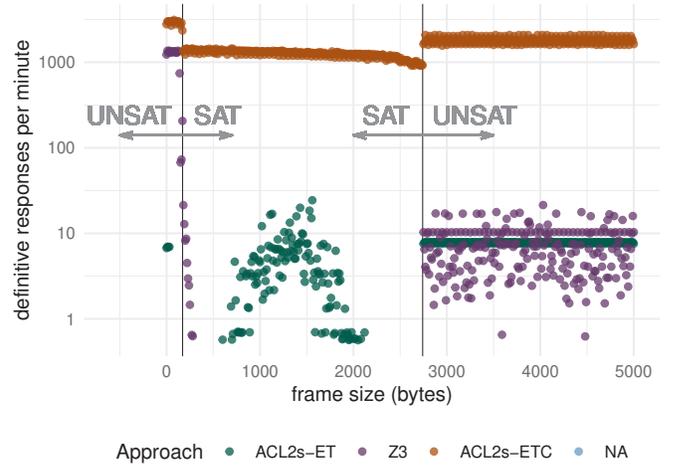


Fig. 7. The number of frames generated per minute using each of the three approaches when queried for frames with a given length. Only instances where the model returned a definitive response (*e.g.* not “unknown” or “timeout”) are shown. The two vertical lines represent the minimum frame size and the maximum frame size; any responses outside of that range were all UNSAT, and any within that range were SAT.

highly variable for determining that larger frame sizes are impossible, and though it was inconsistent, it was always able to show UNSAT in at least one of the five trials performed. It is possible that an alternative encoding of the Z3 model (for example, one that does not make use of Z3’s sequence types) would perform better, but our experience in using the Lustre model with FuzzM does not suggest a significant improvement in performance.

The ACL2s-ET approach was consistently able to show that large frame sizes are impossible, and was able to generate frames for a wider range of frame sizes than the Z3 approach, though it struggled to generate large or small frames and to show that very small frame sizes are impossible. ACL2s is not using information from the frame size constraints to guide its counterexample generation in a meaningful way; `cgen` could be modified to improve its effectiveness here.

VII. FUTURE WORK

This paper introduces the idea of enumerative data types with constraints, or, equivalently, the idea of enumerative dependent types. We believe that this idea will be useful in many applications, *e.g.*, those requiring the analysis and verification of systems and models defined using dependent data types. Such applications include property-based testing, model-based development and distributed systems.

Below we provide a partial list of ideas for future work.

Formalizations and extensions: We plan on developing and formalizing the theory of enumerative data types with constraints for ACL2s and encourage others to develop similar formalizations for other dependent type systems and interactive theorem provers. We suspect that there are numerous interesting directions in which the basic approach can be extended to handle dependent logics of varying expressive power.

A specific extension of interest involves supporting relations of arbitrary arity, not just predicates. Conceptually this is straightforward: the relation can be turned into a predicate by combining all of the relation’s arguments into a single value (a tuple or a record). Then, our approach allows us to represent and handle dependencies between the relation’s arguments. A user can manually perform the conversion from relation to predicate, but ideally this could be done automatically.

ACL2 integration: We plan to provide first-class support for enumerative data types with constraints as part of the ACL2s `defdata` framework, so that ACL2s users can benefit from our work without needing to write custom code. Our proof-of-concept implementation used for this paper’s evaluation uses ACL2s systems programming [36] techniques and is not integrated with ACL2s.

Optimizations: The ACL2s-ETC implementation evaluated in this work was not optimized, and we are confident that there are opportunities for both general and application-specific performance improvements in our method. One such optimization that we have experimented with in the context of stateful protocols is to perform offline (pre-enumeration) analyses of the protocol’s state machine to identify how to efficiently explore interesting regions of the protocol’s state space. This pre-analysis can significantly reduce the amount of work needed at enumeration time to generate appropriate responses to messages from the SUT. There are also interesting questions regarding coverage metrics and “fair” explorations that model analyses can help answer.

Model extraction: One limitation of our current work is that it requires models that are described using dependent types. An interesting question whether it is possible to provide automatic techniques that are able to take existing models and annotate them with the type information requires to use our work. This line of research can include the use of AI techniques such as Natural Language Processing (NLP) to automatically translate legacy prose descriptions of protocols into formal models that can be analyzed using our approach.

VIII. CONCLUSION

In this paper, we introduced the idea of enumerative data types with constraints. This allows us to use formal-methods-in-the-loop in the context of hardware-in-the-loop fuzzing for conformance testing of distributed protocols. We presented a case study where we modeled a portion of the IEEE 802.11 Wi-Fi specification and showed that we are able to generate messages for a wide variety of sizes, something that previous methods cannot do, thereby enabling the use of formal methods in new applications. Interesting directions for future work include adding such capabilities to other formal methods tools and using enumerative data types to analyze other distributed protocols.

Acknowledgments: This work was funded in part by the United States Department of the Navy, Office of Naval Research under contract N68335-17-C-0238. We thank Kristopher Cory and Grant Foudree for their support.

REFERENCES

- [1] “IEEE standard for information technology–telecommunications and information exchange between systems - local and metropolitan area networks–specific requirements - part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications,” *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pp. 1–4379, 2021.
- [2] J. Thomas Barnett, S. Jain, U. Andra, and T. Khurana. Cisco visual networking index (VNI) complete forecast update, 2017–2022. Cisco Systems, Inc. Accessed on May 21st, 2022. [Online]. Available: https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1213-business-services-ckn.pdf
- [3] L. Butti. wifuzzit. [Online]. Available: <https://github.com/0xd012/wifuzzit>
- [4] E7mer. owfuzz. [Online]. Available: <https://github.com/alipay/Owfuzz>
- [5] H. R. Chamarthi, P. C. Dillinger, and P. Manolios, “Data definitions in the ACL2 sedan,” in *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, F. Verbeek and J. Schmaltz, Eds., vol. 152, 2014, pp. 27–48.
- [6] H. R. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon, “The acl2 sedan theorem proving system,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 291–295.
- [7] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore, “ACL2s: “the ACL2 sedan”,” *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 2, pp. 3–18, 2007, proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006). [Online]. Available: <https://doi.org/10.1016/j.entcs.2006.09.018>
- [8] A. T. Walter, D. Greve, and P. Manolios. Enumerative data types with constraints supporting material. [Online]. Available: <https://gitlab.com/acl2s/external-tool-support/enumerative-data-types-with-constraints-supporting-material>
- [9] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.
- [10] M. Kaufmann and J. S. Moore, “ACL2 homepage,” 2022. [Online]. Available: <https://www.cs.utexas.edu/users/moore/acl2/>
- [11] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios, “Integrating testing and interactive theorem proving,” in *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, D. S. Hardin and J. Schmaltz, Eds., vol. 70, 2011, pp. 4–19.
- [12] H. R. Chamarthi and P. Manolios, “Automated specification analysis using an interactive theorem prover,” in *International Conference on Formal Methods in Computer-Aided Design, FMCAD ’11*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 46–53. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2157665>
- [13] H. R. Chamarthi, “Interactive non-theorem disproving,” Ph.D. dissertation, Northeastern University, 2016.
- [14] P. Manolios and D. Vroon, “Termination analysis with calling context graphs,” in *Computer Aided Verification, 18th International Conference, CAV, Proceedings*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 401–414.
- [15] —, “Algorithms for ordinal arithmetic,” in *19th International Conference on Automated Deduction – CADE-19*, ser. LNAI, F. Baader, Ed., vol. 2741. Springer-Verlag, July/August 2003, pp. 243–257.
- [16] —, “Integrating reasoning about ordinal arithmetic into ACL2,” in *Formal Methods in Computer-Aided Design FMCAD*, ser. LNCS. Springer-Verlag, November 2004.
- [17] —, “Ordinal Arithmetic: Algorithms and Mechanization,” *Journal of Automated Reasoning*, vol. 34, no. 4, pp. 387–423, 2005.
- [18] K. Claessen and J. Hughes, “QuickCheck: A lightweight tool for random testing of Haskell programs,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’00. ACM, 2000, p. 268–279. [Online]. Available: <https://doi.org/10.1145/351240.351266>
- [19] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. [Online]. Available: <http://mitpress.mit.edu/catalog/item/default.asp?type=2&tid=10928>
- [20] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, “Automated test generation and mutation testing for Alloy,” in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*. IEEE Computer Society, 2017, pp. 264–275. [Online]. Available: <https://doi.org/10.1109/ICST.2017.31>

- [21] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, “Query-aware test generation using a relational constraint solver,” in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*.
- [22] G. Soltana, M. Sabetzadeh, and L. C. Briand, “Practical constraint solving for generating system test data,” *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, apr 2020. [Online]. Available: <https://doi.org/10.1145/3381032>
- [23] C. Robert, J. Guiochet, H. Waeselynck, and L. V. Sartori, “TAF: a tool for diverse and constrained test case generation,” in *21st IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2021. [Online]. Available: <https://hal.laas.fr/hal-03435959>
- [24] R. Coppa, G. Foudree, and D. Greve, “FuzzM: A model-based approach to grey-box fuzzing,” Rockwell Collins, Tech. Rep., 2018. [Online]. Available: <http://loonwerks.com/publications/pdf/coppa2018techreport.pdf>
- [25] A. Gacek, J. Backes, M. Whalen, L. G. Wagner, and E. Ghassabani, “The JKind model checker,” in *Computer Aided Verification - 30th International Conference, CAV 2018, Proceedings, Part II*, ser. LNCS, H. Chockler and G. Weissenbacher, Eds., vol. 10982. Springer, 2018, pp. 20–27. [Online]. Available: https://doi.org/10.1007/978-3-319-96142-2_3
- [26] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous data flow programming language LUSTRE,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.
- [27] D. A. Greve and A. Gacek, “Trapezoidal generalization over linear constraints,” in *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications*, ser. EPTCS, S. Goel and M. Kaufmann, Eds., vol. 280, 2018, pp. 30–46. [Online]. Available: <https://doi.org/10.4204/EPTCS.280.3>
- [28] M. Vanhoef and F. Piessens, “Key reinstallation attacks: Forcing nonce reuse in WPA2,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. ACM, 2017, p. 1313–1328. [Online]. Available: <https://doi.org/10.1145/3133956.3134027>
- [29] L. Butti and J. Tinnés, “Discovering and exploiting 802.11 wireless driver vulnerabilities,” *Journal in Computer Virology*, vol. 4, no. 1, pp. 25–37, 2008. [Online]. Available: <https://doi.org/10.1007/s11416-007-0065-x>
- [30] P. Biondi. scapy. [Online]. Available: <https://github.com/secdev/scapy>
- [31] M. Vanhoef, D. Schepers, and F. Piessens, “Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing,” in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017*, R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, Eds. ACM, 2017, pp. 360–371. [Online]. Available: <https://doi.org/10.1145/3052973.3053008>
- [32] M. E. Garbelini, C. Wang, and S. Chattopadhyay, “Greyhound: Directed greybox Wi-Fi fuzzing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 817–834, 2022. [Online]. Available: <https://doi.org/10.1109/TDSC.2020.3014624>
- [33] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Proceedings*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.
- [34] P. Manolios and V. Papavasileiou, “ILP modulo theories,” in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 662–677.
- [35] P. Manolios, J. Pais, and V. Papavasileiou, “The Inez mathematical programming modulo theories framework,” in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 53–69.
- [36] A. T. Walter and P. Manolios, “ACL2s systems programming,” in *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, 2022, to be published.
- [37] J. Bielman and L. Oliveira. CFFI—the common foreign function interface. [Online]. Available: <http://common-lisp.net/project/cffi>
- [38] P. Manolios and A. Walter. ACL2s interface. [Online]. Available: <https://gitlab.com/acl2s/external-tool-support/interface>

Reducing NEXP-complete problems to DQBF

Fa-Hsun Chen

National Taiwan University
r10944015@ntu.edu.tw

Shen-Chang Huang

National Taiwan University
b07902135@ntu.edu.tw

Yu-Cheng Lu

National Taiwan University
luyucheng@protonmail.com

Tony Tan

National Taiwan University
tonytan@csie.ntu.edu.tw

Abstract—We present an alternative proof of the NEXP-hardness of the satisfiability of *Dependency Quantified Boolean Formulas* (DQBF). Besides being simple, our proof also gives us a general method to reduce NEXP-complete problems to DQBF. We demonstrate its utility by presenting explicit reductions from a wide variety of NEXP-complete problems to DQBF such as (succinctly represented) 3-colorability, Hamiltonian cycle, set packing and subset-sum as well as NEXP-complete logics such as the Bernays-Schönfinkel-Ramsey class, the two-variable logic and the monadic class. Our results show the vast applications of DQBF solvers which recently have gathered a lot of attention among researchers.

Index Terms—Dependency quantified boolean formulas (DQBF), NEXP-complete problems, polynomial time (Karp) reductions, succinctly represented problems

I. INTRODUCTION

The last few decades have seen a tremendous development of boolean SAT solvers and their applications in many areas of computing [1]. Motivated by applications in verification and synthesis of hardware/software designs [2]–[8], researchers have recently looked at the generalization of boolean formulas known as *dependency quantified boolean formulas* (DQBF).

While solving boolean SAT is “only” NP-complete, for DQBF the complexity jumps to NEXP-complete [9]. This makes solving DQBF quite a challenging research topic. Nevertheless there has been exciting progress. See, e.g., [10]–[18] and the references within, as well as solvers such as iDQ [19], dCAQE [20], HQS [21], [22] and DQBDD [23]. A natural question to ask is if we can use DQBF solvers to solve any NEXP-complete problems – similar to how SAT solvers are used to solve any NP-complete problems.

In this short paper we show how to reduce a wide variety of NEXP-complete problems to DQBF, especially the succinctly represented problems that recently have found applications in hardware/software engineering [24]–[26]. We present another proof for the NEXP-hardness of DQBF. We actually give two proofs. The first is by a very simple reduction from *succinct 3-colorability* [27]. The second is by utilizing the notion that we call *succinct projection*. It is the second one that we view more interesting since it gives us a general method to reduce any NEXP-complete problem to DQBF.

The main idea is quite standard: We encode the accepting runs of a non-deterministic Turing machine (with exponential run time) with boolean functions of polynomial arities. However, we observe that the input-output relation of these functions can actually be “described” by small circuits/formulas. Succinct projections are simply deterministic algorithms that

construct these circuits efficiently. This simple observation is a deviation from the standard definition of NEXP, that a language in NEXP is a language with an exponentially long certificate.

Using succinct projections, we present reductions from various NEXP-complete problems such as (succinct) *Hamiltonian cycle*, *set packing* and *subset sum*. We believe our technique can be easily modified for many other natural problems. Note that the reduction in [9] gives little insight on how it can be used to obtain explicit reductions from concrete NEXP-complete problems.

We also present the reductions from well known NEXP-complete logics such as *the Bernays-Schönfinkel-Ramsey class*, *two-variable logic* (FO^2) and *the Löwenheim class* [28]–[32]. In fact we show that they are essentially equivalent to DQBF. Note that these are logics that have found applications in AI [33], databases [34] and automated reasoning [35], but lack implementable algorithms. Prior to our work, the only algorithm known for these logics is to “guess” a model (of exponential size) and then verify that it is indeed a model of the input formula.

We hope that the technique introduced in this short paper can lead to richer applications of DQBF solvers as well as a wide variety of benchmarks which in turn can lead to further development. It is also open whether the class NEXP has a *bona-fide* problem [27]. Our paper demonstrates that DQBF can be a good candidate – akin to how boolean SAT is the central problem in the class NP.

This paper is organized as follows. In Sect. II we review some definitions and terminology. In Sect. III we reprove the NEXP-completeness of solving DQBF. In Sect. IV and V we present concrete reductions from some NEXP-complete problems and logics to DQBF instances. The full version of this paper can be found in [36].

II. PRELIMINARIES

Let $\Sigma = \{0, 1\}$. We usually use the symbol $\bar{a}, \bar{b}, \bar{c}$ (possibly indexed) to denote a string in Σ^* with $|\bar{a}|$ denoting the length of \bar{a} . We use $\bar{x}, \bar{y}, \bar{z}, \bar{u}, \bar{v}$ to denote vectors of boolean variables. The length of \bar{x} is denoted by $|\bar{x}|$. We write $C(\bar{u})$ to denote a (boolean) circuit C with input gates \bar{u} . When the input gates are not relevant or clear from the context, we simply write C . For $\bar{a} \in \Sigma^{|\bar{u}|}$, $C(\bar{a})$ denotes the value of C when we assign the input gates \bar{u} with \bar{a} . All logarithms have base 2.

A *dependency quantified boolean formula* (DQBF) in prenex normal form is a formula of the form:

$$\Psi := \forall x_1 \cdots \forall x_n \exists y_1(\bar{z}_1) \cdots \exists y_m(\bar{z}_m) \psi \quad (1)$$

where each \bar{z}_i is a vector of variables from $\{x_1, \dots, x_n\}$ and ψ , called *the matrix*, is a quantifier-free boolean formula using variables $x_1, \dots, x_n, y_1, \dots, y_m$. The variables x_1, \dots, x_n are called *the universal variables*, y_1, \dots, y_m *the existential variables* and each \bar{z}_i *the dependency set* of y_i .

A DQBF Ψ in the form (1) is *satisfiable*, if for every $1 \leq i \leq m$, there is a function $s_i : \Sigma^{|\bar{z}_i|} \rightarrow \Sigma$ such that by replacing each y_i with $s_i(\bar{z}_i)$, the formula ψ becomes a tautology. The function s_i is called *the Skolem function* for y_i . In this case, we also say that Ψ is satisfiable by the Skolem functions s_1, \dots, s_m . The problem SAT(DQBF) is defined as: On input DQBF Ψ in the form (1), decide if it is satisfiable.

Since many NEXP-complete problems use circuits as the succinct representations of the inputs, we allow the matrix ψ to be in *circuit form*, i.e., ψ is given as a (boolean) circuit with input gates $x_1, \dots, x_n, y_1, \dots, y_m$. This does not effect the generality of our results, since every DQBF in circuit form can be converted to one in the standard formula form as stated in Proposition 1.

Proposition 1. *Every DQBF Ψ in the form of (1) in circuit form can be converted in polynomial time into an equisatisfiable DQBF formula Ψ' whose matrix is in DNF. Moreover, Ψ and Ψ' have the same existential variables (with the same dependency set).*

The proof is by standard Tseitin's transformation [37]. As an example, consider the following DQBF.

$$\forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2) \neg(x_2 \vee (y_1 \wedge x_1 \wedge y_2))$$

It is equisatisfiable with the following DQBF.

$$\forall x_1 \forall x_2 \forall u_1 \forall u_2 \forall u_3 \forall v_1 \forall v_2 \exists y_1(x_1) \exists y_2(x_2) \left(\begin{array}{l} (v_1 \leftrightarrow y_1) \wedge (v_2 \leftrightarrow y_2) \wedge (u_1 \leftrightarrow v_1 \wedge x_1 \wedge v_2) \\ \wedge (u_2 \leftrightarrow x_2 \vee u_1) \wedge (u_3 \leftrightarrow \neg u_2) \end{array} \right) \rightarrow u_3$$

Intuitively, we use the extra variable v_1 to represent the value y_1 , v_2 the value y_2 , u_1 the value $y_1 \wedge x_1 \wedge y_2$, u_2 the value $x_2 \vee (y_1 \wedge x_1 \wedge y_2)$ and u_3 the value $\neg(x_2 \vee (y_1 \wedge x_1 \wedge y_2))$. Note that the matrix can be easily rewritten into DNF.

III. THE NEXP-COMPLETENESS OF SAT(DQBF)

In this section we present two new proofs that SAT(DQBF) is NEXP-complete, originally proved in [9].

Theorem 2. [9] *SAT(DQBF) is NEXP-complete.*

Note that the membership is straightforward. So we will focus only on the hardness.

A. The first proof: Reduction from succinct 3-colorability

The reduction is from the problem *graph 3-colorability* where the input graphs are given in a succinct form [24]. A (boolean) circuit $C(\bar{u}, \bar{v})$, where $|\bar{u}| = |\bar{v}| = n$, represents a

graph $G(C) = (V, E)$ where $V = \Sigma^n$ and $(\bar{a}, \bar{b}) \in E$ iff $C(\bar{a}, \bar{b}) = 1$. The problem *succinct 3-colorability* is defined as: On input circuit C , decide if $G(C)$ is 3-colorable. This problem is NEXP-complete [27].

The reduction to SAT(DQBF) is as follows. Let $C(\bar{u}, \bar{v})$ be the input circuit, where $|\bar{u}| = |\bar{v}| = n$. We represent a 3-coloring of $G(C)$ as a function $g : \Sigma^n \rightarrow \{01, 10, 11\}$ which can be encoded by the following DQBF.

$$\Psi := \forall \bar{x}_1 \forall \bar{x}_2 \exists y_1(\bar{x}_1) \exists y_2(\bar{x}_1) \exists y_3(\bar{x}_2) \exists y_4(\bar{x}_2) \quad (2)$$

$$\bar{x}_1 = \bar{x}_2 \rightarrow (y_1, y_2) = (y_3, y_4) \quad (2)$$

$$\wedge (y_1, y_2) \neq (0, 0) \wedge (y_3, y_4) \neq (0, 0) \quad (3)$$

$$\wedge C(\bar{x}_1, \bar{x}_2) = 1 \rightarrow (y_1, y_2) \neq (y_3, y_4) \quad (4)$$

Intuitively, we use y_1, y_2 and y_3, y_4 to represent the first and the second bits of the image $g(\bar{x}_1)$ and $g(\bar{x}_2)$, respectively. Lines (2) and (3) state that (y_1, y_2) and (y_3, y_4) must represent the same function from Σ^n to Σ^2 and that their images do not include 00. Line (4) states that the colors of two adjacent vertices must be different. Thus, $G(C)$ is 3-colorable iff Ψ is satisfiable.

B. The second proof: Reduction via succinct projections

Our second proof uses the notion of *succinct projection*. We need some terminology. Let $C(\bar{u}_1, \bar{v}_1, \bar{u}_2, \bar{v}_2)$ be a circuit with input gates $\bar{u}_1, \bar{v}_1, \bar{u}_2, \bar{v}_2$ where $|\bar{u}_1| = |\bar{u}_2| = n$ and $|\bar{v}_1| = |\bar{v}_2| = m$. We say that a function $g : \Sigma^n \rightarrow \Sigma^m$ *agrees* with the circuit C , if $C(w_1, g(w_1), w_2, g(w_2)) = 1$, for every $w_1, w_2 \in \Sigma^n$. In this case, we also say that the circuit C *describes* the function g . In the following whenever we say that a function $g : \Sigma^n \rightarrow \Sigma^m$ *agrees* with $C(\bar{u}_1, \bar{v}_1, \bar{u}_2, \bar{v}_2)$, we implicitly assume that $n = |\bar{u}_1| = |\bar{u}_2|$ and $m = |\bar{v}_1| = |\bar{v}_2|$.

Definition 3. A *succinct projection* for a language L is a polynomial time deterministic algorithm \mathcal{M} such that on input $w \in \Sigma^*$, \mathcal{M} outputs a circuit C such that $w \in L$ iff there is a function g that agrees with C .

Intuitively, we can view the function g as the certificate for the membership of w in L and the circuit C as the succinct description of g . Since succinct projection runs in polynomial time, the output circuit can only have polynomially many gates. The following theorem is a new characterization of languages in NEXP.

Theorem 4. *A language $L \in \text{NEXP}$ iff it has a succinct projection.*

Proof. (if) Suppose that L has a succinct projection. Consider the following algorithm. On input w , first use the succinct projection to construct the circuit C . Then, guess a function g (of exponential size) and verify that it agrees with C . It is obvious that it runs in non-deterministic exponential time. That it is correct follows from the definition of succinct projection.

(only if) It is essentially the Cook-Levin reduction disguised in the form of function certificates. We only sketch it here. Let $L \in \text{NEXP}$ and M be a 1-tape NTM that accepts L in time $2^{p(n)}$ for some polynomial $p(n)$. For a word $w \in L$ of

length n , its accepting run can be represented as a function $g : \Sigma^{p(n)} \times \Sigma^{p(n)} \rightarrow \Sigma^\ell$, where $g(i, j)$ denotes the content of cell i in time j . The tuples in the codomain Σ^ℓ encode the states and the tape symbols of M . To verify that g represents an accepting run, it is sufficient to verify that for every $i_1, j_1, i_2, j_2 \in \Sigma^{p(n)}$, the tuple $(i_1, j_1, g(i_1, j_1), i_2, j_2, g(i_2, j_2))$ satisfies a certain property P which depends only on the input word w and the transitions of M . The desired succinct projection constructs in polynomial time a circuit C describing this property P . \square

The second proof of the NEXP-hardness of SAT(DQBF): Let $L \in \text{NEXP}$. The polynomial time (Karp) reduction from L to SAT(DQBF) is described as Algorithm 1 below.

Algorithm 1: Reducing $L \in \text{NEXP}$ to SAT(DQBF)

Input: $w \in \Sigma^*$.

1: Run the succinct projection of L on w .

2: Let $C(\bar{x}_1, \bar{y}_1, \bar{x}_2, \bar{y}_2)$ be the output circuit where

$$|\bar{x}_1| = |\bar{x}_2| = n, |\bar{y}_1| = |\bar{y}_2| = m, \bar{y}_1 = (y_{1,1}, \dots, y_{1,m})$$

$$\text{and } \bar{y}_2 = (y_{2,1}, \dots, y_{2,m}).$$

3: Output the following DQBF Ψ :

$$\forall \bar{x}_1 \forall \bar{x}_2 \exists y_{1,1}(\bar{x}_1) \cdots \exists y_{1,m}(\bar{x}_1) \exists y_{2,1}(\bar{x}_2) \cdots \exists y_{2,m}(\bar{x}_2)$$

$$C(\bar{x}_1, \bar{y}_1, \bar{x}_2, \bar{y}_2) \wedge (\bar{x}_1 = \bar{x}_2 \rightarrow \bar{y}_1 = \bar{y}_2)$$

We show $w \in L$ iff Ψ is satisfiable. Suppose $w \in L$. Let $g : \Sigma^n \rightarrow \Sigma^m$ be a function that agrees with C . For each $1 \leq i \leq m$, define the Skolem function $s_i : \Sigma^n \rightarrow \Sigma$ where $s_i(\bar{a})$ is the i -th component of $g(\bar{a})$, for every $\bar{a} \in \Sigma^n$. It is routine to verify that Ψ is satisfiable with each s_i being the Skolem function for $y_{1,i}$ and $y_{2,i}$.

Conversely, suppose Ψ is satisfiable. Let $s_{j,i} : \Sigma^n \rightarrow \Sigma$ be the Skolem function for $y_{j,i}$, where $1 \leq j \leq 2$ and $1 \leq i \leq m$. Since $\bar{x}_1 = \bar{x}_2 \rightarrow \bar{y}_1 = \bar{y}_2$, the functions $s_{1,i}$ and $s_{2,i}$ must be the same, for every $1 \leq i \leq m$. Define $g : \Sigma^n \rightarrow \Sigma^m$ where $g(\bar{a}) = (s_1(\bar{a}), \dots, s_{1,m}(\bar{a}))$ for every $\bar{a} \in \Sigma^n$. Since $C(\bar{a}_1, g(\bar{a}_1), \bar{a}_2, g(\bar{a}_2))$ is true for every \bar{a}_1, \bar{a}_2 , the function g agrees with C . That is, there is a function that agrees with C . Hence, $w \in L$. This completes the second proof.

Remark 5. Observe that when Theorem 4 is applied to languages in NP, the accepting run of a non-deterministic Turing machine with polynomial run time $p(n)$ is represented as a function $g : \Sigma^{\log p(n)} \times \Sigma^{\log p(n)} \rightarrow \Sigma^\ell$ and the succinct projection outputs a circuit $C(\bar{x}_1, \bar{y}_1, \bar{x}_2, \bar{y}_2)$ where $|\bar{x}_1| = |\bar{x}_2| = \log p(n)$ and $|\bar{y}_1| = |\bar{y}_2| = \ell$. Thus, for $L \in \text{NP}$, the DQBF output by Algorithm 1 has $4 \log p(n)$ universal variables and 2ℓ existential variables.

IV. SOME CONCRETE REDUCTIONS

In this section we show how to utilize succinct projection to obtain the reductions from concrete NEXP-complete problems to SAT(DQBF). These are (succinct) *Hamiltonian cycle*, *set packing* and *subset sum* [27]. We use the notion of succinctness from [24] which has been explained in Sect. III-A. By Algorithm 1, it suffices to present only the succinct projections.

Some useful notations: For an integer $k \geq 1$, $[k]$ denotes the set $\{0, \dots, k-1\}$. For $i \in [2^n]$, $\text{bin}_n(i)$ is the binary representation of i in n bits. The number represented by $\bar{a} \in \Sigma^n$ is denoted by $\text{num}(\bar{a})$. For $\bar{a}, \bar{b} \in \Sigma^n$, if $\text{num}(\bar{a}) = \text{num}(\bar{b}) + 1 \pmod{2^n}$, we say that \bar{a} is the successor of \bar{b} , denoted by $\bar{a} = \bar{b} + 1$. Note that successor is applied only on two strings with the same length and the successor of 1^n is 0^n . It is not difficult to construct a circuit $C(\bar{x}, \bar{y})$ (in time polynomial in $|\bar{x}| + |\bar{y}|$) such that $C(\bar{a}, \bar{b}) = 1$ iff $\bar{a} = \bar{b} + 1$.

Reduction from succinct Hamiltonian cycle: Succinct Hamiltonian cycle is defined as follows. The input is a circuit $C(\bar{u}, \bar{v})$. The task is to decide if there is a Hamiltonian cycle in $G(C)$.

Let $C(\bar{u}, \bar{v})$ be the input circuit where $|\bar{u}| = |\bar{v}| = n$. We use a function $g : \Sigma^n \rightarrow \Sigma^n$ to represent a Hamiltonian cycle $(\bar{b}_0, \dots, \bar{b}_{2^n-1})$ where $g(\text{bin}_n(i)) = \bar{b}_i$, for every $i \in [2^n]$. To correctly represent a Hamiltonian cycle, the following must hold for every $\bar{a}_1, \bar{a}_2 \in \Sigma^n$.

(H1) If $\bar{a}_1 \neq \bar{a}_2$, then $g(\bar{a}_1) \neq g(\bar{a}_2)$.

(H2) If $\bar{a}_2 = \bar{a}_1 + 1$, then $(g(\bar{a}_1), g(\bar{a}_2))$ is an edge in $G(C)$.

The succinct projection for succinct Hamiltonian cycle simply outputs the circuit that expresses (H1) and (H2), i.e., it outputs the following circuit $D(\bar{x}_1, \bar{y}_1, \bar{x}_2, \bar{y}_2)$ where $|\bar{x}_1| = |\bar{x}_2| = |\bar{y}_1| = |\bar{y}_2| = n$:

$$(\bar{x}_1 \neq \bar{x}_2 \rightarrow \bar{y}_1 \neq \bar{y}_2) \wedge (\bar{x}_2 = \bar{x}_1 + 1 \rightarrow C(\bar{y}_1, \bar{y}_2) = 1)$$

Obviously, a function $g : \Sigma^n \rightarrow \Sigma^n$ represents a hamiltonian cycle in $G(C)$ iff it agrees with D .

Reduction from succinct set packing: In the standard representation the problem *set packing* is defined as follows. The input is a collection \mathcal{K} of finite sets $S_1, \dots, S_\ell \subseteq \Sigma^m$ and an integer k . The task is to decide whether \mathcal{K} contains k mutually disjoint sets. We assume each S_i has a ‘‘name’’ which is a string in $\Sigma^{\log \ell}$.

The succinct representation of the sets S_1, \dots, S_ℓ is a circuit $C(\bar{u}, \bar{v})$ where $|\bar{u}| = m$ and $|\bar{v}| = \log \ell$. A string $\bar{a} \in \Sigma^m$ is in the set $S_{\bar{b}}$, if $C(\bar{a}, \bar{b}) = 1$. We denote by $\mathcal{K}(C)$ the collection of finite sets defined by the circuit C . The problem *succinct set packing* is defined analogously where the input is the circuit $C(\bar{u}, \bar{v})$ and an integer k (in binary).

We now describe its succinct projection. Let $C(\bar{u}, \bar{v})$ and k be the input where $|\bar{u}| = m$ and $|\bar{v}| = n$. We first assume that k is a power of 2. We represent k disjoint sets S_1, \dots, S_k in $\mathcal{K}(C)$ as a function $g : \Sigma^{\log k} \times \Sigma^m \rightarrow \Sigma^n$ where $g(\text{bin}(i), \bar{a})$ is the name of the set S_i . Note that the string \bar{a} is actually ignored in the definition of g .

For a function $g : \Sigma^{\log k} \times \Sigma^m \rightarrow \Sigma^n$ to correctly represent k disjoint sets, the following must hold for every $(\bar{a}_1, \bar{b}_1), (\bar{a}_2, \bar{b}_2) \in \Sigma^{\log k} \times \Sigma^m$.

(P1) If $\bar{a}_1 = \bar{a}_2$, then $g(\bar{a}_1, \bar{b}_1) \neq g(\bar{a}_2, \bar{b}_2)$. That is, the function g does not depend on \bar{b}_1 and \bar{b}_2 .

(P2) If $\bar{a}_1 \neq \bar{a}_2$ and $\bar{b}_1 = \bar{b}_2$, then $C(\bar{b}_1, g(\bar{a}_1, \bar{b}_1)) = 0$ or $C(\bar{b}_1, g(\bar{a}_2, \bar{b}_2)) = 0$. That is, the element \bar{b}_1 is not in the sets whose names are $g(\bar{a}_1, \bar{b}_1)$ and $g(\bar{a}_2, \bar{b}_2)$.

It is routine to verify that g represents k disjoint sets iff (P1) and (P2) hold for every $(\bar{a}_1, \bar{b}_1), (\bar{a}_2, \bar{b}_2) \in \Sigma^{\log k} \times \Sigma^m$.

The succinct projection outputs the following circuit D that formalizes (P1) and (P2):

$$\begin{aligned} &(\bar{x}_1 = \bar{x}_2 \rightarrow \bar{z}_1 = \bar{z}_2) \\ &\wedge (\bar{x}_1 \neq \bar{x}_2 \wedge \bar{y}_1 = \bar{y}_2) \rightarrow \neg(C(\bar{y}_1, \bar{z}_1) = C(\bar{y}_1, \bar{z}_2) = 1) \end{aligned}$$

If k is not a power of 2, we conjunct both atoms $\bar{x}_1 = \bar{x}_2$ and $\bar{x}_1 \neq \bar{x}_2$ with a circuit that tests whether the numbers represented by the bits \bar{x}_1 and \bar{x}_2 is an integer in $[k]$. Such a circuit can be easily constructed in polynomial time in $\lceil \log k \rceil$.

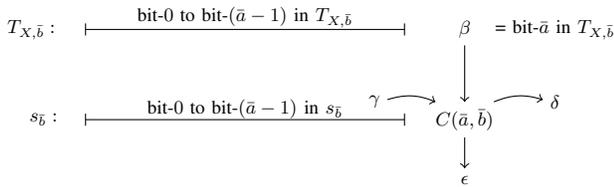
Reduction from succinct subset-sum: In the standard representation the instance of subset-sum is a list of positive integers s_0, \dots, s_{k-1} and t (all written in binary). The task is to decide if there is a subset $X \subseteq [k]$ such that $\sum_{i \in X} s_i = t$. Such X is called the subset-sum solution. The succinct representation is defined as two circuits $C_1(\bar{u}_1, \bar{v})$ and $C_2(\bar{u}_2)$, where $|\bar{u}_1| = \max_{i \in [k]} \log s_i$, $|\bar{v}| = \log k$ and $|\bar{u}_2| = \log t$. Circuit C_1 defines the numbers s_i 's where $C_1(\bar{a}, \bar{b})$ is the i -th least significant bit of s_j , where $i = \text{num}(\bar{a})$ and $j = \text{num}(\bar{b})$. Circuit C_2 defines the number t where $C_2(\bar{a})$ is the i -th least significant bit of t , where $i = \text{num}(\bar{a})$. The subset-sum instance represented by C_1 and C_2 is denoted by $\mathcal{N}(C_1, C_2)$. We will describe the succinct projection for succinct subset-sum.

Let $C_1(\bar{u}_1, \bar{v})$ and $C_2(\bar{u}_2)$ be the input where $|\bar{u}_1| = |\bar{u}_2| = n$ and $|\bar{v}| = m$. We need a few notations. Let s_0, \dots, s_{2^m-1} be the numbers represented by C_1 and t the number represented by C_2 . For a set $X \subseteq [2^m]$, let $T_X = \sum_{i \in X} s_i$. For $0 \leq j \leq 2^m$, let $T_{X,j} = T_{X \cap [j]}$. Abusing the notation, for $\bar{b} \in \Sigma^m$, we write $s_{\bar{b}}$ and $T_{X,\bar{b}}$ to denote s_i and $T_{X,i}$, respectively, where $i = \text{num}(\bar{b})$. For $\bar{a} \in \Sigma^n$, bit- \bar{a} means bit- i where $i = \text{num}(\bar{a})$.

We represent a set $X \subseteq [2^m]$ as a function $g : \Sigma^n \times \Sigma^m \rightarrow \Sigma^5$ where $g(\bar{a}, \bar{b}) = (\alpha, \beta, \gamma, \delta, \epsilon)$ such that:

- $\alpha = 1$ iff $s_{\bar{b}} \in X$.
- β is bit- \bar{a} in $T_{X,\bar{b}}$.
- γ is the carry of adding $T_{X,\bar{b}}$ and $s_{\bar{b}}$ up to bit- $(\bar{a} - 1)$.
- $\delta\epsilon = \beta + \gamma + C(\bar{a}, \bar{b})$, i.e., ϵ is the least significant bit of $\beta + \gamma + C(\bar{a}, \bar{b})$ and δ is the carry.

See the illustration below.



Intuitively, $g(\bar{a}, \bar{b})$ contains the information about the additions performed on bit- \bar{a} in $s_{\bar{b}}$ (with respect to the set X). In particular, the bits of the number T_X are all contained in $g(\bar{a}, 1^m)$ for every $\bar{a} \in \Sigma^n$. These bits can then be compared to those in t by means of the circuit C_2 .

Note that for a function $g : \Sigma^n \times \Sigma^m \rightarrow \Sigma^5$ to properly represent a number T_X , for some $X \subseteq [2^m]$, it suffices to check the values of g on “neighbouring” points in $\Sigma^n \times \Sigma^m$. More precisely, the following conditions must be satisfied for every $(\bar{a}_1, \bar{b}_1), (\bar{a}_2, \bar{b}_2) \in \Sigma^n \times \Sigma^m$, where $g(\bar{a}_1, \bar{b}_1) = (\alpha_1, \beta_1, \gamma_1, \delta_1, \epsilon_1)$ and $g(\bar{a}_2, \bar{b}_2) = (\alpha_2, \beta_2, \gamma_2, \delta_2, \epsilon_2)$.

- (i) If $\bar{b}_1 = \bar{b}_2$, then $\alpha_1 = \alpha_2$. That is, the value α_1 depends only on the index of a number.
- (ii) If $\alpha_1 = 0$, then $\gamma_1 = \delta_1 = 0$ and $\beta_1 = \epsilon_1$.
- (iii) If $\alpha_1 = 1$, then $\gamma_1 + C(\bar{a}_1, \bar{b}_1) + \beta_1 = \delta_1 \epsilon_1$.
- (iv) If $\bar{a}_1 = 0^n$, then $\gamma_1 = 0$.
- (v) If $\bar{a}_1 = 1^n$, then $\delta_1 = 0$.
- (vi) If $\bar{b}_1 = 0^m$, then $\beta_1 = \gamma_1 = 0$.
- (vii) If $\bar{b}_1 = 1^m$, then $\epsilon_1 = C_2(\bar{a}_1)$.
- (viii) If $\alpha_1 = 1$ and $\bar{b}_1 = \bar{b}_2$ and $\bar{a}_2 = \bar{a}_1 + 1$, then $\delta_1 = \gamma_2$.
- (ix) If $\alpha_1 = 1$ and $\bar{b}_2 = \bar{b}_1 + 1$ and $\bar{a}_2 = \bar{a}_1$, then $\epsilon_1 = \beta_2$.

Intuitively, (ii) and (iii) state that the values of $(\alpha_1, \beta_1, \gamma_1, \delta_1, \epsilon_1)$ must have their intended meaning, i.e., when $\alpha_1 = 0$, no addition is performed and when $\alpha_1 = 1$, the addition $\gamma_1 + C(\bar{a}_1, \bar{b}_1) + \beta_1$ is performed and the result is $\delta_1 \epsilon_1$. (iv) states that there is no carry from the previous bit when considering the least significant bit. (v) states that there shouldn't be any carry after adding the most significant bit (if we want T_X equals t). (vi) states that $T_{X,0}$ must be zero. (vii) states that bit- \bar{a} in T_X must equal to bit- \bar{a} in t . Finally, (viii) and (ix) state that when (\bar{a}_1, \bar{b}_1) and (\bar{a}_2, \bar{b}_2) are neighbors, the bits $\beta_1, \gamma_1, \delta_1, \epsilon_1$ and $\beta_2, \gamma_2, \delta_2, \epsilon_2$ must obey their intended meaning.

Obviously, if g satisfies (i)–(ix), then it represents a set X such that $T_X = t$. Conversely, if there is a set X such that $T_X = t$, then there is a function g that satisfies (i)–(ix). It is not difficult to design a succinct projection that constructs a circuit D that describes functions that satisfy (i)–(ix).

V. REDUCTIONS FROM OTHER NEXP-COMPLETE LOGICS

In this section we will consider the following fragments of relational first-order logic (with the equality predicate):

- The *Bernays-Schönfinkel-Ramsey* (BSR) class: The class of sentences of the form:

$$\Psi_1 := \exists x_1 \dots \exists x_m \forall y_1 \dots \forall y_n \psi$$

where ψ is a quantifier-free formula.

- The *two-variable logic* (FO²): The class of sentences using only two variables x and y .

The classic result by Scott [38] states that every FO² sentence can be transformed in linear time into an equisatisfiable FO² sentence of the form:

$$\Psi_2 := \forall x \forall y \alpha(x, y) \wedge \bigwedge_{i=1}^m \forall x \exists y \beta_i(x, y)$$

for some $m \geq 1$, where $\alpha(x, y)$ and each $\beta_i(x, y)$ are quantifier free formulas.

- The *Löwenheim/monadic* class: The class of sentences using only unary predicate symbols. Sentences in this class are also known as monadic sentences.

Let $\text{SAT}(\text{BSR})$, $\text{SAT}(\text{Mon})$ and $\text{SAT}(\text{FO}^2)$ denote their corresponding satisfiability problems. It is well known that all of them are NEXP-complete [28]–[32]. The upper bound is usually established by the so called *Exponential Size Model* (ESM) property stated as follows.

- If the BSR sentence Ψ_1 is satisfiable, then it is satisfiable by a model with size at most $m + 1$ [31, Prop. 6.2.17].
- If the FO^2 sentence Ψ_2 is satisfiable, then it is satisfiable by a model with size $m2^n$, where n is the number of unary predicates used [30].
- If a Löwenheim sentence is satisfiable, then it is satisfiable by a model with size at most $r2^n$, where r is the quantifier rank and n is the number of unary predicates [31, Prop. 6.2.1].

The main idea of the reduction to $\text{SAT}(\text{DQBF})$ is quite simple. We will represent the domain of a model with size at most N as a subset of Σ^t , where $t = \log N$ and use a function $f_0 : \Sigma^t \rightarrow \Sigma$ as the indicator whether an element is in the domain. Every predicate in the input formula can be represented as a function $f : \Sigma^{kt} \rightarrow \Sigma$ where k is the arity of the predicate. All these functions can then be encoded appropriately as existential variables in DQBF. Note that the universal FO quantifier $\forall x \dots$ can be encoded as $\forall \bar{u} f_0(\bar{u}) \rightarrow \dots$. The existential FO quantifier can first be Skolemized and then encoded as existential variables in DQBF.

The rest of this section is organized as follows. For technical convenience, we first introduce the logic Existential Second-order Quantified Boolean Formula ($\exists\text{SOQBF}$) – an alternative, but equivalent formalism of DQBF. The only difference between $\exists\text{SOQBF}$ and DQBF is the syntax in declaring the function symbol. Then, we consider the problem that we call *Bounded FO satisfiability*, denoted by $\text{Bnd-SAT}(\text{FO})$, which subsumes all $\text{SAT}(\text{BSR})$, $\text{SAT}(\text{FO}^2)$ and $\text{SAT}(\text{Mon})$ and show how to reduce it to $\text{SAT}(\exists\text{SOQBF})$.

The logic $\exists\text{SOQBF}$: The class $\exists\text{SOQBF}$ is the extension of quantified boolean formulas (QBF) with existential second-order quantifiers, i.e., formulas of the form:

$$\Psi := \exists f_1 \exists f_2 \dots \exists f_p Q_1 v_1 \dots Q_n v_n \psi$$

where each $Q_i \in \{\forall, \exists\}$ and each f_i is a boolean function symbol associated with a fixed arity $\text{ar}(f_i)$. The formula ψ is a boolean formula using the variables v_i 's and $f(\bar{z})$'s, where $f \in \{f_1, \dots, f_p\}$, $|\bar{z}| = \text{ar}(f)$ and $\bar{z} \subseteq \{v_1, \dots, v_q\}$. We call each $f(\bar{z})$ in ψ a *function variable*.

The semantics of Ψ is defined naturally. We say that Ψ is satisfiable, if there is an interpretation $F_i : \Sigma^{\text{ar}(f_i)} \rightarrow \Sigma$ for each f_i such that $Q_1 v_1 \dots Q_n v_n \psi$ is a true QBF. In this case we say that F_1, \dots, F_p *make Ψ true*. It is not difficult to see that DQBF and $\exists\text{SOQBF}$ can be transformed to each other in linear time while preserving satisfiability.

Bounded FO satisfiability ($\text{Bnd-SAT}(\text{FO})$): The problem $\text{Bnd-SAT}(\text{FO})$ is defined as: On input relational FO sentence φ and a positive integer N (in binary), decide if φ has a model with cardinality at most N . It is a folklore that $\text{Bnd-SAT}(\text{FO})$ is NEXP-complete. Note that due to the ESM property, $\text{Bnd-SAT}(\text{FO})$ trivially subsumes all $\text{SAT}(\text{BSR})$, $\text{SAT}(\text{FO}^2)$ and $\text{SAT}(\text{Mon})$.

Reduction from $\text{Bnd-SAT}(\text{FO})$ to $\text{SAT}(\exists\text{SOQBF})$: Let φ and N be the input to $\text{Bnd-SAT}(\text{FO})$. We may assume that φ is in the Prenex normal form: $\varphi := Q_1 x_1 \dots Q_n x_n \psi$, where each $Q_i \in \{\forall, \exists\}$ and ψ is quantifier-free formula. Adding

redundant quantifier, if necessary, we assume that Q_1 is \forall . Then, we Skolemize each existential quantifier as follows. Let i be the minimal index where $Q_i = \exists$. We rewrite φ into:

$$\begin{aligned} \varphi' &:= \forall x_1 \dots \forall x_{i-1} Q_{i+1} x_{i+1} \dots Q_n x_n \forall z \\ &z = g(x_1, \dots, x_{i-1}) \rightarrow \psi' \end{aligned}$$

where z is a fresh variable, g is the Skolem function representing the existentially quantified variable x_i and ψ' is obtained from ψ by replacing every occurrence of x_i with z . Hence, we may assume that the input sentence φ is of form:

$$\varphi := \forall x_1 \dots \forall x_n \psi \quad (5)$$

where ψ is quantifier-free formula where every (Skolem) function symbol $g(x_1, \dots, x_{i-1})$ only occur in the equality predicate $z = g(x_1, \dots, x_{i-1})$ and z is one of x_i, \dots, x_n .

Let g_1, \dots, g_k be the Skolem function symbols in ψ and P_1, \dots, P_ℓ be the predicates in ψ . Let $\text{ar}(g_i)$ and $\text{ar}(P_i)$ denote the arity of g_i and P_i . Let $t = \lceil \log N \rceil$. Construct the following $\exists\text{SOQBF}$ formula:

$$\begin{aligned} \Phi &:= \exists f_0 \exists f_{1,1} \dots \exists f_{1,t} \dots \exists f_{k,1} \dots \exists f_{k,t} \exists f_{P_1} \dots \exists f_{P_\ell} \\ &\forall \bar{u}_1 \dots \forall \bar{u}_n \left(\begin{array}{l} \bar{u}_1 = 0^t \rightarrow f_0(\bar{u}_1) \\ \wedge \bigwedge_{i=1}^n f_0(\bar{u}_i) \rightarrow \Psi \end{array} \right) \quad (6) \end{aligned}$$

where:

- The arity of f_0 is t .
- For every $1 \leq i \leq k$, the arity of $f_{i,1}, \dots, f_{i,t}$ is $t \cdot \text{ar}(g_i)$.
- For every $1 \leq i \leq \ell$, the arity of $f_{P_1}, \dots, f_{P_\ell}$ is $t \cdot \text{ar}(P_i)$.
- For every $1 \leq i \leq n$, $|\bar{u}_i| = t$.

The formula Ψ is obtained from ψ as follows.

- Each predicate $P_i(x_{j_1}, \dots, x_{j_m})$ is replaced with $f_{P_i}(\bar{u}_{j_1}, \dots, \bar{u}_{j_m})$.
- Each predicate $x_j = g_i(x_{j_1}, \dots, x_{j_m})$ is replaced with $\bar{u}_j = (f_{i,1}(\bar{u}_{j_1}, \dots, \bar{u}_{j_m}), \dots, f_{i,t}(\bar{u}_{j_1}, \dots, \bar{u}_{j_m}))$
- Each predicate $x_j = x_i$ is replaced with $\bar{u}_j = \bar{u}_i$.

Intuitively, we use f_0 as the indicator to determine whether a string in Σ^t is an element in the model. To ensure that the model is not empty, we insist that 0^t belongs to the model, hence, the formula $\bar{u}_1 = 0^t \rightarrow f_0(\bar{u}_1)$. We use the vector of variables \bar{u}_i to represent x_i . For every $1 \leq i \leq k$, the functions $f_{i,1}, \dots, f_{i,t}$ represent the bit representation of $g_i(x_{j_1}, \dots, x_{j_m})$. Finally, for every $1 \leq i \leq \ell$, the function f_{P_i} represents the predicate P_i . Note the part $\bigwedge_{i=1}^n f_0(\bar{u}_i) \rightarrow \Psi$ which means we require Ψ to hold only on the vectors $\bar{u}_1, \dots, \bar{u}_n$ that “passes” the function f_0 , i.e., they are elements of the model. It is routine to verify that the formula φ in Eq. (5) is satisfiable by a model with cardinality at most N iff the $\exists\text{SOQBF}$ formula Φ in Eq. (6) is satisfiable.

ACKNOWLEDGEMENT

We are very grateful to Jie-Hong Roland Jiang for many fruitful discussions on the preliminary drafts of this work. We also thank the anonymous reviewers for their constructive comments. We acknowledge the generous financial support of Taiwan National Science and Technology Council under grant no. 109-2221-E-002-143-MY3.

REFERENCES

- [1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. IOS Press, 2009.
- [2] J. R. Jiang, “Quantifier elimination via functional composition,” in *CAV*, 2009.
- [3] V. Balabanov and J. R. Jiang, “Reducing satisfiability and reachability to DQBF,” in *Talk given at QBF*, 2015.
- [4] C. Scholl and B. Becker, “Checking equivalence for partial implementations,” in *DAC*, 2001.
- [5] K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker, “Equivalence checking of partial designs using dependency quantified boolean formulae,” in *ICCD*, 2013.
- [6] R. Bloem, R. Könighofer, and M. Seidl, “SAT-based synthesis methods for safety specs,” in *VMCAI*, 2014.
- [7] K. Chatterjee, T. Henzinger, J. Otop, and A. Pavlogiannis, “Distributed synthesis for LTL fragments,” in *FMCAD*, 2013.
- [8] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, 2002.
- [9] G. Peterson and J. Reif, “Multiple-person alternation,” in *FOCS*, 1979.
- [10] V. Balabanov, H. K. Chiang, and J. R. Jiang, “Henkin quantifiers and boolean formulae: A certification perspective of DQBF,” *Theor. Comput. Sci.*, vol. 523, pp. 86–100, 2014.
- [11] A. Fröhlich, G. Kovásznai, and A. Biere, “A DPLL algorithm for solving DQBF,” in *POS-12, Third Pragmatics of SAT workshop*, 2012.
- [12] A. Ge-Ernst, C. Scholl, and R. Wimmer, “Localizing quantifiers for DQBF,” in *FMCAD*, 2019.
- [13] O. Kullmann and A. Shukla, “Autarkies for DQCNF,” in *FMCAD*, 2019.
- [14] R. Wimmer, C. Scholl, and B. Becker, “The (D)QBF preprocessor hqspre - underlying theory and its implementation,” *J. Satisf. Boolean Model. Comput.*, vol. 11, no. 1, pp. 3–52, 2019.
- [15] K. Wimmer, R. Wimmer, C. Scholl, and B. Becker, “Skolem functions for DQBF,” in *ATVA*, 2016.
- [16] R. Wimmer, S. Reimer, P. Marin, and B. Becker, “HQSpre – an effective preprocessor for QBF and DQBF,” in *TACAS*, 2017.
- [17] G. Kovásznai, “What is the state-of-the-art in DQBF solving,” in *Join Conference on Mathematics and Computer Science*, 2016.
- [18] C. Scholl and R. Wimmer, “Dependency quantified boolean formulas: An overview of solution methods and applications - extended abstract,” in *SAT*, 2018.
- [19] A. Fröhlich, G. Kovásznai, A. Biere, and H. Veith, “iDQ: Instantiation-based DQBF solving,” in *POS-14, Fifth Pragmatics of SAT workshop*, 2014.
- [20] L. Tentrup and M. Rabe, “Clausal abstraction for DQBF,” in *SAT*, 2019.
- [21] K. Gitina, R. Wimmer, S. Reimer, M. Sauer, C. Scholl, and B. Becker, “Solving DQBF through quantifier elimination,” in *DATE*, 2015.
- [22] R. Wimmer, A. Karrenbauer, R. Becker, C. Scholl, and B. Becker, “From DQBF to QBF by dependency elimination,” in *SAT*, 2017.
- [23] J. Síc and J. Strejcek, “DQBDD: an efficient bdd-based DQBF solver,” in *SAT*, 2021.
- [24] H. Galperin and A. Wigderson, “ Succinct representations of graphs,” *Inf. Control.*, vol. 56, no. 3, pp. 183–198, 1983.
- [25] D. Kini, U. Mathur, and M. Viswanathan, “Data race detection on compressed traces,” in *ESEC/SIGSOFT FSE*, 2018.
- [26] A. Pavlogiannis, N. Schaumberger, U. Schmid, and K. Chatterjee, “Precedence-aware automated competitive analysis of real-time scheduling,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3981–3992, 2020.
- [27] C. Papadimitriou and M. Yannakakis, “A note on succinct representations of graphs,” *Inf. Control.*, vol. 71, no. 3, pp. 181–185, 1986.
- [28] H. Lewis, “Complexity results for classes of quantificational formulas,” *J. Comput. Syst. Sci.*, vol. 21, no. 3, pp. 317–353, 1980.
- [29] M. Fürer, “The computational complexity of the unconstrained limited domino problem (with implications for logical decision problems),” in *Logic and Machines: Decision Problems and Complexity*, 1983, pp. 312–319.
- [30] E. Grädel, P. Kolaitis, and M. Vardi, “On the decision problem for two-variable first-order logic,” *Bull. Symbolic Logic*, vol. 3, no. 1, pp. 53–69, 3 1997.
- [31] E. Börger, E. Grädel, and Y. Gurevich, *The Classical Decision Problem*. Springer, 1997.
- [32] T. Lin, C. Lu, and T. Tan, “Towards a more efficient approach for the satisfiability of two-variable logic,” in *LICS*, 2021.
- [33] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
- [34] S. Itzhaky, T. Kotek, N. Rinetzky, M. Sagiv, O. Tamir, H. Veith, and F. Zuleger, “On the automated verification of web applications with embedded SQL,” in *ICDT*, 2017, pp. 16:1–16:18.
- [35] J. Robinson and A. Voronkov, Eds., *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.
- [36] F.-H. Chen, S.-C. Huang, Y.-C. Lu, and T. Tan, “Reducing NEXP-complete problems to DQBF,” *CoRR*, vol. abs/2208.06014, 2022. [Online]. Available: <http://arxiv.org/abs/2208.06014>
- [37] G. Tseitin, “On the complexity of derivation in propositional calculus,” in *Studies in Constructive Mathematics and Mathematical Logic, Part II*, 1968.
- [38] D. Scott, “A decision method for validity of sentences in two variables,” *The Journal of Symbolic Logic*, p. 377, 1962.

INC: A Scalable Incremental Weighted Sampler

Suwei Yang
National University of Singapore
Singapore
suwei.yang@comp.nus.edu.sg

Victor Liang
GrabTaxi Holdings Pte. Ltd.
Singapore
victor.liang@grab.com

Kuldeep S. Meel
National University of Singapore
Singapore
meel@comp.nus.edu.sg

Abstract—The fundamental problem of weighted sampling involves sampling of satisfying assignments of Boolean formulas, which specify sampling sets, and according to distributions defined by pre-specified weight functions to weight functions. The tight integration of sampling routines in various applications has highlighted the need for samplers to be incremental, i.e., samplers are expected to handle updates to weight functions.

The primary contribution of this work is an efficient knowledge compilation-based weighted sampler, INC¹, designed for incremental sampling. INC builds on top of the recently proposed knowledge compilation language, OBDD[\wedge], and is accompanied by rigorous theoretical guarantees. Our extensive experiments demonstrate that INC is faster than state-of-the-art approach for majority of the evaluation. In particular, we observed a median of 1.69 \times runtime improvement over the prior state-of-the-art approach.

Index Terms—knowledge compilation, sampling, weighted sampling

I. INTRODUCTION

Given a Boolean formula F and weight function W , weighted sampling involves sampling from the set of satisfying assignments of F according to the distribution defined by W . Weighted sampling is a fundamental problem in many fields such as computer science, mathematics and physics, with numerous applications. In particular, constrained-random simulation forms the bedrock of modern hardware and software verification efforts [1].

Sampling techniques are fundamental building blocks, and there has been sustained interest in the development of sampling tools and techniques. Recent years witnessed the introduction of numerous sampling tools and techniques, from approximate sampling techniques to uniform samplers SPUR and KUS, and weighted sampler WAPS [2]–[6]. Sampling tools and techniques have seen continuous adoption in many applications and settings [7]–[12]. The scalability of a sampler is a consideration that directly affects its adoption rate. Therefore, improving scalability continues to be a key objective for the community focused on developing samplers.

The tight integration of sampling routines in various applications has highlighted the importance for samplers to handle incremental weight updates over multiple sampling rounds, also known as incremental weighted sampling. Existing efforts on improving scalability typically focus on single round weighted sampling, and might have overlooked the incremental setting. In particular, existing approaches involving incremental

weighted sampling typically employ off-the-shelf weighted samplers which could lead to less than ideal incremental sampling performance.

The primary contribution of this work is an efficient scalable weighted sampler INC that is designed from the ground up to address scalability issues in incremental weighted sampling settings. The core architecture of INC is based on knowledge compilation (KC) paradigm, which seeks to succinctly represent all satisfying assignments of a Boolean formula with a directed acyclic graph (DAG) [13]. In the design of INC, we make two core decisions that are responsible for outperforming the current state-of-the-art weighted sampler. Firstly, we build INC on top of PROB (Probabilistic OBDD[\wedge] [14]) which is substantially smaller than the KC diagram used in the prior state-of-the-art approaches. Secondly, INC is designed to perform *annotation*, which refers to the computation of joint probabilities, in log-space to avoid the slower alternative of using arbitrary precision math computations.

Given a Boolean formula F and weight function W , INC compiles and stores the compiled PROB in the first round of sampling. The weight updates for subsequent incremental sampling rounds are processed without recompilation, amortizing the compilation cost. Furthermore, for each sampling round, INC simultaneously performs *annotation* and sampling in a single bottom-up pass of the PROB, achieving speedup over existing approaches. We observed that INC is significantly faster than the existing state-of-the-art in the incremental sampling routine. In our empirical evaluations, INC achieved a median of 1.69 \times runtime improvement over the state-of-the-art weighted sampler, WAPS [6]. Additional performance breakdown analysis supports our design choices in the development of INC. In particular, PROB is on median 4.64 \times smaller than the KC diagram used by the competing approach, and log-space *annotation* computations are on median 1.12 \times faster than arbitrary precision computations. Furthermore, INC demonstrated significantly better handling of incremental sampling rounds, with incremental sampling rounds to be on median 5.9% of the initial round, compared to 67.6% for WAPS.

The rest of the paper is organized as follows. We first introduce the relevant background knowledge and related works in Section II. We then introduce PROB and its properties in Section III. In Section IV, we introduce our weighted sampler INC, detail important implementation decisions, and provide theoretical analysis of INC. We then describe the extensive

¹code available at <https://github.com/grab/inc-weighted-sampler/>

empirical evaluations and discuss the results in Section V. Finally, we conclude in Section VI.

II. BACKGROUND AND RELATED WORK

Knowledge Compilation: Knowledge compilation (KC) involves representing logical formulas as directed acyclic graphs (DAG), which are commonly referred to as knowledge compilation diagrams [13]. The goal of knowledge compilation is to allow for tractable computation of certain queries such as model counting and weighted sampling. There are many well-studied forms of knowledge compilation diagrams such as d-DNNF, SDD, BDD, ZDD, OBDD, AOBDD, and the likes [15]–[21]. In this work, we build our weighted sampler upon a variant of OBDD known as OBDD[\wedge] [14].

OBDD[\wedge]: Lee [15] introduced Binary Decision Diagram (BDD) as a way to represent Shannon expansion [22]. [16] introduced fixed variable orderings to BDDs (known as OBDD) [16] for canonical representation and compression of BDDs via shared sub-graphs. Lai et al. [14] introduced conjunction nodes to OBDDs (known as OBDD[\wedge]) [14] to further reduce the size of the resultant DAG to represent a given Boolean formula. In this work, we parameterize an OBDD[\wedge] to form a PROB that is used for weighted sampling.

Sampling: A Boolean variable x can be assigned either *true* or *false*, and its literal refers to either x or its negation. A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, with each clause being a disjunction of literals. A Boolean formula F is satisfiable if there exists an assignment τ of its variables such that the F evaluates to *true*. The model count of Boolean formula F refers to the number of distinct satisfying assignments of F .

Weighted sampling concerns with sampling elements from a distribution according to non-negative weights provided by a user-defined weight function W . In the context of this work, weighted sampling refers to the process of sampling from the space of satisfying assignments of a Boolean formula F . The weight function W assigns a non-negative weight to each literal l of F . The weight of an assignment τ is defined as the product of the weight of its literals.

WAPS: KUS [5] utilizes knowledge compilation techniques, specifically Deterministic Decomposable Negation Normal Form (d-DNNF) [19], to perform uniform sampling in 2 passes of the d-DNNF. *Annotation* is performed in the first pass, followed by sampling. WAPS [6] improves upon KUS by enabling weighted sampling via parameterization of the d-DNNF. WAPS performs sampling in a similar manner to KUS, the main difference being that the *annotation* step in WAPS takes into account the provided weight function. In contrast, we introduce INC which performs weighted sampling in a single pass by leveraging the DAG structure of PROB.

Knowledge compilation-based samplers typically perform incremental sampling as follows. The sampling space is first expressed as satisfying assignments of a Boolean formula, which is then compiled into the respective knowledge compilation form. In the following step, samples are drawn according to the given weight function W . Subsequently, the weights

are updated depending on application logic and weighted sampling is performed again. The process is repeated until an application-specific stopping criterion is met. An example of such an application would be the Baital framework [10], developed to use incremental weighted sampling to generate test cases for configurable systems.

III. PROB: - PROBABILISTIC OBDD[\wedge]

PROB is a DAG composed of four types of nodes - *conjunction*, *decision*, *true* and *false* nodes. The internal nodes of a PROB consist of conjunction and decision nodes whereas the leaf nodes of the PROB consist of true and false nodes. A PROB is recursively made up of sub-PROBs that represent sub-formulas of Boolean formula F . We use $\text{VarSet}(n)$ to refer to the set of variables of F represented by a PROB with n as the root node. $\text{Subdiagram}(n)$ refers to the sub-PROB starting at node n and $\text{Parent}(n)$ refers to the immediate parent of node n in PROB.

A. PROB Structure

Conjunction node (\wedge -node): A \wedge -node n_c represents conjunctions in the assignment space. There are no limits to the number of child nodes that n_c can have. However, the set of variables ($\text{VarSet}(\cdot)$) of each child node of n_c must be disjoint. An example of a \wedge -node would be $n2$ in Figure 1. Notice that $\text{VarSet}(n4) = \{z\}$ and $\text{VarSet}(n5) = \{y\}$ are disjoint.

Decision node: A decision node n_d represents decisions on the associated Boolean variable $\text{Var}(n_d)$ in Boolean formula F that the PROB represents. A decision node can have exactly two children - *lo-child* ($\text{Lo}(n_d)$) and *hi-child* ($\text{Hi}(n_d)$). $\text{Lo}(n_d)$ represents the assignment space when $\text{Var}(n_d)$ is set to *false* and $\text{Hi}(n_d)$ represents otherwise. $\theta_{n_d n_i}$ and $\theta_{n_d n_o}$ refer to the parameters associated with the edge connecting decision node n_d with $\text{Hi}(n_d)$ and $\text{Lo}(n_d)$ respectively in a PROB. Node $n1$ in Figure 1 is a decision node with $\text{Var}(n1) = x$, $\text{Hi}(n1) = n3$ and $\text{Lo}(n1) = n2$.

True and False nodes: True (\top) and false (\perp) nodes are leaf nodes in a PROB. Let τ be an assignment of all variables of Boolean formula F and let PROB ψ represent F . τ corresponds to a traversal of ψ from the root node to leaf nodes. The traversal follows τ at every decision node and visits all child nodes of every conjunction node encountered along the way. τ is a satisfying assignment if all parts of the traversal eventually lead to the true node. τ is not a satisfying assignment if any part of the traversal leads to the false node. With reference to Figure 1, let $\tau_1 = \{x, y, \neg z\}$ and $\tau_2 = \{x, y, z\}$. For τ_1 , the traversal would visit $n1, n3, n6, n7, n9$, and τ_1 is a satisfying assignment since the traversal always leads to \top node ($n9$). As a counter-example, τ_2 is not a satisfying assignment with its corresponding traversal visiting $n1, n3, n6, n7, n8, n9$. τ_2 traversal visits \perp node ($n8$) because variable $z \mapsto \text{true}$ in τ_2 and $\text{Hi}(n6)$ is node $n8$.

B. PROB Parameters

In the PROB structure, each decision node n_d has two parameters $\theta_{\text{Lo}(n_d)}$ and $\theta_{\text{Hi}(n_d)}$, associated with the two branches

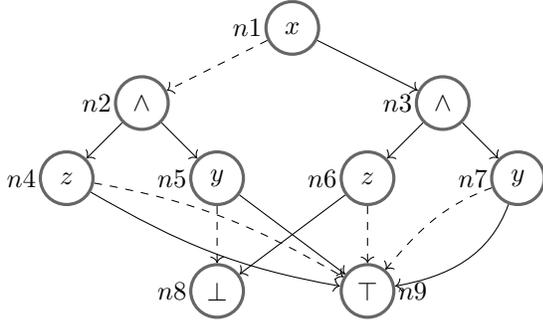


Fig. 1: A smooth PROB ψ_1 with 9 nodes, n_1, \dots, n_9 , representing $F = (x \vee y) \wedge (\neg x \vee \neg z)$. Branch parameters are omitted

of n_d , which sums up to 1. $\theta_{\text{Lo}(n_d)}$ is the normalized weight of the literal $\neg\text{Var}(n_d)$ and similarly, $\theta_{\text{Hi}(n_d)}$ is that of the literal $\text{Var}(n_d)$. One can view $\theta_{\text{Lo}(n_d)}$ to be the probability of picking $\neg\text{Var}(n_d)$ and $\theta_{\text{Hi}(n_d)}$ to be that of picking $\text{Var}(n_d)$ by the *determinism* property introduced later. Let x_i be $\text{Var}(n_d)$. Given a weight function W :

$$\theta_{\text{Lo}(n_d)} = \frac{W(\neg x_i)}{W(\neg x_i) + W(x_i)} \quad \theta_{\text{Hi}(n_d)} = \frac{W(x_i)}{W(\neg x_i) + W(x_i)}$$

C. PROB Properties

The PROB structure has important properties such as *determinism* and *decomposability*. In addition to the *determinism* and *decomposability* properties, we ensure that PROBs used in this work have the *smoothness* property through a smoothing process (Algorithm 1).

Property 1 (Determinism). For every decision node n_d , the set of satisfying assignments represented by $\text{Hi}(n_d)$ and $\text{Lo}(n_d)$ are logically disjoint.

Property 2 (Decomposability). For every conjunction node n_c , $\text{VarSet}(c_i) \cap \text{VarSet}(c_j) = \emptyset$ for all c_i and c_j where $c_i, c_j \in \text{Child}(n_c)$ and $c_i \neq c_j$.

Property 3 (Smoothness). For every decision node n_d , $\text{VarSet}(\text{Hi}(n_d)) = \text{VarSet}(\text{Lo}(n_d))$.

D. Joint Probability Calculation with PROB

In Section III-B, we mention that one can view the branch parameters as the probability of choosing between the positive and negative literal of a decision node. Notice that because of the *decomposability* and *determinism* properties of PROB, it is straightforward to calculate the joint probabilities at every given node. At each conjunction node n_c , since the variable sets of the child nodes of n_c are disjoint by *decomposability*, the joint probability of n_c is simply the product of joint probabilities of each child node. At each decision node n_d , there are only two possible outcomes on $\text{Var}(n_d)$ - positive literal $\text{Var}(n_d)$ or negative literal $\neg\text{Var}(n_d)$. By *determinism* property, the joint probability is the sum of the two possible

scenarios. Formally, the calculations for joint probabilities P' at each node in PROB are as follows:

$$P' \text{ of } \wedge\text{-node } n_c = \prod_{c \in \text{Child}(n_c)} P'(c) \quad (\text{EQ1})$$

$$P' \text{ of decision-node } n_d = \theta_{\text{Lo}(n_d)} \times P'(\text{Lo}(n_d)) + \theta_{\text{Hi}(n_d)} \times P'(\text{Hi}(n_d)) \quad (\text{EQ2})$$

For true node n , $P'(n) = 1$ because it represents satisfying assignments when reached. In contrast $P'(n) = 0$ when n is a *false* node as it represents non-satisfying assignments. In Proposition 2, we show that weighted sampling is equivalent to sampling according to joint probabilities of satisfying assignments of a PROB.

IV. INC - SAMPLING FROM PROB

In this section, we introduce INC - a bottom-up algorithm for weighted sampling on PROB. We first describe INC for drawing one sample and subsequently describe how to extend INC to draw k samples at once. We also provide proof of correctness that INC is indeed performing weighted sampling. As a side note, samples are drawn with replacement, in line with the existing state-of-the-art weighted sampler [6].

A. Preprocessing PROB

In the main sampling algorithm (Algorithm 2) to be introduced later in this section, the input is a smooth PROB. As a preprocessing step, we introduce Smooth algorithm that takes in a PROB ψ and performs smoothing.

The Smooth algorithm processes the nodes in the input PROB ψ in a bottom-up manner while keeping track of $\text{VarSet}(n)$ for every node n in ψ using a map κ . *True* and *false* nodes have \emptyset as they are leaf nodes and do not represent any variables. At each conjunction node, its variable set is the union of variable sets of its child nodes.

The smoothing happens at decision node n in ψ when $\text{VarSet}(\text{Lo}(n))$ and $\text{VarSet}(\text{Hi}(n))$ do not contain the same set of variables as shown by lines 8 and 16 of Algorithm 1. In the smoothing process, a new conjunction node (*lcNode* for $\text{Lo}(n)$ and *rcNode* for $\text{Hi}(n)$) is created to replace the corresponding child of n , with the original child node now set as a child of the conjunction node. Additionally, for each of the missing variables v , a decision node representing v is created and added as a child of the respective conjunction node. The decision nodes created during smoothing have both their lo-child and hi-child set to the *true* node. To reduce memory footprint, we check if there exists the same decision node before creating it in the `checkMakeTrueDecisionNode` function.

As an example, we refer to ψ_2 in Figure 2. It is obvious that ψ_2 is not smooth, because $\text{VarSet}(\text{Lo}(n_1)) = \{y\}$ and $\text{VarSet}(\text{Hi}(n_1)) = \{z\}$. In the smoothing process, we replace $\text{Lo}(n_1)$ with a new conjunction node n_2 and add a decision node n_4 representing missing variable z , with both child set to *true* node n_9 . We repeat the steps for $\text{Hi}(n_1)$ to arrive at PROB ψ_1 in Figure 1.

Algorithm 1 Smooth - returns a smoothed PROB

Input: PROB ψ
Output: smooth PROB

```

1:  $\kappa \leftarrow \text{initMap}()$ 
2: for node  $n$  of  $\psi$  in bottom-up order do
3:   if  $n$  is true node or false node then
4:      $\kappa[n] \leftarrow \emptyset$ 
5:   else if  $n$  is  $\wedge$ -node then
6:      $\kappa[n] \leftarrow \text{unionVarSet}(\text{Child}(n), \kappa)$ 
7:   else
8:     if  $\kappa[\text{Hi}(n)] - \kappa[\text{Lo}(n)] \neq \emptyset$  then
9:        $\text{lset} \leftarrow \kappa[\text{Hi}(n)] - \kappa[\text{Lo}(n)]$ 
10:       $\text{lcNode} \leftarrow \text{new } \wedge \text{-node}()$ 
11:       $\text{lcNode.addChild}(\text{Lo}(n))$ 
12:      for var  $v$  in  $\text{lset}$  do
13:         $\text{dNode} \leftarrow \text{checkMakeTrueDecisionNode}(v)$ 
14:         $\text{lcNode.addChild}(\text{dNode})$ 
15:       $\text{Lo}(n) \leftarrow \text{lcNode}$ 
16:     if  $\kappa[\text{Lo}(n)] - \kappa[\text{Hi}(n)] \neq \emptyset$  then
17:        $\text{rset} \leftarrow \kappa[\text{Lo}(n)] - \kappa[\text{Hi}(n)]$ 
18:        $\text{rcNode} \leftarrow \text{new } \wedge \text{-node}()$ 
19:        $\text{rcNode.addChild}(\text{Hi}(n))$ 
20:       for var  $v$  in  $\text{rset}$  do
21:          $\text{dNode} \leftarrow \text{checkMakeTrueDecisionNode}(v)$ 
22:          $\text{rcNode.addChild}(\text{dNode})$ 
23:        $\text{Hi}(n) \leftarrow \text{rcNode}$ 
24:      $\kappa[n] \leftarrow \text{Var}(n) \cup \text{unionVarSet}(\{\text{Hi}(n), \text{Lo}(n)\})$ 
25: return  $\psi$ 

```

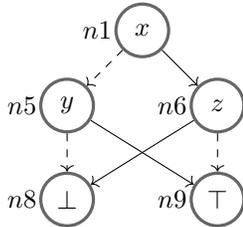


Fig. 2: A PROB ψ_2 representing Boolean formula $F = (x \vee y) \wedge (\neg x \vee \neg z)$, branch parameters are omitted

B. Sampling Algorithm

INC takes a PROB ψ representing Boolean formula F and draws a sample from the space of satisfying assignments of F , the process is illustrated by Algorithm 2. INC performs sampling in a bottom-up manner while integrating the *annotation* process in the same bottom-up pass. Since we want to sample from the space of satisfying assignments we can ignore *false* nodes in ψ entirely by considering a sub-DAG that excludes *false* nodes and edges leading to them, as shown by line 3. As an example, `hideFalseNode` when applied to ψ_1 would remove node $n8$ and the edges immediately leading to it. Next, INC processes each of the remaining nodes in bottom-up order while keeping two caches - ω to store the partial samples from each node, φ to store the joint probability at each node.

Algorithm 2 INC - returns a satisfying assignment based on PROB ψ parameters

Input: smooth PROB ψ
Output: a sampled satisfying assignment

```

1: cache  $\omega \leftarrow \text{initCache}()$ 
2: joint prob cache  $\varphi \leftarrow \text{initCache}()$ 
3:  $\psi' \leftarrow \text{hideFalseNode}(\psi)$ 
4: for node  $n$  of  $\psi'$  in bottom-up order do
5:   if  $n$  is true node then
6:      $\omega[n] \leftarrow \emptyset$ 
7:      $\varphi[n] \leftarrow 1$ 
8:   else if  $n$  is  $\wedge$ -node then
9:      $\omega[n] \leftarrow \text{unionChild}(\text{Child}(n), \omega)$ 
10:     $\varphi[n] \leftarrow \prod_{c \in \text{Child}(n)} \varphi[c]$ 
11:   else
12:      $p_{lo} \leftarrow \theta_{\text{Lo}(n)} \times \varphi[\text{Lo}(n)]$ 
13:      $p_{hi} \leftarrow \theta_{\text{Hi}(n)} \times \varphi[\text{Hi}(n)]$ 
14:      $p_{joint} \leftarrow p_{lo} + p_{hi}$ 
15:      $\varphi[n] \leftarrow p_{joint}$ 
16:      $r \leftarrow x \sim \text{binomial}(1, \frac{p_{hi}}{p_{joint}})$ 
17:     if  $r$  is 1 then
18:        $\omega[n] \leftarrow \omega[\text{Hi}(n)] \cup \text{Var}(n)$ 
19:     else
20:        $\omega[n] \leftarrow \omega[\text{Lo}(n)] \cup \neg \text{Var}(n)$ 
21: return  $\omega[\text{rootnode}(\psi)]$ 

```

INC starts with \emptyset at the *true* node since there is no associated variable.

At each conjunction node, INC takes the union of the child nodes in line 9. Using $n2$ in Figure 1 as an example, if sample drawn at $n4$ is $\omega[n4] = \{\neg z\}$ and at $n5$ is $\omega[n5] = \{y\}$, then $\text{unionChild}(\text{Child}(n2), \omega) = \{y, \neg z\}$. At each decision node n , a decision on $\text{Var}(n)$ is sampled from lines 16 to 20. We first calculate the joint probabilities, p_{lo} and p_{hi} of choosing $\neg \text{Var}(n)$ and choosing $\text{Var}(n)$. Subsequently, we sample decision on $\text{Var}(n)$ using a binomial distribution in line 16 with the probability of success being the joint probability of choosing $\text{Var}(n)$. After processing all nodes, the sampled assignment is the output at root node of ψ .

Extending INC to k samples: It is straightforward to extend the single sample INC shown in Algorithm 2 to draw k samples in a single pass, where k is a user-specified number. At each node, we have to store a list of k independent copies of partial assignments drawn in ω . At each conjunction node n_c , we perform the same union process in line 9 of Algorithm 2 for child outputs in the same indices of the respective lists in ω . More specifically, if n_c has child nodes c_x and c_y , the outputs of index i are combined to get the output of n_c at index i . This process is performed for all indices from 1 to k . At each decision node n_d , we now draw k independent samples instead of a single sample from the binomial distribution as shown in line 16. The sampling step in lines 16 to 20 are performed independently for the k random numbers. There is no change necessary for the calculation of joint probabilities

in Algorithm 2 as there is no change in literal weights.

Incremental sampling: Given a Boolean formula F and weight function W , INC performs incremental sampling with the sampling process shown in Figure 3. In the initial round, INC compiles F and W into a PROB ψ and performs sampling. Subsequent rounds involve applying a new set of weights W to ψ , typically generated based on existing samples by the controller [10], and performing weighted sampling according to the updated weights. The number of sampling rounds is determined by the controller component, whose logic varies according to application.

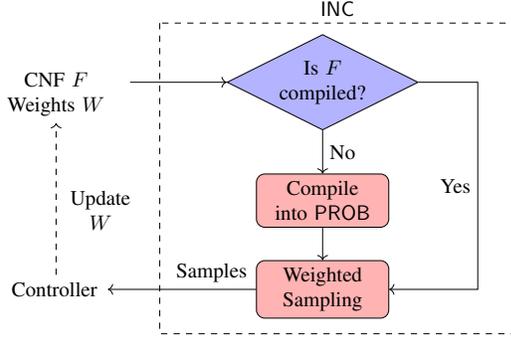


Fig. 3: INC's incremental sampling flow

C. Implementation Decisions

Log-Space Calculations: INC performs *annotation* process - computation of joint probabilities in log space. This design choice is made to avoid the usage of arbitrary precision math libraries, which WAPS utilized to prevent numerical underflow after many successive multiplications of probability values. Using the LogSumExp trick below, it is possible to avoid numerical underflow.

$$\begin{aligned} \log(a + b) &= \log(a) + \log\left(1 + \frac{b}{a}\right) \\ &= \log(a) + \log(1 + \exp(\log(b) - \log(a))) \end{aligned}$$

The joint probability at a decision node n_d is given by $\theta_{\text{Lo}(n_d)} \times$ joint probability of $\text{Lo}(n_d)$ + $\theta_{\text{Hi}(n_d)} \times$ joint probability of $\text{Hi}(n_d)$. Notice that if we were to perform the calculation in log space, we would have to add the two weighted log joint probabilities, termed p_{lo} and p_{hi} in Algorithm 2. Using the LogSumExp trick, we do not need to exponentiate p_{lo} and p_{hi} independently which risks running into numerical underflow. Instead, we only need to exponentiate the difference of p_{lo} and p_{hi} which is more numerically stable. Equations EQ1 and EQ2 can be implemented in log space as follows:

$$\begin{aligned} Q \text{ of } \wedge\text{-node } n_c &= \sum_{c \in \text{Child}(n_c)} Q(c) \\ Q \text{ of decision-node } n_d &= \text{LogSumExp} [\\ &\quad \log(\theta_{\text{Lo}(n_d)}) + Q(\text{Lo}(n_d)), \\ &\quad \log(\theta_{\text{Hi}(n_d)}) + Q(\text{Hi}(n_d))] \end{aligned}$$

In the equations above, Q refers to the corresponding log joint probabilities in EQ1 and EQ2. In the experiments section, we detail the runtime advantages of using log computations compared to arbitrary precision math computations.

Dynamic Annotation: In existing state-of-the-art weighted sampler WAPS, sampling is performed in two passes - the first pass performs *annotation* and the second pass samples assignments according to the joint probabilities. In INC, we combine the two passes into a single bottom-up pass performing *annotation* dynamically while sampling at each node.

D. Theoretical Analysis

Proposition 1. Branch parameters of any decision node n_d are correct sampling probabilities, i.e. $W(x_i) : W(\neg x_i) = \theta_{\text{Hi}(x_i)} : \theta_{\text{Lo}(x_i)}$ where $\text{Var}(n_d) = x_i$.

Proof.

$$\frac{W(x_i)}{W(\neg x_i)} = \frac{\frac{W(x_i)}{W(x_i) + W(\neg x_i)}}{\frac{W(\neg x_i)}{W(x_i) + W(\neg x_i)}} = \frac{\theta_{\text{Hi}(x_i)}}{\theta_{\text{Lo}(x_i)}}$$

We start with the ratio of literal weights of x , multiply both numerator and denominator by $W(x_i) + W(\neg x_i)$ and arrive at the ratio of branch parameters of n_d . Notice that only the ratio matters for sampling correctness and not the absolute value of weights. \square

Remark 1. Let n_d be an arbitrary decision node in PROB ψ . When performing sampling according to a weight function W , $\theta_{\text{Lo}(n_d)}$ is the probability of picking $\neg \text{Var}(n_d)$ and $\theta_{\text{Hi}(n_d)}$ is that of $\text{Var}(n_d)$. The determinism property states that the choice of either literal is disjoint at each decision node.

Proposition 2. INC samples an assignment τ from PROB ψ with probability $\frac{1}{N} \prod_{l \in \tau} W(l)$, where N is a normalization factor.

Proof. The proof consists of two parts, one for \wedge -node and another for decision node.

\wedge -node: Let n_c be an arbitrary conjunction node in PROB ψ . Recall that by decomposability property, $\forall c_i, c_j \in \text{Child}(n_c)$ and $c_i \neq c_j$, $\text{VarSet}(c_i) \cap \text{VarSet}(c_j) = \emptyset$. As such an arbitrary variable $x_i \in \text{VarSet}(n_c)$ only belongs to the variable set of one child node $c_i \in \text{Child}(n_c)$. Therefore, assignment of x_i can be sampled independent of x_j where $x_j \in \text{VarSet}(c_j), \forall c_j \neq c_i$. Let τ'_{c_i} be partial assignment for child node $c_i \in \text{Child}(n_c)$. Notice that each partial assignment τ'_{c_i} is sampled independently of others as there are no overlapping variables, hence their joint probability is simply the product of their individual probabilities. This agrees with the weight of an assignment being the product of its components, up to a normalization factor.

Decision node: Let n_d be an arbitrary decision node in PROB ψ and x_d be $\text{Var}(n_d)$. At n_d , we sample an assignment of x_d based on the parameters $\theta_{\text{Lo}(x_d)}$ and $\theta_{\text{Hi}(x_d)}$, which are probabilities of literal assignment by Proposition 1. By Proposition 1, one can see that the assignment of x_d is sampled

correctly according to W . As the sampling process at n_d is independent of its child nodes by the determinism property, the joint probability of sampled assignment of x_d and the output partial assignment from the corresponding child node would be the product of their probabilities. Notice that the joint probability aligns with the definition of weight of an assignment being the product of the weight of its literals, up to a normalization factor.

Since we do not consider the *false* node and treat it as having 0 probability, we always sample from satisfying assignments by starting at the *true* node in bottom-up ordering. Reconciling the sampling process at the two types of nodes, it is obvious that any combination of decision and \wedge -nodes encountered in the sampling process would agree with a given weight function W up to a normalization factor $1/N$. In fact, $N = \sum_{\tau_i \in S} W(\tau_i)$ where S is the set of satisfying assignments of Boolean formula F that ψ represents. As mentioned in Proposition 1 proof, normalization factors do not affect the correctness of sampling according to W , and we have shown that INC performs weighted sampling correctly under multiplicative weight functions. \square

Remark 2. *From the proof of Proposition 2, the determinism and decomposability property is important to ensure the correctness of INC. The smoothness property is important to ensure that the sampled assignment by INC is complete. For formula $F = (x \vee y) \wedge (\neg x \vee \neg z)$, an assignment τ_1 sampled from a non-smooth PROB could be $\{x, \neg z\}$. Notice that τ_1 is missing assignment for variable y . By performing smoothing, we will be able to sample a complete assignment of all variables in the Boolean formula as both child nodes of each decision node n have the same $\text{VarSet}(\cdot)$.*

V. EXPERIMENTS

We implement INC in Python 3.7.10, using NumPy 1.15 and Toposort package. In our experiments, we make use of an off-the-shelf KC diagram compiler, KCBox [23]. In the later parts of this section, we performed additional comparisons against an implementation of INC using the Gmpy2 arbitrary precision math package (INC_{AP}) to determine the impact of log-space *annotation* computations.

Our benchmark suite consists of instances arising from a wide range of real-world applications such as DQMR networks, bit-blasted versions of SMT-LIB (SMT) benchmarks, ISCAS89 circuits, and configurable systems [6], [10]. For incremental updates, we rely on the weight generation mechanism proposed in the context of prior applications of incremental sampling [10]. In particular, new weights are generated based on the samples from the previous rounds, resulting in the need to recompute joint probabilities in each round. Keeping in line with prior work, we perform 10 rounds (R1-R10) of incremental weighted sampling and 100 samples drawn in each round. The experiments were conducted with a timeout of 3600 seconds on clusters with Intel Xeon Platinum 8272CL processors.

In this section, we detail the extensive experiments conducted to understand INC’s runtime behavior and to compare

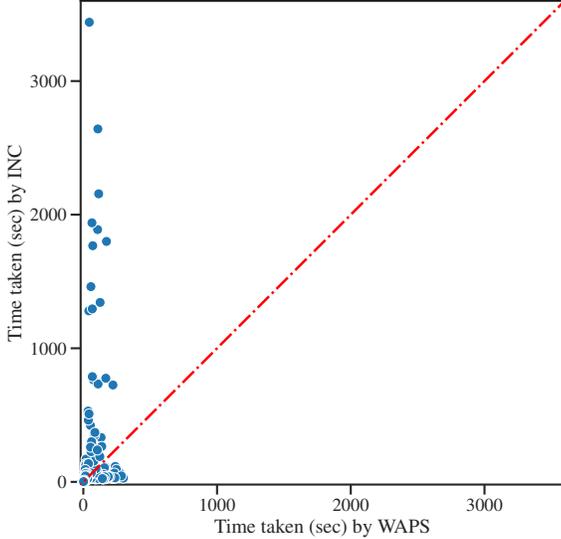
it with the existing state-of-the-art weighted sampler WAPS [6] in incremental weighted sampling tasks. We chose WAPS as it has been shown to achieve significant runtime improvement over other samplers, and accordingly has emerged as a sampler of the choice for practical applications [10]. In particular, our empirical evaluation sought to answer the following questions:

- RQ 1 How does INC’s incremental weighted sampling runtime performance compare to current state-of-the-art?
- RQ 2 How does using PROB affect runtime performance?
- RQ 3 How does log-space calculations impact runtime performance?
- RQ 4 Does INC correctly perform weighted sampling?

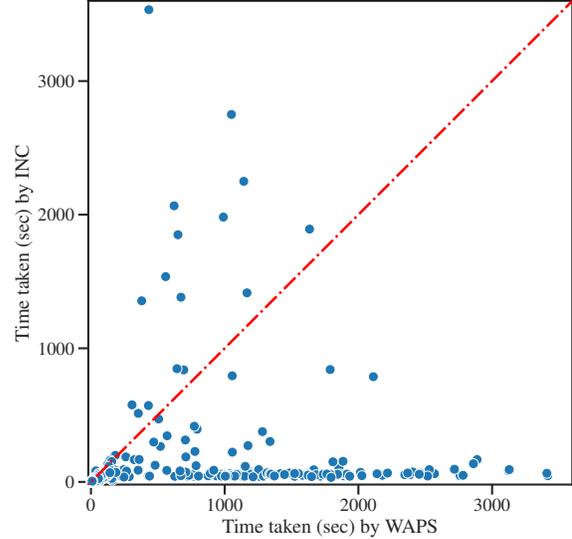
RQ 1: Incremental Sampling Performance: The scatter plot of incremental sampling runtime comparison is shown in Figure 4, with Figure 4a showing runtime comparison for the first round (R1) and Figure 4b showing runtime comparison over 10 rounds. The vertical axes represent the runtime of INC and the horizontal axes represent that of WAPS. In the experiments, INC completed 650 out of 896 benchmarks whereas WAPS completed 674. INC completed 21 benchmarks that WAPS timed out and similarly, WAPS completed 45 benchmarks that INC timed out. In the experiments, INC achieved a median speedup of $1.69\times$ over WAPS.

Further results are shown in Table I. Observe that for runtime taken for R1 (column 3), WAPS is faster and takes around $0.44\times$ of INC’s runtime in the median case. However, INC takes the lead in runtime performance when we examine the total time taken for the incremental rounds R2 to R10 (column 4). For incremental rounds, WAPS always took longer than INC, in the median case WAPS took $4.48\times$ longer than INC. We compare the average incremental round runtime with the first round runtime for both samplers in columns 1 and 2. In the median case, an incremental round for WAPS takes 67% of the time for R1 whereas an incremental round for INC only requires 5.9% of the time R1 takes. We show the per round runtime for 5 benchmarks in Table II to further illustrate INC’s runtime advantage over WAPS for incremental sampling rounds, even though both tools reuse the respective KC diagram compiled in R1. This set of results highlights INC’s superior performance over WAPS in the handling of incremental sampling settings. INC’s advantage in incremental sampling rounds led to better overall runtime performance than WAPS in 75% of evaluations. The runtime advantage of INC would be more obvious in applications requiring more than 10 rounds of samples.

Therefore, we conducted sampling experiments for 20 rounds to substantiate our claims that INC will have a larger runtime lead over WAPS with more rounds. Both samplers are given the same 3600s timeout as before and are to draw 100 samples per round, for 20 rounds. The number of completed benchmarks is shown in Table III. In the 20 sampling round setting, INC completed 649 out of 896 benchmarks, timing out on 1 additional benchmark compared to 10 sampling round setting. In comparison, WAPS completed 596 of 896 benchmarks, timing out on 78 additional benchmarks than in



(a) Single Round (R1) Runtime Scatter Plot



(b) Incremental Runtime Scatter Plot

Fig. 4: Runtime comparisons between INC and state-of-the-art weighted sampler WAPS

Statistic	WAPS MEAN(R2 to R10)	INC MEAN(R2 to R10)	WAPS R1	WAPS SUM(R2 to R10)	WAPS Total
	WAPS R1	INC R1	INC R1	INC SUM(R2 to R10)	INC Total
Mean	0.74	0.064	1.03	15.66	6.12
Std	0.24	0.040	1.47	26.42	10.73
Median	0.67	0.059	0.44	4.48	1.69
Max	1.25	0.188	10.65	172.66	73.96

TABLE I: Incremental weighted sampling runtime ratio statistics for WAPS and INC (Numerators and denominators refer to the corresponding runtimes)

the 10 sampling round setting. In addition, WAPS takes on median $2.17\times$ longer than INC under the 20 sampling round setting, an increase over the $1.69\times$ under the 10 sampling round setting.

The runtime results clearly highlight the advantage of INC for incremental weighted sampling applications and that INC is noticeably better at incremental sampling than the current state-of-the-art.

RQ 2: PROB Performance Impacts: We now focus on the analysis of the impact of using PROB compared to d-DNNF in the design of a weighted sampler. We analyzed the size of both PROB and d-DNNF across the benchmarks that both tools managed to compile and show the results in Table IV. From Table IV, PROB is always smaller than the corresponding d-DNNF. Additionally, PROB is at median $4.64\times$ smaller than the corresponding d-DNNF, and that for PROB is an order of magnitude smaller for at least 25% of the benchmarks. As such, PROB emerges as the clear choice of knowledge compilation diagram used in INC, owing to its succinctness which leads to fast incremental sampling runtimes.

RQ 3: Log-space Computation Performance Impacts:

In the design of INC, we utilized log-space computations to perform *annotation* computations as opposed to naively using arbitrary precision math libraries. In order to analyze the impact of this design choice, we implemented a version of INC where the dynamic *annotation* computations are performed using arbitrary precision math in a similar manner as WAPS. We refer to the arbitrary precision math version of INC as INC_{AP} . As an ablation study, we compare the runtime of both implementations across all the benchmarks and show the comparison in Table V. The statistics shown is for the ratio of INC_{AP} runtime to INC runtime, a value of 1.12 means that INC_{AP} takes $1.12\times$ that of INC for the corresponding statistics.

The results in Table V highlight the runtime advantages of our decision to use log-space computations over arbitrary precision computations. INC has faster runtime than INC_{AP} in majority of the benchmarks. INC displayed a minimum of $0.70\times$, a median of $1.12\times$, and a max of $1.89\times$ speedup over INC_{AP} . Furthermore, INC_{AP} timed out on 2 more benchmarks compared to INC. It is worth emphasizing that log-space

Benchmark	Tool	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Total	Speed
or-50-5-5-UC-10 (100, 253)	WAPS	56.6	56.3	52.5	59.4	52.5	53.6	59.4	53.2	53.4	61.7	558.6	1.0×
	INC	1461.3	7.6	8.4	8.4	8.4	8.4	8.5	8.5	8.4	8.5	1536.3	0.4×
or-100-20-9-UC-30 (200, 528)	WAPS	73.0	69.1	66.7	76.0	66.5	66.9	76.6	66.0	66.9	78.6	706.1	1.0×
	INC	269.5	4.7	4.8	4.8	4.9	5.1	4.8	4.8	4.8	5.1	313.4	2.3×
s953a_15_7 (602, 1657)	WAPS	1.7	1.1	1.1	1.2	1.0	1.1	1.2	1.1	1.1	1.3	11.9	1.0×
	INC	4.9	0.7	11.5	1.0×								
h8max (1202, 3072)	WAPS	90.3	104.2	92.4	116.0	94.3	94.1	112.9	92.9	94.4	120.4	1011.9	1.0×
	INC	34.1	2.1	2.2	2.4	2.3	2.4	2.2	2.4	2.4	2.3	55.7	18.2×
innovator (1256, 50452)	WAPS	195.5	221.9	201.3	244.4	200.1	206.7	247.2	202.0	202.9	257.4	2179.3	1.0×
	INC	32.8	1.6	1.8	1.9	1.9	1.9	1.8	1.9	1.9	1.9	49.4	44.1×

TABLE II: Runtime (seconds) breakdowns for each of ten rounds (R1-R10) between WAPS and INC for benchmarks of different sizes e.g. ‘h8max’ benchmark consists of 1202 variables and 3072 clauses.

Number of rounds	WAPS	INC
10	674	650
20	596	649

TABLE III: Number of completed benchmarks within 3600s, for 10 and 20 round settings

Statistic	$\frac{\text{WAPS KC size}}{\text{INC KC size}}$
Mean	18.92
Std	81.19
Median	4.64
Max	1734.08

TABLE IV: Statistics for number of nodes in d-DNNF (WAPS KC diagram) over that of smoothed PROB (INC KC diagram).

computations do not introduce any error, and our usage of them sought to improve on the naive usage of arbitrary precision math libraries.

RQ 4: INC Sampling Quality: We conducted additional evaluation to further substantiate evidence of INC’s sampling correctness, apart from theoretical analysis in Section IV-D. Specifically, we compared the samples from INC and WAPS, which has proven theoretical guarantees [6], on the ‘case110’ benchmark that is extensively used by prior works [4]–[6]. We gave each positive literal weight of 0.75 and each negative literal 0.25, and subsequently drew one million samples using both INC and WAPS and compare them in Figure 5.

Statistic	$\frac{\text{INC}_{AP} \text{ runtime}}{\text{INC runtime}}$
Mean	1.14
Std	0.16
Median	1.12
Max	1.89

TABLE V: Runtime comparison of INC and INC_{AP}

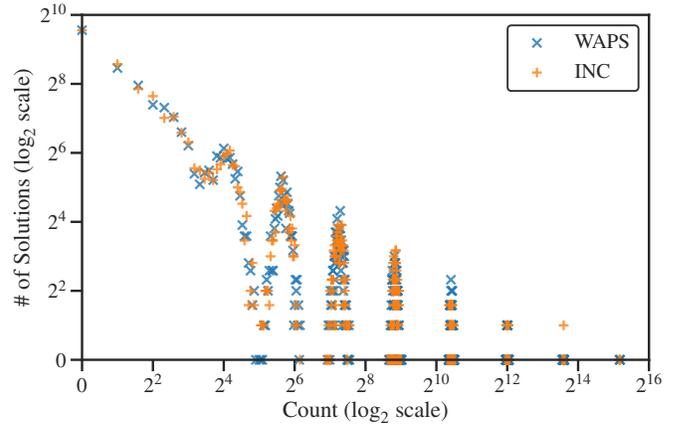


Fig. 5: Distribution comparison for Case110, with log scale for both axes

Figure 5 shows the distributions of samples drawn by INC and WAPS for ‘case110’ benchmark. A point (x, y) on the plot represents y number of unique solutions that were sampled x times in the sampling process by the respective samplers. The almost perfect match between the weighted samples drawn by INC and WAPS, coupled with our theoretical analysis in Section IV-D, substantiates our claim INC’s correctness in performing weighted sampling. Additionally, it also shows that INC can be a functional replacement for existing state-of-the-art sampler WAPS, given that both have theoretical guarantees.

Discussion: We demonstrated the runtime performance advantages of INC and the two main contributing factors - a choice of succinct knowledge compilation form and dynamic log-space *annotation*. INC takes longer than WAPS for single-round sampling, mainly because WAPS takes less time for KC diagram compilation than INC, leading to WAPS being faster in single-round sampling. In the incremental sampling setting, the compilation costs of KC diagrams are amortized, and since INC is substantially better at handling incremental updates, it thus took the overall runtime lead from WAPS

in the majority of the benchmarks. Extrapolating the trend, it is most likely that INC would have a larger runtime lead over WAPS for applications requiring more than 10 sampling rounds. The runtime breakdown demonstrates that INC is able to amortize the compilation time over the incremental sampling rounds, with subsequent rounds being much faster than WAPS. In summary, we show that INC is substantially better at incremental sampling than existing state-of-the-art.

VI. CONCLUSION AND FUTURE WORK

In conclusion, we introduced a bottom-up weighted sampler, INC, that is optimized for incremental weighted sampling. By exploiting the succinct structure of PROB and log-space computations, INC demonstrated superior runtime performance in a series of extensive benchmarks when compared to the current state-of-the-art weighted sampler WAPS. The improved runtime performance, coupled with correctness guarantees, makes a strong case for the wide adoption of INC in future applications.

For future work, a natural step would be to seek further runtime improvements for PROB compilation since INC takes longer than SOTA for the initial sampling round, due to slower compilation. Another extension would be to investigate the design of a partial *annotation* algorithm to reduce computations when only a small portion of the weights have been updated. It would also be of interest if we could store partial sampled assignments at each node as a succinct sketch to reduce memory footprint, for instance we could store each unique assignment and its count.

ACKNOWLEDGEMENT

We sincerely thank Yong Lai for the insightful discussions. Suwei Yang is supported by the Grab-NUS AI Lab, a joint collaboration between GrabTaxi Holdings Pte. Ltd., National University of Singapore, and the Industrial Postgraduate Program (Grant: S18-1198-IPP-II) funded by the Economic Development Board of Singapore. Kuldeep S. Meel is supported in part by National Research Foundation Singapore under its NRF Fellowship Programme (NRF-NRFFAI1-2019-0004), Ministry of Education Singapore Tier 2 grant (MOE-T2EP20121-0011), and Ministry of Education Singapore Tier 1 Grant (R-252-000-B59-114).

REFERENCES

[1] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, pp. 258–265, IEEE, 2007.

[2] M. Jerrum and A. Sinclair, "The markov chain monte carlo method: an approach to approximate counting and integration," 1996.

[3] T. Shi, J. Steinhardt, and P. Liang, "Learning where to sample in structured prediction," in *AISTATS*, 2015.

[4] D. Achlioptas, Z. Hammoudeh, and P. Theodoropoulos, "Fast sampling of perfectly uniform satisfying assignments," in *SAT*, 2018.

[5] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge compilation meets uniform sampling," in *LPAR*, 2018.

[6] R. Gupta, S. Sharma, S. Roy, and K. S. Meel, "Waps: Weighted and projected sampling," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2019.

[7] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI Mag.*, vol. 28, pp. 13–30, 2007.

[8] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[9] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, p. 307–392, 2019.

[10] E. Baranov, A. Legay, and K. S. Meel, "Baital: An adaptive weighted sampling approach for improved t-wise coverage," in *Proc. 28th European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[11] R. Peharz, S. Lang, A. Vergari, K. Stelzner, A. Molina, M. Trapp, G. V. den Broeck, K. Kersting, and Z. Ghahramani, "Einsum networks: Fast and scalable learning of tractable probabilistic circuits," in *ICML*, 2020.

[12] T. Baluta, Z. L. Chua, K. S. Meel, and P. Saxena, "Scalable quantitative verification for deep neural networks," in *Proceedings of International Conference on Software Engineering (ICSE)*, 5 2021.

[13] A. Darwiche and P. Marquis, "A knowledge compilation map," *J. Artif. Intell. Res.*, vol. 17, pp. 229–264, 2002.

[14] Y. Lai, D. Liu, and M. Yin, "New canonical representations by augmenting obdds with conjunctive decomposition," *J. Artif. Intell. Res.*, vol. 58, pp. 453–521, 2017.

[15] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, pp. 985–999, 1959.

[16] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, 1986.

[17] S. ichi Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," *30th ACM/IEEE Design Automation Conference*, pp. 272–277, 1993.

[18] A. Darwiche, "Decomposable negation normal form," *J. ACM*, vol. 48, pp. 608–647, 2001.

[19] A. Darwiche, "A compiler for deterministic, decomposable negation normal form," in *AAAI/IAAI*, 2002.

[20] R. Mateescu, R. Dechter, and R. Marinescu, "And/or multi-valued decision diagrams (aomdds) for graphical models," *J. Artif. Intell. Res.*, vol. 33, pp. 465–519, 2008.

[21] A. Darwiche, "Sdd: A new canonical representation of propositional knowledge bases," in *IJCAI*, 2011.

[22] G. Boole, "An investigation of the laws of thought: On which are founded the mathematical theories of logic and probabilities," 1854.

[23] Y. Lai, K. S. Meel, and R. Yap, "The power of literal equivalence in model counting," in *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 2 2021.

Bounded Model Checking for LLVM

Siddharth Priya ^{*}
siddharth.priya@uwaterloo.ca

Yusen Su^{*}
y256su@uwaterloo.ca

Yuyan Bao^{*}
yuyan.bao@uwaterloo.ca

Xiang Zhou^{*}
x245zhou@uwaterloo.ca

Yakir Vizel[†]
yvizel@cs.technion.ac.il

Arie Gurfinkel ^{*}
arie.gurfinkel@uwaterloo.ca

^{*}University of Waterloo
Waterloo, Canada

[†]The Technion
Haifa, Israel

Abstract—Bounded Model Checking (BMC) is an effective and precise static analysis technique that reduces program verification to satisfiability (SAT) solving. In this paper, we present the design and implementation of a new BMC engine (SEABMC) in the SEAHORN verification framework for LLVM. SEABMC precisely models arithmetic, pointer, and memory operations of LLVM. Our key design innovation is to structure verification condition generation around a series of transformations, starting with a custom IR (called SEA-IR) that explicitly purifies all memory operations by explicating dependencies between them. This transformation-based approach enables supporting many different styles of verification conditions. To support memory safety checking, we extend our base approach with fat pointers and shadow bits of memory to keep track of metadata, such as the size of a pointed-to object. To evaluate SEABMC, we have used it to verify `aws-c-common` library from AWS. We report on the effect of different encoding options with different SMT solvers, and also compare with CBMC, SMACK, KLEE and SYMBIOTIC. We show that SEABMC is capable of providing order of magnitude improvement compared with state-of-the-art.

I. INTRODUCTION

Bounded Model Checking (BMC) is an effective technique for precise software static analysis. It encodes a bounded (i.e., loop- and recursion-free) program P with assertions into a verification condition VC in (propositional) logic, such that VC is satisfiable iff P has an execution that violates an assertion. The satisfiability of VC is decided by a SAT-solver (or, more commonly, by an SMT-solver). BMC can be extremely precise, including path-sensitivity, bit-precision, and precise memory model. Its key weakness is scalability – precise reasoning requires careful selection of what details to include into the analysis.

A BMC engine can be implemented directly at the level of program source code, as best illustrated by CBMC [1] – the oldest and most mature BMC for C. This allows verifying absence of undefined behaviour and other source-level properties, and improves error reporting since it can be done at the source level. However, this complicates the implementation because modern programming languages are incredibly complex. Moreover, most industrial code relies on *de-facto*, rather than the standard language semantics [2] and on non-

standard features that are supported by mainstream compilers. An alternative is to implement BMC on an intermediate representation (IR) of a compiler. LLVM IR [3], called *bitcode*, is a common choice. This simplifies implementation to focus only on capturing semantics of the IR, allows sharing infrastructure with the compiler, simplifies integration of verification into current build systems, and simplifies supporting multiple source languages (e.g., SMACK [4] supports 8 languages [5]). This is the approach we take in this paper.

Over the years, there have been multiple BMC tools developed for LLVM, including SEAHORN (that we build on), SMACK, and LLBMC [6]. However, the issue still remains that existing tools are either not maintained, commercial (and not publicly available, e.g. LLBMC), or are not effective at bit- and memory-precise reasoning (SEAHORN and SMACK). Our goal is to address this deficiency, while re-examining and re-evaluating many of the design decisions. Thus, while BMC is a mature technique, we have two objectives. First, we want different strategies for generating verification conditions (VCGen) through program transformations. This allows us to examine which encoding works best in practice for production code, and why. Second, we want to provide mechanisms to express safety properties, e.g. memory safety, succinctly. In accomplishing these objectives, we believe that we have identified a new interesting point in the design space.

For our first objective, we propose a new pipeline. A source program is translated to a new IR, called SEA-IR, that extends LLVM IR, with explicit dependency between memory operations. This, effectively, purifies memory operations, i.e., there is no global memory, and no side-effects. A SEA-IR program then goes through a series of program transformations for VCGen. The program is progressively reduced to a pure data-flow form in which all instructions execute in parallel, and is only then, converted to SMT-LIB supported logic. This allows experimenting with different strategies of VCGen by controlling these transformations. For example, we can generate VCs using a control flow representation of the program like DAFNY [7] or a pure data flow representation like CBMC. VCs depend on memory representation. Thus, we explore two different forms of representing memory content: lambda-

based [8] that represents memory as nested ITE-expressions¹, and array-based that uses SMT theory of arrays [9]. In particular, lambda-based representation allows precise and efficient modelling of wide memory operations such as `memcpy`. We also explore the space of memory models between flat memory in which memory is a flat array, and an object memory where memory is represented by a set of arrays.

To improve checking for safety properties, our second objective, we attach additional information to pointers (so called *fat*) and to memory (so called *shadow*). This simplifies tracking of various program metadata for modelling safety properties. As an example, we can use fat pointers to check for out of bounds array access and shadow memory to check for immutability of read only memory. While existing tools report memory safety analysis, SEABMC can capture metadata of arbitrary size since we are not constrained by concrete pointer or memory width. Additionally, we model pointer provenance. This allows us to catch out-of-bounds accesses which might be missed by tools like LLBMC and ASAN [10].

We evaluate SEABMC on verification tasks of `aws-c-common` C library developed by Amazon Web Services (AWS). The library is a collection of common data-structures for C (including buffers, arrays, lists, etc.). We chose it for several reasons. First of all, it has been recently verified using CBMC. Thus, it includes many meaningful verification tasks. Second, it is a live industrial project, thus, it provides an example of how to integrate SEABMC into a real project, and shows that SEABMC supports all of the necessary language features. Third, it provides an opportunity to compare head-to-head against a mature tool (CBMC) on industrial code. We feel this is a more interesting comparison than, for example, comparing on isolated verification benchmarks of SVCOMP [11]. We show that SEABMC is an order of magnitude faster than CBMC, and outperforms three mature LLVM-based tools: SMACK, SYMBIOTIC [12] and KLEE [13]. Note that we focus on SEABMC design and performance. An extensive case study comparing different *kinds* of verification tools on `aws-c-common` is available in [14].

In summary, this paper makes the following contributions: an IR, SEA-IR, for LLVM bitcode that purifies memory operations; a VCGen that combines program transformations with encoding into logic allowing for many different styles of VCs; a memory model that combines fat-pointers with shadow-memory to represent metadata; an open-sourced BMC tool; and, a thorough evaluation against the state-of-the-art verification tools on production C code.

II. GENERATING VERIFICATION CONDITIONS

This section presents our main verification condition generation (VCGen) algorithm. We start with a new intermediate representation, that we call SEA-IR. This representation extends LLVM bitcode with purified memory operations. We then describe a series of transformations that transform a

program in SEA-IR to a pure data-flow (PD) form where no part of computation depends on control. Each transformation progressively simplifies the program for generating verification conditions. The PD form is one from which verification conditions can be generated in the most straightforward way. Finally, we show how PD programs can be converted to verification conditions in SMT-LIB. In this section, we assume that the input program contains only one function, no loops or global variables. In practice, this is achieved by inlining all functions, unrolling loops to a fixed depth, and eliminating global variables. The loop unroll bound is often detected automatically, but can also be set by the user.

SEA-IR SEABMC transforms LLVM bitcode to an intermediate representation, called SEA-IR, that extends LLVM bitcode by making dependency information between memory operations explicit. In LLVM IR, this information does not exist in the program. Fig. 1 shows the simplified syntax of SEA-IR. Here, we present a simplified version with many features removed, e.g., types, expressions, function calls, etc. However, we assume that the type of each register is known (but not shown). We use `R` to represent a scalar register, `P` for a pointer register and `M` for a memory register. A legal SEA-IR program is assumed to be in a Static Single Assignment (SSA) form with all registers are assigned before use, all expressions well-typed and a program always ending with a `halt`.

We use the term *object* to refer to an allocated sequence of bytes in memory. Interestingly, we do not use a single addressable memory that maps from addresses to values. Instead, a SEA-IR program uses a set of memory regions or *memories*, which collectively contains all objects in a program. Each memory, in-turn, contains a subset of objects used in the program. To maintain compatibility with de-facto semantics, addresses are assigned from a single address space and are, thus, globally unique. To aid program analysis, all memories are pure: storing in memory creates a new memory i.e., *definition*; loading from a memory is a *use*. This def-use scheme [15] is known as MemorySSA in LLVM. Partitioning memory into multiple memories relieves the SMT-solver from some of the alias analysis reasoning.

To explain SEA-IR, we use a simple C program in Fig. 2. The program initializes variable `x` with a non-deterministic 8-bit integer obtained by the return value of function `nd_char()`. The value of `x` is further constrained by the `assume`, such that `x > 0 && x < 10`. Then, the program non-deterministically allocates 1- or 2-byte memory region and assigns the address to the variable `p`. The first byte that `p` points to is assigned by the value of `x`. The second byte (if any) is assigned 0. For the moment, ignore that the second assignment might be undefined behaviour (we expand on this in Sec III). Finally, the two `asserts` describe the post-condition.

Fig. 3a shows the SEA-IR program transformed from the C program. In this presentation, we do not strictly follow the syntax of SEA-IR. For example, we allow immediate values to appear in place of registers, and write expressions in infix form. The program is a single function `main`, which consists of four basic blocks labeled by `BB0`, `BB1`, `BB2` and `BB3`. A basic

¹ITE stands for If-Then-Else.

```

PR ::= fun main() {BB+}
BB ::= L : PHI* S+ (BR | halt)
BR ::= br E, L, L | br L
PHI ::= R = phi [R, L](, [R, L])* |
        M = phi [M, L](, [M, L])* |
        P = phi [P, L](, [P, L])*
S ::= RDEF | MDEF | VS
RDEF ::= R = E | P, M = alloca R, M |
        P, M = malloc R, M | R = load P, M |
        P = load P, M | M = free P, M
MDEF ::= M = store R, P, M | M = store P, P, M
VS ::= assert R | assume R

```

Fig. 1: Simplified grammar of SEA-IR, where E , L , R , P and M are expressions, labels, scalar registers, pointer registers and memory registers, respectively.

block consists of a label, zero or more `PHI`-statements, one or more statements, an optional branch statement or a `halt`.

A SEA-IR program has three types of registers: scalar registers, pointer registers and memory registers. Scalar registers store values of basic datatype – integers. Pointers store pointer values. Memory registers store memory regions, and map from addresses to values. Each memory register maps to a unique *memory* and we use memory register and memory interchangeably. For example, in Fig. 3a, R_0 is a scalar register which stores an integer and P_1 is a register for storing a pointer. M_0 and M_1 are memory registers. Since each program is finite, the number of registers is finite as well.

An assignment statement defines the register by the value of a given expression. We assume that expressions include the usual set of operations, e.g., arithmetic, bitwise operations, cast operations and pointer arithmetic. For example, in BB_0 of Fig. 3a, $R_2 = R_0 < 10$ defines the value of register R_2 by the value of the expression $R_0 < 10$, where $<$ is an unsigned 8-bit less-than operator.

A `phi` selects a value from a list of values when a control flow merges. For example, $M_3 = \text{phi}[M_1, BB_1], [M_2, BB_2]$ in BB_3 of Fig. 3a assigns M_1 (M_2) to M_3 if the previously executed basic block was BB_1 (BB_2).

SEA-IR provides `alloca` and `malloc` instructions to allocate memory on the stack and the heap, respectively. A given number of bytes are allocated in memory on RHS of the statement, defining a new memory on the LHS. While the allocation does not change memory, it does define it. This is explained in Sec. III. Consider $P_1, M_1 = \text{malloc } 2, M_0$ in BB_1 of Fig. 3a. It allocates 2 bytes (on the heap) in memory M_0 , defines memory M_1 and a fresh pointer in P_1 .

A `store`, e.g., $M_5 = \text{store } 0, P_5, M_4$ in BB_3 , defines memory M_5 by writing the value 0 to the address pointed-to by the pointer register P_5 in memory M_4 . Note that the instruction is pure; i.e., all effects of the instructions are on the output registers only. The result of the modification is in M_5 , while M_4 is unchanged. Similarly, a `load` reads the value pointed-to by a pointer register in memory register M , and assigns the value to a new register. `assert` and `assume` are the usual verification statements for assertions and assumptions, respectively.

```

1 int main() {
2   uint8_t x = nd_char();
3   assume(x > 0 && x < 10);
4   uint8_t *p = nd_bool() ? malloc(2*sizeof(uint8_t))
5                       : malloc(sizeof(uint8_t));
6   *p = x;
7   *(p + 1) = 0;
8   assert(0 < *p && *p < 10);
9   assert(*(p + 1) == 0); // potential UB
10  return 0;
11 }

```

Fig. 2: An example C program.

Program Transformation Before generating verification conditions, a series of program transformations, as given below, are applied to a SEA-IR program.

Single Assert Form. A program is in a Single Assert (SA) form if it only contains one `assert`, which appears as the last instruction (before `halt`) in the last block of a program. Fig. 3b shows the code in a SA form transformed from the one in Fig. 3a, where an `ERR` label is added to the original code, and denotes an error state. In BB_3 , `assert R6` is transformed into `br R6, BB4, ERR`, meaning that if R_6 is false, then the program’s execution trace is diverted to `ERR`. Similarly, `assert 0 = 0` in BB_3 is transformed into `assume 0 != 0` and `br ERR`.

Single Assume Single Assert (SASA) Form. A program is in SASA form if it is in SA form, and contains a single `assume` immediately followed by a single `assert`. For example, the two definition of registers R_1 and R_2 in BB_0 of Fig. 3b are combined into one definition of R_1 in Fig. 3c, where the two boolean expressions are combined by a conjunction. A `phi`-statement, $A = \text{phi}[R_6, BB_4], [R_1, BB_3]$, is added to `ERR`, so that register A tracks the value of the conjunction. The `assume` ensures that A is true prior to the assertion.

Gated Single Static Assignment Form. A program in SASA form is further transformed into a Gated Single Static Assignment (GSSA) form, where `phi`-functions are replaced by `select` expressions². For example, $\text{phi}[M_1, BB_1], [M_2, BB_2]$ in `ERR` of Fig. 3c is transformed into `select R2, M1, M2` in Fig. 3d, where R_2 is the condition that the program trace is diverted to BB_1 or BB_2 .

Pure Dataflow Form. A (loop-free) program is in a Pure Dataflow (PD) form if it is in GSSA form and contains a single basic block. As shown in Fig. 4a, all the labels and `br` are removed from Fig. 3d, and the five basic blocks are merged into one single basic block.

Reduced Pure Dataflow Form. A program is in a reduced PD form if every definition appears on a def-use chain of either `assume` or `assert`. Each such definition is said to be in the cone of influence (COI). In Fig. 4a, the highlighted code is not in the cone of influence and is not considered.

A reduced PD program has no control dependencies. It is essentially a sequence of equations with two side-conditions determined by `assume` and `assert`. All definitions are used,

²In LLVM, `select` is the usual ternary ITE such as $a ? c : b$ in C.

```

fun main() {
BB0:
M0 = mem.init()
R0 = nd_char()
R1 = R0 > 0
assume R1
R2 = R0 < 10
assume R2
R3 = nd_bool()
br R3, BB1, BB2
BB1:
P1, M1 = malloc 2, M0
br BB3
BB2:
P2, M2 = malloc 1, M0
br BB3
BB3:
M3 = phi [M1, BB1], [M2, BB2]
P4 = phi [P1, BB1], [P2, BB2]
M4 = store R0, P4, M3
P5 = P4 + 1
M5 = store 0, P5, M4
R6 = R0 > 0 && R0 < 10
assert R6
assert 0 == 0
halt
}

```

(a) SEA-IR

```

fun main() {
BB0:
M0 = mem.init()
R0 = nd_char()
R1 = R0 > 0
assume R1
R2 = R0 < 10
assume R2
R3 = nd_bool()
br R3, BB1, BB2
BB1:
P1, M1 = malloc 2, M0
br BB3
BB2:
P2, M2 = malloc 1, M0
br BB3
BB3:
M3 = phi [M1, BB1], [M2, BB2]
P4 = phi [P1, BB1], [P2, BB2]
M4 = store R0, P4, M3
P5 = P4 + 1
M5 = store 0, P5, M4
R6 = R0 > 0 && R0 < 10
br R6, BB4, ERR
BB4:
assume 0 != 0
br ERR
ERR:
assert false
halt
}

```

(b) Single Assert (SA)

```

fun main() {
BB0:
M0 = mem.init()
R0 = nd_char()
R1 = R0 > 0 && R0 < 10
R2 = nd_bool()
br R2, BB1, BB2
BB1:
P1, M1 = malloc 2, M0
br BB3
BB2:
P2, M2 = malloc 1, M0
br BB3
BB3:
M3 = phi [M1, BB1], [M2, BB2]
P3 = phi [P1, BB1], [P2, BB2]
M4 = store R0, P3, M3
P4 = P3 + 1
M5 = store 0, P4, M4
R5 = R0 > 0 && R0 < 10
br R5, BB4, ERR
BB4:
R6 = false
br ERR
ERR:
A = phi [R6, BB4], [R1, BB3]
assume A
assert false
halt
}

```

(c) Single Assume (SASA)

```

fun main() {
BB0:
M0 = mem.init()
R0 = nd_char()
R1 = R0 > 0 && R0 < 10
R2 = nd_bool()
br R2, BB1, BB2
BB1:
P1, M1 = malloc 2, M0
br BB3
BB2:
P2, M2 = malloc 1, M0
br BB3
BB3:
M3 = select R2, M1, M2
P3 = select R2, P1, P2
M4 = store R0, P3, M3
P4 = P3 + 1
M5 = store 0, P4, M4
R5 = R0 > 0 && R0 < 10
br R5, BB4, ERR
BB4:
R6 = false
br ERR
ERR:
A = select R5, R6, R1
assume A
assert false
halt
}

```

(d) Gated SSA (GSSA)

Fig. 3: Program from Fig. 2 in: (a) SEA-IR, (b) SA, (c) SASA, and (d) GSSA forms.

```

fun main() {
entry:
M0 = mem.init()
R0 = nd_char()
R1 = R0 > 0 && R0 < 10
R2 = nd_bool()
P1, M1 = malloc 2, M0
P2, M2 = malloc 1, M0
M3 = select R2, M1, M2
P3 = select R2, P1, P2
M4 = store R0, P3, M3
P4 = P3 + 1
M5 = store 0, P4, M4
R5 = R0 > 0 && R0 < 10
R6 = false
A = select R5, R6, R1
assume A
assert 0
halt
}

```

(a) Pure-Dataflow (PD)

```

 $r_1 = (0 < r_0 \wedge r_0 < 10) \wedge$ 
 $p_1 = \text{addr}_0 \wedge m_1 = m_0 \wedge$ 
 $p_2 = \text{addr}_0 + 4 \wedge m_2 = m_0 \wedge$ 
 $p_3 = \text{ite}(r_2, p_1, p_2) \wedge$ 
 $p_4 = p_3 + 1 \wedge$ 
 $r_5 = (r_0 > 0 \wedge r_0 < 10) \wedge$ 
 $r_6 = \text{false} \wedge$ 
 $a = \text{ite}(r_5, r_6, r_1) \wedge$ 
 $a \wedge$ 
 $\neg \text{false}$ 

```

(b) SMT-LIB

Fig. 4: Program from Fig. 2 in PD and SMT-LIB forms. The highlighted lines are removed from the program.

directly, or indirectly, by either **assume** or **assert** (or both). Now, generating VC implies mapping each definition into a logic equation.

Verification Condition Generation We now describe the translation function sym that encodes a program into a VC. Throughout the section, we illustrate sym using the program in Fig. 4a and the corresponding VC in Fig. 4b.

The input to sym is a SEA-IR program in a reduced PD form, and the output is a SMT-LIB program. For simplicity of presentation, we assume that two fundamental sorts are used in the encoding: bit-vector of 64 bits, $bv(64)$, and a map

between bit-vectors, $bv(64) \rightarrow bv(64)$.³ In addition, we use the following helper sorts: $scalr : bv(64)$, $ptrs : scalr$, and $mems : bv(64) \rightarrow bv(64)$, where $scalr$ is sorts of scalars, $ptrs$ of pointers, and $mems$ of memories.

sym is defined recursively, bottom up, on the abstract syntax tree of SEA-IR. First, each register, R , is mapped to a symbolic constant $sym(R)$ of an appropriate sort. To simplify the presentation, we use a lower-case math font for constants corresponding to the register. For example, in Fig. 4a, $sym(R0)$ is r_0 of $scalr$ sort, $sym(P2)$ is p_2 of $ptrs$ sort, and $sym(M0)$ is m_0 of $mems$ sort, respectively.

Second, each expression E in SEA-IR is mapped into a corresponding SMT-LIB expression $sym(E)$. We omit the details of this step since they are fairly standard. For example, a **select** is translated into an *ite*, scalar addition, such as $R9 + 1$ is translated into bit-vector addition $bvadd$, etc. Pointer manipulating expressions, such as pointer arithmetic (**gep**) and pointer-to-integer cast (**ptoi**) are described in Sec. III.

Finally, sym translates each statement into an equality. For example, $R = E$ is translated into $r = e$, where e is $sym(E)$. For example, in Fig. 4a, $A = \text{select } R5, R6, R1$ is translated into $a = \text{ite}(r_5, r_6, r_1)$ in Fig. 4b.

Translating **alloca** and **malloc** requires a memory allocator. We parameterize sym by an allocation function $alloc : A \rightarrow ptrs$ that maps allocation expressions in A to values of pointer sort. For example, in Fig. 5, $P1, M1 = \text{alloca } R0, M0$ is translated into $p_1 = alloc(\text{alloca } R0 M0) \wedge m_1 = m_0$, and is reduced to $p_1 = \text{addr}_0 \wedge m_1 = m_0$, where addr_0 is the return value of $alloc$.

³In practice, SEABMC supports multiple bit-widths for scalars, and different ranges for values for maps.

$$\begin{aligned}
\text{sym}(R = E) &\triangleq r = e & \text{sym}(\text{assume } R) &\triangleq r & \text{sym}(\text{assert } R) &\triangleq \neg r \\
\text{sym}(M1 = \text{store } R1, P2, M0) &\triangleq m_1 = \text{write}(m_0, r_1, p_2) \\
\text{sym}(R1 = \text{load } P0, M) &\triangleq p_1 = \text{read}(m, p_0) \\
\text{sym}(P1, M1 = \text{alloca } R0, M0) &\triangleq p_1 = \text{alloc}(\text{alloca } R0, M0) \wedge m_1 = m_0 \\
\text{sym}(P1, M1 = \text{malloc } R0, M0) &\triangleq p_1 = \text{alloc}(\text{malloc } R0, M0) \wedge m_1 = m_0
\end{aligned}$$

Fig. 5: Definition of *sym*.

$$\begin{aligned}
\forall a \in A \cdot \text{size}(a) \text{ is known} & \quad \forall a \in A \cdot (\text{alloc}(a) \bmod \text{align}(a)) = 0 \\
\forall a_1 \neq a_2 \in A \cdot (\text{alloc}(a_1) + \text{size}(a_1) \leq \text{alloc}(a_2)) & \vee (\text{alloc}(a_2) + \\
& \text{size}(a_2) \leq \text{alloc}(a_1))
\end{aligned}$$

Fig. 6: Specifications for *size*, *align*, and *alloc*.

For *sym*, *alloc* must satisfy the basic specifications of a memory allocator. The spec is formalized in Fig. 6, where *size* and *align* return the size and alignment of each allocation expression in A . Intuitively, each allocated segment must have a statically known bound on size, all pointers returned by an allocation are aligned, and all allocations are mutually disjoint. For example, in Fig. 4a, the memory allocations in $P2$, $M1 = \text{malloc } 2$, $M0$ and $P1$, $M2 = \text{malloc } 1$, $M0$ are guaranteed to be disjoint since Fig. 4b adds a constraint that $p_1 = \text{addr}_0 \wedge p_2 = \text{addr}_0 + 4$. In practice, we also enforce that stack allocations (**alloca**) return high addresses, and heap allocations (**malloc**) return low addresses. Other constraints, such as separating kernel- and user-space addresses can be easily added.

The semantics for memory operations depends on the representation of memories (see Sec. III). We use two functions, *read* and *write*, to encapsulate the actual translation when defining the meaning of **load** and **store**, respectively. The function $\text{read}(m, p)$ represents the value of the memory register m at index p . The function $\text{write}(m, r_1, p_2)$ represents a new memory obtained by writing the value r_1 at index p_2 in m . In Fig. 5, **load** $P0, M$ and **store** $R1, P2, M0$ are translated into $\text{read}(m, p_0)$, and $\text{write}(m_0, r_1, p_2)$, respectively.

SEABMC has two memory representations: *Arrays* and *Lambdas*.

Arrays. Memories are modeled by an SMT-LIB theory of extensional arrays `ArraysEx`⁴. A memory register M is mapped to a symbolic constant m , where m is of sort *mems*. As shown in Fig. 7, a *write* is translated into an `ArrayEx store`, and a *read* is translated into an `ArrayEx select`.

Lambdas. Memories are modelled by λ -functions of the form $\lambda x.e$, where e is an expression with free occurrences of x . A memory register M is translated into an uninterpreted function m of sort *mems*. As shown in Fig. 7, $\text{read}(m, r_0)$ is translated into a function application $m(r_0)$, and $\text{write}(m_0, r_1, p_2)$ is translated into a new λ -function, $\lambda x.\text{ite}(x = p_2, r_1, m_0)$. In the final VC, function applications are β -reduced to substitute formal arguments with actual parameters. Thus, the VC only

⁴<http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml>.

	Array	λ
$\text{read}(m, p_0)$	select $m \ p_0$	$m(p_0)$
$\text{write}(m_0, r_1, p_2)$	store $m_0 \ r_1 \ p_2$	$\lambda x.\text{ite}(x = p_2, r_1, m_0(x))$

Fig. 7: Translation of *read* and *write*.

$$\text{RDEF} ::= R = \text{isderefer } R, R \mid R = \text{isalloc } R, M \mid R = \text{ismod } R, M$$

Fig. 8: SEA-IR syntax for memory safety.

has *ites*, and does not require `ArrayEx` support in the SMT-solver.

Overall, for a program P in a reduced PD form with a sequence of statements $S_0 \cdots S_k$, followed by **assume** $R0$ and **assert** $R1$, $\text{sym}(P)$ is defined as follows:

$$\text{sym}(P) \triangleq \left(\bigwedge_{0 \leq i \leq k} \text{sym}(S_i) \right) \wedge \text{sym}(R0) \wedge \text{sym}(R1).$$

For example, the VC for a program in Fig. 4a is shown in Fig. 4b. Definitions in Fig. 4a are translated into a conjunction of equalities, and **assert** 0 is translated into $\neg \text{false}$. The VC is *unsatisfiable* since A evaluates to *false*.

Theorem 1: $\text{sym}(P)$ is satisfiable iff P has an execution that satisfies the assumption and violates the assertion.

III. VERIFYING MEMORY SAFETY

In most languages, including C, memory safety is difficult to specify directly. To make such specifications possible, we use fat pointers [16] and shadow memory to keep metadata about pointers and memory, respectively. Moreover, we present a general extension of both memory and pointer semantics.

Intuitively, we want to represent each fat pointer as a tuple of values that collectively represent the value of the pointer and all the metadata (i.e., *fat*) that is cached at it. We do not put restrictions on the number of values nor their sorts. However, we assume that there is a function *addr* that maps a pointer to an expression representing an address. Thus, for a pointer register P , $\text{sym}(P)$ is a tuple $\langle t_1, \dots, t_j \rangle$ of j constants that represents the pointer, and $\text{addr}(\langle t_1, \dots, t_j \rangle)$ is an address of that pointer. For example, a common case is to use the first element of the tuple to represent the address: $\text{addr}(\langle t_1, \dots, t_j \rangle) = t_1$. Fig. 13 presents a small program (on the left) that writes a fat pointer $P0$ to memory at address $P1$. Memory is divided into five parts with *val* memory used to store the actual program data. Here, *val* stores the *base* value of the fat pointer and *offset* and *size* store the fat. Memory operations are tracked by *alloc* and *mod* memory that mark whether an address is allocated and whether it has been written to, respectively. Fig. 13 shows the memory state after the **store** operation. Both *alloc* and *mod* are set to 1 because $P1$ is allocated and has been modified.

Formally, we re-define *ptrs* to be a tuple of sorts, written as $\langle s_1, \dots, s_j \rangle$. We say that a tuple $\tau = \langle c_1, \dots, c_p \rangle$ of p constants is of a tuple sort $\langle s_1, \dots, s_p \rangle$ iff, for each $0 < i \leq p$, c_i is of sort s_i . Tuples of sorts, and tuples of constants are only present during VCGen, but not in the final verification

```

fun main() {
BB0:
M0 = mem.init()
R0 = nd_char()
R1 = R0 > 0 && R0 < 10   r1 = r0 > 0 ∧ r0 < 10 ∧
R2 = nd_bool()
P1, M1 = malloc 2, M0    p1.base = addr0 ∧ p1.offset = 0 ∧ p1.size = 4 ∧ m1 = m0 ∧
P2, M2 = malloc 1, M0    p2.base = addr0 + 4 ∧ p2.offset = 0 ∧ p2.size = 4 ∧ m2 = m0 ∧
M3 = select R2, M1, M2
P3 = select R2, P1, P2   p3 = ite(r2, p1, p2) ∧
R4 = isderef R3, 1       r4 = 0 ≤ (1 + p3.offset) < p3.size ∧
M4 = store R0, P3, M3
P5 = gep P3, 1           p5.base = r3.base ∧ p5.offset = r3.offset + 1 ∧ p5.size = r3.size ∧
R6 = isderef P5, 1       r6 = 0 ≤ (1 + p5.offset) < p5.size ∧
M5 = store 0, P5, M4
R7 = R0 > 0 && R0 < 10   r7 = r0 > 0 ∧ r0 < 10 ∧
R8 = false               r8 = false ∧
A0 = select R7, R8, R1   a0 = ite(r7, r8, r1) ∧
A1 = select R6, A0, R1   a1 = ite(r6, a0, r1) ∧
A2 = select R4, A1, R1   a2 = ite(r4, a1, r1) ∧
assume (A2)              a2 ∧
assert (0)                ¬false
halt
}

```

(b) VC in SMT-LIB

(a) Pure-Dataflow (PD)

Fig. 9: Program from Fig. 2 in PD and SMT-LIB forms. The **isderef** instruction checks for spatial memory safety.

$$\begin{aligned}
\text{sym}(P1, M1 = \text{malloc } R0, M0) &\triangleq \\
p_1 = \text{alloc}(\text{malloc } R0, M0) \wedge m_1 = \text{alloc}_{sh}(m_0, p_1) \\
\text{sym}(M1 = \text{free } P0, M0) &\triangleq m_1 = \text{free}_{sh}(m_0, p_0) \\
\text{sym}(MR = \text{store } R1, P2, M1) &\triangleq \\
\langle m_{r_1}, \dots, m_{r_j} \rangle &= \langle \text{write}(m_0.1, r_1, \text{addr}(p_2.1)), \dots, \\
&\quad \text{write}(m_0.j, r_1, \text{addr}(p_2.j)) \rangle \wedge \\
\langle m_{1_{j+1}}, \dots, m_{1_k} \rangle &= \text{store}_{sh}(\langle m_{0_{j+1}}, \dots, m_{0_k} \rangle, p_2) \\
\text{sym}(R1 = \text{load } P0, M0) &\triangleq \\
r_1 = \langle \text{read}(m_0.1, \text{addr}(p_0)), \dots, \text{read}(m_0.j, \text{addr}(p_0)) \rangle
\end{aligned}$$

Fig. 10: Memory-safety aware VCGen semantics.

condition. For that, we rewrite equality between two tuples as conjunction of equalities between their elements, and use $\tau.i$ for the i th element of tuple τ .

Similarly, we re-define *mems* for a memory register M to be a tuple of values that store the program and the shadow states. Thus, $\text{sym}(M) = \langle v_0, \dots, v_k \rangle$, where each v_i is the sort $bv(64) \rightarrow bv(64)$. If a pointer is represented by a j -tuple, we assume that memory is represented by a k -tuple, with $k \geq j$, so that the first j entries in a memory register are wide enough to store the fat pointer. Specifically, we require that the sort of v_j is same as sort of t_j for $1 \leq j \leq k$.

We modify the semantics of **malloc** by storing meta data along with explicit program states. The modification is defined in Fig. 10 ($M1$ is now a memory tuple). The signature of *alloc* is unchanged, but now returns a fat pointer. Given a pointer p of sort *ptrs*, a function $\text{size}(\langle t_1, \dots, t_j \rangle)$ returns the size of a memory object pointed-to by p . An additional function $\text{alloc}_{sh} : \text{mems} \rightarrow \text{mems}$ operates on shadow memory. The semantics of alloc_{sh} and free_{sh} is described later.

A **store** is divided into two parts. First is the store of the actual program data. Since the data can be of sort *scalr* or *ptrs*, a store of a k -tuple of data on memory m_0 is translated into k writes, on each element of $\langle m_{01}, \dots, m_{0j} \rangle$. Second is updating metadata, done by store_{sh} that works

$$\begin{aligned}
\text{alloc}_{sh}(m, p) &\triangleq \\
&\langle m.val, m.offset, m.size, \text{write}(m.alloc, 1, p.base), m.mod \rangle \\
\text{free}_{sh}(m, p) &\triangleq \\
&\langle m.val, m.offset, m.size, \text{write}(m.alloc, 0, p.base), m.mod \rangle \\
\text{store}_{sh}(\langle m.alloc, m.mod \rangle, p) &\triangleq \\
&\langle m.alloc, \text{write}(m.mod, 1, p.base) \rangle
\end{aligned}$$

Fig. 11: Shadow memory semantics for memory safety.

$$\begin{aligned}
\text{sym}(R1 = \text{isderef } P0 B) &\triangleq r_1 = 0 \leq p_0.offset < p_0.size \\
\text{sym}(R1 = \text{isalloc } P0 M) &\triangleq r_1 = \text{read}(m.alloc, p_0.base) \\
\text{sym}(R1 = \text{ismod } P0 M) &\triangleq r_1 = \text{read}(m.mod, p_0.base)
\end{aligned}$$

Fig. 12: Semantics for verifying memory safety.

on $\langle m_{0_{j+1}}, \dots, m_{0_k} \rangle$. The details of store_{sh} are described later in this section. Similarly, a **load** expects to read $\langle m_{01}, \dots, m_{0j} \rangle$ of sort *ptrs*. This allows representing arbitrary fat and shadows. We illustrate its specializations for memory safety next.

Spatial memory safety A program satisfies spatial memory safety iff every read and write is always inside an allocated object. A fat pointer is defined as a tuple of three constants $\langle s_1, s_2, s_3 \rangle$ denoted as $\langle base, offset, size \rangle$ for convenience. Here *base* is the start address of the object, *offset* is an index into the object, and *size* is its size. The address *addr* is given by $base + offset$.

With fat pointers, we introduce instructions for pointer arithmetic and pointer integer casts. The instruction **gep** is used for pointer arithmetic. Fig. 9a shows an example use in $R5 = \text{gep } R3, 1$. Here, semantically, a new pointer $R5$ is created that has the same *base* and *size* as $R3$, with *offset* incremented by 1. We also introduce **ptoi** instruction that casts a pointer to an integer by adding *offset* to *base*. For an integer to pointer cast, we use the **itop** instruction. This instruction sets *base* to the integer value and fat (i.e., metadata) to zero.

To assert that a pointer dereference is spatially safe, we provide an **isderef** instruction, whose semantics is shown in Fig. 12. For example, the program in Fig. 9a executes **assert**(0) as $R6 = \text{isderef } R5, 1$ evaluates to *false* causing $A1$ and $A2$ to evaluate to $R1$ and *true*, respectively. Thus, the VC in Fig. 9b is *satisfiable* which exposes the out of bounds error in Fig. 2 line 9. Note that this error is not caught by the VC in Sec. II. In SEABMC, we automatically add **isderef** assertions before memory accesses. Many of such assertions are statically and, thus, cheaply resolved to *true* or *false* prior to SMT solving.

Note that SEABMC semantics for spatial safety differs from LLBMC [17]. LLBMC treats only accesses to unallocated memory as unsafe. This implies that it is valid for a pointer to overflow into another object allocated just below or above. In SEABMC, jumping across the allocated boundary is invalid. SEABMC also differs from CBMC in this regard. In CBMC [1], the pointer representation is fixed and a few bits in the pointer representation are reserved for fat data. These

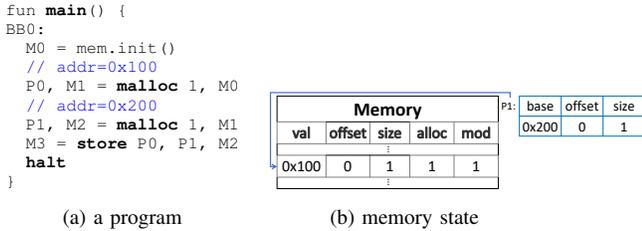


Fig. 13: Memory state M3 - when P0 is stored at location P1.

constraint the available address range. Additionally, only limited metadata can be stored in each pointer. In SEABMC, we support composite pointer representations that maintain parity with concrete pointer representation while allowing for rich metadata in the fat region of the pointer.

Temporal memory safety A program satisfies temporal memory safety iff it never does one of the following: **(UAF)** an object is used after it has been freed; and **(RO)** an object marked as read-only (by programmers) is modified. We detect a violation of memory safety by tracking the status of a memory object using shadow memory. Each memory is a tuple $\langle v_1, \dots, v_5 \rangle$ of constants of sort $bv(64) \rightarrow bv(64)$, denoted $\langle val, offset, size, alloc, mod \rangle$, where $\langle val, offset, size \rangle$ maps to pointer data $\langle base, offset, size \rangle$, and $alloc$ and mod track the allocated and modified status of an object, respectively.

An object can be in allocated or freed state. To track allocated state, sym in Sec. II is extended for `alloca`, `malloc`, and `free`. The new semantics is shown in Fig. 10. The function $alloc_{sh} : mems \rightarrow mems$ is defined, for temporal memory safety, as shown in Fig. 11. Note that $alloc_{sh}(m, r)$ marks $m.mod$ memory only at the start of an object, i.e., $r.base$. For this reason it is necessary to use the fat pointer representation since it records the $base$ for every pointer. The `isalloc` instruction, shown in Fig. 8, is used to check the allocated state of an object at any point in the program. The semantics for `isalloc` is defined in Fig. 12.

A C program has no native mechanism for verifying that an object remains unmodified when passed to a function. To remedy this, we extend the semantics for `store` (see Fig. 10). The function $store_{sh} : mems \rightarrow mems$ is implemented for temporal memory safety (see Fig. 11). The `ismod` in Fig. 8 is used to check the read-only state of an object at any program point. The semantics for `ismod` is given in Fig. 12. We also provide a companion instruction `resetmod R, M` that resets $m.mod$ at address $r.base$ to zero. This allows initializing an object, resetting modified state, and then checking that the subsequent program does not modify the object. We track memory state only at object granularity, therefore, the current implementation is tied to using the fat pointer representation.

IV. EXPERIMENTS

In this section, we describe the evaluation of SEABMC⁵ on verification tasks from `aws-c-common`. Each task verified post-conditions and memory safety of a single function from

⁵Source at <https://github.com/seahorn/seahorn/tree/dev10>.

`aws-c-common`. Overall, there are 169 tasks in 20K LOC. Results and tasks are available at <https://github.com/seahorn/verify-c-common>⁶. We have chosen these tasks because they represent a real industrial use-case of BMC. We have adapted them from CBMC to be compatible with LLVM-based C verification tools. Note that here we focus on SEABMC performance. A detailed comparison of different *kinds* of verification tools on `aws-c-common` is presented in [14].

Comparing Different VCGen Strategies We evaluate the effectiveness of the different VCGen strategies by controlling which transformations are enabled. The main performance metric is *time solved* – the time to solve *all* solved tasks⁷ (i.e., with timeout excluded). The time limit is 600s per task.

First, we evaluate the two memory representations: Arrays vs Lambdas. We use Z3 [18] and YICES2 [19] to account for the difference between SMT-solvers. The results are summarized in Tab. Ia. For Z3, we find that Arrays are less efficient than Lambdas. For YICES2, the results are comparable, suggesting that the choice of the representation is less important. Z3 with Lambdas is the overall winner, and we use it for the rest of the experiments.

Second, we evaluate the effectiveness of the transformations in Sec. II. The results are in Tab. Ib. Here, *optimal* means applying all of the transformation involved, plus eagerly simplifying VC during VCGen. β -reducing lambdas introduces many nested ITE-terms, so simplifying them early is useful.

To evaluate, we compare with 5 additional strategies by disabling some transformations: 1) `rel_alloc` – use $alloc$ that returns relative addresses from some symbolic start of stack and heap, rather than concrete addresses 2) `flat_mem` – one flat memory instead of using alias analysis to partition memory into disjoint memories as much as possible 3) `no_coi` – disable cone-of-influence 4) `no_simp` – disable eager simplification 5) `p_cond` – generate VC directly from SSA form by using path condition to encode `phi`-functions as in [6], [20]. Removing any of the transformations either noticeably degrades performance, or causes a timeout.

SEABMC supports memory word size of 1 byte ($bv(8)$), 4 bytes ($bv(32)$) and 8 bytes ($bv(64)$). The 1-byte words are most precise and support arbitrary memory accesses, while 8-byte words require aligned accesses. The comparison between the two is shown in Tab. Ic. Wider words significantly improve performance, but can be unsound for some benchmarks. By supporting both, SEABMC lets the user pick most appropriate choice per benchmarks. In other experiments, we adjust word size per individual benchmarks.⁸

Shadow memory performance A C program has no builtin mechanism for verifying that an object is not modified by a function. To overcome this limitation, the verification tasks in `aws-c-common` record the value of a byte from a non deterministic offset within an allocated object and then verify that this byte is unchanged in all executions. While this is a

⁶This website includes instructions for reproducing the experiments.

⁷This analysis uses 172 tasks instead of 169. 3 tasks are SEABMC specific.

⁸CBMC uses a similar per-benchmark configuration as well.

config	solver	unsat	timeout	failed	solved time(s)	config	unsat	timeout	solved time(s)	avg(s)	std(s)	word size	unsat	timeout	failed	solved time(s)	config	unsat	solved time(s)
array	z3	158	8	6	1647	optimal	172	0	836	5	10	bv(64)	156	0	16	679	no shadow memory	70	143
	yices2	170	0	2	1016	rel_alloc	172	0	1456	8	19	bv(8)	171	1	0	2546	shadow memory	70	90
lambdas	z3	172	0	0	836	flat_mem	163	9	2689	16	55								
	yices2	172	0	0	912	no_coi	170	2	849	5	10								
						no_simp	166	6	1429	9	33								
						p_cond	170	2	659	4	6								

(a) Different memory representations.

(b) Different encodings.

(c) Different word sizes.

(d) Different memory features.

TABLE I: Evaluations of different configuration.

clever technique, setting it up in a verification task is complex. The `ismod` instruction added in SEABMC (see Sec. III) offers a user friendly alternative. We also found it to be more performant in the SEABMC implementation. We ported 70 tasks in `aws-c-common` to use `ismod`. Ported tasks ran 55% faster, on average, than their originals (see Tab. Id). This strengthens the case for shadow memory from both usability and performance perspectives.

SEABMC vs. State-of-the-Art Overall, the results for our configurations in previous discussion suggest that the *optimal* strategy provides best performance in terms of precision and efficiency. We also consider four tools comparing against: CBMC [1], SMACK [4], KLEE [13], and SYMBIOTIC [12]. LLBMC is another interesting BMC tool, however, we decided to exclude it from comparisons due to the lack of an easily accessible public version⁹ for user to reproduce LLBMC results. CBMC is, perhaps, the oldest and most well-known BMC for C programs (not based on LLVM). It is actively used by AWS, and was used for the verification of `aws-c-common`. SMACK is an LLVM-based BMC tool that uses Boogie [21] and Corral [4] for bounded and deductive verification. SYMBIOTIC is a KLEE-based tool that combines program instrumentation, slicing, and symbolic execution [22]. Both SMACK and SYMBIOTIC performed very well on the “SoftwareSystems” category in SV-COMP’21. KLEE is a LLVM-based symbolic execution tool that does not encode the VC in one shot but rather explores satisfiability of path conditions in a program one path-at-a-time. It is a practical alternative to BMC.

The results collected on an AMD Ryzen(TM) 5 5600X CPU with 32 GB memory are shown in Tab. II. Only SEABMC and CBMC solve all verification tasks from `aws-c-common`. SMACK in bit-precise mode times out on most instances, and in arithmetic mode times out on 20 and fails on 4. SYMBIOTIC times out on 5 and fails on 10. It is best-performing on `priority_queue` and `ring_buffer`. However, it also failed to detect seeded bugs¹⁰, which questions its results. KLEE is particularly effective on `linked_list` – showing the benefit of exploring path-at-a-time, when number of paths is small.

Bugs found In [14], we discuss bugs found and reported to AWS. One example, in Fig. 14, concerns the `byte` buffer data structure that is defined as a length delimited byte string.

⁹LLBMC source code is not publicly available; Binary download on website is broken.

¹⁰Details at <https://github.com/seahorn/verify-c-common/issues/124>

```

1 typedef
2 struct byte_buf {
3     char* buf;
4     int len, cap;
5 } BB;
6 bool BB_is_ok(BB *b)
7 { return (b->len == 0
8           || b->buf); }

```

Fig. 14: Incorrect `byte_buf` invariant

Its data representation should be either the buffer (`buf`) is `NULL` or its capacity (`cap`) is 0 (not the `len` as defined in `BB_is_ok` Line 7). Under the correct model (a `malloc` that can potentially fail), SEABMC produces counter examples in 50 seconds, CBMC in 112 seconds. However, KLEE cannot detect this bug since it needs an allocated buffer with an explicit size to proceed with analysis.

Overall, SEABMC outperforms competitors on most categories and in the overall run-time. Thus, we conclude that SEABMC is a highly efficient BMC engine.

We have compared SEABMC with tools from SV-COMP, but not with the benchmarks. There are two reasons. First, while a version of `aws-c-common` appears in SV-COMP, it is pre-processed with CBMC harnesses, and, therefore, includes undefined behaviors (e.g., uninitialized variables). This is not supported by SEABMC front-end. Second, we felt it is more important to validate tools in an actively developed code-base. Thus, we focused our effort on building an infrastructure for continuously verifying current `aws-c-common` using many existing tools, rather than integrating SEABMC into the rules of SV-COMP.

V. RELATED WORK

Bounded Software Model Checking is a mature program analysis technique. We briefly review only some of the closest related work. Over the years, there have been many model checking tools built on top of the LLVM platform. The closest to ours is the work of Babic [23] and LLBMC [17]. Similarly to [23], we rely on the Gated SSA form to remove all control dependence leaving only data-flows to be represented. However, our encoding is significantly simplified by an intermediate representation that purifies memory flows. Unfortunately, [23] has not been maintained making head-to-head comparison difficult.

We borrow the idea of using lambda-encoding for representing memory from LLBMC [17]. One important advantage of lambdas is that we can represent memory operations such as `memcpy` efficiently (while with arrays, these have to be

category	Statistics		SEABMC			CBMC			SMACK				SYMBIOTIC				KLEE					
	cnt	loc	avg (s)	std (s)	time (s)	avg (s)	std (s)	time (s)	cnt	fld/to	avg (s)	std (s)	time (s)	cnt	fld/to	avg (s)	std (s)	time (s)	cnt	avg (s)	std (s)	time (s)
arithmetic	6	202	1	0	3	4	0	22	6	2/0	3	1	18	6	0/0	135	281	809	6	1	0	5
array	4	390	2	1	7	6	0	23	4	0/1	53	98	213	4	0/0	11	4	44	4	26	2	103
array_list	24	3,150	3	4	71	19	33	450	24	0/0	5	1	126	23	0/0	43	68	980	24	41	38	994
byte_buf	29	2,908	1	1	29	9	10	252	29	0/2	27	50	788	29	0/0	40	162	1,168	27	59	96	1,592
byte_cursor	24	2,365	1	0	23	6	3	153	16	0/2	32	66	519	17	0/0	7	4	125	17	10	11	169
hash_callback	3	347	6	5	18	8	5	25	3	0/0	4	2	11	3	0/0	40	62	120	3	50	38	151
hash_tier	4	708	9	15	37	10	6	39	4	0/0	91	58	363	3	0/1	37	44	112	3	14	6	41
hash_table	19	3,295	6	8	105	19	28	366	19	2/4	54	79	1,025	15	8/4	472	1,261	7,088	15	33	72	492
linked_list	18	2,127	2	2	37	33	112	595	18	0/5	96	91	1,735	18	0/0	8	5	143	18	1	0	12
others	2	31	0	0	1	4	0	7	1	0/0	2	0	2	1	0/0	5	0	5	1	1	0	1
priority_queue	15	3,004	14	22	202	286	700	4,284	15	0/1	20	50	307	15	0/0	10	20	152	15	32	8	473
ring_buffer	6	934	21	22	128	13	8	78	6	0/3	133	98	796	6	1/0	10	9	63	6	30	16	180
string	15	1,329	3	2	49	7	5	104	15	0/2	31	69	467	15	1/0	9	11	137	15	102	106	1,528
total	169	20,790			710			6,398	4/20				6,370	10/5			10,946					5,741

TABLE II: Verification results for SEABMC, CBMC, SMACK, SYMBIOTIC, and KLEE. Timeout for SMACK and SEABMC is 200s, and 5,000s for SYMBIOTIC. **cnt**, **fld**, **to**, **avg**, **std** and **time**, are the number of verification tasks, failed cases, timeout cases, average run-time, standard deviation, and total run-time in seconds, per category.

unfolded). In particular, this allows for unbounded verification of loop-free programs that use these operations. The most significant difference from LLBMC is in our encoding of memory safety. In particular, we cache bounds information in the pointer, and check that every access is inside the allocated memory object. In contrast, LLBMC assumes an arbitrary allocator and checks that all accesses are into some allocated memory, not necessarily into the expected object. Unfortunately, there is no public version of LLBMC available, precluding a head-to-head comparison.

SMACK [4], [5] is probably the most known BMC for LLVM. It is based on Boogie and Corral from Microsoft Research. It is most effective for arithmetic abstraction of software (i.e., abstracting machine integers by arbitrary precision integers). Its model for memory safety relies on complex encoding using universally quantified axioms in Boogie, leading to quantified reasoning in SMT. In contrast, our representation is tuned to perform well with modern SMT solvers. SMACK shares SEADSA [24], [25] alias analysis with SEABMC. DIVINE4 [26] is an explicit state model checker that also targets LLVM. However, it uses LLVM 7 which makes head-to-head comparison difficult. It targets parallel programs, which SEABMC does not. For sequential programs, it is related to libFuzzer and KLEE that we compare with.

VI. CONCLUSION

We have presented the techniques behind SEABMC, a new LLVM-base Bounded Model Checker for C. SEABMC is path-sensitive, bit-precise, and provides a precise model of memory. It extends the traditional memory model with *fat* pointers and *shadow* memory that allow attaching metadata to pointers and memory. We have evaluated SEABMC against CBMC, SMACK, SYMBIOTIC, and KLEE and show significant performance improvements over the competition.

REFERENCES

- [1] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.
- [2] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of C: elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krutz and E. Berger, Eds. ACM, 2016, pp. 1–15.
- [3] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88.
- [4] Z. Rakamaric and M. Emmi, "SMACK: decoupling source language details from verifier implementations," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 106–113.
- [5] J. J. Garzella, M. S. Baranowski, S. He, and Z. Rakamaric, "Leveraging compiler intermediate representation for multi- and cross-language verification," in *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. Beyer and D. Zufferey, Eds., vol. 11990. Springer, 2020, pp. 90–111.
- [6] F. Merz, S. Falke, and C. Sinz, "LLBMC: bounded model checking of C and C++ programs using a compiler IR," in *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Joshi, P. Müller, and A. Podelski, Eds., vol. 7152. Springer, 2012, pp. 146–161.
- [7] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370. [Online]. Available: https://doi.org/10.1007/978-3-642-17511-4_20
- [8] S. Falke, F. Merz, and C. Sinz, "Extending the theory of arrays: memset, memcpy, and beyond," in *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. Cohen and A. Rybalchenko, Eds., vol. 8164. Springer, 2013, pp. 108–128. [Online]. Available: https://doi.org/10.1007/978-3-642-54108-7_6
- [9] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016.
- [10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>
- [11] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science,

- J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 401–422.
- [12] J. Slaby, J. Strejcek, and M. Trtík, “Symbiotic: Synergy of instrumentation, slicing, and symbolic execution - (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Piterman and S. A. Smolka, Eds., vol. 7795. Springer, 2013, pp. 630–632. [Online]. Available: https://doi.org/10.1007/978-3-642-36742-7_50
- [13] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.
- [14] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel, “Verifying verified code,” in *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Proceedings*, ser. Lecture Notes in Computer Science. Springer, 2021.
- [15] F. C. Chow, S. Chan, S. Liu, R. Lo, and M. Streich, “Effective representation of aliases and indirect memory operations in SSA form,” in *Compiler Construction, 6th International Conference, CC’96, Linköping, Sweden, April 24-26, 1996, Proceedings*, ser. Lecture Notes in Computer Science, T. Gyimóthy, Ed., vol. 1060. Springer, 1996, pp. 253–267.
- [16] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, “Cyclone: A safe dialect of C,” in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, C. S. Ellis, Ed. USENIX, 2002, pp. 275–288.
- [17] C. Sinz, S. Falke, and F. Merz, “A precise memory model for low-level bounded model checking,” in *5th International Workshop on Systems Software Verification, SSV’10, Vancouver, BC, Canada, October 6-7, 2010*, R. Huuck, G. Klein, and B. Schlich, Eds. USENIX Association, 2010.
- [18] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.
- [19] B. Dutertre, “Yices 2.2,” in *Computer-Aided Verification (CAV’2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, July 2014, pp. 737–744.
- [20] A. Gurfinkel, S. Chaki, and S. Sapra, “Efficient predicate abstraction of program summaries,” in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 131–145.
- [21] K. R. M. Leino, “This is Boogie 2,” 2008.
- [22] J. Slaby, J. Strejcek, and M. Trtík, “Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution,” in *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Stoelinga and R. Pinger, Eds., vol. 7437. Springer, 2012, pp. 207–221. [Online]. Available: https://doi.org/10.1007/978-3-642-32469-7_14
- [23] D. Babić, “Exploiting Structure for Scalable Software Verification,” Ph.D. dissertation, University of British Columbia, Canada, 2008.
- [24] A. Gurfinkel and J. A. Navas, “A context-sensitive memory model for verification of C/C++ programs,” in *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, ser. Lecture Notes in Computer Science, F. Ranzato, Ed., vol. 10422. Springer, 2017, pp. 148–168.
- [25] J. Kuderski, J. A. Navas, and A. Gurfinkel, “Unification-based pointer analysis without oversharing,” in *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, C. W. Barrett and J. Yang, Eds. IEEE, 2019, pp. 37–45.
- [26] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkal, and V. Štill, “Model checking of C and C++ with DIVINE 4,” in *Automated Technology for Verification and Analysis*, ser. LNCS, vol. 10482. Springer, 2017, pp. 201–207.

flag	meaning
--unwind 1	number of times to unwind loops
--flush	print to stdout
--object-bits 8	number of pointer bits to store meta information
--malloc-may-fail	malloc may fail
--malloc-fail-null	malloc may fail and return NULL

TABLE V: CBMC options for **no-mem-safe**.

flag	meaning
--unwind 1	number of times to unwind loops
--flush	print to stdout
--object-bits 8	number of pointer bits to store meta information
--malloc-may-fail	malloc may fail
--malloc-fail-null	malloc may fail and return NULL

TABLE VI: CBMC options **no-memmove**.

verification task	config	tool	run-time (s)
aws-array-list-erase	all	SEABMC	4
		CBMC	98
aws-array-list-erase	no-mem-safe	SEABMC	2
		CBMC	98
aws-array-list-erase	no-memmove	SEABMC	3
		CBMC	40

TABLE III: SEABMC vs. CBMC for `aws-array-list-erase`.

flag	meaning
--unwind 1	number of times to unwind loops
--flush	print to stdout
--object-bits 8	number of pointer bits to store meta information
--malloc-may-fail	malloc may fail
--malloc-fail-null	malloc may fail and return NULL
--bounds_check	check access is within bounds
--pointer_check	check access is within bounds

TABLE IV: CBMC options for **all**.

APPENDIX

Performance of SEABMC vs CBMC In this section we look at performance of SEABMC vs CBMC more closely. In App. A we study tool performance on a single task by using different features of the tools. In App. B, we look at the CBMC flags used for the analysis.

A. Comprehensive Analysis w.r.t. CBMC

SEABMC outperforms CBMC on many of the categories. To ensure that the comparison is “fair”, we have done a

comprehensive manual analysis with a few verification tasks.

For a fair comparison, one must show that the verification problem being solved is the same. While both tools verify user-supplied assertions in `aws-c-common`, they also verify internal properties such as memory safety, integer overflow, etc., depending on how they are invoked. For example, CBMC checks for integer overflow, while SEABMC does not. Hence, as a first step, we identified all such options in CBMC and disabled them.

There are many other factors that differentiate SEABMC and CBMC including: IRs (i.e., GOTO program vs. LLVM-IR), model of memory operations, and VCGen. Thus, we identified the differences that benefit SEABMC. We chose one verification task `aws-array-list-erase`, and derived 3 configurations based on the above analysis¹¹: 1) *All*: SEABMC and CBMC verify a similar set of properties, namely, user-supplied assertions and memory safety. 2) *No Memory Safety*: SEABMC and CBMC verify user-supplied assertions only. 3) *No memmove*: `aws-array-list-erase` uses `memmove` in its implementation. Since `memmove` has custom implementations in both SEABMC and CBMC, we evaluated run-time when disabling the assertions for it.¹²

The results are shown in Tab. III. We present the analysis for one verification task, however, the same applied to other verification tasks where SEABMC outperforms CBMC— even

¹¹See App. B for CBMC flags used.

¹²These assertions guarantee spatial memory safety of `memmove`. when verifying similar properties. Further manual analysis shows that most difference is due to the model of memory in SEABMC and CBMC. Specifically, memory operations on large blocks, are very expensive for CBMC (40s vs. 98s due to pre-conditions for `memmove` in Tab. III).

B. Command line options for CBMC

This section lists the CBMC command line flags used for `aws-array-list-erase` verification job for different configuration.

all Options to enable user assertions and memory safety checks

no-mem-safe Options to enable user assertions only

no-memmove Options to enable user assertions and remove memory safety and `memmove` checks.

The `memmove` checks are disabled manually in source code.

Automatic Repair and Deadlock Detection for Parameterized Systems

Swen Jacobs 
CISPA, Saarbrücken, Germany

Mouhammad Sakr 
SnT, University of Luxembourg

Marcus Völz 
SnT, University of Luxembourg

Abstract—We present an algorithm for the repair of parameterized systems. The repair problem is, for a given process implementation, to find a refinement such that a given safety property is satisfied by the resulting parameterized system, and deadlocks are avoided. Our algorithm uses a parameterized model checker to determine the correctness of candidate solutions and employs a constraint system to rule out candidates. We apply this algorithm on systems that can be represented as well-structured transition systems (WSTS), including disjunctive systems, pairwise rendezvous systems, and broadcast protocols. Moreover, we show that parameterized deadlock detection can be decided in EXPTIME for disjunctive systems, and that deadlock detection is in general undecidable for broadcast protocols.

I. INTRODUCTION

Concurrent systems are hard to get correct, and are therefore a promising application area for formal methods. For systems that are composed of an arbitrary number of processes n , methods such as parameterized model checking can provide correctness guarantees that hold regardless of n . While the parameterized model checking problem (PMCP) is undecidable even if we restrict systems to uniform finite-state processes [1], there exist several approaches that decide the problem for specific classes of systems and properties [2]–[10].

However, if parameterized model checking detects a fault in a given system, it does not tell us how to repair the latter such that it satisfies the specification. To repair the system, the user has to find out which behavior of the system causes the fault, and how it can be corrected. Both tasks may be nontrivial.

For faults in the internal behavior of a process, the approach we propose is based on a similar idea as existing repair approaches [11], [12]: we start with a non-deterministic implementation, and restrict non-determinism to obtain a correct implementation. This non-determinism may have been added by a designer to “propose” possible repairs for a system that is known or suspected to be faulty.

However, repairing a process internally will not be enough in the presence of concurrency. We need to go beyond existing repair approaches, and also repair the communication between processes to ensure the large number of possible interactions between processes is correct as well. We do so by choosing the right options out of a set of possible interactions, combining the idea above with that of synchronization synthesis [13], [14].

In addition to guaranteeing safety properties, we aim for an approach that avoids introducing deadlocks, which is particularly important for a repair algorithm, since often the

easiest way to “repair” a system is to let it run into a deadlock as quickly as possible. Unlike non-determinism for repairing internal behavior, we are even able to introduce non-determinism for repairing communication automatically.

Regardless of whether faults are fixed in the internal behavior or in the communication of processes, we aim for a parameterized correctness guarantee, i.e., the repaired implementation should be correct in a system with any number of processes. We show how to achieve this by integrating techniques from parameterized model checking into our repair approach.

High-Level Parameterized Repair Algorithm. Figure 1 sketches the basic idea of our parameterized repair algorithm.

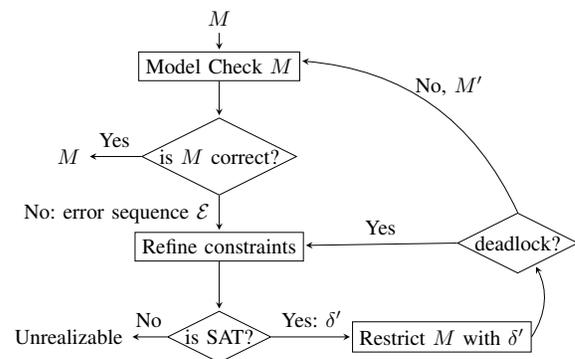


Fig. 1: Parameterized repair of concurrent systems

The algorithm starts with a representation M of the parameterized system, based on non-deterministic models of the components, and checks if error states are reachable for any size of M . If not, the components are already correct. Otherwise, the parameterized model checker returns an error sequence \mathcal{E} , i.e., one or more concrete error paths. \mathcal{E} is then encoded into constraints that ensure that any component that satisfies them will avoid the error paths detected so far. A SAT solver is used to find out if any solution still exists, and if so we restrict M to components that avoid previously found errors. To guarantee that this restriction does not introduce deadlocks, the next step is a parameterized deadlock detection. This provides similar information as the model checker, and is used to refine the constraints if deadlocks are reachable. Otherwise, M' is sent to the parameterized model checker for the next iteration.

Research Challenges. Parameterized model checking in general is known to be undecidable, but different decision pro-

cedures exist for certain classes of systems, such as guarded protocols with disjunctive guards (or disjunctive systems) [4], pairwise systems [2] and broadcast protocols [3]. However, these theoretical solutions are not uniform and do not provide practical algorithms that allow us to extract the information needed for our repair approach. Therefore, the following challenges need to be overcome to obtain an effective parameterized repair algorithm for a broad class of systems:

- C1 The parameterized model checking algorithm should be uniform, and needs to provide information about error paths in the current candidate model that allow us to avoid such error paths in future repair candidates.
- C2 We need an effective approach for parameterized deadlock detection, preferably supplying similar information as the model checker.
- C3 We need to identify an encoding of the discovered information into constraints such that the repair process is sufficiently flexible¹, and sufficiently efficient to handle examples of interesting size.

Parameterized Repair: an Example. Consider a system with one scheduler (Fig. 2) and an arbitrary number of reader-writer processes (Fig. 3), running concurrently and communicating via pairwise rendezvous, i.e., every send actions (e.g. *write!*) needs to synchronize with a receive action (e.g. *write?*) by another process. In this system, multiple processes can be in the *writing* state at the same time, which must be avoided if they use a shared resource. We want to repair the system by restricting communication of the scheduler.

According to the idea in Fig. 1, the parameterized model checker searches for reachable errors, and it may find that after two consecutive *write!* transitions by different reader-writer processes, they both occupy the *writing* state at the same time. This information is then encoded into constraints on the behavior of processes, which restrict non-determinism and communication and make the given error path impossible. To repair the system we mainly need somehow to oblige a process to wait for the action *done_w!* (done writing) before entering the *writing* state. However, in our example all errors could be avoided by simply removing all outgoing transitions of state $q_{A,0}$ of the scheduler. To avoid such repairs, our algorithm uses *initial constraints* (see section IV) that enforce totality on the transition relation. Another undesirable solution would be the scheduler shown in Fig. 4, because the resulting system will deadlock immediately. This is avoided by checking reachability of deadlocks on candidate repairs. We get a solution that is safe and deadlock-free if we take Fig. 4 and flip all transitions.

Contributions. Our main contribution is a counterexample-guided parameterized repair approach, based on model checking of well-structured transition systems (WSTS) [15], [16]. We investigate which information a parameterized model checker needs to provide to guide the search for candidate

¹For example, to allow the user to specify additional properties of the repair, such as keeping certain states reachable.

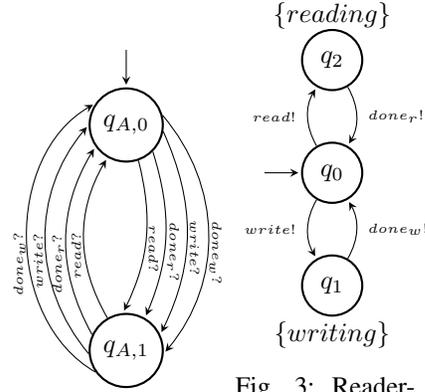


Fig. 2: Scheduler

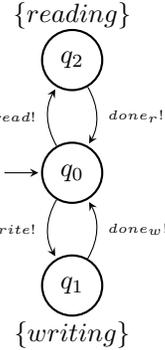


Fig. 3: Reader-Writer

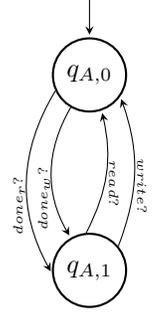


Fig. 4: Deadlocked Scheduler

repairs, and how this information can be encoded into propositional constraints. Our repair algorithm supports internal repairs and repairs of the communication behavior, while systematically avoiding deadlocks in many classes of systems, including disjunctive systems, pairwise systems and broadcast protocols.

Since existing model checking algorithms for WSTS do not support deadlock detection, our approach has a subprocedure for this problem, which relies on *new theoretical results*: (i) for disjunctive systems, we provide a novel deadlock detection algorithm, based on an abstract transition system, that improves on the complexity of the best known solution; (ii) for broadcast protocols we prove that deadlock detection is in general undecidable, so approximate methods have to be used. We also discuss approximate methods to detect deadlocks in pairwise systems, which can be used as an alternative to the existing approach that has a prohibitive complexity.

Finally, we evaluate an implementation of our algorithm on benchmarks from different application domains, including a distributed lock service and a robot-flocking protocol.

II. SYSTEM MODEL

For simplicity, we first restrict our attention to disjunctive systems, other systems will be considered in Sect. V-B. In the following, let Q be a finite set of states.

Processes. A *process template* is a transition system $U = (Q_U, \text{init}_U, \mathcal{G}_U, \delta_U)$, where $Q_U \subseteq Q$ is a finite set of states including the initial state init_U , $\mathcal{G}_U \subseteq \mathcal{P}(Q)$ is a set of guards, and $\delta_U : Q_U \times \mathcal{G}_U \times Q_U$ is a guarded transition relation.

We denote by t_U a transition of U , i.e., $t_U \in \delta_U$, and by $\delta_U(q_U)$ the set of all outgoing transitions of $q_U \in Q_U$. We assume that δ_U is *total*, i.e., for every $q_U \in Q_U$, $\delta_U(q_U) \neq \emptyset$. Define the *size* of U as $|U| = |Q_U|$. An instance of template U will be called a *U-process*.

Disjunctive Systems. Fix process templates A and B with $Q = Q_A \dot{\cup} Q_B$, and let $\mathcal{G} = \mathcal{G}_A \cup \mathcal{G}_B$ and $\delta = \delta_A \cup \delta_B$. We

consider systems $A\|B^n$, consisting of one A -process and n B -processes in an interleaving parallel composition.²

The systems we consider are called “disjunctive” since guards are interpreted disjunctively, i.e., a transition with a guard g is enabled if there *exists* another process that is currently in one of the states in g . Figures 5 and 6 give examples of process templates. An example disjunctive system is $A\|B^n$, where A is the writer and B the reader, and the guards determine which transition can be taken by a process, depending on its own state and the state of other processes in the system. Transitions with the trivial guard $g = Q$ are displayed without a guard. We formalize the semantics of disjunctive systems in the following.

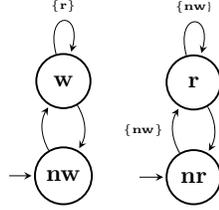


Fig. 5: Writer

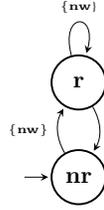


Fig. 6: Reader

Counter System. A *configuration* of a system $A\|B^n$ is a tuple (q_A, \mathbf{c}) , where $q_A \in Q_A$, and $\mathbf{c} : Q_B \rightarrow \mathbb{N}_0$. We identify \mathbf{c} with the vector $(\mathbf{c}(q_0), \dots, \mathbf{c}(q_{|B|-1})) \in \mathbb{N}_0^{|B|}$, and also use $\mathbf{c}(i)$ to refer to $\mathbf{c}(q_i)$. Intuitively, $\mathbf{c}(i)$ indicates how many processes are in state q_i . We denote by \mathbf{u}_i the unit vector with $\mathbf{u}_i(i) = 1$ and $\mathbf{u}_i(j) = 0$ for $j \neq i$.

Given a configuration $s = (q_A, \mathbf{c})$, we say that the guard g of a local transition $(q_U, g, q'_U) \in \delta_U$ is *satisfied* in s , denoted $s \models_{q_U} g$, if one of the following conditions holds:

- (a) $q_U = q_A$, and $\exists q_i \in Q_B$ with $q_i \in g$ and $\mathbf{c}(i) \geq 1$
(A takes the transition, a B -process is in g)
- (b) $q_U \neq q_A$, $\mathbf{c}(q_U) \geq 1$, and $q_A \in g$
(B -process takes the transition, A is in g)
- (c) $q_U \neq q_A$, $\mathbf{c}(q_U) \geq 1$, and $\exists q_i \in Q_B$ with $q_i \in g$, $q_i \neq q_U$ and $\mathbf{c}(i) \geq 1$
(B -process takes the transition, another B -process is in different state in g)
- (d) $q_U \neq q_A$, $q_U \in g$, and $\mathbf{c}(q_U) \geq 2$
(B -process takes the transition, another B -process is in same state in g)

We say that the local transition (q_U, g, q'_U) is *enabled* in s .

Then the *configuration space* of all systems $A\|B^n$, for fixed A, B but arbitrary $n \in \mathbb{N}$, is the transition system $M = (S, S_0, \Delta)$ where:

- $S \subseteq Q_A \times \mathbb{N}_0^{|B|}$ is the set of states,
- $S_0 = \{\text{init}_A, \mathbf{c} \mid \mathbf{c}(q) = 0 \text{ if } q \neq \text{init}_B\}$ is the set of initial states,
- Δ is the set of transitions $((q_A, \mathbf{c}), (q'_A, \mathbf{c}'))$ s.t. one of the following holds:
 - 1) $\mathbf{c} = \mathbf{c}' \wedge \exists (q_A, g, q'_A) \in \delta_A : (q_A, \mathbf{c}) \models_{q_A} g$ (transition of A)
 - 2) $q_A = q'_A \wedge \exists (q_i, g, q_j) \in \delta_B : \mathbf{c}(i) \geq 1 \wedge \mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j \wedge (q_A, \mathbf{c}) \models_{q_i} g$ (transition of a B -process)

We will also call M the *counter system* (of A and B), and will call configurations *states* of M , or *global states*.

²The form $A\|B^n$ is only assumed for simplicity of presentation. Our results extend to systems with an arbitrary number of process templates.

Let $s, s' \in S$ be states of M , and $U \in \{A, B\}$. For a transition $(s, s') \in \Delta$ we also write $s \rightarrow s'$. If the transition is based on the local transition $t_U = (q_U, g, q'_U) \in \delta_U$, we also write $s \xrightarrow{t_U} s'$ or $s \xrightarrow{g} s'$. Let $\Delta^{\text{local}}(s) = \{t_U \mid s \xrightarrow{t_U} s'\}$, i.e., the set of all enabled outgoing local transitions from s , and let $\Delta(s, t_U) = s'$ if $s \xrightarrow{t_U} s'$. From now on we assume wlog. that each guard $g \in \mathcal{G}$ is a singleton.³

Runs. A *path* of a counter system is a (finite or infinite) sequence of states $x = s_1, s_2, \dots$ such that $s_m \rightarrow s_{m+1}$ for all $m \in \mathbb{N}$ with $m < |x|$ if the path is finite. A *maximal path* is a path that cannot be extended, and a *run* is a maximal path starting in an initial state. We say that a run is *deadlocked* if it is finite. Note that every run s_1, s_2, \dots of the counter system corresponds to a run of a fixed system $A\|B^n$, i.e., the number of processes does not change during a run. Given a set of error states $E \subseteq S$, an *error path* is a finite path that starts in an initial state and ends in E .

The Parameterized Repair Problem. Let $M = (S, S_0, \Delta)$ be the counter system for process templates $A = (Q_A, \text{init}_A, \mathcal{G}_A, \delta_A)$, $B = (Q_B, \text{init}_B, \mathcal{G}_B, \delta_B)$, and $ERR \subseteq Q_A \times \mathbb{N}_0^{|B|}$ a set of error states. The *parameterized repair problem* is to decide if there exist process templates $A' = (Q_A, \text{init}_A, \mathcal{G}_A, \delta'_A)$ with $\delta'_A \subseteq \delta_A$ and $B' = (Q_B, \text{init}_B, \mathcal{G}_B, \delta'_B)$ with $\delta'_B \subseteq \delta_B$ such that the counter system M' for A' and B' does not reach any state in ERR .

If they exist, we call $\delta' = \delta'_A \cup \delta'_B$ a *repair* for A and B . We call M' the *restriction* of M to δ' , also denoted $\text{Restrict}(M, \delta')$.

Note that by our assumption that the local transition relations are total, a trivial repair that disables all transitions from some state is not allowed.

III. PARAMETERIZED MODEL CHECKING OF DISJUNCTIVE SYSTEMS

In this section, we address research challenges **C1** and **C2**: after establishing that counter systems can be framed as well-structured transition systems (WSTS) (Sect. III-A), we introduce a parameterized model checking algorithm for disjunctive systems that suits our needs (Sect. III-B), and finally show how the algorithm can be modified to also check for the reachability of deadlocked states (Sect. III-C). Full proofs for the lemmas in this section can be found in the extended version [17].

A. Counter Systems as WSTS

Well-quasi-order. Given a set of states S , a binary relation $\preceq \subseteq S \times S$ is a *well-quasi-order* (wqo) if \preceq is reflexive, transitive, and if any infinite sequence $s_0, s_1, \dots \in S^\omega$ contains a pair $s_i \preceq s_j$ with $i < j$. A subset $R \subseteq S$ is an *antichain* if any two distinct elements of R are incomparable wrt. \preceq . Therefore, \preceq

³This is not a restriction as any local transition (q_U, g, q'_U) with a guard $g \in \mathcal{G}$ and $|g| > 1$ can be split into $|g|$ transitions $(q_U, g_1, q'_U), \dots, (q_U, g_{|g|}, q'_U)$ where for all $i \leq |g| : g_i \in g$ is a singleton guard.

is a wqo on S if and only if it is well-founded and has no infinite antichains.

Upward-closed Sets. Let \preceq be a wqo on S . The *upward closure* of a set $R \subseteq S$, denoted $\uparrow R$, is the set $\{s \in S \mid \exists s' \in R : s' \preceq s\}$. We say that R is *upward-closed* if $\uparrow R = R$. If R is upward-closed, then we call $B \subseteq S$ a *basis* of R if $\uparrow B = R$. If \preceq is also antisymmetric, then any basis of R has a unique subset of minimal elements. We call this set the *minimal basis* of R , denoted $\text{minBasis}(R)$.

Compatibility. Given a counter system $M = (S, S_0, \Delta)$, we say that a wqo $\preceq \subseteq S \times S$ is *compatible* with Δ if the following holds: $\forall s, s', r \in S$: if $s \rightarrow s'$ and $s \preceq r$ then $\exists r'$ with $s' \preceq r'$ and $r \rightarrow^* r'$. We say \preceq is *strongly compatible* with Δ if the above holds with $r \rightarrow r'$ instead of $r \rightarrow^* r'$.

WSTS [15]. We say that (M, \preceq) with $M = (S, S_0, \Delta)$ is a *well-structured transition system* if \preceq is a wqo on S that is compatible with Δ .

Lemma 1: Let $M = (S, S_0, \Delta)$ be a counter system for process templates A, B , and let $\lesssim \subseteq S \times S$ be the binary relation defined by:

$$(q_A, \mathbf{c}) \lesssim (q'_A, \mathbf{d}) \Leftrightarrow (q_A = q'_A \wedge \mathbf{c} \lesssim \mathbf{d}),$$

where \lesssim is the component-wise ordering of vectors. Then (M, \lesssim) is a WSTS.

Predecessor, Effective pred-basis [16]. Let $M = (S, S_0, \Delta)$ be a counter system and let $R \subseteq S$. Then the set of *immediate predecessors* of R is

$$\text{pred}(R) = \{s \in S \mid \exists r \in R : s \rightarrow r\}.$$

A WSTS (M, \lesssim) has *effective pred-basis* if there exists an algorithm that takes as input any finite set $R \subseteq S$ and returns a finite basis of $\uparrow \text{pred}(\uparrow R)$. Note that, since \lesssim is strongly compatible with Δ , if a set $R \subseteq S$ is upward-closed with respect to \lesssim then $\text{pred}(R)$ is also upward-closed.⁴

For backward reachability analysis, we want to compute $\text{pred}^*(R)$ as the limit of the sequence $R_0 \subseteq R_1 \subseteq \dots$ where $R_0 = R$ and $R_{i+1} = R_i \cup \text{pred}(R_i)$. Note that if we have strong compatibility and effective pred-basis, we can compute $\text{pred}^*(R)$ for any upward-closed set R . If we can furthermore check intersection of upward-closed sets with initial states (which is easy for counter systems), then reachability of arbitrary upward-closed sets is decidable.

The following lemma, like Lemma 1, can be considered folklore. We present it here mainly to show *how* we can effectively compute the predecessors, which is an important ingredient of our model checking algorithm.

Lemma 2: Let $M = (S, S_0, \Delta)$ be a counter system for guarded process templates A, B . Then (M, \lesssim) has effective pred-basis.

⁴For a formal proof, check the extended version [17].

B. Model Checking Algorithm

Our model checking algorithm is based on the known backwards reachability algorithm for WSTS [15]. We present it in detail to show how it stores intermediate results to return an *error sequence*, from which we derive concrete error paths.

Algorithm 1 Parameterized Model Checking

```

1: procedure MODELCHECK(Counter System  $M, ERR$ )
2:    $\text{tempSet} \leftarrow ERR, E_0 \leftarrow ERR, i \leftarrow 1, \text{visited} \leftarrow \emptyset$ 
   // A fixed point is reached if  $\text{visited} = \text{tempSet}$ 
3:   while  $\text{tempSet} \neq \text{visited}$  do
4:      $\text{visited} \leftarrow \text{tempSet}$ 
5:      $E_i \leftarrow \text{minBasis}(\text{pred}(\uparrow E_{i-1}))$ 
6:   //  $\text{pred}$  is computed as in the proof of Lemma 2
7:     if  $E_i \cap S_0 \neq \emptyset$  then //intersect with initial states?
8:       return  $False, \{E_0, \dots, E_i \cap S_0\}$ 
9:      $\text{tempSet} \leftarrow \text{minBasis}(\text{visited} \cup E_i)$ 
10:     $i \leftarrow i + 1$ 
11:  return  $True, \emptyset$ 

```

Given a counter system M and a finite basis ERR of the set of error states, algorithm 1 iteratively computes the set of predecessors until it reaches an initial state, or a fixed point. The procedure returns either *True*, i.e. the system is safe, or an *error sequence* E_0, \dots, E_k , where $E_0 = ERR$, $\forall 0 < i < k : E_i = \text{minBasis}(\uparrow \text{pred}(\uparrow E_{i-1}))$, and $E_k = \text{minBasis}(\uparrow \text{pred}(\uparrow E_{k-1})) \cap S_0$. That is, every E_i is the minimal basis of the states that can reach ERR in i steps.

Properties of Algorithm 1. Correctness of the algorithm follows from the correctness of the algorithm by Abdulla et al. [15], and from Lemma 2. Termination follows from the fact that a non-terminating run would produce an infinite minimal basis, which is impossible since a minimal basis is an antichain.

Example. Consider the reader-writer system in Figures 5 and 6. Suppose the error states are all states where the writer is in w while a reader is in r . In other words, the error set of the corresponding counter system M is $\uparrow E_0$ where $E_0 = \{(w, (0, 1))\}$ and $(0, 1)$ means zero reader-processes are in nr and one in r . Note that $\uparrow E_0 = \{(w, (i_0, i_1)) \mid (w, (0, 1)) \lesssim (w, (i_0, i_1))\}$, i.e. all elements with the same w , $i_0 \geq 0$ and $i_1 \geq 1$. If we run Algorithm 1 with the parameters $M, \{(w, (0, 1))\}$, we get the following error sequence: $E_0 = \{(w, (0, 1))\}$, $E_1 = \{(nw, (0, 1))\}$, $E_2 = \{(nw, (1, 0))\}$, with $E_2 \cap S_0 \neq \emptyset$, i.e., the error is reachable.

C. Deadlock Detection in Disjunctive Systems

The repair of concurrent systems is much harder than fixing monolithic systems. One of the sources of complexity is that a repair might introduce a deadlock, which is usually an unwanted behavior. In this section we show how we can detect deadlocks in disjunctive systems.

Note that a set of deadlocked states is in general not upward-closed under \lesssim (defined in Sect. III-A): let $s = (q_A, \mathbf{c}), r =$

(q_A, \mathbf{d}) be global states with $s \lesssim r$. If s is deadlocked, then $\mathbf{c}(i) = 0$ for every q_i that appears in a guard of an outgoing local transition from s . Now if $\mathbf{d}(i) > 0$ for one of these q_i , then some transition is enabled in r , which is therefore not deadlocked.

A natural idea is to refine the wqo such that deadlocked states are upward closed. To this end, consider $\lesssim_0 \subseteq \mathbb{N}_0^{|B|} \times \mathbb{N}_0^{|B|}$ where

$$\mathbf{c} \lesssim_0 \mathbf{d} \Leftrightarrow (\mathbf{c} \lesssim \mathbf{d} \wedge \forall i \leq |B| : (\mathbf{c}(i) = 0 \Leftrightarrow \mathbf{d}(i) = 0)),$$

and $\lesssim_0 \subseteq S \times S$ where $(q_A, \mathbf{c}) \lesssim_0 (q'_A, \mathbf{d}) \Leftrightarrow (q_A = q'_A \wedge \mathbf{c} \lesssim_0 \mathbf{d})$.

Then, deadlocked states are upward closed with respect to \lesssim_0 . However, it is not easy to adopt the WSTS approach to this case, since for our counter systems $\text{pred}(R)$ will in general not be upward closed if R is upward closed. Instead of using \lesssim_0 to define a WSTS, in the following we will use it to define a counter abstraction (similar to the approach of Pnueli et al. [18]) that can be used for deadlock detection.

The idea is that we use vectors with counter values from $\{0, 1\}$ to represent their upward closure with respect to \lesssim_0 . These upward closures will be seen as abstract states, and in the usual way define that a transition between abstract states \hat{s}, \hat{s}' exists iff there exists a transition between concrete states $s \in \uparrow \hat{s}, s' \in \uparrow \hat{s}'$. We formalize the abstract system in the following, assuming wlog. that δ_B does not contain transitions of the form $(q_i, \{q_i\}, q_j)$, i.e., transitions from q_i that are guarded by q_i .⁵

01-Counter System. For a given counter system M , we define the 01-Counter System $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\Delta})$, where:

- $\hat{S} \subseteq Q_A \times \{0, 1\}^{|B|}$ is the set of states,
- $\hat{s}_0 = (\text{init}_A, \mathbf{c})$ with $\mathbf{c}(q) = 1$ iff $q = \text{init}_B$ is the initial state,
- $\hat{\Delta}$ is the set of transitions $((q_A, \mathbf{c}), (q'_A, \mathbf{c}'))$ s.t. one of the following holds:
 - 1) $\mathbf{c} = \mathbf{c}' \wedge \exists (q_A, g, q'_A) \in \delta_A : (q_A, \mathbf{c}) \models_{q_A} g$ (transition of A)
 - 2) $q_A = q'_A \wedge \exists (q_i, g, q_j) \in \delta_B : (q_A, \mathbf{c}) \models_{q_i} g \wedge \mathbf{c}(i) = 1 \wedge [(\mathbf{c}(j) = 0 \wedge (\mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j \vee \mathbf{c}' = \mathbf{c} + \mathbf{u}_j)) \vee (\mathbf{c}(j) = 1 \wedge (\mathbf{c}' = \mathbf{c} - \mathbf{u}_i \vee \mathbf{c}' = \mathbf{c}))]$ (transition of a B -process)

Define runs and deadlocks of a 01-counter system similarly as for counter systems. For a state $s = (q_A, \mathbf{c})$ of M , define the corresponding abstract state of \hat{M} as $\alpha(s) = (q_A, \hat{\mathbf{c}})$ with $\hat{\mathbf{c}}(i) = 0$ if $\mathbf{c}(i) = 0$, and $\hat{\mathbf{c}} = 1$ otherwise.

Theorem 1: The 01-counter system \hat{M} has a deadlocked run if and only if the counter system M has a deadlocked run.

Proof idea: Suppose $x = s_1, s_2, \dots, s_f$ is a deadlocked run of M . Note that for any $s \in S$, a transition based on local transition $t_U \in \delta_U$ is enabled if and only if a transition based on t_U is enabled in the abstract state $\alpha(s)$ of \hat{M} . Then it is

⁵A system that does not satisfy this assumption can easily be transformed into one that does, with a linear blowup in the number of states, and preserving reachability properties including reachability of deadlocks.

easy to see that $\hat{x} = \alpha(s_1), \alpha(s_2), \dots, \alpha(s_f)$ is a deadlocked run of \hat{M} .

Now, suppose $\hat{x} = \hat{s}_1, \hat{s}_2, \dots, \hat{s}_f$ is a deadlocked run of \hat{M} . Let b be the number of transitions $(\hat{s}_k, \hat{s}_{k+1})$ based on some $t_B = (q_i, g, q_j) \in \delta_B$ with $\hat{s}_{k+1}(i) = 1$, i.e., the transitions where we keep a 1 in position i . Furthermore, let t_1, \dots, t_{f-1} be the sequence of local transitions that \hat{x} is based on. Then we can construct a deadlocked run of M in the following way: We start in $s_1 = (\text{init}_A, \mathbf{c}_1)$ with $\mathbf{c}_1(\text{init}_B) = 2^b$ and for every t_k in the sequence do:⁶

- if $t_k \in \delta_A$, we take the same transition once,
- if $t_k = (q_i, g, q_j) \in \delta_B$ with $\hat{s}_{k+1}(i) = 0$, we take the same local transition until position i becomes empty, and
- if $t_k = (q_i, g, q_j) \in \delta_B$ with $\hat{s}_{k+1}(i) = 1$, we take the same local transition $\frac{c}{2}$ times, where c is the number of processes that are in position i before (i.e., we move half of the processes to j , and keep the other half in i).

By construction, after any of the transitions in t_1, \dots, t_{f-1} , the same positions as in \hat{x} will be occupied in our constructed run, thus the same transitions are enabled. Therefore, the constructed run ends in a deadlocked state. ■

Corollary 1: Deadlock detection in disjunctive systems is decidable in EXPTIME (in $|Q_B|$).

An Algorithm for Deadlock Detection. Now we can modify the model-checking algorithm to detect deadlocks in a 01-counter system \hat{M} : instead of passing a basis of the set of errors in the parameter ERR , we pass a finite set of deadlocked states $DEAD \subseteq \hat{S}$, and predecessors can directly be computed by pred . Thus, an *error sequence* is of the form E_0, \dots, E_k , where $E_0 = DEAD$, $\forall 0 < i < k : E_i = \text{pred}(E_{i-1})$, and $E_k = E_{k-1} \cap S_0$.

IV. PARAMETERIZED REPAIR ALGORITHM

Now, we can introduce a parameterized repair algorithm that interleaves the backwards model checking algorithm (Algorithm 1) with a forward reachability analysis and the computation of candidate repairs.

Forward Reachability Analysis. In the following, for a set $R \subseteq S$, let $\text{Succ}(R) = \{s' \in S \mid \exists s \in R : s \rightarrow s'\}$. Furthermore, for $s \in S$, let $\Delta^{\text{local}}(s, R) = \{t_U \in \delta \mid t_U \in \Delta^{\text{local}}(s) \wedge \Delta(s, t_U) \in R\}$.

Given an error sequence E_0, \dots, E_k , let the *reachable error sequence* $\mathcal{RE} = RE_0, \dots, RE_k$ be defined by $RE_k = E_k$ (which by definition only contains initial states), and $RE_{i-1} = \text{Succ}(RE_i) \cap \uparrow E_{i-1}$ for $1 \leq i \leq k$. That is, each RE_i contains a set of states that can reach $\uparrow ERR$ in i steps, and are reachable from S_0 in $k - i$ steps. Thus, it represents a set of concrete error paths of length k .

Constraint Solving for Candidate Repairs. The generation of candidate repairs is guided by constraints over the local transitions δ as atomic propositions, such that a satisfying assignment of the constraints corresponds to the candidate

⁶Note that a similar, but more involved construction is also possible with $\mathbf{c}_1(\text{init}_B) = b$.

repair, where only transitions that are assigned **true** remain in δ' . During an execution of the algorithm, these constraints ensure that all error paths discovered so far will be avoided, and include a set of fixed constraints that express additional desired properties of the system, as explained in the following.

Initial Constraints. To avoid the construction of repairs that violate the totality assumption on the transition relations of the process templates, every repair for disjunctive systems has to satisfy the following constraint:

$$TRConstr_{Disj} = \bigwedge_{q_A \in Q_A} \bigvee_{t_A \in \delta_A(q_A)} t_A \wedge \bigwedge_{q_B \in Q_B} \bigvee_{t_B \in \delta_B(q_B)} t_B$$

Informally, $TRConstr_{Disj}$ guarantees that a candidate repair returned by the SAT solver never removes all local transitions of a local state in $Q_A \cup Q_B$. Furthermore a designer can add constraints that are needed to obtain a repair that conforms with their requirements, for example to ensure that certain states remain reachable in the repair (see the extended version [17] for more examples).

A Parameterized Repair Algorithm. Given a counter system M , a basis ERR of the error states, and initial Boolean constraints $initConstr$ on the transition relation (including at least $TRConstr_{Disj}$), Algorithm 2 returns either a *repair* δ' or the string *Unrealizable* to denote that no repair exists.

Properties of Algorithm 2.

Theorem 2 (Soundness): For every repair δ' returned by Algorithm 2:

- $Restrict(M, \delta')$ is safe, i.e., $\uparrow ERR$ is not reachable, and
- the transition relation of $Restrict(M, \delta')$ is total in the first two arguments.

Proof: The parameterized model checker guarantees that the transition relation is safe, i.e., $\uparrow ERR$ is not reachable. Moreover, the transition relation constraint $TRConstr$ is part of $initConstr$ and guarantees that, for any candidate repair returned by the SAT solver, the transition relation is total. ■

Theorem 3 (Completeness): If Algorithm 2 returns “Unrealizable”, then the parameterized system has no repair.

Proof: Algorithm 2 returns “Unrealizable” if $accConstr \wedge initConstr$ has become unsatisfiable. We consider an arbitrary $\delta' \subseteq \delta$ and show that it cannot be a repair. Note that for the given run of the algorithm, there is an iteration i of the loop such that δ' , seen as an assignment of truth values to atomic propositions δ , was a satisfying assignment of $accConstr \wedge initConstr$ up to this point, and is not anymore after this iteration.

If $i = 0$, i.e., δ' was never a satisfying assignment, then δ' does not satisfy $initConstr$ and can clearly not be a repair. If $i > 0$, then δ' is a satisfying assignment for $initConstr$ and all constraints added before round i , but not for the constraints $\bigwedge_{s \in RE_k} BuildConstr(s, [RE_{k-1}, \dots, RE_0])$ added in this iteration of the loop, based on a reachable error sequence $\mathcal{RE} = RE_k, \dots, RE_0$. By construction of $BuildConstr$, this means we can construct out of δ' and \mathcal{RE} a concrete error path in $Restrict(M, \delta')$, and δ' can also not be a repair. ■

Algorithm 2 Parameterized Repair

```

1: procedure PARAMREPAIR( $M, ERR, InitConstr$ )
2:    $accConstr \leftarrow InitConstr, isCorrect \leftarrow False$ 
3:   while  $isCorrect = False$  do
4:      $isCorrect, [E_0, \dots, E_k] \leftarrow MC(M, ERR)$ 
5:     if  $isCorrect = False$  then
6:        $RE_k \leftarrow E_k$  //  $E_k$  contains only initial states
7:        $RE_{k-1} \leftarrow Succ(RE_k) \cap \uparrow E_{k-1}, \dots,$ 
8:        $RE_0 \leftarrow Succ(RE_1) \cap \uparrow E_0$ 
9:     //for every initial state in  $RE_k$  compute its constraints
10:     $newConstr \leftarrow \bigwedge_{s \in RE_k} BuildConstr(s, [RE_{k-1}, \dots, RE_0])$ 
11:    //accumulate iterations' constraints
12:     $accConstr \leftarrow newConstr \wedge accConstr$ 
13:    //reset deadlock constraints
14:     $ddlockCnstr \leftarrow True$ 
15:     $\delta', isSAT \leftarrow SAT(accConstr \wedge ddlockCnstr)$ 
16:    if  $isSAT = False$  then
17:      return Unrealizable
18:      //compute a new candidate using the repair  $\delta'$ 
19:       $M = Restrict(M, \delta')$ 
20:    //if M reaches a deadlock get a new repair
21:    if  $HasDeadlock(M)$  then
22:       $ddlockCnstr \leftarrow \neg \delta' \wedge ddlockCnstr$ 
23:      jump to line 14
24:    else return  $\delta'$  //a repair is found!

1: procedure BUILDCONSTR(State  $s, \mathcal{RE}$ )
2:   //  $s$  is a state,  $\mathcal{RE}[1 : ]$  is a list obtained by removing
3:   // the first element from  $\mathcal{RE}$ 
4:   if  $\mathcal{RE}[1 : ]$  is empty then
5:     //if  $t_U \in \Delta^{local}(s)$  leads directly to error set, delete it ( $\neg t_U$ 
6:     // must set to true by the SAT solver)
7:     return  $\bigwedge_{t_U \in \Delta^{local}(s, \mathcal{RE}[0])} \neg t_U$ 
8:   else
9:     //else either delete  $t_U$  or delete outgoing transitions of the
10:    // target state of  $t_U$  recursively
11:    return  $\bigwedge_{t_U \in \Delta^{local}(s, \mathcal{RE}[0])} (\neg t_U \vee$ 
12:     $BuildConstr(\Delta(s, t_U), \mathcal{RE}[1 : ]))$ 

```

Theorem 4 (Termination): Algorithm 2 always terminates.

Proof: For a counter system based on A and B , the number of possible repairs is bounded by $2^{|\delta|}$. In every iteration of the algorithm, either the algorithm terminates, or it adds constraints that exclude at least the repair that is currently under consideration. Therefore, the algorithm will always terminate. ■

What can be done if a repair doesn't exist? If Algorithm 2 returns “unrealizable”, then there is no repair for the given input. To still obtain a repair, a designer can add more non-determinism and/or allow for more communication between processes, and then run the algorithm again on the new instance of the system. Moreover, unlike in monolithic systems, even if the result is “unrealizable”, it may still be possible to obtain a solution that is good enough in practice. For instance,

we can change our algorithm slightly as follows: When the SAT solver returns “UNSAT” after adding the constraints for an error sequence, instead of terminating we can continue computing the error sequence until a fixed point is reached. Then, we can determine the minimal number of processes m_e that is needed for the last candidate repair to reach an error, and conclude that this candidate is safe for any system up to size $m_e - 1$.

V. EXTENSIONS

A. Beyond Reachability

Algorithm 2 can also be used for repair with respect to general safety properties, based on the automata-theoretic approach to model checking. We assume that the reader is familiar with finite-state automaton and with the automata-theoretic approach to model checking.

Checking Safety Properties. Let $M = (S, S_0, \Delta)$ be a counter system of process templates A and B that violates a safety property φ over the states of A , and let $\mathcal{A} = (Q^A, q_0^A, Q_A, \delta^A, \mathcal{F})$ be the automaton that accepts all words over Q_A that violate φ . To repair M , the composition $M \times \mathcal{A}$ and the set of error states $ERR = \{((q_A, \mathbf{c}), q_{\mathcal{F}}^A) \mid (q_A, \mathbf{c}) \in S \wedge q_{\mathcal{F}}^A \in \mathcal{F}\}$ can be given as inputs to the procedure *ParamRepair*.

Corollary 1: Let $\lesssim_{\mathcal{A}} \subseteq (M \times \mathcal{A}) \times (M \times \mathcal{A})$ be a binary relation defined by:

$$((q_A, \mathbf{c}), q^A) \lesssim_{\mathcal{A}} ((q'_A, \mathbf{c}'), q'^A) \Leftrightarrow \mathbf{c} \lesssim \mathbf{c}' \wedge q_A = q'_A \wedge q^A = q'^A$$

then $((M \times \mathcal{A}), \lesssim_{\mathcal{A}})$ is a WSTS with effective *pred*-basis. Similarly, the algorithm can be used for any safety property $\varphi(A, B^{(k)})$ over the states of A , and of k B -processes. To this end, we consider the composition $M \times B^k \times \mathcal{A}$ with $M = (S, S_0, \Delta)$, $B = (Q_B, \text{init}_B, \mathcal{G}_B, \delta_B)$, and $\mathcal{A} = (Q^A, q_0^A, Q_A \times Q_{B^k}, \delta^A, \mathcal{F})$ is the automaton that reads states of $A \times B^k$ as actions and accepts all words that violate the property.⁷

Example. Consider again the simple reader-writer system in Figures 5 and 6 where we use the following abbreviations: $(n)w$ for (non-)writing, and $(n)r$ for (non-)reading. Assume that instead of local transition $(nr, \{nw\}, r)$ we have an unguarded transition (nr, Q, r) . We want to repair the system with respect to the safety property $\varphi = G[(w \wedge nr_1) \Longrightarrow (nr_1 Wnw)]$ where G, W are the temporal operators *always* and *weak until*, respectively. Figure 7 depicts the automaton equivalent to $\neg\varphi$. To repair the system we first need to split the guards as mentioned in Section II, i.e., (nr, Q, r) is split into $(nr, \{nr\}, r)$, $(nr, \{r\}, r)$, $(nr, \{nw\}, r)$, and $(nr, \{w\}, r)$. Then we consider the composition $\mathcal{C} = M \times B \times \mathcal{A}$ and we run Algorithm 2 on the parameters \mathcal{C} , $((-, -, (*, *), q_2^A))$ (where $(-, -)$ means any writer state and any reader state, and $*$ means 0 or 1), and $TRC\text{onstr}_{Disj}$. The model checker

⁷By symmetry, property $\varphi(A, B^{(k)})$ can be violated by these k explicitly modeled processes iff it can be violated by any combination of k processes in the system.

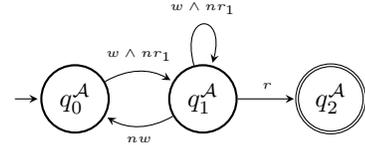


Fig. 7: Automaton for $\neg\varphi$

in Line 4 may return the following error sequences, where we only consider states that didn't occur before:

$$\begin{aligned} E_0 &= \{((-, -, (*, *), q_2^A)\}, \\ E_1 &= \{((w, r_1, (0, 0)), q_1^A)\}, \\ E_2 &= \{((w, nr_1, (0, 0)), q_0^A), ((w, nr_1, (0, 1)), q_0^A), \\ &\quad ((w, nr_1, (1, 0)), q_0^A)\}, \\ E_3 &= \{((nw, nr_1, (0, 0)), q_0^A), ((nw, nr_1, (0, 1)), q_0^A), \\ &\quad ((w, r_1, (0, 0)), q_0^A), ((w, r_1, (0, 1)), q_0^A), ((w, r_1, (1, 0)), q_0^A)\} \end{aligned}$$

In Line 14 we find out that the error sequence can be avoided if we remove the transitions $\{(nr, \{nr\}, r), (nr, \{r\}, r), (nr, \{w\}, r)\}$. Another call to the model checker in Line 4 finally assures that the new system is safe. Note that some states were omitted from error sequences in order to keep the presentation simple.

B. Beyond Disjunctive Systems

Furthermore, we have extended Algorithm 2 to other systems that can be framed as WSTS, in particular pairwise systems [2] and systems based on broadcasts or other global synchronizations [3], [19]. We summarize our results here, more details can be found in the extended version [17].

Both types of systems are known to be WSTS, and there are two remaining challenges:

- 1) how to find suitable constraints to determine a restriction δ' , and
- 2) how to exclude deadlocks.

The first is relatively easy, but the constraints become more complicated because we now have synchronous transitions of multiple processes. Deadlock detection is decidable for pairwise systems, but the best known method is by reduction to reachability in VASS [2], which has recently been shown to be TOWER-hard [20]. For broadcast protocols we can show that the situation is even worse:

Theorem 5: Deadlock detection in broadcast protocols is undecidable.

The main ingredient of the proof is the following lemma:

Lemma 3: There is a polynomial-time reduction from the reachability problem of affine VASS with broadcast matrices to the deadlock detection problem in broadcast protocols.

Proof: We modify the construction from the proofs of Theorems 3.17 and 3.18 from German and Sistla [2], using affine VASS instead of VASS and broadcast protocols instead of pairwise rendezvous systems.

Starting from an arbitrary affine VASS G that only uses broadcast matrices and where we want to check if configuration (q_2, \mathbf{c}_2) is reachable from (q_1, \mathbf{c}_1) , we first transform it to an affine VASS G^* with the following properties

- each transition only changes the vector \mathbf{c} in one of the following ways: (i) it adds to or subtracts from \mathbf{c} a unit vector, or (ii) it multiplies \mathbf{c} with a broadcast matrix M (this allows us to simulate every transition with a single transition in the broadcast system), and
- some configuration $(q'_2, 0)$ is reachable from some configuration $(q'_1, 0)$ in G^* if and only if (q_2, \mathbf{c}_2) is reachable from (q_1, \mathbf{c}_1) in G .

The transformation is straightforward by splitting more complex transitions and adding auxiliary states. Now, based on G^* we define process templates A and B such that $A\|B^n$ can reach a deadlock iff $(q'_2, 0)$ is reachable from $(q'_1, 0)$ in G^* .

The states of A are the discrete states of G^* , plus additional states q', q'' . If the state vector of G^* is m -dimensional, then B has states q_1, \dots, q_m , plus states init, v . Then, corresponding to every transition in G^* that changes the state from q to q' and either adds or subtracts unit vector \mathbf{u}_i , we have a rendezvous sending transition from q to q' in A , and a corresponding receiving transition in B from init to q_i (if \mathbf{u}_i was added), or from q_i to init (if \mathbf{u}_i was subtracted). For every transition that changes the state from q to q' and multiplies \mathbf{c} with a matrix M , A has a broadcast sending transition from q to q' , and receiving transitions between the states q_1, \dots, q_m that correspond to the effect of M .

The additional states q', q'' of A are used to connect reachability of $(q'_2, 0)$ to a deadlock in $A\|B^n$ in the following way: (i) there are self-loops on all states of A except on q' , i.e., the system can only deadlock if A is in q' , (ii) there is a broadcast sending transition from q'_2 to q' in A , which sends all B -processes that are in q_1, \dots, q_m to special state v , and (iii) from v there is a broadcast sending transition to init in B , and a corresponding receiving transition from q' to q'' in A . Thus, $A\|B^n$ can only deadlock in a configuration where A is in q' and there are no B -processes in v , which is only reachable through a transition from a configuration where A is in q_2 and no B -processes are in q_1, \dots, q_m . Letting q_1 be the initial state of A and init the initial state of B , such a configuration is reachable in $A\|B^n$ if and only if $(q'_2, 0)$ is reachable from $(q'_1, 0)$ in G^* . ■

Approximate Methods for Deadlock Detection. Since solving the problem exactly is impractical or impossible in general, we propose to use approximate methods. For pairwise systems, the 01-counter system introduced as a precise abstraction for disjunctive systems in Sect. III-C can also be used, but in this case it is not precise, i.e., it may produce spurious deadlocked runs. Another possible overapproximation is a system that simulates pairwise transitions by a pair of disjunctive transitions. For broadcast protocols we can use lossy broadcast systems, for which the problem is decidable [21].⁸ Another alternative is to add initial constraints that restrict the repair algorithm and imply deadlock-freedom.

⁸Note that in the terminology of Delzanno et al., deadlock detection is a special case of the TARGET problem.

VI. IMPLEMENTATION & EVALUATION

We have implemented a prototype of our parameterized repair algorithm that supports the three types of systems (disjunctive, pairwise and broadcast), and safety and reachability properties. For disjunctive and pairwise systems, we have evaluated it on different variants of reader-writer-protocols, based on the ones given in Sect. I.II, where we replicated some of the states and transitions to test the performance of our algorithm on bigger benchmarks. For disjunctive systems, all variants have been repaired successfully in less than 2s. For pairwise systems, these benchmarks are denoted “RW*i* (PR)” in Table I. A detailed treatment of one benchmark, including an explanation of the whole repair process is given in the extended version [17].

For broadcast protocols, we have evaluated our algorithm on a range of more complex benchmarks taken from the parameterized verification literature [22]: a distributed **Lock Service** (DLS) inspired by the Chubby protocol [23], a distributed **Robot Flocking** protocol (RF) [24], a distributed **Smoke Detector** (SD) [19], a sensor network implementing a **Two-Object Tracker** (TOT) [25], and the cache coherence protocol **MESI** [26] in different variants constructed similar as for RW.

Typical desired safety properties are mutual exclusion and similar properties. Since deadlock detection is undecidable for broadcast protocols, the absence of deadlocks needs to be ensured with additional initial constraints.

On all benchmarks, we compare the performance of our algorithm based on the valuations of two flags: SEP and EPT. The SEP (“single error path”) flag indicates that, instead of encoding all the model checker’s computed error paths, only one path is picked and encoded for SAT solving. When the EPT (“error path transitions”) flag is raised the SAT formula is constructed so that only transitions on the extracted error paths may be suggested for removal. Note that in the default case, even transitions that are unrelated to the error may be removed. Table I summarizes the experimental results we obtained.

We note that the algorithm deletes fewer transitions when the EPT flag is raised (EPT=T). This is because we tell the SAT solver explicitly not to delete transitions that are not on the error paths. Removing fewer transitions might be desirable in some applications. We observe the best performance when the SEP flag is set to true (SEP=T) and the EPT flag is false. This is because the constructed SAT formulas are much simpler and the SAT solver has more freedom in deleting transitions, resulting in a small number of iterations.

VII. RELATED WORK

Many automatic repair approaches have been considered in the literature, most of them restricted to monolithic systems [11], [12], [27]–[30]. Additionally, there are several approaches for synchronization synthesis and repair of *concurrent systems*. Some of them differ from ours in the underlying approach, e.g., being based on automata-theoretic synthesis [31], [32]. Others are based on a similar underlying counterexample-guided synthesis/repair principle, but differ in

TABLE I: Running time, number of iterations, and number of deleted transitions (#D.T.) for the different configurations. Each benchmark is listed with its number of local states, and edges. We evaluated the algorithms on different sets of errors with $P_1 \cup P_2 = C$ where P_1 and P_2 are two distinct error sets that differ from one benchmark to another. Smallest number of iterations, runtime per benchmark, deleted transitions are highlighted in boldface.

Benchmark	Size		Errors	[SEP=F & EPT=F]			[SEP=T & EPT=F]			[SEP=F & EPT=T]			[SEP=T & EPT=T]		
	States	Edges		#Iter	Time	#D.T.	#Iter	Time	#D.T.	#Iter	Time	#D.T.	#Iter	Time	#D.T.
RW1 (PW)	5	12	C	3	2.5	4	3	2.9	4	2	1.7	2	2	1.7	2
RW2 (PW)	15	42	C	3	3.8	14	3	4.8	14	2	3.2	7	7	8.4	7
RW3 (PW)	35	102	C	3	820.7	34	3	7.6	34	2	552.3	17	17	40.3	17
RW4 (PW)	45	132	C	TO	TO	TO	3	11.8	44	TO	TO	TO	22	99.2	22
DLS	10	95	P1	1	0.8	13	1	0.8	13	3	2.4	5	5	5.6	5
DLS	10	95	P2	1	0.8	13	2	1.7	13	3	2.6	9	7	5.5	9
DLS	10	95	C	2	4.2	13	2	1.5	13	3	3	9	9	8.1	9
RF	10	147	P1	1	2.5	32	1	1.2	32	TO	TO	TO	8	12.4	13
RF	10	147	P2	1	1.2	32	1	1.3	32	TO	TO	TO	8	11.3	14
RF	10	147	C	1	7.8	32	1	1.4	32	TO	TO	TO	8	12.5	12
SD	6	39	C	1	1	4	1	1	4	3	2.4	4	3	3	4
2OT	12	128	P1	12	18.8	26	6	8.3	26	16	73.8	17	16	34	17
2OT	12	128	P2	1	1.8	26	1	1.8	26	4	2958	11	8	16.5	12
2OT	12	128	C	11	17.2	Unreal.	6	11.7	Unreal.	TO	TO	TO	11	48.6	Unreal.
MESI1	4	26	C	1	2.4	6	1	0.9	6	2	1.8	5	4	3.5	5
MESI2	9	71	C	1	1.1	26	1	1.1	26	3	56.4	20	6	6.8	15
MESI3	14	116	C	1	109.4	46	1	108.1	46	TO	TO	TO	6	289.9	15

other aspects from ours. For instance, there are approaches that repair the program by adding atomic sections, which forbid the interruption of a sequence of program statements by other processes [13], [33]. *Assume-Guarantee-Repair* [34] combines verification and repair, and uses a learning-based algorithm to find counterexamples and restrict transition guards to avoid errors. In contrast to ours, this algorithm is not guaranteed to terminate. From *lazy synthesis* [35] we borrow the idea to construct the set of *all* error paths of a given length instead of a single concrete error path, but this approach only supports systems with a fixed number of components. Some of these existing approaches are more general than ours in that they support certain infinite-state processes [13], [33], [34], or more expressive specifications and other features like partial information [31], [32].

The most important difference between our approach and all of the existing repair approaches is that, to the best of our knowledge, none of them provide correctness guarantees for systems with a parametric number of components. This includes also the approach of McClurg et al. [14] for the synthesis of synchronizations in a software-defined network. Although they use a variant of Petri nets as a system model, which would be suitable to express parameterized systems, their restrictions are such that the approach is restricted to a fixed number of components. In contrast, we include a parameterized model checker in our repair algorithm, and can therefore provide parameterized correctness guarantees. There exists a wealth of results on parameterized model checking, collected in several good surveys recently [36]–[38].

VIII. CONCLUSION AND FUTURE WORK

We have investigated the parameterized repair problem for systems of the form $A \parallel B^n$ with an arbitrary $n \in \mathbb{N}$. We introduced a general parameterized repair algorithm, based on interleaving the generation of candidate repairs with parameterized

model checking and deadlock detection, and instantiated this approach to different classes of systems that can be modeled as WSTS: disjunctive systems, pairwise rendezvous systems, and broadcast protocols.

Since deadlock detection is an important part of our method, we investigated this problem in detail for these classes of systems, and found that the problem can be decided in EXPTIME for disjunctive systems, and is undecidable for broadcast protocols.

Besides reachability properties and the absence of deadlocks, our algorithm can guarantee general safety properties, based on the automata-theoretic approach to model checking. On a prototype implementation of our algorithm, we have shown that it can effectively repair non-deterministic overapproximations of many examples from the literature. Moreover, we have evaluated the impact of different heuristics or design choices on the performance of our algorithm in terms of repair time, number of iterations, and number of deleted transitions.

A limitation of the current algorithm is that it cannot guarantee any *liveness properties*, like termination or the absence of undesired loops. Also, it cannot automatically *add behavior* (states, transitions, or synchronization options) to the system, in case the repair for the given input is unrealizable. We consider these as important avenues for future work. Moreover, in order to improve the practicality of our approach we want to examine the inclusion of symbolic techniques for counter abstraction [39], and advanced parameterized model checking techniques, e.g., *cutoff* results for disjunctive systems [6], [40], [41], or recent *pruning* results for immediate observation Petri nets, which model exactly the class of disjunctive systems [42].

REFERENCES

- [1] I. Suzuki, "Proving properties of a ring of finite state machines," *Inf. Process. Lett.*, vol. 28, no. 4, pp. 213–214, 1988.

- [2] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, no. 3, pp. 675–735, 1992.
- [3] J. Esparza, A. Finkel, and R. Mayr, "On the verification of broadcast protocols," in *LICS*. IEEE Computer Society, 1999, pp. 352–359.
- [4] E. A. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *CADE*, ser. LNCS, vol. 1831. Springer, 2000, pp. 236–254.
- [5] E. A. Emerson and K. S. Namjoshi, "On reasoning about rings," *Foundations of Computer Science*, vol. 14, no. 4, pp. 527–549, 2003.
- [6] E. A. Emerson and V. Kahlon, "Model checking guarded protocols," in *LICS*. IEEE Computer Society, 2003, pp. 361–370.
- [7] E. M. Clarke, M. Talupur, T. Touili, and H. Veith, "Verification by network decomposition," in *CONCUR*, ser. LNCS, vol. 3170. Springer, 2004, pp. 276–291.
- [8] B. Aminof, S. Jacobs, A. Khalimov, and S. Rubin, "Parameterized model checking of token-passing systems," in *VMCAI*, ser. LNCS, vol. 8318. Springer, 2014, pp. 262–281.
- [9] B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith, "Parameterized model checking of rendezvous systems," in *CONCUR*, ser. LNCS, vol. 8704. Springer, 2014, pp. 109–124.
- [10] B. Aminof and S. Rubin, "Model checking parameterised multi-token systems via the composition method," in *IJCAR*, ser. LNCS, vol. 9706. Springer, 2016, pp. 499–515.
- [11] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *17th Conference on Computer Aided Verification (CAV'05)*. Springer, 2005, pp. 226–238, INCS 3576.
- [12] P. C. Attie, K. D. A. Bab, and M. Sakr, "Model and program repair via sat solving," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, pp. 1–25, 2017.
- [13] R. Bloem, G. Hofferek, B. Könighofer, R. Könighofer, S. Außerlechner, and R. Spörk, "Synthesis of synchronization using uninterpreted functions," in *2014 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2014, pp. 35–42.
- [14] J. McClurg, H. Hojjat, and P. Černý, "Synchronization synthesis for network programs," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 301–321.
- [15] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1996, pp. 313–321.
- [16] A. Finkel and P. Schnoebelen, "Well-structured transition systems everywhere!" *Theoretical Computer Science*, vol. 256, no. 1-2, pp. 63–92, 2001.
- [17] S. Jacobs, M. Sakr, and M. Völpl, "Parameterized repair of concurrent systems," 2021. [Online]. Available: <https://doi.org/10.48550/arXiv.2111.03322>
- [18] A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infity)-counter abstraction," in *CAV*, ser. Lecture Notes in Computer Science, vol. 2404. Springer, 2002, pp. 107–122.
- [19] N. Jaber, S. Jacobs, C. Wagner, M. Kulkarni, and R. Samanta, "Parameterized verification of systems with global synchronization and guards," in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 12224. Springer, 2020, pp. 299–323.
- [20] W. Czerwinski, S. Lasota, R. Lazic, J. Leroux, and F. Mazowiecki, "The reachability problem for petri nets is not elementary," *J. ACM*, vol. 68, no. 1, pp. 7:1–7:28, 2021.
- [21] G. Delzanno, A. Sangnier, and G. Zavattaro, "Parameterized verification of ad hoc networks," in *CONCUR*, ser. LNCS, vol. 6269. Springer, 2010, pp. 313–327.
- [22] N. Jaber, C. Wagner, S. Jacobs, M. Kulkarni, and R. Samanta, "Quicksilver: modeling and parameterized verification for distributed agreement-based systems," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–31, 2021.
- [23] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*. USENIX Association, 2006, pp. 335–350.
- [24] D. Canepa and M. G. Potop-Butucaru, "Stabilizing flocking via leader election in robot networks," in *SSS*, ser. Lecture Notes in Computer Science, vol. 4838. Springer, 2007, pp. 52–66.
- [25] C. Chang and J. Tsai, "Distributed collaborative surveillance system based on leader election protocols," *IET Wirel. Sens. Syst.*, vol. 6, no. 6, pp. 198–205, 2016.
- [26] E. A. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *CHARME*, ser. LNCS, vol. 2860. Springer, 2003, pp. 247–262.
- [27] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, 2003, pp. 78–95.
- [28] A. Griesmayer, R. Bloem, and B. Cook, "Repair of Boolean programs with an application to C," in *18th Conference on Computer Aided Verification (CAV'06)*, 2006, pp. 358–371, LNCS 4144.
- [29] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conference (GECCO'09)*. ACM, 2009, pp. 947–954.
- [30] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, 2018.
- [31] B. Finkbeiner and S. Schewe, "Bounded synthesis," *STTT*, vol. 15, no. 5-6, pp. 519–539, 2013.
- [32] S. Bansal, K. S. Namjoshi, and Y. Sa'ar, "Synthesis of coordination programs from linear temporal specifications," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 54:1–54:27, 2020. [Online]. Available: <https://doi.org/10.1145/3371122>
- [33] M. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2010, pp. 327–338.
- [34] H. Frenkel, O. Grumberg, C. Pasareanu, and S. Sheinvald, "Assume, guarantee or repair," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 211–227.
- [35] B. Finkbeiner and S. Jacobs, "Lazy synthesis," in *VMCAI*, ser. LNCS, vol. 7148. Springer, 2012, pp. 219–234.
- [36] J. Esparza, "Keeping a crowd safe: On the complexity of parameterized verification (invited talk)," in *STACS*, ser. LIPIcs, vol. 25. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014, pp. 1–10.
- [37] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder, *Decidability of Parameterized Verification*, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
- [38] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018, vol. 10.
- [39] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening, "Symbolic counter abstraction for concurrent software," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 64–78.
- [40] S. Außerlechner, S. Jacobs, and A. Khalimov, "Tight cutoffs for guarded protocols with fairness," in *VMCAI*, ser. LNCS, vol. 9583. Springer, 2016, pp. 476–494.
- [41] S. Jacobs and M. Sakr, "Analyzing guarded protocols: Better cutoffs, more systems, more expressivity," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2018, pp. 247–268.
- [42] J. Esparza, M. A. Raskin, and C. Weil-Kennedy, "Parameterized analysis of immediate observation petri nets," in *Petri Nets*, ser. Lecture Notes in Computer Science, vol. 11522. Springer, 2019, pp. 365–385.

Synthesizing Locally Symmetric Parameterized Protocols from Temporal Specifications

Ruoxi Zhang
University of Waterloo
 Waterloo, Canada
 r378zhan@uwaterloo.ca

Richard Trefler
University of Waterloo
 Waterloo, Canada
 trefler@uwaterloo.ca

Kedar S. Namjoshi
Nokia Bell Labs
 Murray Hill, USA
 kedar.namjoshi@nokia-bell-labs.com

Abstract—Scalable protocols and web services are typically parameterized: that is, each instance of the system is formed by linking together isomorphic copies of a representative process. Verification of such systems is difficult due to state explosion for large instances and the undecidability of verifying properties over all instances at once. This work turns instead to the derivation of a parameterized protocol from its specification. We exploit a reduction theorem showing that it suffices to construct a representative process P that meets a local specification under interference by neighboring copies of P . Every instance of the parameterized protocol is built by deploying replicated instances of P . While the reduction from the original to a local specification is done by hand, the construction of P is fully automated. This is a new and challenging synthesis question, as one must synthesize an unknown process P while simultaneously considering interference by copies of this unknown process. We present two algorithms: an eager reduction to the synthesis of a transformed specification, and a lazy, iterative, tableau construction which incorporates fresh interference at each step. The tableau method has worst-case complexity that is exponential in the length of the local specification. We have implemented the tableau construction and show that it is capable of synthesizing parameterized protocols for mutual exclusion, leader election, and dining philosophers.

I. INTRODUCTION

Scalable systems, such as network communication protocols, distributed algorithms, and multi-core hardware models, are typically parameterized – that is, they are composed of many isomorphic copies of a representative process. These processes interact with each other according to an underlying communication scheme. Automated verification of such systems quickly runs into state explosion with increasing instance size, as an instance with K processes can have a reachable state space that is exponential in K . The alternative of “once and for all” verification of all instances at once is undecidable in general [1].

In this work, we turn instead to the construction (synthesis) of a parameterized system from its specification. The key to the presented methodology is a compositional (i.e. assume-guarantee) reduction theorem from [2] which exploits the symmetry inherent in these systems, showing that it suffices to verify that a localized property holds of a representative process P under interference from neighboring copies of P . The first step of the methodology is to reduce the global specification of the desired parameterized system to a localized property. This reduction varies by application, as the global

specification is itself parameterized and quantified (e.g., “all instances satisfy mutual exclusion”) while the local specification is quantifier-free. The second step is to synthesize an appropriate process P from the local specification, which is carried out automatically.

This synthesis question is of a new and challenging type. The standard formulation of temporal synthesis is to construct a process P satisfying a given temporal specification φ . However, our reduction requires the construction of a process P whose closure under interference by copies of (the unknown) P satisfies a temporal specification φ . That is, the synthesis procedure must somehow derive a suitable process while simultaneously taking into account the effects of interference by adjacent copies of this unknown process. Every instance of the protocol is built by deploying replicated instances of the synthesized P .

We provide two algorithms for the synthesis question. The first is an ‘eager’ method that transforms a given specification φ to a new specification $\mathcal{I}(\varphi)$ which incorporates self-interference; one can then apply standard synthesis methods to $\mathcal{I}(\varphi)$. The second is a ‘lazy’ method which iteratively constructs a sequence of tableaux starting with a tableau for φ ; at each iteration, the current tableau is extended with interference transitions. The limit tableau is then pruned to obtain the solution. Although the eager method is direct, the transformation from φ to $\mathcal{I}(\varphi)$ always incurs an exponential blowup in the number of proposition symbols in φ . For this reason, we implement the lazy method and show that it can synthesize solutions for mutual exclusion, leader election, and dining philosophers specifications.

This approach does not provide a complete solution to the parameterized synthesis question, for several reasons. The first is that the reduction from a quantified global specification to an unquantified local specification is carried out by hand. The second is that the process P to be derived can only have a fixed-size neighborhood, as otherwise one would require an unbounded quantification over the neighbors of P . Hence, the method can derive solutions for rings, tori, wrap-around mesh, and other networks where the degree of a node is independent of the number of nodes in an instance. (The use of localized abstractions, e.g., [3], may help bypass this limitation; we plan to investigate this in future work.) Finally, both algorithms produce a process P where any two

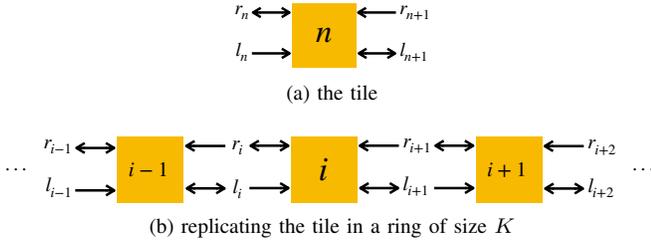


Fig. 1. The tile of the dining philosophers protocol.

states that satisfy the same propositions have identical future behavior. This rules out the synthesis of auxiliary state beyond that defined by propositional valuations. Nonetheless, despite these limitations, one can synthesize correct-by-construction parameterized protocols for the specifications listed above.

In the sequel, for ease of exposition, we limit attention to parameterized protocols on ring networks. Local symmetry ensures that a single representative process suffices. This process has two neighbors, one to the left and one to the right. Specifications are expressed in CTL, augmented with unconditional fairness on process schedules. The tableau method builds on the classical tableau constructions for CTL and Fair CTL and their associated synthesis procedures based on pruning states and transitions from the tableau.

It is worth noting that while the construction of a single instance with a fixed number of processes is a closed synthesis question, the derivation of a representative for all instances is an open synthesis question.

II. PRELIMINARIES

A. Rings: Structure, Semantics, and Interference

A *ring* of size K is a directed graph with node set $N = [0..K)$, and edge set $\{E_i\}$ for $i \in [0..K)$. Node i is connected to edges E_i (on its left) and $E_{(i+1)}$ (on its right). (Arithmetic is implicitly modulo K .) Edge E_i is connected to nodes $(i-1)$ (on its left) and i (on its right). Two nodes are *neighbors* if they have a common connected edge. The set of neighbors of node i is denoted $nbr(i)$.

The parameterized networks of interest are *uniform* rings of arbitrary size, in that the process at each node is a copy of a single ‘tile’ process (cf. [2]). Figure 1 shows a tile and the construction of an instance through replication. The external variables of a process are those assigned to adjacent edges. (In the figure, an incoming arrow represents read access; an outgoing arrow represents write access.) A process may also have internal state variables assigned to the node.

For readability, we denote the representative process by P_n , so we can speak of its neighbors as P_{n-1} and P_{n+1} (and either of them as P_m). It is important that P_n is not viewed as the n ’th process in a particular instance but rather as the representative process for all instances.

The external and internal variables of P_n together form the state space of P_n , which is the collection of valuations to these variables. The state machine for P_n is a tuple $(S_n, S_n^0, T_n, \lambda_n)$, where S_n is the state space; S_n^0 is a non-empty set of initial

states; $T_n \subseteq S_n \times S_n$ is a transition relation; and $\lambda_n : S_n \rightarrow 2^{\Sigma_n}$ is a function that labels each state with a subset of atomic propositions from the set Σ_n .

As defined, the state machine of P_n is a labeled state transition system that describes the behavior of the representative process alone in its neighborhood. A neighbor can interfere with P_n by changing the values of commonly shared (necessarily external) variables. A *joint state* is a pair of states (s, t) , with s from P_n and t from a neighbor P_m that agree on the valuation to their shared variables. A joint transition from joint state (s, t) to joint state (s', t') by process P_m is defined if (t, t') is in T_m and the values of variables of P_n that are not shared with P_m are equal in s and s' . We say that (s, s') is an *interference transition* caused by P_m . For example, ‘ P_m passes a token to P_n ’ is an interference transition.

We denote the i ’th copy of the representative process P_n in an instance by P_i . The K process instance formed by copies $P_0 \dots P_{K-1}$ has the global state transition relation $G = (S, S^0, T, \lambda)$. Here each state $s \in S$ is a valuation to the internal variables of each process, together with a valuation to the external edge variables; S^0 is a non-empty set of initial states, where each state in S^0 projects to an initial state of P_i , for all i . The transition relation T defines non-deterministic interleaving: (s, i, s') is in T if $(s[i], s'[i])$ is in T_i and the value of any variables not in process P_i is the same in s and s' . Here, the notation $s[i]$ represents the projection of s on the variables of P_i . The labeling λ of a state s is the indexed union of all local labelings $\lambda_i(s[i])$.

From G one can define a machine G_i by projecting out the labels of transitions other than those of the i ’th process. I.e., consider a transition (s, k, s') of G . If $k = i$, retain the transition as is; otherwise, replace the label with τ .

The effect of interference on P_n is given by a transition system H_n^θ defined in [2]; we repeat the definition here. A compositional inductive invariant θ of an instance is a set of local assertions $\{\theta_n\}$ with the following properties: for every n , (1) θ_n includes the initial states of P_n ; (2) transitions by P_n preserve θ_n ; and (3) interference transitions by P_m from joint states satisfying θ_n and θ_m preserve θ_n . These properties can be converted to simultaneous pre-fixpoint form over $\{\theta_n\}$. By the Knaster-Tarski theorem, the least fixpoint is the strongest compositional invariant, denoted by θ^* .

States of H_n^θ are the local states S_n that satisfy θ_n ; transitions of H_n^θ are of two types: (1) a transition by P_n , denoted (s, n, s') , where $\theta_n(s)$ holds and (s, s') is in T_n , and (2) an interference transition denoted (s, m, s') representing a transition by P_m from a joint state (s, t) where $\theta_n(s)$ and $\theta_m(t)$ hold, to a joint state (s', t') .

This transition system is linked to the global transition system with respect to local properties.

Theorem II.1. ([2]) H_i^θ stuttering-simulates G_i for every i . Moreover, if H_i^θ satisfies an ‘outward-facing’ restriction, then H_i^θ and G_i are stuttering-bisimilar.

The systems are equivalent only up to stuttering as H_n^θ does not take into account transitions by processes ‘far away’

from position i , while G of course contains all transitions. The outward-facing restriction says (informally) that the interference by a neighboring process m depends only on the valuation of the variables shared by P_n and P_m .

The transition system $H_n^{\theta^*}$ induced by the strongest compositional invariant θ^* is of special interest; we abbreviate it as H_n^* . It is constructed by an inductive, least fixpoint process. (1) The initial structure H_n consists of the initial states of P_n . Apply steps (2) or (3) in any fairly interleaved order until no new transitions can be added; the result is H_n^* . Step (2) applies an enabled transition of P_n to a reachable state of H_n , labeling it by n . Step (3) views the currently reachable n -transitions of H_n as transitions from its (isomorphic) neighboring copy H_m and adds an enabled interference transition to a reachable state of H_n , labeling it by m .

B. Local Fair CTL

Let the scheduling of the process network be unconditionally fair. We use fair computation tree logic (Fair CTL) [4] to represent a local correctness property φ_n , e.g., ‘ P_n accesses the shared resource if P_n owns the token’. The induced parametric global correctness property is the conjunction $\bigwedge_i \varphi_i$.

Syntax. The language of Fair CTL contains Σ_n , Boolean operators $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$, linear time temporal operators X_n (process indexed strong next-time), Y_n (process indexed weak next-time), G (always), F (sometime), U (until), W (dual of until), and path quantifiers A (for all paths), E (there exists a path).

We have the following syntax for Fair CTL. If $p \in \Sigma_n$, then p is a formula. If f, g are formulae, then so are $\neg f, f \wedge g, f \vee g, f \Rightarrow g, f \Leftrightarrow g, AY_n f, EX_n f, AGf, EGf, AFf, EFf, AfUg, EfUg, AfWg, \text{ and } EfWg$.

As given in [5], we use indexed next-time operators X_n and Y_n in place of the unindexed ones, where $X_n f$ means that the immediate successor state s' (along any maximal path designated by a path quantifier) is reached by executing one step of P_n , and f is true in s' ; and $Y_n f$ means that if the immediate successor state s' (along any maximal path designated by a path quantifier) is reached by executing one step of P_n , then f is true in s' .

Globally, unconditionally fair scheduling asserts that all processes are selected for execution infinitely often by the scheduler. Locally, the fairness assumption is expressed as ‘ P_n and its neighbors are executed infinitely often’. The path quantifiers A and E in Fair CTL are subscripted by the fixed local fairness assumption, Φ , indicating that quantifications are performed only on fair paths.

In Fair CTL, a path quantifier is followed by a linear-time temporal operator. The pairs are the basic modalities. A formula whose basic modality is $A_\Phi U, E_\Phi U, A_\Phi F, E_\Phi F$, or $E_\Phi G$ is an *eventuality* formula corresponding to a liveness property. Formulae $A_\Phi G$ are invariants corresponding to safety properties. In addition, we assume all formulae are converted into positive normal form, which means the negations are driven inwards to atomic propositions.

Semantics. A local Fair CTL formula φ_n is interpreted on the local state transition system H_n^* and the global state transition system G_n . Let $M = (S, S^0, T, \lambda)$ be a structure. A path, $\pi = (s_0, s_1, \dots)$, is a sequence of states such that $(s_i, s_{i+1}) \in T$ for all i , and $\pi^j = (\pi_j, \pi_{j+1}, \dots)$ is the suffix of π starting at state π_j . A full path is an infinite path, and self-loops are allowed. A full path is fair iff it satisfies Φ .

We use $M, s \models_\Phi f$ to mean that the formula f is true in M at state s under the fairness assumption Φ . We define \models_Φ inductively as follows:

- $M, s \models_\Phi p$ iff $p \in \lambda(s)$ for atomic proposition p .
- $M, s \models_\Phi \neg f$ iff not $(M, s \models_\Phi f)$.
- $M, s \models_\Phi f \wedge g$ iff $M, s \models_\Phi f$ and $M, s \models_\Phi g$.
- $M, s_0 \models_\Phi E_\Phi X_n f$ iff there exists $\pi = (s_0, s_1, \dots)$, such that $(s_0, s_1) \in T_n, M, \pi \models \Phi$, and $M, s_1 \models_\Phi f$.
- $M, s_0 \models_\Phi A_\Phi Y_n f$ iff for all $\pi = (s_0, s_1, \dots)$, if $(s_0, s_1) \in T_n$ and $M, \pi \models \Phi$, then $M, s_1 \models_\Phi f$.
- $M, s_0 \models_\Phi E_\Phi (fUg)$ iff there exists $\pi = (s_0, s_1, \dots)$, such that $M, \pi \models \Phi$, and there exists $i \geq 0$, such that $M, s_i \models_\Phi g$, and for all $0 \leq j < i, M, s_j \models_\Phi f$.
- $M, s_0 \models_\Phi A_\Phi (fUg)$ iff for all $\pi = (s_0, s_1, \dots)$, if $M, \pi \models \Phi$, then there exists $i \geq 0$, such that $M, s_i \models_\Phi g$, and for all $0 \leq j < i, M, s_j \models_\Phi f$.

By abbreviations, $f \vee g \equiv \neg(\neg f \wedge \neg g)$, $A(fWg) \equiv \neg E(\neg f U \neg g)$, $E(fWg) \equiv \neg A(\neg f U \neg g)$, $AG \equiv \neg EF \neg f$, and $EG \equiv \neg AF \neg f$ (hence, $AFf \equiv A(\text{true}Uf)$, $EFf \equiv E(\text{true}Uf)$, $AGf \equiv A(\text{false}Wf)$, and $EGf \equiv E(\text{false}Wf)$). A formula f is *satisfiable* iff there exists a model M such that $M, s \models_\Phi f$ for some state s of M .

C. Fairness and Outward-Facing

The local fairness assumption $\Phi = F^\infty ex_n \wedge \bigwedge_m F^\infty ex_m$. The path formula $F^\infty ex_n$ asserts that P_n is selected for execution infinitely often by the scheduler. The infinitary linear time operator F^∞ abbreviates GF and is interpreted as $M, \pi \models F^\infty g$ iff for every $i \geq 0$, there exists $j \geq i$, such that $M, \pi^j \models g$.

Formally, outward-facing is defined relative to Φ , extending the definition in [2]. Let s and t be two states on H_n^* ; s and t are related by a relation $B_{n,m}$ if $s[e] = t[e]$ for every common connected edge e between n and m . The notation $s[e]$ denotes the value of the external variable assigned to e at s . Process P_n is *outward-facing* in its interactions with P_m if $B_{n,m}$ is a stuttering bisimulation on H_n^* .

D. Parameterized Synthesis

We can now explain precisely how the reduction theorem supports parameterized synthesis.

Theorem II.2. *Let φ_n be a local FairCTL specification. Let P_n be a process such that its derived H_n^* satisfies φ_n . Every instance of the parameterized system constructed from isomorphic copies of P_n satisfies the global property $\bigwedge_i \varphi_i$.*

Proof. Consider P_n and its induced H_n^* which satisfies the local correctness property φ_n . By symmetry, each copy P_i of

the representative P_n has an isomorphic H_i^* which satisfies the corresponding φ_i .

Consider an instance of the parameterized system constructed from isomorphic copies of P_n . Let G be the global state space of the instance. Let i be a node of the instance. By Theorem II.1, G_i satisfies φ_i ; hence, by the locality of φ_i , it follows that G satisfies φ_i . As this holds for every node, G satisfies the global property $(\bigwedge_i \varphi_i)$. By the first part of Theorem II.1, this ‘inflationary’ consequence holds for any universal Fair CTL property. It holds for all Fair CTL properties if H_n^* is outward-facing. \square

The synthesis procedures of the following sections will, in effect, simultaneously construct both the strongest invariant θ^* and the resulting H_n^* .

III. EAGER SYNTHESIS

We describe the eager method of synthesizing a representative process P_n whose interference closure H_n^* satisfies the Fair CTL formula φ_n . The atomic propositions in φ_n are divided into two disjoint groups: X , representing properties of the external state, and L , representing properties of the internal state. We use a, b, a', b' to refer to valuations of variables in X , and k, l, k', l' to refer to valuations of variables in L . The notation $X = a$ means that each variable in X has the value given to it in a .

Given a local property φ_n , the eager method produces a Fair CTL formula $\mathcal{I}(\varphi_n)$ that is a conjunction of φ_n with several constraints. The constraints are expressed in CTL extended with the modal operators $\langle c \rangle$ and its negation dual $[c]$, where $\langle c \rangle f$ is the set of states from which there is a transition labeled c to a state satisfying f . It is straightforward to adjust the Fair CTL synthesis procedure for this variant of the EX operator.

The candidate models are labeled transition systems where transitions are labeled either by n (the representative) or by m (a neighbor). States are labeled with propositions from X and L . The constraints added to φ_n , intuitively, make the models ‘look’ similar to H_n^* .

A pair (a, a') of valuations to X is an *interference pair* if $\text{EF}((X = a) \wedge \langle n \rangle (X = a'))$ holds at the initial state of a candidate model; i.e., if there is a reachable state labeled a with an n -successor labeled a' . By symmetry, the n -transition producing this pair may be viewed as an m -transition of a neighbor. A pair (b, b') of valuations to X is considered the result of interference by (a, a') viewed as a neighboring m -transition if (1) the X -variables shared between m and n have the same valuations in b and a , and in b' and a' , and (2) the X -variables not shared between m and n have the same valuation in b and b' . The set of such pairs is denoted $\iota_m(a, a')$.

The Fair CTL formula $\mathcal{I}(\varphi_n)$ is the conjunction of φ_n with the constraints (1)-(4) given below. The added constraints are expressible in CTL as X and L have finitely many valuations.

- 1) Every interference pair induces an interference transition at all matching states. I.e., for every interference pair (a, a') and every (b, b') in $\iota_m(a, a')$, the property $\text{AG}((X = b) \Rightarrow \langle m \rangle (X = b'))$ holds.

- 2) m -transitions do not modify local state. I.e., $\text{AG}((L = l) \Rightarrow [m](L = l))$ for every valuation l of the local propositions.
- 3) Every m -transition is induced by an interference pair. I.e., for every b, b' such that $\text{EF}((X = b) \wedge \langle m \rangle (X = b'))$, there is an interference pair (a, a') such that $(b, b') \in \iota_m(a, a')$.
- 4) States with the same propositional label have similar successors. I.e., for c ranging over m and n : if $\text{EF}((X = a \wedge L = l) \wedge \langle c \rangle (X = a' \wedge L = l'))$ holds, then $\text{AG}((X = a \wedge L = l) \Rightarrow \langle c \rangle (X = a' \wedge L = l'))$.

A specification is *realizable* if it has a satisfying model.

Theorem III.1. $\mathcal{I}(\varphi_n)$ is realizable if and only if there is a process P_n with state space $2^X \times 2^L$ whose interference-closure H_n^* satisfies φ_n .

Proof. We show that any solution to the right-hand condition induces a solution to $\mathcal{I}(\varphi_n)$, and vice-versa.

From right-to-left, consider a process P_n meeting the right-hand condition. We claim that H_n^* satisfies conditions (1)-(4) by its inductive construction. If an interference pair (a, a') becomes reachable at some stage of the construction, it is used to construct interference transitions at all subsequent stages; thus, condition (1) holds. Interference transitions do not modify local state, meeting condition (2). Moreover, all interference transitions stem from an interference pair introduced at an earlier stage, meeting condition (3). Finally, as the closure is defined over the same state space as P_n , there is a unique state for each propositional labeling, satisfying condition (4).

The proof for the left-to-right direction is more involved, as we cannot *a priori* restrict the models of $\mathcal{I}(\varphi_n)$ to the state space $2^X \times 2^L$. Thus, consider any model M_0 of $\mathcal{I}(\varphi_n)$. We may assume that every transition of M_0 is reachable. (If not, limiting M_0 to its reachable state space still satisfies $\mathcal{I}(\varphi_n)$.)

Let \sim be the relation defined by $s \sim t$ if states s and t satisfy the same propositions. Condition (4) implies that \sim is a strong bisimulation on M_0 . (Proof: Consider states s, t such that $s \sim t$ and a c -successor s' of s . Let a, l be the propositions over X and L (respectively) that are satisfied by s , and let a', l' be the corresponding propositions satisfied by s' . The transition from s to s' is a witness to the assumption of (4); hence t must have a c -successor t' satisfying a', l' . By definition, $s' \sim t'$ holds.)

Let M_1 be the quotient of M_0 under \sim . As \sim is a strong bisimulation, M_0 and M_1 are strongly bisimilar; hence, both satisfy the same Fair CTL formulas; in particular, M_1 also satisfies $\mathcal{I}(\varphi_n)$. Let process P be the subgraph formed by the n -transitions of M_1 . We show that M_1 is the interference closure of P .

Note that by the definition of \sim and the quotient construction, every propositional valuation is associated with at most one state of M_1 , so we can consider M_1 to be isomorphic to a process with state space $2^X \times 2^L$.

We first show that the interference closure of P is a subgraph of M_1 , by induction on the stages of the closure construction. Initially, that is true as P is a subgraph of

M_1 . Suppose that this condition holds at the current stage. Consider the transition added at the next step. If this is a transition of P , it is already present in M_1 . If the transition is an interference transition applied at a state s , it must be derived from an n -transition present at the current stage. By the induction hypothesis, the inducing n -transition and the state s both belong to M_1 . By conditions (1) and (2), the derived interference transition from s also belongs to M_1 . It follows that the closure process constructed as the limit of these steps is a subgraph of M_1 .

We also need to rule out the existence of transitions in M_1 that are not in the closure process. Let t be a transition of M_1 , from a state (b, k) to (b', k') . If this is an n -transition, it belongs to P and hence to the closure. Consider the case where it is an m -transition. By (2), k' must equal k . From (3), there is an interference pair (a, a') in M_1 induced by an n -transition t' such that $(b, b') \in \iota_m(a, a')$. The n -transition t' is in P by definition and hence in the closure. Therefore the interference transition t induced by t' is also in the closure. \square

The eager method is technically interesting as it transforms the new, self-referential synthesis question into a standard form, simply by adding constraints that encode interference. However, the transformation results in an exponential blowup as the added constraints range over all propositional valuations. Hence, this method is likely to be impractical. The following section formulates a lazy procedure that gradually introduces interference into a *tableau* of the original formula.

IV. THE TABLEAU APPROACH

A *tableau* of n is a tuple $\mathcal{T}_n = (V_n, R, L)$, where V_n is a set of nodes; R is a transition relation over V_n , and $L : V_n \rightarrow 2^{Prop}$ is a labeling function. A tableau has two types of nodes, $V_n = V_n^C \cup V_n^D$ such that $V_n^C \cap V_n^D = \emptyset$, where V_n^C is a set of *AND-nodes* that are potential states of P_n , and V_n^D is a set of *OR-nodes*. The transition relation $R = R^{DC} \cup R^{CD}$, where $R^{DC} \subseteq V_n^D \times V_n^C$, $R^{CD} \subseteq V_n^C \times V_n^D$, and transitions in R^{CD} are labeled with n or $m \in nbr(n)$. Each node $v_n \in V_n$ is labeled with a subset of $Prop$, where $Prop$ is the extended Fischer-Ladner closure of φ_n [6], [7]. The closure $Prop$ describes the negation, subset, and fixpoint closure of the temporal operators.

We adopt the two-pass tableau approach of [8], [4], i.e., first construct a tableau from the specification, then prune and unravel the tableau into a model. The local property φ_n of interest is in the format of *init-spec* \wedge *other-spec*. Hence, *init-spec* specifies a single initial state. For multiple initial states, a set of local properties $\{\varphi_n^0, \varphi_n^1, \dots\}$ is generated, each with the same *other-spec* but a different *init-spec*.

We modify the classical tableau approach to synthesize H_n^* from φ_n , such that H_n^* is outward-facing and closed under interference. Subsection IV-A shows how to derive the initial tableau \mathcal{T}_n^0 closely following the original procedure [8]. Our main innovation is that we assume the neighbors are isomorphic copies of \mathcal{T}_n^i and subsection IV-B shows how to construct \mathcal{T}_n^{i+1} by adding interference transitions to \mathcal{T}_n^i . The iterative

TABLE I
THE α - β EXPANSION RULES.

$\alpha = f \wedge g$	$\alpha_1 = f$	$\alpha_2 = g$
$\alpha = A_\Phi(fWg)$	$\alpha_1 = g$	$\alpha_2 = f \vee A_\Phi YA_\Phi(fWg)$
$\alpha = E_\Phi(fWg)$	$\alpha_1 = g$	$\alpha_2 = f \vee E_\Phi XE_\Phi(fWg)$
$\alpha = A_\Phi Gg$	$\alpha_1 = g$	$\alpha_2 = A_\Phi YA_\Phi Gg$
$\alpha = E_\Phi Gg$	$\alpha_1 = g$	$\alpha_2 = E_\Phi XE_\Phi Gg$
$\alpha = A_\Phi Yg$	$\alpha_1 = A_\Phi Y_n g$	$\alpha_{nbr(n)} = A_\Phi Y_{nbr(n)} g$
$\beta = f \vee g$	$\beta_1 = f$	$\beta_2 = g$
$\beta = A_\Phi(fUg)$	$\beta_1 = g$	$\beta_2 = f \wedge A_\Phi YA_\Phi(fUg)$
$\beta = E_\Phi(fUg)$	$\beta_1 = g$	$\beta_2 = f \wedge E_\Phi XE_\Phi(fUg)$
$\beta = A_\Phi Fg$	$\beta_1 = g$	$\beta_2 = A_\Phi YA_\Phi Fg$
$\beta = E_\Phi Fg$	$\beta_1 = g$	$\beta_2 = E_\Phi XE_\Phi Fg$
$\beta = E_\Phi Xg$	$\beta_1 = E_\Phi X_n g$	$\beta_{nbr(n)} = E_\Phi X_{nbr(n)} g$

procedure continues until a fixpoint tableau \mathcal{T}_n^* is reached such that \mathcal{T}_n^* is closed under interference by isomorphic copies of \mathcal{T}_n^* . We then apply deletion rules (in Subsection IV-C), extract a model H_n^* from the pruned fixpoint tableau, and obtain P_n from H_n^* by removing interference transitions (in Subsection IV-D). These steps follow the original tableau procedure with slight variations.

A. The Initial Tableau

Similar to the classical tableau approach [8], [4], the root of the tableau, d_{root} , is an OR-node labeled with $\{\varphi_n\}$. Starting with d_{root} , the initial tableau \mathcal{T}_n^0 is constructed by repeatedly creating successors and appending them to the leaf nodes. In the case of duplicate labels and types, the newly created node is merged with the existing node, i.e. the new node is deleted and its incoming and outgoing edges are added to the existing node. The construction of \mathcal{T}_n^0 terminates when there are no more leaf nodes. If there are multiple initial states, we repeat the steps of constructing the initial tableau with different *init-specs* while merging duplicates.

For each OR-node d , $blocks(d)$ is a set of successors of d such that each AND-node $c_i \in blocks(d)$ represents a way of satisfying the formulae in $L(d)$. The generation of $blocks(d)$ follows the classical tableau approach, with a slightly different α - β expansion: as listed in Table I (c.f. [6]), most expansions are binary, except for AY and EX, which expand to a list of operators indexed by n and the neighbors in $nbr(n)$. The unindexed next-time operators are not part of φ_n but can be added to node labels during formula expansion. Formulae in $L(d)$ are satisfiable iff there exists a node in $blocks(d)$ whose label is satisfiable.

For each AND-node c , $tiles(c)$ is the minimal set of n -successors of c , i.e. the next-time states reachable through transitions labeled with n . Let $CA_n = \{f \mid A_\Phi Y_n f \in L(c)\}$ and $CE_n = \{g \mid E_\Phi X_n g \in L(c)\}$. For each $g \in CE_n$, an OR-node labeled $CA_n \cup \{g\}$ is created as a successor node of c . Edges from c to nodes in $tiles(c)$ are labeled with n . Here, we only consider a single edge case. I.e., if both CA_n and CE_n are empty sets, then we add a ‘dummy’ successor d_n to c and set $blocks(d_n) = \{c\}$. If $L(c)$ is satisfiable, then the labels of all nodes in $tiles(c)$ are satisfiable.

In the classical tableau approach, for each neighbor m , the set of m -successors of c are created in a similar way to n -

successors. However, since the local property φ_n only specifies the behavior of P_n , interference transitions by neighboring processes P_m are not specified in φ_n . Instead, we infer the transitions labeled with m based on transitions labeled with n . The next subsection shows the detailed steps of adding interference transitions and m -successors.

B. The Fixpoint Tableau

Starting from the initial tableau \mathcal{T}_n^0 containing only transitions labeled with n , we construct \mathcal{T}_n^{i+1} from \mathcal{T}_n^i through the following steps.

First, we summarize the interferences contained in the tableau so far. We search \mathcal{T}_n^i for n -transitions that change the values of shared variables and convert these n -transitions to a set of m -transitions for each neighbor m by bijection. That is, for each pair of AND-nodes c and c' such that $c' \in \text{blocks}(d)$ for $d \in \text{tiles}(c)$, let Y and Y' be the values of the shared variables in $L(c)$ and $L(c')$, respectively. If Y' is different from Y , then we use one or more tuples (m, Y_m, Y'_m) to record m -transitions that change the values of shared variables between n and m from Y_m to Y'_m .

Next, we add interference transitions to the current tableau. For each unique tuple (m, Y_m, Y'_m) , we add the interference transition to each applicable AND-node and label the transition with m . An AND-node c in \mathcal{T}_n^i is *applicable* to an interference transition (m, Y_m, Y'_m) if the values of the shared variables in Y_m match those in $L(c)$, and the interference transition is not already added to c .

For each AND-node c , $\text{bricks}_m(c)$ is a possibly empty set of m -successors of c , and $\text{bricks}(c) = \bigcup_{m \in \text{nbr}(n)} \text{bricks}_m(c)$. An empty $\text{bricks}_m(c)$ indicates an implicit self-loop by m in c , i.e., transitions labeled with m do not interfere with n in c .

The set $\text{bricks}_m(c)$ is generated as follows. Let $CA_m = \{f \mid A_\Phi Y_m f \in L(c)\}$ and $CE_m = \{g \mid E_\Phi X_m g \in L(c)\}$. These m -indexed properties are not sub-formulae of φ_n but are added to node labels as a result of α - β expansion. For example, $A_\Phi Gp$ expands to p , $A_\Phi Y_n A_\Phi Gp$, and $A_\Phi Y_m A_\Phi Gp$ for each m . For each unique interference (m, Y_m, Y'_m) and applicable AND-node c , we create an OR-node successor d_m . The label of d_m contains formulae in Y'_m , CA_m , and values of variables in $L(c)$ that are not shared with m .

In addition to that, we also create an OR-node successor of c for each $E_\Phi X_m g \in L(c)$. These successors capture the changes to shared variables as well as the satisfaction of existential next-time properties. For a given Y_m , consider the set of Y'_m such that (m, Y_m, Y'_m) is a tuple. Those Y'_m form the possible interference to shared variables. The changes to Y_m are translated into a disjunctive formula. Each change is represented as a conjunct of values of variables in Y'_m . For each $g \in CE_m$ and applicable AND-node c , we create an OR-node d_m , and $L(d_m)$ contains g , the disjunctive formula, formulae in CA_m , and values of variables in $L(c)$ that are not shared with m . For each newly created node d_m , we connect c to d_m by an edge labeled m and merge d_m if duplicated.

Figure 2 is an example of adding bricks_m to a given AND-node (n -successor nodes are omitted from the figure).

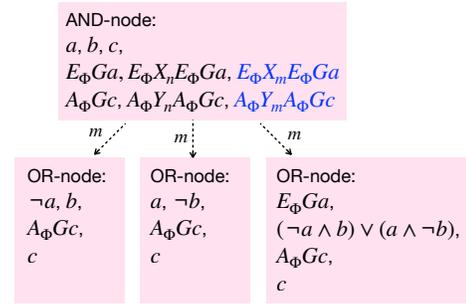


Fig. 2. The interference transitions and m -successors of an AND-node.

In this example, a and b are two external variables shared between n and m , and c is an internal variable of n . Suppose m interferes with n only by changing (a, b) to $(\neg a, b)$ or $(a, \neg b)$. The disjunctive formula representing changes of (a, b) is $(\neg a \wedge b) \vee (a \wedge \neg b)$. Property $E_\Phi G a$ is propagated to exactly one m -successor, and $A_\Phi G c$ is propagated to all m -successors. The propagation is done through blue formulae in the figure.

Finally, for each newly added OR-node d , we create descendants of d that are reachable via n -transitions. The construction terminates when there are no more leaf nodes. The size of the resulting tableau \mathcal{T}_n^{i+1} is greater than or equal to the size of \mathcal{T}_n^i . We repeat these steps until no more transitions or nodes can be added, i.e., when $\mathcal{T}_n^{i+1} = \mathcal{T}_n^i$. The resulting tableau captures all the changes to values of shared variables by neighboring processes as interference transitions.

Based on the fairness constraint, interference transitions will eventually be executed. At each AND-node c where the value of shared variables between n and m is represented by Y_m , we need to distinguish between two cases: (1) m changes Y_m to Y'_m through a (stuttering) transition such that $Y_m \neq Y'_m$, and (2) m keeps Y_m unchanged in a fair cycle. The first case was captured as interference transitions and the second as implicit self-loops. However, if both cases happen at the same Y_m , the corresponding node c should have the interference transition indicating the change as well as an explicit m -labeled self-loop indicating the choice of ‘remaining unchanged forever’. We add these self-loops to applicable AND-nodes in \mathcal{T}_n^* by using dummy nodes.

When no more transitions can be added, the tableau has reached its fixpoint, \mathcal{T}_n^* , and the construction terminates.

C. Tableau Pruning

The goal is to construct a model H_n^* such that P_n is outward-facing in H_n^* . Since H_n^* is extracted from the pruned \mathcal{T}_n^* , we added a *restricted outward-facing assumption* to only focus on tableaux where all the encoded models are outward-facing. For each neighbor m and each set of values of shared variables Y_m , the restricted outward-facing assumption requires the representative n to make the same set of changes to the shared state Y_m no matter which AND-node child is selected to be in the model. This guarantees a strictly stronger form of the outward-facing property.

TABLE II
THE DELETION RULES FOR TABLEAU PRUNING.

<i>deleteP</i>	Delete any node whose label is propositionally inconsistent.
<i>deleteOR</i>	Delete any OR-node all of whose successors are deleted.
<i>deleteAND</i>	Delete any AND-node one of whose successors is deleted.
<i>deleteEU</i>	Delete any node v if $E_{\Phi}(fUg) \in L(v)$, and there does not exist an AND node c' reachable from v through a finite path π , such that $g \in L(c')$ and $f \in L(c)$ for all AND-nodes c on π except c' .
<i>deleteAU</i>	Delete any node v if $A_{\Phi}(fUg) \in L(v)$, and there does not exist a subdag U rooted at v such that $g \in L(c')$ for all leaf nodes c' in U and $f \in L(c)$ for all internal AND-nodes c of U .
<i>deleteEG</i>	Delete any node v if $E_{\Phi}Gg \in L(v)$, and there does not exist a fair full path π starting at v such that $g \in L(c)$ for all nodes c on π .
<i>deleteJoint</i>	Delete any AND-node c_n if every AND-node c_m of a neighbor m that forms a joint state (c_n, c_m) is deleted.

Before pruning, we verify the assumption on \mathcal{T}_n^* and terminate the synthesis procedure if the assumption is violated. (Relaxing the assumption and finding an outward-facing model from any tableau is a future research direction.)

Similar to the classical approach, pruning tableau \mathcal{T}_n^* is done by deleting *inconsistent* nodes. As shown in Table II (c.f. [4]), an additional rule *deleteJoint* is added. *DeleteJoint* deletes any AND-node in \mathcal{T}_n^* that fails to form joint states with neighboring isomorphic tableau, \mathcal{T}_m^* . For each neighbor m and each value of shared variables Y_m , let C_n^Y be a set of AND-nodes of n such that $Y_m \subseteq L(c_n)$ for each node c_n in the set. Let C_m^Y be a set of AND-nodes of m such that each $c_m \in C_m^Y$ forms joint states with the nodes in C_n^Y . If all the isomorphic AND-nodes b_n of c_m in C_m^Y are deleted, we delete all the AND-nodes c_n in C_n^Y because the joint states of Y_m no longer hold, and vice versa.

The pruning process eventually terminates because the number of nodes in \mathcal{T}_n^* is finite. Upon termination, if the root of the tableau is deleted, then φ_n is not satisfiable by our procedure. Otherwise, we extract a model H_n^* from the pruned tableau.

D. Extraction of a Model

We reuse the existing procedure in [8], [6] to ‘unravel’ the pruned tableau \mathcal{T}_n^* into a model.

For each AND-node c in \mathcal{T}_n^* , we construct a fragment of c following the standard tableau approach. The structure of a fragment is taken from \mathcal{T}_n^* . All nodes in a fragment are AND-nodes. Nodes s and t are connected with a directed edge in a fragment if there exists transitions $(c, d), (d, c') \in R$ in \mathcal{T}_n^* , such that s and t are copies of c and c' , respectively. The fragment of c certifies the fulfillment of all eventualities in $L(c)$. When it comes to universal eventualities like $A_{\Phi}(fUg)$, if there are multiple subdags in the tableau, we choose the one with the least number of unfair cycles.

A model H_n^* is formed by connecting fragments together following the standard tableau approach. The process P_n is obtained from H_n^* by removing the interference transitions.

E. Soundness and Complexity

Theorem IV.1. *Soundness. If a labeled transition system H_n^* is constructed from φ_n , then H_n^* satisfies φ_n , H_n^* is closed under neighboring interference, and process P_n is outward-facing in H_n^* .*

Proof. During tableau construction, *blocks(d)* computes successors of an OR-node d , *tiles(c)* computes n -successors of an AND-node c , and *bricks(c)* computes m -successors of c for neighbors m . Based on the constructions of the tableau, all formulae in node labels are propagated correctly in the tableau of n , including \mathcal{T}_n^0 , any intermediate \mathcal{T}_n^i , and \mathcal{T}_n^* (similar to the proofs in [6]). For example, $A_{\Phi}(fUg)$ in the label of a node propagates to successor nodes as either g or f , $A_{\Phi}Y_nA_{\Phi}(fUg)$, and $A_{\Phi}Y_mA_{\Phi}(fUg)$. The propagation continues forever along each path until g is reached.

Since all the nodes in the pruned \mathcal{T}_n^* are consistent, all the eventualities in the label of any AND-node in the pruned \mathcal{T}_n^* are fulfilled in a fragment rooted at the node. Since H_n^* is constructed by concatenating fragments, starting with a root that automatically satisfies φ_n , H_n^* is a model of φ_n .

The size of the tableau increases monotonically until it reaches a fixpoint, \mathcal{T}_n^* . Since the size of \mathcal{T}_n is bounded, the tableau construction eventually terminates at the fixpoint. By construction, each intermediate tableau \mathcal{T}_n^i fully reflects the interference of neighboring isomorphic copies of \mathcal{T}_n^{i-1} . The construction continues until no more nodes can be added to the tableau. Therefore, \mathcal{T}_n^* is closed under self-interference.

Then, we show that the model is also closed under self-interference. Based on *deleteJoint*, in the pruned tableau \mathcal{T}_n^* , for each m -labeled transition $Y_m \rightarrow Y'_m$ and each AND-node c whose label contains Y_m , c forms joint states with neighbors m , and there exists transitions isomorphic to $Y_m \rightarrow Y'_m$ in the pruned \mathcal{T}_n^* . On the other hand, in the pruned \mathcal{T}_n^* , the set of interference transitions reflects exactly the set of transitions labeled with n that change the values of shared variables. Based on model extraction, H_n^* is closed.

Since \mathcal{T}_n^* satisfies the restricted outward-facing tableau assumption, for all the encoded models H_n^* , process P_n is outward-facing in H_n^* . \square

Lemma IV.2. *Let φ_n be a local property of n , and Σ_n^{share} the set of shared variables in Σ_n . The size of tableau \mathcal{T}_n is bounded by $\exp(|\varphi_n| + \exp(|\Sigma_n^{share}|))$.*

Proof. For each n -successor v_n in \mathcal{T}_n , $L(v_n) \subseteq Prop$, so the number of formulae in $L(v_n)$ is less than or equal to $|Prop|$. Since duplicate nodes are merged, the number of n -successors in \mathcal{T}_n is bounded by $\exp(|Prop|)$.

As in Section IV-B, an extra disjunctive formula is added to the labels of some OR-nodes to represent the interference transitions of neighbors m . Considering binary variables, the number of different values of shared variables is $\exp(|\Sigma_n^{share}|)$. Hence, there are at most $\exp(\exp(|\Sigma_n^{share}|))$ different ways related to the presence of a disjunctive formula in node labels. Therefore, the number of nodes in \mathcal{T}_n is bounded by $\exp(|Prop|) + \exp(\exp(|\Sigma_n^{share}|))$. Since $|Prop|$

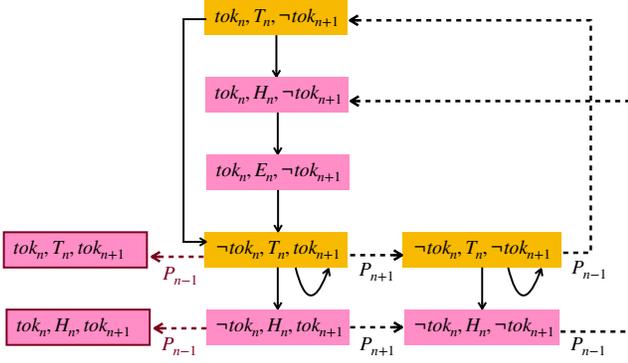


Fig. 3. A process model of the mutual exclusion protocol.

is linear in terms of $|\varphi_n|$, the number of nodes in \mathcal{T}_n is in $O(\exp(|\varphi_n| + \exp(|\Sigma_n^{share}|)))$. In applications, $\exp(|Prop|)$ is more likely to dominate $\exp(\exp(|\Sigma_n^{share}|))$. In most cases, the size of \mathcal{T}_n is exponential in the length of the input local Fair CTL property φ_n . \square

Lemma IV.3. *The cost of constructing P_n is in time polynomial in the size of the tableau.*

Proof. For each node v in tableau \mathcal{T}_n , the sum of the lengths of the formulae in $L(v)$ is in $O(|\varphi_n|^2)$. The cost of computing successor for v is polynomial in $|\varphi_n|$. Fixpoint construction, tableau pruning, and unraveling all require time polynomial in the size of the tableau. Therefore, the total cost of constructing P_n is in time $O(\exp(|\varphi_n| + \exp(|\Sigma_n^{share}|)))$. \square

The tableau approach constructs P_n as a template for the locally symmetric processes. To deploy the template throughout the process network, the subscript indices on all state and transition labels are changed accordingly.

V. APPLICATIONS

We illustrate our approach with three ring-based protocols, namely, mutual exclusion, leader election, and dining philosophers. Our approach is implemented in Python with the CTL module provided in the *pyModelChecking* API. We tested the synthesis procedure on a 2.5 GHz CPU and 16 GB of memory, and each ran for 5.3, 297, and 261 seconds, respectively. In each case, the procedure converged within three tableau iterations.

A. Mutual Exclusion

Mutual exclusion is a mechanism that prevents processes from accessing a shared resource simultaneously. Globally, the mutual exclusion property asserts that no two processes can be in the critical section at the same time. Locally, the property is achieved through token passing.

For any $K \geq 2$ and a generic n , the external variables tok_n and tok_{n+1} are shared with $n-1$ and $n+1$, respectively. The internal variable N_n stands for non-critical, T_n for trying, and C_n for critical. We specify φ_n as follows.

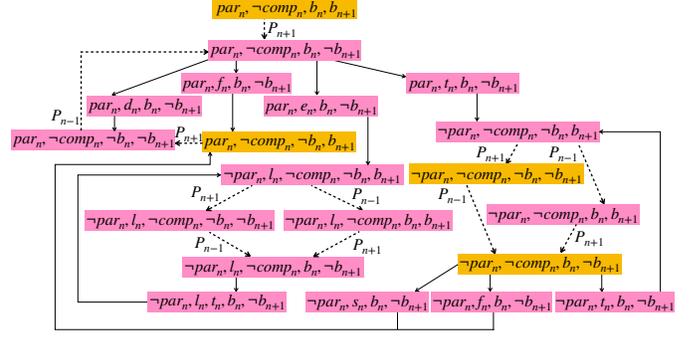


Fig. 4. A process model of the leader election protocol.

- Three initial conditions: $N_n \wedge tok_n \wedge \neg tok_{n+1}$ (n has the token), $N_n \wedge \neg tok_n \wedge tok_{n+1}$ (the right neighbor has the token), and $N_n \wedge \neg tok_n \wedge \neg tok_{n+1}$ (no token locally).
- Local mutual exclusion: $A_\Phi G(\neg tok_n \vee \neg tok_{n+1})$.
- Moves of n from non-critical to trying (while keeping the token) or remains in non-critical (while passing the token): $A_\Phi G((N_n \wedge \neg tok_n) \Rightarrow (E_\Phi X_n(N_n \wedge \neg tok_n) \wedge E_\Phi X_n(T_n \wedge \neg tok_n)))$, $A_\Phi G((N_n \wedge tok_n) \Rightarrow (E_\Phi X_n(N_n \wedge \neg tok_n \wedge tok_{n+1}) \wedge E_\Phi X_n(T_n \wedge tok_n)))$.
- Moves of n from trying to critical with the token: $A_\Phi G((T_n \wedge tok_n) \Rightarrow A_\Phi Y_n(C_n \wedge tok_n))$.
- Moves of n from critical to non-critical while passing the token, $A_\Phi G(C_n \Rightarrow A_\Phi Y_n(N_n \wedge \neg tok_n \wedge tok_{n+1}))$.
- The liveness property: $A_\Phi G(T_n \Rightarrow A_\Phi F C_n)$
- One at a time: $A_\Phi G(N_n \vee T_n \vee C_n)$, $A_\Phi G(N_n \Rightarrow (\neg T_n \wedge \neg C_n))$, $A_\Phi G(T_n \Rightarrow (\neg N_n \wedge \neg C_n))$, and $A_\Phi G(C_n \Rightarrow (\neg N_n \wedge \neg T_n))$.

Properties that ensure variables remain unchanged are omitted from the list for the sake of clarity. By induction on the size K , assuming the initial condition that exactly one process owns a single token, if φ_n is true for all processes in a ring, then it guarantees that each process eventually gets and passes the token, and there is exactly one token (i.e., tokens are not generated or lost). Hence, no two processes access the critical resource simultaneously.

Fig. 3 is a model of φ_n . Rectangles represent local states, where yellow corresponds to initial states. Solid arrows are transitions by P_n , and dashed arrows are interference transitions. Rectangles with red borders are inconsistent states because φ_n has no information about the initial conditions of non-neighboring processes. I.e., in the perspective of n , there is at most one token locally, but globally, n does not know. Instead of deleting the parents of these inconsistent states according to the deletion rules, we manually refine the set of interference transitions by taking into account the initialization of all processes in the ring. I.e., there is only one token.

B. Chang and Roberts Leader Election

Suppose each process has a finite and unique competing value (abbr. cv). The goal of the protocol is to select the process with the largest cv to be the leader. Globally, the correctness of the protocol is specified as a safety property, i.e.,

there is never more than one leader, and a liveness property, i.e., eventually there will be a leader. The specification can be written locally from the perspective of a generic n [9]. The cv of n may or may not be the greatest on the network.

Initially, some but not all processes detect the absence of the leader, i.e., P_n may become a participant in the election and send out an election message containing its cv to the right. When P_n receives an election message from its left, P_n compares the competing value in the message, denoted by cv' , with its own cv . In general, the comparison yields three different outcomes, i.e., $cv' > cv$, $cv' < cv$, and $cv' = cv$. If $cv' > cv$, P_n forwards the message to the right. If $cv' < cv$, P_n sends a message of its own cv . If $cv' = cv$, P_n becomes the leader. A non-participant becomes a participant after forwarding or sending an election message, and a participant no longer sends election messages of its own cv . A new leader sends a message to the right to terminate the election. Upon receiving the termination message, a process becomes non-participant and forwards the message.

A constructed model H_n^* is shown in Fig. 4. External variables b_n and b_{n+1} are shared with left and right neighbors, representing shared message buffers of size one. Internal variables par_n denotes that P_n is a participant, l_n denotes that P_n is the leader. Comparisons are abstracted into boolean variables. When $comp_n$ is true indicating a comparison in progress, one of the following is true, f_n (greater/forward), s_n (smaller/send), d_n (smaller/discard), e_n (equal), and t_n (election termination). For comparison results other than d_n , b_{n+1} becomes true, i.e., a message is sent to the right.

The global reasoning for this protocol is as follows. Globally, there exists one process whose competing value is the greatest. Based on the global initialization and local specification, and supposing the message comparison always yields correct results, the process with the greatest cv sends and receives a message with its own competing value. For all the other processes, messages with their competing values will not go through the full round of message passing, and these messages will eventually be discarded by processes with a greater competing value. Therefore, there will eventually be a leader and never more than one leader.

C. Dining Philosophers

In a standard dining philosopher protocol [10], the internal state of P_n is one of T_n (thinking), H_n (hungry), or E_n (eating). Fig. 1 indicates the external variables of n , where r_n means that P_n picks up its left fork, and l_n means that the left neighbor picks up the fork. Similarly, l_{n+1} means that P_n picks up its right fork, and r_{n+1} means that the right neighbor picks up the fork. The variables r_n and l_n cannot be true at the same time, nor can r_{n+1} and l_{n+1} . Both variables r and l are false means the corresponding fork is available. Process P_n can read and write r_n and l_{n+1} , but P_n has read-only access to l_n and r_{n+1} .

Process P_n can stay in thinking or move to hungry at any time, and P_n in its hungry state picks up available forks. While holding both the left and the right forks (i.e., $r_n \wedge l_{n+1}$),

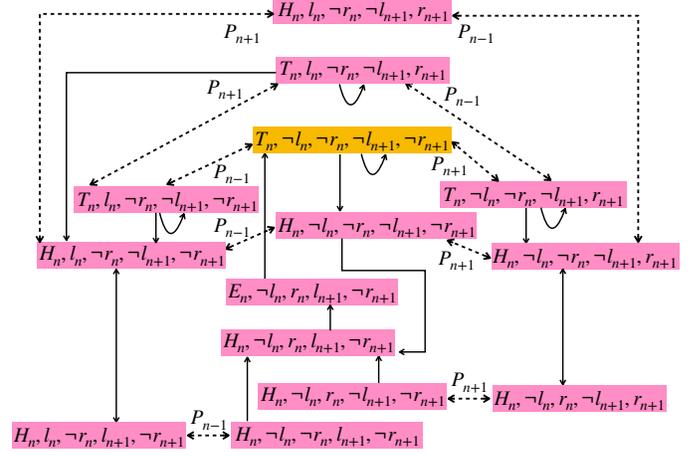


Fig. 5. A process model of dining philosophers.

P_n should enter into the eating state. After eating, P_n goes back to thinking and returns the forks (i.e., $\neg r_n \wedge \neg l_{n+1}$). The specification guarantees that no two neighboring processes are eating simultaneously.

Fig. 5 shows a model of φ_n . Adding a liveness property, $A_{\Phi}G(H_n \Rightarrow A_{\Phi}FE_n)$ would make φ_n unsatisfiable. Livelock and starvation are possible and are observed locally in the model. The unsatisfiability of appending the liveness property to φ_n does not mean there is no local solution to the dining philosopher problem. On the contrary, the problem can be solved using acyclic precedence graphs as in [10] (i.e., by modifying φ_n and introducing more variables).

VI. RELATED WORK AND CONCLUSION

In this paper, we reduce the synthesis problem for a parameterized protocol to the problem of synthesizing a representative process that meets a local specification under interference from neighboring copies of itself. The algorithm runs in time exponential in the length of the local property, which is expressed in Fair CTL and may include safety as well as liveness aspects, using both universal and existential path quantification. The approach is incomplete and not fully automated, but it succeeds on several interesting cases.

The novelty is in our solution to the new ‘self-referential’ synthesis question. Our tableau construction builds on the classical one of [8] for CTL and that of [4] for Fair CTL. These constructions work in closed synthesis settings where the environment is assumed to be cooperative. A fully open synthesis procedure was devised for LTL in [11]. In our case, the environment is formed of copies of the unknown to-be-synthesized process, which is an open synthesis problem of a special type.

The work relies on the compositional inductive invariant under local symmetry given in [12], [13], and [2]. We capture the behaviors of a representative in its neighborhood as a fixpoint tableau. Other work related to inductive invariants (c.f. [14]) uses similar fixpoint characterizations to compute thread-modular rely-guarantee assertions under abstractions.

Synthesis of a distributed system is undecidable, even with a fixed number of components [15]. Decidable architectures are known [16] as are decision procedures (c.f. [17], [18]), but the complexity is exponential or even nonelementary in the number of processes. In contrast, our procedure produces a representative process that is replicated to form arbitrary-size instances, so its complexity is independent of the instance size.

Reduction or generalization theorems are also central to prior work on parametrized synthesis. In [5] representative processes are constructed from synthesis of pair-systems. The paper [19] decides ‘almost always satisfiability’ for indexed but restricted CTL properties. Cutoff results for parametrized verification are applied in [20] to synthesize ring protocols; however, the dining philosophers and leader election examples fall outside the class for which cutoffs are known. The paper [21] takes an automata-theoretic approach to rotation-symmetric architectures. Synthesis of symmetric processes in self-stabilizing parameterized unidirectional rings is explored by [22]. The paper [23] focuses on round-bounded parameterized systems.

The different approaches that exploit symmetry in the system structures make use of a kind of global symmetry c.f. [24], [25], and [26]. In contrast, the work presented in this paper relies on notions of local symmetry as introduced in [12], [13], and [2]. The differences are important because local symmetry properly generalizes ‘global symmetry,’ often allowing for exponentially more reduction, for instance in the case of ring architectures. Our work here is the first to show how the notation of local symmetry can be used to form the basis of a synthesis procedure whose output is a single representative that can be deployed across all network instances in the parametric family of networks.

The reduction theorem on which the work in this paper is based is of an assume-guarantee type. Existing formulations of assume-guarantee synthesis (c.f. [27], [28]) however do not allow for the self-referential form of synthesis that is required by the reduction theorem.

We are currently working on applications to other protocols, including those with several representative processes, to fault tolerant protocols [6], and towards relaxing the outward-facing assumption.

Acknowledgments. Kedar Namjoshi was supported in part by DARPA under contract HR001120C0159. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Richard Trefer and Ruoxi Zhang were supported, in part, by an Individual Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

- [1] K. R. Apt and D. Kozen, “Limits for automatic verification of finite-state concurrent systems,” *Inf. Process. Lett.*, vol. 22, no. 6, pp. 307–309, 1986.
- [2] K. S. Namjoshi and R. J. Trefer, “Symmetry reduction for the local mu-calculus,” in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 379–395.
- [3] —, “Loop freedom in AODVv2,” in *FORTE 2015*, ser. LNCS, vol. 9039, 2015, pp. 98–112.
- [4] E. A. Emerson and C.-L. Lei, “Temporal reasoning under generalized fairness constraints,” in *STACS 86*, B. Monien and G. Vidal-Naquet, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 21–36.
- [5] P. C. Attie and E. A. Emerson, “Synthesis of concurrent systems with many similar processes,” *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 1, pp. 51–115, Jan. 1998.
- [6] P. C. Attie, A. Arora, and E. A. Emerson, “Synthesis of fault-tolerant concurrent programs,” *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 1, pp. 125–185, Jan. 2004.
- [7] M. J. Fischer and R. E. Ladner, “Propositional dynamic logic of regular programs,” *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 194–211, 1979.
- [8] E. A. Emerson and E. M. Clarke, “Using branching time temporal logic to synthesize synchronization skeletons,” *Science of Computer Programming*, vol. 2, no. 3, pp. 241–266, 1982.
- [9] E. Chang and R. Roberts, “An improved algorithm for decentralized extrema-finding in circular configurations of processes,” *Commun. ACM*, vol. 22, no. 5, pp. 281–283, May 1979.
- [10] K. M. Chandy and J. Misra, “The drinking philosophers problem,” *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 4, p. 632–646, oct 1984.
- [11] A. Pnueli and R. Rosner, “On the synthesis of a reactive module,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’89. New York, NY, USA: Association for Computing Machinery, 1989, pp. 179–190.
- [12] K. S. Namjoshi and R. J. Trefer, “Local symmetry and compositional verification,” in *VMCAI*, ser. LNCS, vol. 7148, 2012, pp. 348–362.
- [13] —, “Parameterized compositional model checking,” in *TACAS*, ser. LNCS, vol. 9636, 2016, pp. 589–606.
- [14] A. Miné, “Relational thread-modular static value analysis by abstract interpretation,” in *VMCAI*, 2014, pp. 39–58.
- [15] A. Pnueli and R. Rosner, “Distributed reactive systems are hard to synthesize,” in *Proceedings 31st Annual Symposium on Foundations of Computer Science*, 1990, pp. 746–757 vol.2.
- [16] B. Finkbeiner and S. Schewe, “Uniform distributed synthesis,” in *20th Annual IEEE Symposium on Logic in Computer Science (LICS’ 05)*, 2005, pp. 321–330.
- [17] O. Kupferman and M. Vardi, “Synthesizing distributed systems,” in *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, 2001, pp. 389–398.
- [18] S. Mohalik and I. Walukiewicz, “Distributed games,” in *FSTTCS*, 2003, pp. 338–351.
- [19] E. A. Emerson and J. Srinivasan, “A decidable temporal logic to reason about many processes,” in *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC. New York, NY, USA: Association for Computing Machinery, 1990, pp. 233–246.
- [20] S. Jacobs and R. Bloem, “Parameterized synthesis,” *Log. Methods Comput. Sci.*, vol. 10, no. 1, 2014.
- [21] R. Ehlers and B. Finkbeiner, “Symmetric synthesis,” in *FSTTCS*, ser. LIPIcs, vol. 93. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 26:1–26:13.
- [22] A. P. Klinkhamer and A. Ebnenasir, “Synthesizing Parameterized Self-stabilizing Rings with Constant-Space Processes,” in *FSEN*, ser. LNCS, vol. 10522, 2017, pp. 100–115.
- [23] B. Bollig, M. Lehaut, and N. Sznajder, “Round-Bounded Control of Parameterized Systems,” in *ATVA*, ser. LNCS, vol. 11138, 2018, pp. 370–386.
- [24] E. A. Emerson and A. P. Sistla, “Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach,” *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 4, p. 617–638, jul 1997.
- [25] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, “Exploiting symmetry in temporal logic model checking,” *Formal Methods in System Design*, vol. 9, no. 1, p. 77–104, Aug. 1996.
- [26] C. N. Ip and D. L. Dill, “Better verification through symmetry,” *Form. Methods Syst. Des.*, vol. 9, no. 1–2, p. 41–75, aug 1996.
- [27] K. Chatterjee and T. A. Henzinger, “Assume-guarantee synthesis,” in *In Proceedings of TACAS’07*, 2007, pp. 261–275.
- [28] R. Majumdar, K. Mallik, A. Schmuck, and D. Zufferey, “Assume-guarantee distributed synthesis,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3215–3226, 2020.

Synthesizing Self-Stabilizing Parameterized Protocols with Unbounded Variables

Ali Ebneenasir 

Department of Computer Science
Michigan Technological University
Houghton, MI 49931, U.S.A.
Email: aebneenas@mtu.edu

Abstract—The focus of this paper is on the synthesis of unidirectional symmetric ring protocols that are self-stabilizing. Such protocols have an unbounded number of processes and unbounded variable domains, yet they ensure recovery to a set of legitimate states from any state. This is a significant problem as many distributed systems should preserve their fault tolerance properties when they scale. While previous work addresses this problem for constant-space protocols where domain size of variables are fixed regardless of the ring size, this work tackles the synthesis problem assuming that both variable domains and the number of processes in the ring are unbounded (but finite). We present a sufficient condition for synthesis and develop a sound algorithm that takes a conjunctive state predicate representing legitimate states, and generates the parameterized actions of a protocol that is self-stabilizing to legitimate states. We characterize the unbounded nature of protocols as semilinear sets, and show that such characterization simplifies synthesis. The proposed method addresses a longstanding problem because recovery is required from any state in an unbounded state space. For the first time, we synthesize some self-stabilizing unbounded protocols, including a near agreement and a parity protocol.

Index Terms—Parameterized Systems, Synthesis and Verification, Self-Stabilization

I. INTRODUCTION

This paper investigates the problem of synthesizing Self-Stabilizing unidirectional Symmetric ring protocols with Unbounded number of processes and unbounded variable domains, called SS-SymU protocols (a.k.a. unbounded uni-rings). A *process* contains a set of atomic actions. When an action of a process is executed, it is disabled until enabled again by the neighboring processes; i.e., *self-disabling* actions. In a *symmetric* ring, the actions of each process are generated from a *template* process by a simple variable re-indexing. A self-stabilizing protocol automatically recovers (in a finite number of steps) to a set of legitimate states \mathcal{I} from *any* arbitrary state [1]; i.e., all states are initial states. Such recovery should be achieved without the intervention of a central authority. The significance of this synthesis problem is multi-fold. First, while uni-ring is a simple topology, it is of practical importance in distributed systems where the underlying communication topology may include cyclic structures. Second, the unboundedness of the ring size and variable domains is a requirement where networks scale up and buffer sizes grow. The elegance of many distributed protocols/algorithms (e.g., logical clocks [2], Dijkstra’s token passing [1], unbounded registers [3])

is due to the assumption of unbounded variable domains and processes, which makes it significant to develop tools that can synthesize such protocols under the unboundedness assumption. Third, self-stabilization is an important fault tolerance property that enables decentralized recovery in the presence of transient faults, which perturb the system state without causing permanent damages. While previous work [4], [5], [6], [7], [8] addresses the verification and synthesis of parameterized symmetric uni-rings, the domain size of variables remains constant regardless of the ring size. To the best of our knowledge, this paper presents the first method for the synthesis of SS-SymU protocols that are unbounded in terms of both the number of processes and variable domains.

Most existing methods for the synthesis of self-stabilizing protocols either focus on fixed-size protocols or consider an unbounded number of processes only; variable domains are considered bounded. For example, specification-based methods [9] compose a pair of template processes to reason about the global safety and local liveness properties of parameterized synchronization skeletons. Methods for fixed-size synthesis [10], [11], [12], [13] consider a fixed upper bound k on the number of processes, and generate a solution that is correct up to k processes. To enable the synthesis of parameterized self-stabilizing systems where solutions work for an arbitrary number of n processes, some approaches rely on parameterized synthesis [14] where an implementation is generated for a parameterized specification and a parameterized architecture. Such methods employ bounded [15] and SMT-based [11] synthesis to show the correctness of a solution with cutoff number of processes, where a solution exists for a protocol with *cutoff* number of processes *iff* (if and only if) a solution exists for the parameterized protocol with unbounded number of processes. Other methods [7] present cutoffs for the synthesis of self-stabilizing protocols in symmetric networks, however, such cutoffs can be quadratic/exponential in the bounded variable domains depending on the structure of \mathcal{I} . Synthesis of parameterized systems with threshold guards [4] starts with a sketch automaton (whose transitions have incomplete guard conditions capturing the number of received messages), and complete the guards towards satisfying program specifications. Our previous work [5] addresses the synthesis of self-stabilizing parameterized protocols where the local state space of the template process remains constant.

Contributions. In contrast to most existing methods, we propose a novel approach based on the synthesis of semilinear sets in the unbounded local state space of the template process of SS-SymU for conjunctive predicates. Specifically, we start with a global state predicate $\mathcal{I} = \forall i \in \mathbb{N} :: L(x_{i-1}, x_i)$ where $L(x_{i-1}, x_i)$ denotes a local state predicate of the template process P_i and x_i is an abstraction of the local state of P_i . We then generate a protocol that self-stabilizes to \mathcal{I} regardless of network size and the domain size of variables. Domain size is of particular importance as some protocols may not exist for specific domain sizes (e.g., Dijkstra’s token ring [1] requires a domain size of at least $N - 1$ in a ring of N processes). We utilize necessary and sufficient conditions identified in [5], [6] for the livelock-freedom of a solution with constant-space processes in order to impose a structure on the unbounded transition system of the template process. Such conditions require the existence of a value γ in the domain of x_i for which $L(\gamma, \gamma)$ holds. Moreover, necessary and sufficient conditions for livelock-freedom (under an unfair scheduler) require a tree-like structure rooted at γ for the local state transition system of the template process. While these results are for constant-space processes, we generalize them for unbounded domain sizes. Specifically, we show that if the state transition system of the template process is a semilinear set represented as an infinite tree rooted at γ , then a solution exists. A semilinear set is the finite union of a set of linear sets, where a linear set contains periodic integer vectors. Based on this sufficient condition, we develop a sound algorithm that takes $L(x_{i-1}, x_i)$ and generates the periodic linear sets of a semilinear set in a way that their vectors are organized in a potentially infinite tree rooted at γ . Each synthesized linear set represents the unbounded structure of a protocol action. We then use such linear sets to synthesize the parameterized actions of a protocol that self-stabilizes to \mathcal{I} for unbounded number of processes and unbounded domain sizes. We demonstrate the proposed method using a near-agreement and a parity protocol.

Organization. Section II provides some basic concepts. Section III presents the proposed synthesis method. Section IV demonstrates the application of the synthesis method for a parity protocol. Section V discusses related work. Section VI makes concluding remarks and discusses future research.

II. PRELIMINARIES

This section represents the definition of state predicates, parameterized protocols and their representation as locality graphs (adopted from [16], [17], [5], [6]), and semilinear sets. We use the term *parameterized protocol* to refer to uni-ring symmetric protocols that have both unbounded number of processes and unbounded variable domains. A protocol p includes $N > 1$ *symmetric* processes on a uni-ring, where the code of each process is derived from the code of a template process P_i by variable re-indexing. The template process P_i has a variable x_i whose domain abstracts the set of valuations to all writable variables of P_i . The domain of x_i , denoted $M = \text{Dom}(x_i)$, can be unbounded (but finite). Any *local state* of a process (a.k.a. *locality/neighborhood*) is determined by a

unique valuation of its readable variables. We assume that any writable variable is also readable. Network topology defines the set of readable variables of a process. For example, in a uni-ring consisting of N processes, each process P_i (where $i \in \mathbb{Z}_N$, i.e., $0 \leq i \leq N - 1$) has a predecessor P_{i-1} , where subtraction is in modulo N . That is, P_i can read the values of x_i and x_{i-1} , but can update only x_i . The *global state* of a protocol is defined by a snapshot of the local states of all processes. The *state space* of a protocol p , denoted by Σ_p , is the universal set of all global states of p . A *state predicate* is a subset of Σ_p . A process *acts* (i.e., *transitions*) when it atomically updates its state based on its locality.

We assume that processes act one at a time (i.e., interleaving semantics). Thus, each *global transition* corresponds to the action of a single process from some global state. An *execution/computation* of a protocol is a sequence of states s_0, s_1, \dots, s_k where there is a transition from s_i to s_{i+1} for every $i \in \mathbb{Z}_k$. The transition function $\delta : \Sigma_p \times \Sigma_p \rightarrow \Sigma_p$ of the template process captures its set of actions $x_{i-1} = a \wedge x_i = b \rightarrow x_i := c$, which can also be captured as triples of the form (a, b, c) . That is, $\delta(a, b) = c$ iff (if and only if) P_i has an action $x_{i-1} = a \wedge x_i = b \rightarrow x_i := c$. An action has two components; a *guard*, which is a Boolean expression in terms of readable variables and a *statement* that atomically updates the state (i.e., writable variables) of the process once the guard holds; i.e., the action is *enabled*. Previous work [18] shows that assuming self-disabling and deterministic processes simplifies synthesis without undermining soundness and completeness. An action (a, b, c) cannot co-exist with action (a, c, d) in a *self-disabling* process for any d . A *deterministic* process cannot have two actions enabled at the same time; i.e., an action (a, b, c) cannot co-exist with an action (a, b, d) where $d \neq c$.

Definition II.1 (Action Graph). For a fixed domain size M , we can depict the set of actions of the template process of a symmetric uni-ring by a labeled directed multigraph $G = (V, A)$, called the *action graph*, where each vertex $v \in V$ represents a value in \mathbb{Z}_M , and each arc $(a, c) \in A$ with a label b captures an action $x_{i-1} = a \wedge x_i = b \rightarrow x_i := c$.

For example, consider the Parity protocol introduced in [6]. Each process P_i has a variable $x_i \in \mathbb{Z}_3$ (i.e., $M = 3$) and actions $x_{i-1} = 0 \wedge x_i = 1 \rightarrow x_i := 0$, $x_{i-1} = 1 \wedge x_i = 2 \rightarrow x_i := 0$, and $x_{i-1} = 2 \wedge x_i = 1 \rightarrow x_i := 0$. This protocol ensures that, from any global state of a symmetric uni-ring, a state is reached where processes agree on a common odd/even parity. We formally specify these states as the state predicate $\mathcal{I}_{Par} = \forall i \in \mathbb{Z}_N : ((|x_{i-1} - x_i| \bmod 2 = 0))$. Throughout this paper, the subscript operations are modulo number of processes, and the arithmetic operations in the state predicates, and in the guard and assignment of actions are performed modulo M . Figure 7b illustrates this protocol as an action graph containing arcs $(0, 1, 0)$, $(1, 2, 0)$, and $(2, 1, 0)$.

Definition II.2 (Self-Stabilization and Convergence). A protocol p is *self-stabilizing* [1] to a state predicate \mathcal{I} iff from any state in $\neg\mathcal{I}$, every computation of p reaches a state in

\mathcal{I} (i.e., *convergence*) and remains in \mathcal{I} (i.e., *closure*). A state predicate \mathcal{I} is *closed* in p iff there is no transition (s, s') , where $s \in \mathcal{I}$ and $s' \notin \mathcal{I}$. Convergence of p to \mathcal{I} requires that p does not reach a deadlock, nor does it reach a livelock in $\neg\mathcal{I}$. A *deadlock* state is a global state where no process has any enabled action. A *livelock* is an infinite cyclic computation $l = \langle s_0, s_1, \dots, s_0 \rangle$, where s_i is a global state, for $i \geq 0$.

Definition II.3 (Locality Graph). Consider a global state predicate $\mathcal{I} = \forall i \in \mathbb{Z}_N : L(x_{i-1}, x_i)$ for a protocol, and a domain size M . The local predicate $L(x_{i-1}, x_i)$ captures a set of local states, representing an acceptable relation between the states of each process P_i and the states of its predecessor P_{i-1} . We represent $L(x_{i-1}, x_i)$ as a digraph $G = (V, A)$, called the *locality graph*, such that each vertex $v \in V$ represents a value in \mathbb{Z}_M , and an arc (a, b) is in A iff $L(a, b)$ holds.

Figure 7a illustrates the locality graph of the Parity protocol introduced in this section for $M = 3$ and the state predicate $L(x_{i-1}, x_i) \equiv ((|x_{i-1} - x_i| \bmod 2) = 0)$. We have extensively studied [5], [6] the use of locality and action graphs in local reasoning about global properties (e.g., livelocks). Our previous work [17], [5] investigates the following synthesis problem, whereas in Section III we solve this problem when its assumption is lifted.

Problem II.4 (Synthesis of Symmetric Uni-Rings).

- **Input:** $L(x_{i-1}, x_i)$, and the domain size M of x_i .
- **Output:** The transition function δ (represented as an action graph or parameterized actions) of a protocol p such that the entire ring is self-stabilizing to $\mathcal{I} = \forall i : i \in \mathbb{Z}_N : L(x_{i-1}, x_i)$ for any ring size $N \geq 3$.
- **Assumption:** M is fixed regardless of the ring size N ; i.e., p has *constant-space* processes.

The following theorem (proved in [17], [5]) provides the foundation of a synthesis method for parameterized uni-rings with constant-space processes. In the rest of this section, we present an overview of the synthesis method of [5] since its knowledge is required for our exposition.

Theorem II.5. *There is a symmetric uni-ring protocol p (with deterministic, self-disabling and constant-space processes) that self-stabilizes to $\mathcal{I} = \forall i \in \mathbb{Z}_N : L(x_{i-1}, x_i)$ for an unbounded (but finite) number of N processes iff there is a vertex γ in the locality graph G of $L(x_{i-1}, x_i)$, where $L(\gamma, \gamma)$ holds, and the action graph of p is a directed spanning tree of G , sinking at γ as its root [17], [5].*

Algorithm 1 (introduced in [5]) takes as input the local predicate $L(x_{i-1}, x_i)$ and generates the set of parameterized actions of a self-stabilizing uni-ring protocol. For example, Step 1 takes the local predicate $(|x_{i-1} - x_i| \bmod 2 = 0)$ of \mathcal{I}_{Par} in Parity with domain size 3, and initially generates its locality graph illustrated in Figure 7a. This occurs because there is some γ for which $L(\gamma, \gamma)$ holds. Selecting γ as 0, Algorithm 2 generates the spanning tree of Figure 7b in Step 3 (excluding the labels). Notice that, the output of Algorithm

2 is a spanning tree over the vertices of the locality graph of $L(x_{i-1}, x_i)$ rooted at γ , including a self-loop on γ . Step 4 of Algorithm 1 then includes the arc labels, where a value b becomes a label for an arc (a, c) iff $\neg L(a, b) \wedge (b \neq c)$. For example, when labeling the arc $(0, 0)$ in Figure 7b, $a = 0$, and the algorithm looks for any value b in \mathbb{Z}_3 such that $(|0 - b| \bmod 2) \neq 0$ modulo 3. For $M = 3$, the value $b = 1$ is the only acceptable label.

Algorithm 1. *SynUniRing($L(x_{i-1}, x_i)$): state predicate, M : domain size)*

- 1: Check if a value $\gamma \in \mathbb{Z}_M$ exists such that $L(\gamma, \gamma) = \mathbf{true}$.
- 2: If no such γ exists, then **return** \emptyset and declare that no solution exists.
- 3: $\tau := \mathit{ConstructSpanningTree}(L(x_{i-1}, x_i), M, \gamma)$.
- 4: Transform τ into an action graph of a protocol by the following step:

For each arc (a, c) in τ , where $a, c \in \mathbb{Z}_M$, label (a, c) with every value $b \in \mathbb{Z}_M$ for which $L(a, b) = \mathbf{false}$ and $b \neq c$ hold.

- 5: Return the actions represented by the arcs of τ .

end

Algorithm 2. *ConstructSpanningTree($L(x_{i-1}, x_i)$): state predicate, M : positive integer, $\gamma \in \mathbb{Z}_M$)*

- 1: Construct the locality graph $G = (V, A)$ of $L(x_{i-1}, x_i)$ for domain size M .
- 2: Induce a subgraph $G' = (V', A')$ that contains all arcs of G that participate in cycles involving γ .
- 3: Construct a spanning tree τ rooted at γ for G' . Use backward reachability to construct the spanning tree.
- 4: For each node $v \in G$ that is absent from G' , include an arc from v to the root of τ . The resulting graph would still be a tree, denoted τ' .
- 5: Include a self-loop (γ, γ) at the root of τ' .
- 6: Return τ' .

end

Theorem II.5 explains why Algorithm 2 includes a self-loop at the root γ (in Step 5). Moreover, the reason why Algorithm 1 constructs a spanning tree is to ensure deadlock and livelock-freedom. We have shown [5] that the existence of such a spanning tree is necessary and sufficient for convergence to \mathcal{I} in symmetric uni-rings with constant-space processes.

Definition II.6 (Vector). A vector of dimension $d \geq 1$ of non-negative integers is a tuple $(a_1, a_2, \dots, a_d) \in \mathbb{N}^d$, where $a_i \in \mathbb{N}$ for $1 \leq i \leq d$, and \mathbb{N} denotes the set of non-negative integers.

Definition II.7 (Linear Set). Any non-empty subset of \mathbb{N}^d is *linear* [19] if it can be represented as a periodic set of vectors $\mathcal{L} = \{v_b + \sum_{i=1}^n \lambda_i \cdot p_i : \lambda_i \in \mathbb{N}\}$, $v_b \in \mathbb{N}^d$ is the *base vector* and $\{p_1, \dots, p_n\} \subseteq \mathbb{N}^d$ is a finite set of *period vectors*.

For example, a singleton set $\mathcal{L}_1 = \{(5, 7)\}$ is linear (with dimension $d = 2$) because the base vector is $(5, 7)$, and there

is a unique period vector $(0, 0)$. Moreover, the linear set $\mathcal{L}_2 = \{(3, 2), (4, 3), (5, 4), \dots\}$ has a base vector $(3, 2)$ and a period vector $p_1 = (1, 1)$. That is, $\mathcal{L}_2 = \{v_b + \lambda p_1 : \lambda \in \mathbb{N}\}$, where $v_b = (3, 2)$, $n = 1$, $d = 2$, $p_1 = (1, 1)$, and $\lambda \in \mathbb{N}$.

Definition II.8 (Semilinear Set). A *semilinear* set [19] is a finite union of some linear sets. Semilinear sets provide a finite representation for finite and infinite subsets of \mathbb{N}^d .

Ginsburg and Spanier [20] show that semilinear sets capture the sets of integers that are definable in the first-order theory of integers with addition and order; i.e., Presburger arithmetic. Semilinear sets are closed under Boolean operations [20].

III. SYNTHESIS METHOD

This section first presents a sufficient condition for the existence of a SS-SymU protocol, and then provides a sound algorithm for generating such protocols. We use the Near Agreement (NA) protocol as a running example to ease the presentation of this section.

Problem Statement. We solve Problem II.4 without its assumption of constant-space processes; i.e., processes have unbounded state spaces due to unbounded variable domains.

Example: Near Agreement (NA) Protocol. A node P_i in a ring of N symmetric nodes *nearly agrees* with P_{i-1} iff $(x_{i-1} = x_i) \vee (x_{i-1} = x_i + 1)$, where subtraction is in modulo N and addition is done modulo M . Thus, the entire ring should self-stabilize to $\mathcal{I}_{NA} = \forall i \in \mathbb{N} :: L(x_{i-1}, x_i)$, where $L(x_{i-1}, x_i) \equiv (x_{i-1} = x_i) \vee (x_{i-1} = x_i + 1)$. Figure 3a illustrates the locality graph of $L(x_{i-1}, x_i)$ for $M = 3$. Our objective is to synthesize an NA protocol that is self-stabilizing regardless of the number of processes and the domain size M .

A. Sufficient Condition for Solvability

Since Algorithm 1 is a sound and complete algorithm for any fixed domain size M , one can enumeratively increase the domain size and utilize Algorithm 1 to generate a self-stabilizing protocol for each particular M . However, such an approach would not bear fruit for unbounded domain sizes unless we can ensure that the structure of the spanning tree (and in turn the action graph) that Algorithm 1 generates for M , will be inductively preserved for $M+1$ and beyond. This is a challenge because when the domain size increases to $M+1$, the locality graph of $L(x_{i-1}, x_i)$ may be totally different. For example, observe how the locality graphs in Figures 2a and 3a change when M is increased from 2 to 3 for the NA protocol. To ensure that the spanning tree's structure would be preserved when domain size increases, one approach is to keep the arcs of the spanning tree τ_M for domain size M , and systematically include one more arc (a, a') in τ_M to derive another spanning tree τ_{M+1} for the domain size $M+1$. In turn, expanding the domain of x_i from $M+1$ to $M+2$ should ensure that τ_{M+2} preserves all arcs of τ_{M+1} and includes an additional arc (b, b') through some function f such that $f[(a, a')] = (b, b')$ and $b = M+1$ modulo $M+2$. Moreover, if $f[(b, b')] = (c, c')$ when the domain size increases to $M+3$, then $c - b = b - a$ and $c' - b' = b' - a'$ must hold. That is, the growth of the

spanning tree must be periodic. Moreover, the root remains to be γ . If such conditions are met, then for any domain size M , the conditions of Theorem II.5 hold. Since the vertices of the spanning tree are non-negative integers, each arc (a, b) in a tree is an integer vector. As such, the vector (a, a') would be the base vector of a linear set and $(b - a, b' - a')$ gives the period vector of that linear set. Each one of the arcs in the first tree τ_M for the initial domain size M would also form a finite linear set. Therefore, the arcs of the unbounded spanning tree would form a semilinear set.

Theorem III.1. *Let $\mathcal{I} = \forall i \in \mathbb{N} :: L(x_{i-1}, x_i)$, and let there be a value γ for which $L(\gamma, \gamma)$ holds starting from some domain size M onward. If the arcs of the γ -rooted spanning trees built for each domain size $k \geq M$ represent the periodic growth of a semilinear set, then there is a symmetric uni-ring protocol that self-stabilizes to \mathcal{I} regardless of the ring size and domain size. (Proof is due to Algorithm 3 and its soundness.)*

B. Overview of the Synthesis Method

An implication of Theorem III.1 is that we no longer have a finite spanning tree. Instead, we have an unbounded set of spanning trees τ_0, τ_1, \dots as the domain size M grows. Put it another way, for an unbounded domain size, we have an unbounded spanning tree that has an unbounded branching factor, or an unbounded depth (or both). *How do we formally represent such unbounded structures to facilitate the synthesis of actions?* Theorem III.1 points us to semilinear sets. For example, Algorithm 1 generates the tree in Figure 2b for the NA protocol and domain size 2, whose arcs represent a set of integer vectors $\{(1, 1), (0, 1)\}$. Likewise, the trees in Figures 3b to 5b respectively capture these three sets of integer vectors: $\{(1, 1), (0, 1), (2, 1)\}$, $\{(1, 1), (0, 1), (2, 1), (3, 2)\}$ and $\{(1, 1), (0, 1), (2, 1), (3, 2), (4, 3)\}$ for domain sizes 3 to 5. The vectors $(1, 1)$ and $(0, 1)$ exist in the *intersection* of all four sets and will be there for larger domain sizes too. We call this set of vectors the *common core*, denoted \mathcal{C} . The remaining vectors can be generalized as the linear set $\mathcal{UC} = \{(2, 1), (3, 2), \dots\}$ with the base vector $(2, 1)$ and the period vector $(1, 1)$. We call the linear set \mathcal{UC} the *unbounded core* of the protocol. Since the common core is finite, each vector in it can be represented as a linear set. Thus, we first generate the linear sets of a semilinear set that represents the unbounded spanning tree of a protocol (Figure 1). Then, we synthesize the parameterized action from linear sets.

C. Generating Linear Sets

This section presents an algorithm for the generation of a semilinear set representing the unbounded spanning tree of a protocol. This problem is divided into the formal specifications of the common and unbounded cores of a protocol as linear sets. A tree is acceptable as long as it has a vertex corresponding to each value in a domain size M and its root is a value $\gamma \in \mathbb{Z}_M$ for which $L(\gamma, \gamma)$ holds. Algorithm 3 generates the linear sets of an unbounded tree as Presburger formulas. Naturally, we start with the domain size of 2. Steps 2 and 3 of Algorithm 3 search for a value γ for which $L(\gamma, \gamma)$

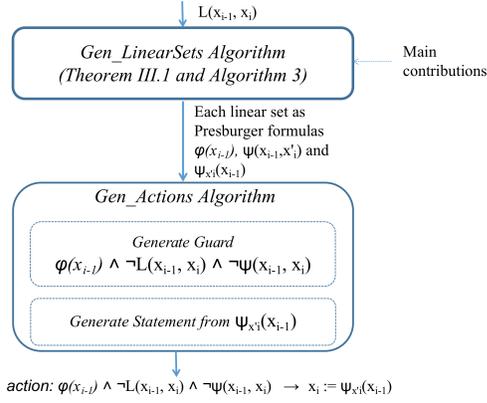


Fig. 1: Overview of the proposed synthesis method.

holds for two consecutive odd and even domain sizes. This search continues up to a preset upper bound \mathcal{B} . Without such an upper bound, the algorithm may never terminate. Step 4 invokes Algorithm 2 for the construction of a spanning tree for M and γ found in Step 3. The common core \mathcal{C} (see Step 5) then includes the integer vectors corresponding to the arcs of the spanning tree τ built in Step 4. After forming the common core, Algorithm 3 increases the domain size in Step 6. Such an increase introduces a new value in the domain of x_i , denoted v_M , which corresponds to a new vertex added to τ . To determine how v_M should be included in the tree, Algorithm 3 identifies the set U of all vertices u for which $L(v_M, u)$ holds. We ignore the arcs $L(u, v_M)$ because connecting any non-leaf node to v_M creates a cycle in the tree. Moreover, connecting a leaf node l to v_M would result in two parents for l . Thus, the only option for connecting v_M to the tree is to include an outgoing arc from v_M to some other tree node. If the set U is empty (Step 7), then v_M is directly connected to the root γ ; i.e., an arc (v_M, γ) is included in τ . In this case, we consider (v_M, γ) as the base vector of a linear set and $(1, 0)$ as the period vector. Such a linear set captures the unbounded growth of the domain size as new arcs connected to the root. That is, the root γ would have an unbounded number of children. If U is non-empty (Step 8), then a value $w \in U$ is randomly selected to be the parent of v_M in the tree; i.e., the arc (v_M, w) is included in the tree. Every time the domain size increases, the value of v_M is incremented. For this reason, the first element of the period vector must be 1. For simplicity, we consider the growth of w in an incremental fashion too. That is, the period vector is $(1, 1)$ and the base vector is (v_M, w) . Overall, Steps 7 and 8 determine the values of the base vector (b, b') and the period vector (p, p') of the unbounded core.

Algorithm 3. $Gen_LinearSets(L(x_{i-1}, x_i):$ state predicate, \mathcal{B} : positive integer)

- 1: $M := 2$.
- 2: If $M \geq \mathcal{B}$ then declare that γ could not be found and exit; // **Upper bound reached**.
- 3: If there is a solution for some value γ where $L(\gamma, \gamma)$ holds modulo M and $M + 1$, then go to Step 4;

otherwise, $M := M + 1$ and go to Step 2.

- 4: $\tau := ConstructSpanningTree(L(x_{i-1}, x_i), M, \gamma)$.
- 5: $\mathcal{C} := S_\tau$ where S_τ represents the set of arcs of τ as a set of integer vectors. // **The common core detected**
- 6: $M' := M + 1$ and let v_M denote the new vertex (i.e., value M modulo M') due to domain size increase. Calculate the set $U = \{u \mid L(v_M, u) \text{ holds}\}$;
- 7: If $U = \emptyset$ then include arc (v_M, γ) every time the domain is increased. Set the base vector to (v_M, γ) , and the period vector to $(1, 0)$. Thus, $(b, b') := (v_M, \gamma)$, and $(p, p') := (1, 0)$. // **Unbounded core \mathcal{UC}** .
- 8: Else select an arc (v_M, w) for some value $w \in U$ as the base vector. Set the base vector to (v_M, w) , and the period vector to $(1, 1)$. Thus, $(b, b') := (v_M, w)$, and $(p, p') := (1, 1)$. // **Unbounded core \mathcal{UC}** .
- 9: For each integer vector $(c, d) \in \mathcal{C}$, return formulas $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = c)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = d)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} d$.
- 10: Corresponding to the unbounded core \mathcal{UC} constructed in Steps 7 and 8, return formulas $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = b + \lambda p)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = x_{i-1} + (b' - b) + \lambda(p' - p))$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} + (b' - b) + \lambda(p' - p))$, where $\lambda \in \mathbb{N}$.

end

Steps 9 and 10 specify the linear sets corresponding to the common core and the unbounded core as Presburger formulas [20]. Each integer vector (a, b) in a linear set actually represents an atomic action of the protocol specified as $x_{i-1} = a \wedge C(x_{i-1}, x_i) \rightarrow x_i := b$, where $C(x_{i-1}, x_i)$ is a Boolean expression specified in terms of x_i and x_{i-1} . Since the second element of each vector (a, b) represents the updated value of x_i , we use the notation x'_i instead of x_i when formally specifying the linear sets of a semilinear set. For example, we specify the linear set $\{(0, 1)\}$ as $x_{i-1} = 0 \wedge x'_i = 1$. Each such formula provides an incomplete sketch of an action, which should be completed in subsequent steps of synthesis. In general, we specify a linear set \mathcal{L} with the base vector (b, b') and the period vector (p, p') as $\{(x_{i-1}, x'_i) \mid \forall \lambda \in \mathbb{N} :: (x_{i-1} = b + \lambda p) \wedge (x'_i = b' + \lambda p')\}$. Since x_{i-1} and x'_i are free variables and λ is known to be a natural value, we eliminate the quantifications in Steps 9 and 10 of Algorithm 3. Let $\mathcal{F}_1 = (x_{i-1} = b + \lambda p)$ and $\mathcal{F}_2 = (x'_i = b' + \lambda p')$. Subtracting \mathcal{F}_1 from \mathcal{F}_2 relates x'_i with x_{i-1} as $\psi(x_i, x'_i) \stackrel{\text{def}}{=} x'_i = x_{i-1} + (b' - b) + \lambda(p' - p)$ (Step 10). Factoring out x'_i , we get $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} + (b' - b) + \lambda(p' - p))$. In fact, $\psi_{x'_i}(x_{i-1})$ represents the expression that should be assigned to x_i in the action corresponding to the linear set \mathcal{L} .

The NA protocol. Figures 2a and 2b respectively represent the locality graph and the spanning tree of NA for $M = 2$. Figures 3 to 5 illustrate the locality graphs and the spanning trees for domain sizes 3 to 5. The semilinear set of the NA protocol can be specified as the union of the following linear sets:

- *Linear set 1:* The base vector is $(1, 1)$, and the period vector is $(0, 0)$. That is, for the unbounded domain M , this set would be equal to $\{(x_{i-1}, x'_i) \mid x_{i-1} = (1 +$

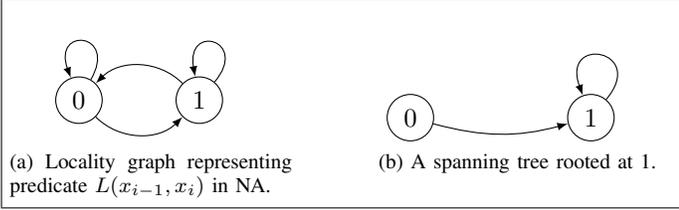


Fig. 2: Locality graph and a spanning tree of NA for $M = 2$.

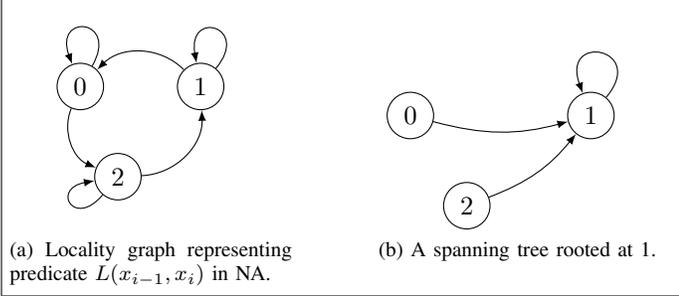


Fig. 3: Locality graph and a spanning tree of NA for $M = 3$.

$\lambda 0) = 1$ and $x'_i = (1 + \lambda 0) = 1$ where $\lambda \in \mathbb{N}$. Since the period vector is $(0, 0)$, this set includes just a single vector; i.e., $\{(1, 1)\}$. Thus, we have $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 1)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 1)$ and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 1$ for this linear set.

- **Linear set 2:** The base vector is $(0, 1)$, and the period vector is $(0, 0)$. Thus, we have $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 0)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 1)$ and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 1$.
- **Linear set 3:** Using the base vector $(2, 1)$, and the period vector $(1, 1)$, this linear set is specified as $\{(x_{i-1}, x'_i) \mid x_{i-1} = 2 + \lambda \text{ and } x'_i = 1 + \lambda \text{ where } \lambda \in \mathbb{N}\}$. Step 10 gives $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 2 + \lambda)$, which means $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} \geq 2)$. Moreover, we have $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = x_{i-1} - 1)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 1)$.

The union of the above linear sets forms a semilinear set that captures the unbounded spanning tree of the NA protocol.

Theorem III.2. *Algorithm 3 terminates and is sound. That is, it correctly generates a semilinear set representing an unbounded spanning tree rooted at γ .*

Proof. Due to space constraint, we provide a proof sketch here and refer interested readers to [21] for the complete proof. The proof of termination follows from the finiteness of the upper bound \mathcal{B} . The proof of soundness includes two parts. First, we show that the common core \mathcal{C} constructed in Step 5 is a finite union of some linear sets. Second, we prove that the union of \mathcal{C} and the unbounded core generated in Steps 7 and 8 is a semilinear set representing an unbounded spanning tree rooted at γ . We show this by induction on M . \square

D. Synthesizing Parameterized Actions from Linear Sets

This section presents a method for the synthesis of parameterized actions of self-stabilizing protocols from linear sets. Each linear set in the semilinear set represents the structure of an individual action in a protocol with deterministic and

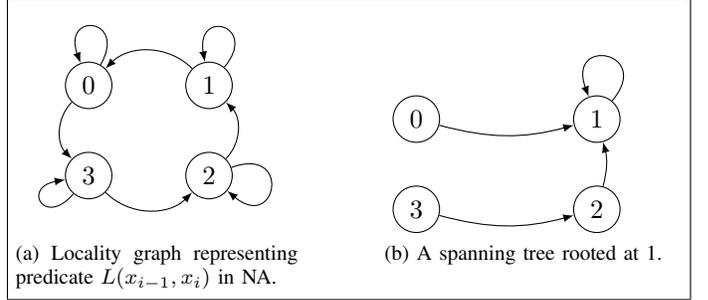


Fig. 4: Locality graph and a spanning tree of NA for $M = 4$.

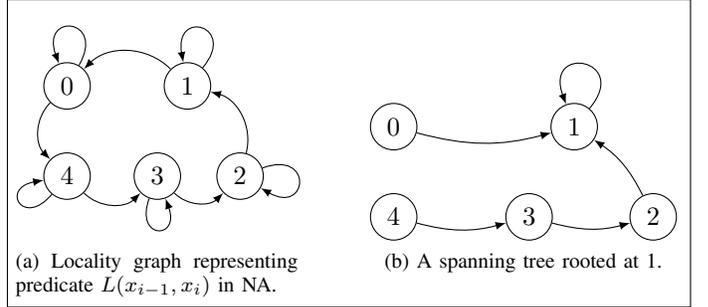


Fig. 5: Locality graph and a spanning tree of NA for $M = 5$.

self-disabling process. However, such a structure lacks details of the guard and statement of each action. Thus, the question is: *how do we synthesize the guard of each action?* and *how do we synthesize the statement of each action?* The guard of each action includes three components: (1) its structure (taken from a linear set); (2) $\neg L(x_{i \ominus 1}, x_i)$, and (3) the self-disabling condition, which is the negation of the statement of the action. Since a linear set contains integer vectors (a, b) where a represents the value that x_{i-1} should have before the value of x_i is updated to b , the first component of a guard includes all values of x_{i-1} that make the formula $\phi(x_{i-1})$ true, and the statement of the guard should make $\psi(x_{i-1}, x'_i)$ true. Moreover, an action is enabled for all values of x_i (in the current state of a process) that make $L(x_{i-1}, x_i)$ false, which is why $\neg L(x_{i-1}, x_i)$ is a part of the guard condition. The statement of the action should make $L(x_{i-1}, x_i)$ true. Moreover, once an action is executed, it should disable itself; i.e., self-disabling assumption. This means that the guard of an action should contain the negation of the expression that holds after the execution of the action. Thus, the third component of a guard is $\neg \psi(x_{i-1}, x_i)$. In the computation of $\psi(x_{i-1}, x_i)$, Algorithm 4 uses the values of x_{i-1} and x_i in the current state of process P_i , before x_i is updated. In summary, the guard of each action would be equal to $\phi(x_{i-1}) \wedge \neg L(x_{i-1}, x_i) \wedge \neg(x_i = \psi_{x'_i}(x_{i-1}))$ (see Algorithm 4). Since x'_i represents the updated value of x_i in $\psi(x_{i-1}, x'_i)$, one can refactor $\psi(x_{i-1}, x'_i)$ in order to generate $\psi_{x'_i}(x_{i-1})$, which denotes $\psi(x_{i-1}, x'_i)$ modulo x'_i . That is, $\psi_{x'_i}(x_{i-1})$ treats x'_i as a function of x_{i-1} . This way, we create the assignment $x_i := \psi_{x'_i}(x_{i-1})$ in Line 2 of Algorithm 4.

Algorithm 4. *Gen_Actions*($\phi(x_{i-1}), \psi(x_{i-1}, x'_i)$): Presburger formula corresponding to a linear set, $L(x_{i-1}, x_i)$: State predicate)

- 1: $\mathcal{G} \stackrel{\text{def}}{=} \phi(x_{i-1}) \wedge \neg L(x_{i-1}, x_i) \wedge (x_i \neq \psi_{x'_i}(x_{i-1}))$
- 2: $\mathcal{A} \stackrel{\text{def}}{=} (x_i := \psi_{x'_i}(x_{i-1}))$
- 3: Return $\mathcal{G} \rightarrow \mathcal{A}$

end

1) *Example: Synthesis of the Actions of the NA Protocol:*

We first demonstrate how we generate the action corresponding to the linear set $(1, 1)$. We take the output of Algorithm 3 for this linear set (i.e., $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 1)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 1)$ and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 1$) and generate its action.

- $\neg L(x_{i-1}, x_i)$: Since $L(x_{i-1}, x_i) = (x_{i-1} = x_i) \vee (x_{i-1} = x_i + 1)$, we include the constraint $(x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1)$ in the guard of this action.
- *Linear set constraint*: This linear set imposes the constraint $\phi(x_{i-1}) \equiv (x_{i-1} = 1)$ on the guard of the action.
- *Self-disabling constraint*: We use $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 1)$ to specify this constraint. To this end, we first determine the assignment of the action using $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 1$. Thus, the assignment is just $x_i := 1$. As a result, the self-disabling constraint is the negation of $x_i = 1$; i.e., $x_i \neq 1$.

Thus, the synthesized action is $(x_{i-1} = 1) \wedge (x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1) \wedge (x_i \neq 1) \rightarrow x_i := 1$. Likewise, the action generated from the linear set $\{(0, 1)\}$ is $(x_{i-1} = 0) \wedge (x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1) \wedge (x_i \neq 1) \rightarrow x_i := 1$. We now generate the action corresponding to the linear set $\{(x_{i-1}, x'_i) \mid x_{i-1} = 2 + \lambda \text{ and } x'_i = 1 + \lambda \text{ where } \lambda \in \mathbb{N}\}$. Corresponding to this unbounded linear set, Algorithm 3 generates $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} \geq 2)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = x_{i-1} - 1)$ and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 1)$. We first synthesize the three components of the guard of this action, and then generate its assignment.

- $\neg L(x_{i-1}, x_i)$: This part is again $(x_{i-1} \neq x_i) \wedge (x_{i-1} \neq x_i + 1)$ for the same reason discussed for the first action.
- *Linear set constraint*: The constraint $\phi(x_{i-1})$ requires that we include $(x_{i-1} \geq 2)$ as part of the guard condition.
- *Self-disabling constraint*: Using $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 1)$, we realize that the assignment of this action establishes the condition $(x_i = x_{i-1} - 1)$. Thus, we include the constraint $(x_i \neq x_{i-1} - 1)$ in the guard, and $x_i := x_{i-1} - 1$ as the assignment of this action.

Putting everything together, we get the following action for this unbounded linear set: $(x_{i-1} \geq 2) \wedge (x_{i-1} \neq x_i) \wedge (x_i \neq x_{i-1} - 1) \rightarrow x_i := x_{i-1} - 1$.

Sample executions. Consider a computation of a ring of four processes for a domain size $M = 4$ (i.e., $x_i \in \mathbb{Z}_4$) starting at the state $s_0 = (\underline{0}, \underline{2}, 1, \underline{3})$, where the underlined values indicate the enabled processes based on the synthesized actions. That is, processes P_0, P_1 and P_3 are enabled. For example, P_0 is enabled because $x_0 = 0 \wedge x_3 = 3$ and the third action is enabled. Using a similar reasoning, one can figure out why P_1 and P_3 are enabled at s_0 . For brevity, we demonstrate a synchronous execution of this ring, but one can extract an asynchronous interleaving of processes that converges to the

same final state. Starting at s_0 , all three enabled processes can execute, where the entire ring transitions to the state $s_1 = \langle \underline{2}, 1, 1, 1 \rangle$, and then reaches the state $s_2 = \langle 1, 1, 1, 1 \rangle$, where everyone agrees with its predecessor. For a domain size $M = 5$ and an arbitrary start state $\langle \underline{0}, \underline{2}, 0, \underline{3} \rangle$, the NA protocol generates the following computation: $\langle \underline{2}, 1, 1, 1 \rangle, \langle 1, 1, 1, 1 \rangle$. As another example, consider a larger ring of five processes and $M = 5$. Starting at $\langle 0, 4, \underline{2}, \underline{3}, 1 \rangle$, the NA protocol will converge through the following states: $\langle \underline{0}, 4, 3, \underline{1}, \underline{2} \rangle, \langle 1, \underline{4}, 3, 2, 1 \rangle, \langle 1, 1, \underline{3}, 2, 1 \rangle, \langle 1, 1, 1, \underline{2}, 1 \rangle, \langle 1, 1, 1, 1, 1 \rangle$. Yet another example includes a case of $M = 7$ and six processes in the ring. Starting at $\langle \underline{6}, \underline{2}, 0, \underline{3}, \underline{6}, \underline{4} \rangle$, the NA protocol has the following converging computation: $\langle \underline{3}, \underline{5}, \underline{1}, 1, \underline{2}, \underline{5} \rangle, \langle \underline{4}, \underline{2}, \underline{4}, \underline{1}, 1, 1 \rangle, \langle 1, \underline{3}, \underline{1}, \underline{3}, 1, 1 \rangle, \langle 1, 1, \underline{2}, 1, 1, 1 \rangle, \langle 1, 1, 1, 1, 1, 1 \rangle$. Observe that, the synthesized NA protocol is self-stabilizing for different ring sizes and domain sizes.

IV. PARITY PROTOCOL

This section demonstrates the synthesis of a Parity protocol, where processes in the uni-ring should converge to an agreed-upon parity starting from any arbitrary state. Formally, the entire ring should self-stabilize to states where $\forall i : i \in \mathbb{N} : (|x_{i-1} - x_i| \bmod 2) = 0$ holds. (Notice that, $|x_{i-1} - x_i| = \max(x_{i-1} - x_i, x_i - x_{i-1})$.) Figures 6 to 9 illustrate how the spanning tree of Parity grows as the domain size increases. The common core is $\{(0, 0), (1, 0), (2, 0)\}$ because $M = 3$ is the first domain size for which there is a solution. We synthesize an action corresponding to each linear set.

- *Linear set 1*: The self-loop on 0 can be represented as a linear set with the base vector $(0, 0)$ and the period vector $(0, 0)$. Algorithm 3 outputs $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 0)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 0)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 0$. Thus, the assignment of the action is $x_i := 0$, and the requirement of having self-disabling actions would be $x_i \neq 0$. The constraint $\neg L(x_{i-1}, x_i)$ provides $(|x_{i-1} - x_i| \bmod 2) \neq 0$. Thus, the synthesized action is $(x_{i-1} = 0) \wedge ((|x_{i-1} - x_i| \bmod 2) \neq 0) \wedge (x_i \neq 0) \rightarrow x_i := 0$.
- *Linear set 2*: The base vector of this linear set is $(1, 0)$ and its period vector is $(0, 0)$. As a result, we have $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 1)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 0)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 0$. The assignment of the action is $x_i := 0$, which leads to the self-disabling constraint $x_i \neq 0$. The constraint $\neg L(x_{i-1}, x_i)$ provides $((|x_{i-1} - x_i| \bmod 2) \neq 0)$. Thus, the synthesized action is $(x_{i-1} = 1) \wedge ((|x_{i-1} - x_i| \bmod 2) \neq 0) \wedge (x_i \neq 0) \rightarrow x_i := 0$.
- *Linear set 3*: The base vector of this linear set is $(2, 0)$ and its period vector is $(0, 0)$. As a result, we have $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 2)$, $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = 0)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} 0$. The assignment of the action is $x_i := 0$, which leads to the self-disabling constraint $x_i \neq 0$. The constraint $\neg L(x_{i-1}, x_i)$ provides $((|x_{i-1} - x_i| \bmod 2) \neq 0)$. Thus, the synthesized action is $(x_{i-1} = 2) \wedge ((|x_{i-1} - x_i| \bmod 2) \neq 0) \wedge (x_i \neq 0) \rightarrow x_i := 0$.
- *Linear set 4*: Using the base vector $(3, 1)$ and the period vector $(1, 1)$, this linear set contains integer vectors $S_4 =$

$\{(x_{i-1}, x'_i) \mid (x_{i-1} = 3 + \lambda) \wedge (x'_i = 1 + \lambda) \text{ where } \lambda \in \mathbb{N}\}$. Algorithm 3 gives us $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} = 3 + \lambda)$, which can be written as $\phi(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} \geq 3)$. Algorithm 3 also outputs $\psi(x_{i-1}, x'_i) \stackrel{\text{def}}{=} (x'_i = x_{i-1} - 2)$, and $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 2)$. The assignment of the action is obtained from $\psi_{x'_i}(x_{i-1}) \stackrel{\text{def}}{=} (x_{i-1} - 2)$, leading to $x_i := x_{i-1} - 2$. Thus, the synthesized action for this linear set is $(x_{i-1} \geq 3) \wedge ((|x_{i-1} - x_i| \bmod 2) \neq 0) \wedge (x_i \neq x_{i-1} - 2) \rightarrow x_i := x_{i-1} - 2$.

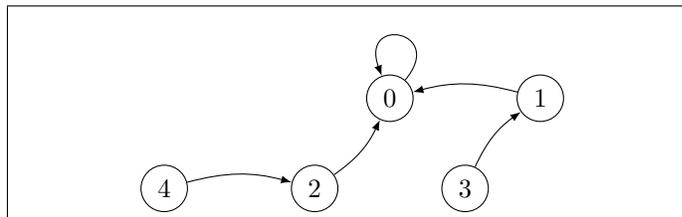


Fig. 9: A spanning tree of the Parity protocol for domain size 5.

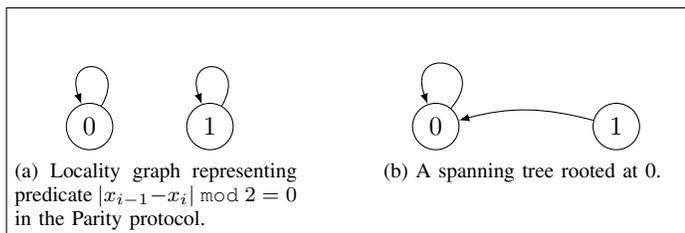


Fig. 6: Locality graph and a spanning tree of the Parity protocol for domain size 2.

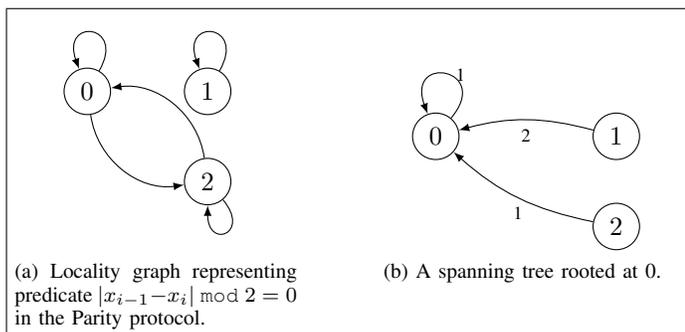


Fig. 7: Locality graph and a spanning tree of the Parity protocol for domain size 3.

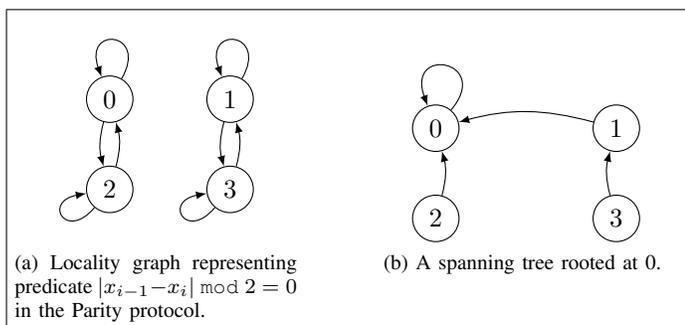


Fig. 8: Locality graph and a spanning tree of the Parity protocol for domain size 4.

V. RELATED WORK

This section discusses the state-of-the-art in the verification and synthesis of parameterized systems, especially unbounded and infinite-state systems. For example, predicate abstraction [22], [23] enables a method for creating a finite-state representation of infinite-state systems where safety properties can

be verified. Constraint language programming [24] enables the verification of safety properties of concurrent systems with unbounded data. Approaches for reachability analysis of generalized Petri nets [25], [26] apply over-approximation towards generating a finite model, and then develop an efficient semi-decision procedure for forward reachability analysis. Counter abstraction [27] utilizes integer counters to count the number of processes in a specific state, but such abstractions are too coarse for the design of self-stabilizing protocols where recovery must be ensured from every concrete state. Environment abstraction [28] extends counter abstraction in order to model the abstract state and the environment of each process. Invisible invariants [29], [30] infer an invariant of a parameterized system by examining a few small instantiations of protocols. Indexed predicates [31] provide a method for the generation and verification of invariant predicates specified in terms of the process indices in infinite-state systems. The aforementioned methods mostly aim at the verification of safety and local liveness properties, and it is unclear how they can synthesize self-stabilizing unbounded protocols.

Most methods for the synthesis of parameterized unbounded systems provide little results for the synthesis of unbounded self-stabilizing protocols, where a global liveness property (i.e., convergence) must be met from *any* state in an unbounded state space. For example, synthesis of Petri nets [32], [33], [34] mainly focuses on the transformation of behavioral specifications in the form of labeled transition systems to Petri nets. UCLID5 [35], [36] provides a framework for modular verification and synthesis of the artifacts (e.g., invariants, assume-guarantee conditions) that are used during verification. Syntax-Guided Synthesis (SyGus) [37] generates the implementation of a set of functions (each adhering to a grammar) in the specification of a system for a background logic theory. It is unclear how one can use SyGus to synthesize the actions of SS-SymU protocols which must interact asynchronously to ensure convergence in a specific topology. Moreover, methods that combine SyGus with reactive synthesis are mostly applied to centralized systems [38]. Oracle-Guided Inductive Synthesis (OGIS) [39], [40], [41] is based on iterative query-response interactions between a learner and a teacher towards synthesizing a system that adheres to formal specifications. Utilizing OGIS in the synthesis of self-stabilizing unbounded systems may not converge to a solution that must recover from any state rather than recovery from a proper set of initial states. While the existing synthesis methods inspire our work, the novelty of

our approach mainly lies in the characterization of unbounded actions as semilinear sets for the synthesis of SS-SymU.

VI. CONCLUSIONS AND FUTURE WORK

This paper investigated the problem of synthesizing self-stabilizing symmetric protocols (SS-SymU) on uni-rings, where a ring can have an unbounded number of processes and processes have unbounded variables. While previous research [5] has addressed this problem for rings of unbounded size, we are not aware of any work that synthesizes self-stabilizing protocols having unbounded variables too. We first showed that the ability to represent unbounded actions of a protocol as semilinear sets is sufficient for synthesis. This reduces the synthesis of SS-SymU to the synthesis of semilinear sets. Then, we presented a sound algorithm that generates a semilinear set for a protocol from which the parameterized actions of the protocol are derived. We demonstrated how our algorithm can generate SS-SymU protocols (e.g., near agreement and parity on unbounded uni-rings) that were previously infeasible. We are currently implementing the proposed method as a synthesizer and are investigating the feasibility of synthesis for more complicated protocols and topologies. We would also like to know how semilinear sets can be utilized for the verification and synthesis of unbounded protocols that satisfy general temporal properties (instead of just self-stabilization).

REFERENCES

- [1] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.
- [2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.
- [3] L. Lamport and N. Lynch, *Handbook of Theoretical Computer Science: Chapter 18, Distributed Computing: Models and Methods*. Elsevier Science Publishers B. V., 1990.
- [4] M. Lazic, I. Konnov, J. Widder, and R. Bloem, "Synthesis of distributed algorithms with parameterized threshold guards," in *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- [5] A. Ebnenasir and A. P. Klinkhamer, "Topology-specific synthesis of self-stabilizing parameterized systems with constant-space processes," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 614–629, 2019.
- [6] A. Klinkhamer and A. Ebnenasir, "On the verification of livelock-freedom and self-stabilization on parameterized rings," *ACM Transactions on Computational Logic*, vol. 20, no. 3, pp. 1–36, 2019.
- [7] N. Mirzaie, F. Faghiih, S. Jacobs, and B. Bonakdarpour, "Parameterized synthesis of self-stabilizing protocols in symmetric networks," *Acta Informatica*, vol. 57, no. 1, pp. 271–304, 2020.
- [8] H. Moloodi, F. Faghiih, and B. Bonakdarpour, "Parameterized distributed synthesis of fault-tolerance using counter abstraction," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2021, pp. 67–77.
- [9] P. C. Attie and E. A. Emerson, "Synthesis of concurrent systems with many similar processes," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 1, pp. 51–115, 1998.
- [10] A. Farahat and A. Ebnenasir, "A lightweight method for automated design of convergence in network protocols," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 4, pp. 38:1–38:36, Dec. 2012.
- [11] F. Faghiih and B. Bonakdarpour, "SMT-based synthesis of distributed self-stabilizing systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2015, to appear.
- [12] A. Klinkhamer and A. Ebnenasir, "Shadow/puppet synthesis: A stepwise method for the design of self-stabilization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3338 – 3350, Feb. 2016.
- [13] R. Bloem, N. Braud-Santoni, and S. Jacobs, "Synthesis of self-stabilising and byzantine-resilient distributed systems," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 157–176.
- [14] S. Jacobs and R. Bloem, "Parameterized synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 362–376.
- [15] B. Finkbeiner and S. Schewe, "Bounded synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 519–539, 2013.
- [16] A. Klinkhamer and A. Ebnenasir, "Verifying livelock freedom on parameterized rings and chains," in *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2013, pp. 163–177.
- [17] A. P. Klinkhamer and A. Ebnenasir, "Synthesizing parameterized self-stabilizing rings with constant-space processes," in *International Conference on Fundamentals of Software Engineering*. Springer, 2017, pp. 100–115.
- [18] A. Klinkhamer and A. Ebnenasir, "Shadow/puppet synthesis: A stepwise method for the design of self-stabilization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3338–3350, 2016.
- [19] R. J. Parikh, "On context-free languages," *Journal of the ACM (JACM)*, vol. 13, no. 4, pp. 570–581, 1966.
- [20] S. Ginsburg and E. H. Spanier, "Bounded algol-like languages," *Transactions of the American Mathematical Society*, vol. 113, no. 2, pp. 333–368, 1964.
- [21] A. Ebnenasir, "Synthesizing self-stabilizing parameterized protocols with unbounded variables," Michigan Technological University, Tech. Rep. CS-TR-22-01, May 2022, <https://www.mtu.edu/cs/research/papers/pdfs/ebnenasir-synthesizing-self-stabilizing-22-01.pdf>.
- [22] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 203–213.
- [23] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in c," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 388–402, 2004.
- [24] G. Delzanno, "An overview of msr (c): A clp-based framework for the symbolic verification of parameterized concurrent systems," *Electronic Notes in Theoretical Computer Science*, vol. 76, pp. 65–82, 2002.
- [25] W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki, "The reachability problem for petri nets is not elementary," *Journal of the ACM (JACM)*, vol. 68, no. 1, pp. 1–28, 2020.
- [26] N. Amat, S. D. Zilio, and T. Hujsa, "Property directed reachability for generalized petri nets," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 505–523.
- [27] A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infty)-counter abstraction," in *International Conference on Computer Aided Verification (CAV)*, 2002, pp. 107–122.
- [28] E. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parameterized verification," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2006, pp. 126–141.
- [29] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2001, pp. 82–97.
- [30] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck, "Liveness with invisible ranking," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 223–238.
- [31] S. K. Lahiri and R. E. Bryant, "Indexed predicate discovery for unbounded system verification," in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 135–147.
- [32] P. Darondeau, "Unbounded petri net synthesis," in *Advanced Course on Petri Nets*. Springer, 2003, pp. 413–438.
- [33] E. Badouel, L. Bernardinello, and P. Darondeau, *Petri net synthesis*. Springer, 2015.
- [34] E. Best and R. Devillers, "Pre-synthesis of petri nets based on prime cycles and distance paths," *Science of Computer Programming*, vol. 157, pp. 41–55, 2018.
- [35] S. A. Seshia and P. Subramanian, "Uclid5: Integrating modeling, verification, synthesis and learning," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2018, pp. 1–10.

- [36] F. Mora, K. Cheang, E. Polgreen, and S. A. Seshia, "Synthesis in uclid5," *arXiv preprint arXiv:2007.06760*, 2020.
- [37] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 2013, pp. 1–17.
- [38] W. Choi, "Can reactive synthesis and syntax-guided synthesis be friends?" in *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2021, pp. 3–5.
- [39] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 215–224.
- [40] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *Acta Informatica*, vol. 54, no. 7, pp. 693–726, 2017.
- [41] E. Polgreen, A. Reynolds, and S. A. Seshia, "Satisfiability and synthesis modulo oracles," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2022, pp. 263–284.

The RAPID Software Verification Framework

Pamina Georgiou^a, Bernhard Gleiss^a, Ahmed Bhayat^b, Michael Rawson^a, Laura Kovács^a, Giles Regeer^b
^a TU Wien, Vienna, Austria

^b University of Manchester, Manchester, United Kingdom

{ pamina.georgiou, bernhard.gleiss, michael.rawson, laura.kovacs }@tuwien.ac.at, { ahmed.bhayat, giles.regeer }@manchester.ac.uk

Abstract—We present the RAPID framework for automatic software verification by applying first-order reasoning in *trace logic*. RAPID establishes partial correctness of programs with loops and arrays by inferring invariants necessary to prove program correctness using a saturation-based automated theorem prover. RAPID can heuristically generate *trace lemmas*, common program properties that guide inductive invariant reasoning. Alternatively, RAPID can exploit nascent support for induction in modern provers to fully automate inductive reasoning without the use of trace lemmas. In addition, RAPID can be used as an invariant generation engine, supplying other verification tools with quantified loop invariants necessary for proving partial program correctness.

I. INTRODUCTION

State-of-the-art deductive verification tools for programs containing inductive data structures ([1], [2], [3], [4], [5]) largely depend on satisfiability modulo theories (SMT) solvers to discharge verification conditions and establish software correctness. These approaches are mostly limited to reasoning over universally-quantified properties in fragments of first-order theories: arrays, integers, etc. In contrast, RAPID supports reasoning with arbitrary quantifiers in full first-order logic with theories [6]. Program semantics and properties are directly encoded in trace logic by quantifying over *timepoints* of program execution. This allows simultaneous reasoning about *sets* of program states, unlike model-checking approaches [2][7]. The gain in expressiveness is beneficial for reasoning about programs with unbounded arrays [6] or to prove security properties [8], for example.

This paper presents what RAPID can do, sketches its design (Section III), and describes its main components and implementation aspects (Sections IV–VII). Experimental evaluation using the SV-COMP benchmark [9] shows RAPID’s efficacy in verification (Section VIII).

Given a program loop annotated with pre/post-conditions, RAPID offers two modes for proving partial program correctness. In the first, RAPID relies on so-called *trace lemmas*, apriori-identified inductive properties that are automatically instantiated for a given program. In the second, RAPID delegates inductive reasoning to the underlying first-order theorem prover [10][11], without instantiating trace lemmas. In either mode, the automated theorem prover used by RAPID is VAMPIRE [12]. RAPID can also synthesize quantified invariants from program semantics, complementing other invariant-generation methods.

```

1 func main() {
2   const Int[] a;
3   const Int alength;
4   Int[] b, c;
5   Int blength, clength, i = 0, 0, 0;
6   while(i < alength) {
7     if(a[i] >= 0) {
8       b[blength] = a[i];
9       blength = blength+1;
10    } else {
11      c[clength] = a[i];
12      clength = clength+1;
13    } i = i+1;
14  }
15 }
```

Fig. 1: Program partitioning an array a into two arrays b , c containing positive and negative elements of a respectively.

Related Work: Verifying programs with unbounded data structures can use model checking for invariant synthesis. Tools like Spacer/Quic3 ([4], [2]), SEAHORN [1] or FREQHORN [7] are based on constrained horn clauses (CHC) and use either fixed-point calculation or sampling/enumerating invariants until a given safety assertion is proved. These approaches use SMT solvers to check validity of invariants and are limited to quantifier-free or universally-quantified invariants. Recurrence solving and data-structure-specific tactics can be used to infer and prove quantified program properties [3]. DIFFY [13] and VAJRA [5] derive relational invariants of two mutations of a program such that inductive properties can be enforced over the entire program, without invariants for each individual loop.

II. MOTIVATING EXAMPLE

We motivate RAPID using the program in Figure 1, written in a standard while-like programming language \mathcal{W} . Each program in \mathcal{W} consists of a single top-level function `main`, with arbitrary nestings of if-then-else and while statements. \mathcal{W} includes optionally-mutable integer (array) variables, and standard side-effect-free expressions over Booleans and integers. Semantics and properties of \mathcal{W} -programs are expressed in *trace logic* \mathcal{L} , an instance of many-sorted first-order logic with theories and equality [6]. A *timepoint* in trace logic is a term of sort \mathbb{L} that refers to a program location. For example, l_5 refers to `line 5` in Figure 1. If a program location occurs in a loop, a timepoint is represented by a function $l : \mathbb{N} \mapsto \mathbb{L}$, where the argument is a natural number representing a loop iteration.

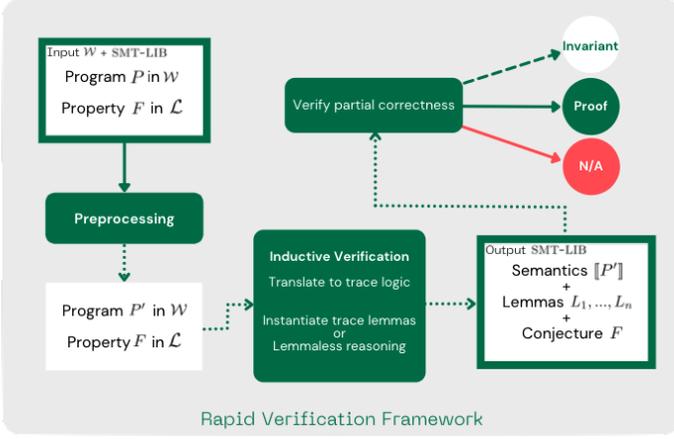


Fig. 2: Overview of the RAPID verification framework.

For example, $l_6(0)$ denotes the first iteration of the loop before entering the loop body. A mutable scalar variable v is modeled as a function over time $v : \mathbb{L} \mapsto \mathbb{I}$. An array variable is modeled as a function $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$, where array indices are represented by integer arguments. For constant variables we omit the timepoint argument. We use a constant $nl_i : \mathbb{N}$ to denote the last iteration of the loop starting at l_i . When a loop is nested within other loops, the last iteration is a *function* over timepoints of all enclosing loops; l_{end} denotes the timepoint after program execution. For Figure 1, $l_6(nl_6)$ denotes the program location of the loop at its last iteration, when the loop condition no longer holds. We assume that programs terminate, and hence RAPID focuses on *partial* correctness.

Figure 1 creates two new arrays, b and c , containing positive and negative elements from the input array a respectively. Note that the arrays are unbounded, and we use the symbolic, non-negative constant $alength$ to bound the length of the input array a . The constraint that $alength$ be non-negative can be expressed within a conjecture (see (1) below for example). A safety property we want to check is that for any position in b there exists a position in a such that both values are equal within the respective array bounds (and similarly for c). This equates to the following conjecture expressed in trace logic¹:

$$\forall pos_{\mathbb{I}}. \exists pos'_{\mathbb{I}}. 0 \leq pos < blength(l_{end}) \wedge alength \geq 0 \rightarrow 0 \leq pos' < alength \wedge b(l_{end}, pos) = a(pos'), \quad (1)$$

To the best of our knowledge other verification approaches cannot automatically validate (1) due to quantifier alternation, but RAPID proves this property for Figure 1.

III. THE RAPID FRAMEWORK

The RAPID framework consists of approximately 10,000 lines of C⁺ ². Figure 2 summarizes the RAPID workflow. Inputs to RAPID are programs P written in \mathcal{W} along with properties F expressed in \mathcal{L} . *Preprocessing* in RAPID applies program transformations for common loop-altering programming con-

¹we write $\forall x_S. F$ or $\exists x_S. F$ to mean that x has sort S in F

²available at <https://github.com/vprover/rapid>

```

1  while(i < alength) {
2      if (a[i] == x) {
3          break;
4      }
5      i = i + 1;
6  }
7
1  Bool break = false;
2  while(i < alength && !break ) {
3      if (a[i] == x) {
4          break = true;
5      }
6      if (!break) {
7          i = i + 1;
8      }
9  }
10

```

Fig. 3: Loop transformation for **break**-statement.

structs, as well as timepoint inlining to obtain a simplified program P' from P (see Section IV).

Next, RAPID performs *inductive verification* (see Section V) by generating the axiomatic semantics $\llbracket P' \rrbracket$ expressed in \mathcal{L} and instantiating a set L_1, \dots, L_n of inductive properties — so-called *trace lemmas* — for the respective program variables of P' . For establishing some property F , RAPID supports two modes of inductive verification: *standard* and *lemmaless* mode. The difference in both versions relates to the underlying support for automating inductive reasoning while proving F . The *standard* verification mode equips the verification task with the trace lemmas L_1, \dots, L_n , providing the necessary induction schemes for proving F . The *lemmaless* verification mode uses built-in inductive reasoning and relies less, or not at all, on trace lemmas. In either mode, the verification tasks of RAPID are encoded in the SMT-LIB format. Finally, a third and recent RAPID mode can be used for invariant generation (see Section VII). In this mode, RAPID “only” outputs quantified invariants using the SMT-LIB syntax; these invariants can further be used by other verification tools.

IV. PREPROCESSING IN RAPID

a) Program Transformations: We use standard program transformations to translate away **break**, **continue** and **return** statements. For these, RAPID introduces fresh Boolean program variables indicating whether a statement has been executed. The program is adjusted accordingly: **return** statements end program execution; **break** statements invalidate the first enclosing loop condition; and for **continue** the remaining code of the first enclosing loop body is not executed. **Example 1:** Figure 3 shows a standard transformation for a **break**-statement.

b) Timepoint Inlining: RAPID uses SSA-style inlining [14], [15], [16] for timepoints to simplify axiomatic program semantics and trace lemmas of a verification task. Specifically, RAPID caches (i) for each integer variable the current program

```

1  a = a + 2;
2  b = 3;
3  c = a + b;
4
5  assert (a(lend) < c(lend))

```

(a) block assignments

```

1  if (x < 1) {
2    x = 0;
3  } else {
4    skip;
5  }
6  while (y > 0) {
7    y = y - 1;
8  }
9
10 assert (x(lend) ≥ 0)

```

(b) simple branching

Fig. 4

expression assigned to it, and (ii) for each integer-array variable the last timepoint where it was assigned. Cached values are used during traversal of the program tree to simplify later program expressions. Thus we avoid defining irrelevant equalities of program variable values over unused timepoints, and only reference timepoints relevant to the property. We illustrate this on two examples:

Example 2 (Inlining assigned integer expressions): The effect of inlined semantics can be observed when we encounter block assignments to integer variables: we can skip assignments and use the last assigned expression directly in any reference to the original program variable. Consider the partial program in Figure 4a. Our axiomatic semantics in trace logic [6] would result in

$$\begin{aligned}
a(l_2) = a(l_1) + 2 & \quad \wedge \quad b(l_2) = b(l_1) & \quad \wedge \\
c(l_2) = c(l_1) & \quad \wedge \quad a(l_3) = a(l_2) & \quad \wedge \\
b(l_3) = 3 & \quad \wedge \quad c(l_3) = c(l_2) & \quad \wedge \\
a(l_{end}) = a(l_3) & \quad \wedge \quad b(l_{end}) = b(l_3) & \quad \wedge \\
c(l_{end}) = a(l_3) + b(l_3) & &
\end{aligned}$$

whereas the inlined version of semantics is drastically shorter:

$$a(l_{end}) = a(l_1) + 2 \quad \wedge \quad c(l_{end}) = (a(l_1) + 2) + 3.$$

In contrast to the extended semantics that define all program variables for each timepoint, the inlined version only considers the values of referenced program variables at the timepoint of their last assignment. Thus, when c is defined, RAPID directly references the (symbolic) values assigned to a and b . While b is not defined at all, note that a is defined as $a(l_{end})$ is referenced in the conjecture. Furthermore, the inlined semantics only make use of two timepoints, l_1 , and l_{end} , as the remaining timepoints are irrelevant to the conjecture.

Example 3 (Inlining equalities with branching.): Figure 4b shows another program that benefits from inlining equalities,

as well as only considering timepoints relevant to the conjecture. The original semantics defines program variables x and y for all program locations: $l_1, l_2, l_3, l_4, l_6(it), l_6(nl_6), l_{end}$, for some iteration it and final iteration nl_6 . While the program contains two variables x and y , only x is used in the property we want to prove. Since no assignments to x contain references to y , the loop semantics do not interfere with x , so we have

$$\begin{aligned}
x(l_3) < 1 & \rightarrow x(l_6(0)) = 0 & \quad \wedge \\
x(l_3) \geq 1 & \rightarrow x(l_6(0)) = x(l_3) & \quad \wedge \\
x(l_{end}) & = x(l_6(0))
\end{aligned}$$

where the semantics of the loop defining y are omitted. Note that all timepoints of the if-then-else statements are flattened into the timepoint at the beginning of the loop at l_6 in iteration 0. The axiomatic semantics thus reduce to three conjuncts defining the value of x throughout the execution. However, x is not defined in any loop iteration other than the first as they are irrelevant to the property.

c) User-defined input: RAPID is fully automated. However, it may still benefit from manually-defined invariants to support the prover. Users can therefore extend the input to RAPID with first-order axioms written in the SMT-LIB format.

V. INDUCTIVE VERIFICATION IN RAPID

As mentioned above, RAPID implements two verification modes; in the default *standard* mode, RAPID uses trace lemmas to prove inductive properties of programs. In its *lemmaless* mode RAPID relies on built-in induction support in saturation-based first-order theorem proving. In this section we elaborate on both modes further.

A. Standard Verification Mode: Reasoning with Trace Lemmas

RAPID's *standard* mode relies on trace lemma reasoning to automate inductive reasoning. Trace lemmas are sound formulas that are: (i) derived from bounded induction over loop iterations; (ii) represent common inductive program properties for a set of similar input programs; and (iii) are automatically instantiated for all relevant program variables of a specific input program during its translation to trace logic; see [6].

In all of our experiments from Section VIII, including the example from Figure 1, we only instantiate three generic inductive trace lemmas to establish partial correctness. One such trace lemma asserts, for example, that a program variable is not mutated after a certain execution timepoint.

Example 4: Consider the safety assertion (1) of our running example from Figure 1. In its standard verification mode, RAPID proves correctness of (1) by using, among others, the following trace lemma instance

$$\begin{aligned}
& \forall j_{\mathbb{I}}. \forall b_{L\mathbb{N}}. \forall b_{R\mathbb{N}}. \left(\right. \\
& \quad \forall it_{\mathbb{N}}. \left((b_L \leq it < b_R \wedge b(l_9(b_L), j) = b(l_9(it), j)) \right. \\
& \quad \quad \left. \rightarrow b(l_9(b_L), j) = b(l_9(s(it), j)) \right) \\
& \quad \left. \rightarrow (b_L \leq b_R \rightarrow b(l_9(b_L), j) = b(l_9(b_R), j)) \right)
\end{aligned}$$

stating that the value of b at some position j is unchanged between two bounds b_L and b_R if, for any iteration it and its successor $s(it)$, values of b are unchanged.

Multitrace Generalization: RAPID can also be used to prove k -safety properties over k traces, useful for security-related hyperproperties such as non-interference and sensitivity [8]. For such problems it is sufficient to extend program variables to functions over time and trace, such that program variables are represented as $(\mathbb{L} \times \mathbb{T} \mapsto \mathbb{I})$. Program locations, and hence timepoints, are similarly parameterized by an argument of sort \mathbb{T} to denote the same timepoint in different executions.

B. Lemmaless Verification Mode

When in *lemmaless* mode RAPID does not add any trace lemma to its verification task but relies on first-order theorem proving to derive inductive loop properties. An extended version of SMT-LIB (see Section VI) is used to provide the underlying prover with additional information to guide the search for necessary inductive schemes, such as likely symbols for induction. We further equip saturation-based theorem proving with two new inference rules that enable induction on such terms; see [17] for details. *Multi-clause goal induction* takes a formula derived from a safety assertion that contains a final loop counter, that is a symbol denoting last loop iterations, and inserts an instance of the induction schema for natural numbers with the negation of this formula as its conclusion into the proof search space. For example, consider the formula $x(l_5(nl_5)) < 0$. Multi-clause goal induction introduces the induction hypothesis $x(l_5(0)) \geq 0 \wedge \forall it_{\mathbb{N}}. (it < nl_5 \wedge x(l_5(it)) \geq 0) \rightarrow x(l_5(s(it))) \geq 0 \rightarrow x(l_5(nl_5)) \geq 0$. If the base and step cases can be discharged, a contradiction can be easily produced from the conclusion and original clause.

Array mapping induction also introduces an instance of the induction schema to the search space, but is not based on formulas derived from the goal. Instead, this rule uses clauses derived from program semantics to generate a suitable conclusion for the induction hypothesis.

VI. VERIFYING PARTIAL CORRECTNESS IN RAPID

For proving the verification tasks of Section V, and thus verifying partial program correctness, RAPID relies on saturation-based first-order theorem proving. To this end, each verification mode of RAPID uses the VAMPIRE prover, for which we implemented the following, RAPID-specific adjustments.

a) *Extending SMT-LIB:* Each verification task of RAPID is expressed in extensions of SMT-LIB, allowing us to treat some terms and definitions in a special way during proof search:

- (i) `declare-nat`: The VAMPIRE prover has been extended with an axiomatization of the natural numbers as a term algebra, especially for RAPID-style verification purposes. We use the command `(declare-nat Nat zero s p Sub)` to declare the sort `Nat`, with constructors `zero` and successor `s`, predecessor `p` and ordering relation `Sub`.
- (ii) `declare-lemma-predicate`: Our trace lemmas are usually of the form $(P_1 \wedge \dots \wedge P_n) \rightarrow Conclusion_L$ for some trace lemma L with premises $P_1 \wedge \dots \wedge P_n$. In terms

of reasoning, it makes sense for the prover to derive the premises of such a lemma before using its conclusion to derive more facts, as we have many automatically instantiated lemmas of which we can only prove the premises of some from the semantics. To enforce this, we adapt literal selection such that inferences from premises are preferred over inferences from conclusions. Lemmas are split into two clauses $(P_1 \wedge \dots \wedge P_n) \rightarrow Premise_L$ and $Premise_L \rightarrow Conclusion_L$, where $Premise_L$ is declared as a *lemma literal*. We ensure our literal selection function selects either a negative lemma literal³ if available, or a positive lemma literal only in combination with another literal, requiring the prover to resolve premises before using the conclusion.

The *lemmaless* mode of RAPID introduces the following additional declarations to SMT-LIB:

- (i) `declare-const-var`: assign symbols representing constant program variables a large weight in the prover's term ordering, allowing constant variables to be rewritten to non-constant expressions.
- (ii) `declare-timepoint`: distinguish a symbol representing a timepoint from program variables, guiding VAMPIRE to apply induction upon timepoints.
- (iii) `declare-final-loop-count`: declare a symbol as a final loop count symbol, eligible for induction.

b) *Portfolio Modes:* We further developed a collection of RAPID-specific proof options in VAMPIRE, using for example extensions of theory split queues [18] and equality-based rewritings [19]. Such options have been distilled into a RAPID portfolio schedule that can be run with `--mode portfolio -sched rapid`. Moreover, the multi-clause goal induction rule and the array mapping induction inference of RAPID have been compiled to a separate portfolio mode, accessed via `--mode portfolio -sched induction_rapid`.

VII. INVARIANT GENERATION WITH RAPID

RAPID can also be used as an invariant generation engine, synthesizing first-order invariants using the VAMPIRE theorem prover. To do so, we use a special mode of VAMPIRE to derive logical consequences of the semantics produced by RAPID. Some of these consequences may be loop invariants. The *symbol elimination* approach of [20] defined some set of program symbols undesirable, and only reports consequences that have *eliminated* such symbols from their predecessors. In RAPID, we adjust symbol elimination for deriving invariants in trace logic using VAMPIRE. These invariants may contain quantifier alternations, and some conjunction of them may well be enough to help other verification tools show some property. When RAPID is in *invariant generation* mode, the encoding of the problem is optimized for invariant generation. We limit trace lemmas to more specific versions of the bounded induction scheme. We also remove RAPID-specific symbols such as lemma literals so that they do not appear in consequences.

³Note that lemma literals become negative in the premise definition after CNF-transformation.

Symbol Elimination: Loop invariants should only contain symbols from the input loop language, with no timepoints. To remove such constructs, we apply symbol elimination: any symbol representing a variable v used on the left-hand side of an assignment is eliminated. However, we still want to generate invariants containing otherwise-eliminated variables at specific locations, so for each eliminated variable v we define $v_init = v(l_1)$ and $v_final = v(l_2)$ for appropriate locations l_1, l_2 : these new symbols need not be eliminated.

We further adjusted symbol elimination in RAPID to output fully-simplified consequences during proof search in VAMPIRE (the so-called *active set* [12]) at the end of a user-specified time limit. Consequences that contain undesirable symbols or are pure consequences of theories are removed at this stage.

Reasoning with Integers vs. Naturals: In the standard setting, RAPID uses natural numbers (internally Nat) to describe loop iterations. However, in some situations it is advantageous to use the theory of integers: loop counter variable i of sort \mathbb{I} will have the same numerical value as nl of sort \mathbb{N} at the end of a loop. Integer-based timepoints allow deriving $i(l(nl)) = nl$. Such a clause can be very helpful for invariant generation, as shown in Example 5.

Example 5: Consider the property $\forall x. 0 \leq x \leq \text{alength} \rightarrow a(x) = b(x)$. The property essentially requires us to prove that two arrays a, b are equal in all positions between 0 and alength . Such a property might for example be useful to prove when we copy from an array b into array a in a loop with loop condition $i < \text{alength}$ where i is the loop counter variable incremented by one in each iteration. Now when we run RAPID in the invariant generation mode, we might be able to derive a property $\forall x. 0 \leq x \leq nl \rightarrow a(x) = b(x)$, essentially stating that the property holds for all iterations of the loop. The prover can further easily deduce that $i(l(nl)) \geq \text{alength}$ thanks to our semantics.

However, in case of natural numbers we cannot deduce that $i(l(nl)) = nl$ since the sorts of i and nl differ. In order to derive an invariant strong enough to prove the postcondition we depend upon the prover to find the invariant $\forall x. 0 \leq x \leq i(l(nl)) \rightarrow a(x) = b(x)$ directly which cannot be deduced by the prover as our loop semantics are bounded by loop iterations rather than the loop counter values.

When using `-integerIterations` on we can circumvent this problem as the prover can then simply deduce the equality $i(l(nl)) = nl$ which makes the conjunction of clauses strong enough to prove the desired postcondition.

VIII. EXPERIMENTAL EVALUATION

We evaluated the two verification modes of RAPID and compare against the state-of-the-art solvers DIFFY and SEAHORN, as summarized below.

Benchmark Selection: Our benchmarks⁴ are based on the `c/ReachSafety-Array` category of the SV-COMP repository [21], specifically from the `array-examples/*` subcategory⁵ as it contains problems suitable for our input language.

⁴<https://github.com/vprover/rapid/tree/main/examples/arrays>

⁵<https://github.com/sosy-lab/sv-benchmarks/tree/master/c/array-examples>

TABLE I: Experimental Results

Total	RAPID _{std}	RAPID _{lemmaless}	DIFFY	SEAHORN
140	91 (5)	103 (10)	61 (1)	17 (0)

Other examples are not yet expressible in \mathcal{W} due to the presence of function calls and/or unsupported memory access constructs. We manually translate all programs to \mathcal{W} and express pre/post-conditions as trace logic properties. Additionally, we extend some SV-COMP examples with new conjectures containing existential and alternating quantification.

In general SV-COMP benchmarks are bounded to a certain array size N . By contrast, we treat arrays as unbounded in RAPID and reason using symbolic array lengths. Some benchmarks in the original SV-COMP repository are minor variations of each other that differ only in one concrete integer value, e.g to increment a program variable by some integer. Instead of copying each such variation for different digits, we abstract such constant values to a single symbolic integer constant such that just one of our benchmark covers numerous cases in the original SV-COMP setup.

Results: We compare our two RAPID verification modes, indicated by RAPID_{std} and RAPID_{lemmaless} respectively, against SEAHORN and DIFFY. All experiments were run on a cluster with two 2.5GHz 32-core CPUs with a 60-seconds timeout. Note that DIFFY produced the same results as its precursor VAJRA in this experiment. Table I summarizes our results, parentheticals indicating uniquely solved problems. Of a total of 140 benchmarks, RAPID_{std} solves 91 problems, while RAPID_{lemmaless} surpasses this by 12 problems. Particularly, RAPID_{lemmaless} could solve more variations with quantifier alternations of our running example 1, as property-driven induction works well for such problems. A small number of instances, however, was solved by RAPID_{std} but not by RAPID_{lemmaless} within the time limit, indicating that trace lemma reasoning can help to fast-forward proof search. In total, RAPID solves 112 benchmarks, whereas SEAHORN and DIFFY could respectively prove 17 and 61 problems (with mostly universally quantified properties). For more detailed experimental data on subsets of these benchmarks we refer to [6], [17].

IX. CONCLUSION

We described the RAPID verification framework for proving partial correctness of programs containing loops and arrays, and its applications towards efficient inductive reasoning and invariant generation. Extending RAPID with function calls, and automation thereof, is an interesting task for future work.

Acknowledgements. This research was partially supported by the ERC consolidator grant ARTIST 101002685, the FWF research project LogiCS W1255-N23, the TU Wien SecInt doctoral program, and the EUProofNet Cost Action CA20111. Our research was partially funded by the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

REFERENCES

- [1] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The SeaHorn verification framework,” in *CAV*, 2015, pp. 343–361.
- [2] A. Gurfinkel, S. Shoham, and Y. Vizek, “Quantifiers on demand,” in *ATVA*, 2018, pp. 248–266.
- [3] P. Rajkhowa and F. Lin, “Extending viap to handle array programs,” in *VSTTE*, 2018, pp. 38–49.
- [4] H. G. V. Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, “Global guidance for local generalization in model checking,” in *CAV*. Springer, 2020, pp. 101–125.
- [5] S. Chakraborty, A. Gupta, and D. Unadkat, “Verifying array manipulating programs with full-program induction,” in *TACAS*, 2020, pp. 22–39.
- [6] P. Georgiou, B. Gleiss, and L. Kovács, “Trace logic for inductive loop reasoning,” in *FMCAD*. IEEE, 2020, pp. 255–263.
- [7] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, “Quantified invariants via syntax-guided synthesis,” in *CAV*, 2019, pp. 259–277.
- [8] G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei, “Verifying relational properties using trace logic,” in *FMCAD*, 2019, pp. 170–178.
- [9] D. Beyer, “Software verification: 10th comparative evaluation (SV-COMP 2021),” in *TACAS*, 2021, pp. 401–422.
- [10] P. Hozzová, L. Kovács, and A. Voronkov, “Integer induction in saturation,” in *CADE*, 2021, pp. 361–377.
- [11] G. Reger and A. Voronkov, “Induction in saturation-based proof search,” in *CADE*, 2019, pp. 477–494.
- [12] L. Kovács and A. Voronkov, “First-Order Theorem Proving and Vampire,” in *CAV*, 2013, pp. 1–35.
- [13] S. Chakraborty, A. Gupta, and D. Unadkat, “Diffy: Inductive reasoning of array programs using difference invariants,” in *CAV*, 2021.
- [14] P. Briggs and K. D. Cooper, “Effective partial redundancy elimination,” *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 159–170, 1994.
- [15] A. W. Appel, “SSA is functional programming,” *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, 1998.
- [16] —, *Modern compiler implementation in C*. Cambridge university press, 2004.
- [17] A. Bhayat, P. Georgiou, C. Eisenhofer, L. Kovács, and G. Reger, “Lemmaless induction in trace logic,” Preprint, https://github.com/vprover/vampire_publications/blob/master/paper_drafts/rapid_induction.pdf.
- [18] B. Gleiss and M. Suda, “Layered clause selection for theory reasoning,” in *IJCAR*, 2020, pp. 297–315.
- [19] B. Gleiss, L. Kovács, and J. Rath, “Subsumption demodulation in first-order theorem proving,” in *IJCAR*, 2020, pp. 297–315.
- [20] L. Kovács and A. Voronkov, “Finding loop invariants for programs over arrays using a theorem prover,” in *FASE*, 2009, pp. 470–485.
- [21] SV-COMP. <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks>.

ACORN: Network Control Plane Abstraction using Route Nondeterminism

Divya Raghunathan
Princeton University
Princeton, USA
dr31@cs.princeton.edu

Ryan Beckett
Microsoft Research
Redmond, USA
ryan.beckett@microsoft.com

Aarti Gupta
Princeton University
Princeton, USA
aartig@cs.princeton.edu

David Walker
Princeton University
Princeton, USA
dpw@cs.princeton.edu

Abstract—Networks are hard to configure correctly, and misconfigurations occur frequently, leading to outages or security breaches. Formal verification techniques have been applied to guarantee the correctness of network configurations, thereby improving network reliability. This work addresses verification of distributed network control planes, with two distinct contributions to improve the scalability of verification. Our first contribution is a hierarchy of abstractions of varying precision which introduce nondeterminism into the procedure that routers use to select the best available route. We prove the soundness of these abstractions and show their benefits. Our second contribution is a novel SMT encoding which uses symbolic graphs to encode all possible stable routing trees that are compliant with the given network control plane configurations. We have implemented our abstractions and SMT encoding in a prototype tool called ACORN. Our evaluations show that our abstractions can provide significant relative speedups (up to 323x) in performance, and ACORN can scale up to $\approx 37,000$ routers in data center benchmarks (with FatTree topologies, running shortest-path routing and valley-free policies) for verifying reachability. This far exceeds the performance of existing control plane verifiers.

I. INTRODUCTION

Bugs in configuring networks can lead to expensive outages or critical security breaches, and misconfigurations occur frequently [1], [2], [3], [4], [5], [6]. Thus, there has been great interest in formal verification of computer network configurations. Many initial efforts targeted the network *data plane*, *i.e.*, the forwarding rules in each router that determine how a given packet is forwarded to a destination. Many of these methods have been successfully applied in large data centers in practice [7], [8], [9]. In comparison, formal verification of the network *control plane* is more challenging.

Traditional control planes use distributed protocols such as OSPF, BGP, and RIP [10] to compute a network data plane based on the route announcements received from peer networks, the current failures detected, and the router configurations. In control plane verification, one must check that *all* data planes that emerge due to the router configurations are correct. There has been much recent progress in control plane verification. Fully symbolic SMT-based verifiers [11], [12], [13] usually work well for small-sized networks, but have not been shown to scale to medium-to-large networks. Simulation-based verifiers [14], [15], [16], [13], [17], [18] scale better, but in general, do not provide full symbolic reasoning, *e.g.*, for considering all external route announcements. Our work

is motivated by this gap: we aim to provide full symbolic reasoning *and* improve the scalability of verification. We address this challenge with two main contributions – a novel hierarchy of control plane abstractions, and a new symbolic graph-based SMT encoding for control plane verification.

Hierarchy of nondeterministic abstractions. Our novel control plane abstractions introduce nondeterminism in the procedure that routers use to select a route – we call these the *Nondeterministic Routing Choice (NRC) abstractions*. Instead of forcing a router to pick the *best* available route, we allow it to *nondeterministically choose* a route from a subset of available routes which includes the best route. The number of non-best routes in this set determines the precision of the abstraction; our least precise abstraction corresponds to picking *any* available route that is compliant with policy.

Our main insight here is that determining the best route may not be needed for verification of many correctness properties that network operators care about, such as reachability (*e.g.*, when the number of hops may not matter), valley-freedom, or no-transit (Gao-Rexford conditions [19]). On the other hand, for policy-based routing, it is still important to model other protocol features such as route filters. Our results show, for the first time, that nondeterministic routing abstractions can successfully verify such properties and provide significant gains in performance and scalability. Although some other efforts [12], [20] have also proposed to abstract the decision process in BGP (details in §VII), we elucidate and study the general principle for generic distributed protocols, prove it sound, and reveal a range of precision-cost tradeoffs.

The potential downside of considering non-best routes is that our abstractions may lead to false positives, *i.e.*, we could report property violations although the *best* route may actually satisfy the property. In such cases, we propose using a more precise abstraction that models more of the route selection procedure. Our experiments (§VI) demonstrate that the NRC abstractions can successfully verify a wide range of networks and common policies and offer significant performance and scalability benefits in symbolic SMT-based verification. Although our abstractions are sound for verification of specified failures (§IV), we focus on verification without failures here, and plan to consider failures in future work.

Symbolic graph-based SMT encoding. Our novel SMT

encoding uses *symbolic graphs* [21] (where a Boolean variable is associated with each edge in the network topology) to model the stable states of a network control plane. Our encoding can leverage specialized SMT solvers such as MonoSAT [21] that provide support for graph-based reasoning, as well as standard SMT solvers such as Z3 [22].

Experimental evaluation. We have implemented our NRC abstractions and symbolic graph-based SMT encoding in a prototype tool called ACORN (Abstracting the Control plane using Route Nondeterminism). We present a detailed evaluation on benchmark examples that include synthetic data center examples with FatTree topologies [23], as well as real topologies from Topology Zoo [24] and BGPStream [25] running well-known network policies, where we verify reachability and other properties of interest. All benchmark examples are successfully verified using an NRC abstraction (96% of examples with our least precise abstraction, and the remaining 4% using a more precise abstraction). These benchmarks, including some new examples that we created, are publicly available [26]. ACORN could verify reachability in large FatTree benchmarks with about 37,000 nodes (running common policies) within an hour. This kind of scalability is needed in modern data centers with tens of thousands of routers that run distributed routing protocols such as BGP [27]. We compared ACORN with two publicly available state-of-the-art control plane verifiers on the data center benchmarks, and our results show that our tool scales an order of magnitude better.

To summarize, we make the following contributions:

- 1) We present a hierarchy of novel control plane abstractions, called the NRC abstractions, that add nondeterminism to a general route selection procedure (§IV). We prove our abstractions sound and empirically show that they enable a precision-cost tradeoff in verification. Although our focus is on SMT-based verification, these abstractions could be used with other methods as well.
- 2) We present a novel SMT encoding (§V) (based on symbolic graphs [21]) to capture distributed control plane behavior. This leverages SMT solvers that support graph-based reasoning, as well as standard SMT solvers.
- 3) We implemented our abstractions and SMT encoding in a prototype tool called ACORN and present a detailed evaluation (§VI) on synthetic data center benchmarks and real-world topologies with well-known network policies.

II. MOTIVATING EXAMPLES

In a distributed routing protocol, routers exchange *route announcements* containing information on how to reach various destinations. On receiving a route announcement, a router updates its internal state and sends a route announcement to neighboring routers after processing it as per the routing configurations. In well-behaved networks, this distributed decision process converges to a *stable state* [28] in which the internal routing information of each router does not change upon receiving additional route announcements. The best route selected by each router defines a *routing tree*: if router u selects

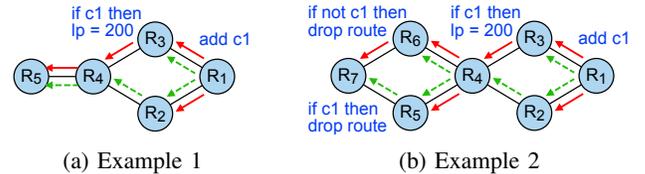


Fig. 1: Examples showing correct verification result with an NRC abstraction. Red arrows show the routing tree in the real network, and green arrows show an additional routing tree allowed in the abstraction.

the route announcement sent by router v for destination d , then u will forward data packets with destination d to v .

Example 1 (Motivating example). Consider the network in Figure 1a (from ShapeShifter [16]) with five routers running the Border Gateway Protocol (BGP), described in Appendix A, where actions taken by routers are shown along the edges.

The verification task is to check whether routes announced at R_1 can reach R_5 . The network uses the BGP community attribute, a list of string tags, to ensure that R_4 prefers to route through R_3 : the community tag $c1$ is added along the edge (R_1, R_3) , which causes the local preference (lp) to be set to 200 along the edge (R_3, R_4) . Routes with higher local preference are preferred (the default local preference is 100). Thus, the best route at R_4 is through R_3 and the corresponding routing tree is shown by red (solid) arrows.

Note that R_5 can receive a route even if R_4 chooses to route through R_2 instead, though this route is not the best for R_4 . Thus, R_5 can reach the destination regardless of the choice R_4 makes. This observation captures the basic idea in our NRC abstractions—intuitively, we explore *multiple* available routes at a node: the best route as well as other routes. Then we check if R_5 receives a route under *each of these possibilities*. Since R_5 can reach R_1 in all routing trees considered by our abstraction we correctly conclude that it can reach R_1 .

False positives and refinement. The NRC abstractions are sound, *i.e.*, when verification with an abstraction is successful, the property is guaranteed to hold in the network. However, verification with an abstraction could report a false positive, *i.e.*, a property violation even when the network satisfies the property. In Figure 1a, suppose R_5 drops routes without the tag $c1$. In the real network, R_5 will receive a route, since the route sent by R_4 has the tag $c1$. However, verification with an abstraction that considers all possible routes would report that R_5 *cannot* reach the destination, with a counterexample where R_4 routes through R_2 and its route announcement is dropped by R_5 . Here, an NRC abstraction higher up in the precision hierarchy, *e.g.*, one which chooses a route with maximum local preference and minimum path length, will verify that R_5 receives a route, thereby eliminating the false positive.

Path-sensitive reasoning. Even our least precise abstraction can verify many interesting policies due to our symbolic SMT-based approach which tracks correlations between choices made at different routers, which other tools [16] do not track.

<p>SRP instance: $SRP = (G, A, a_d, \prec, \text{trans})$, $G = (V, E, d)$</p> <p>SRP solution: $\mathcal{L} : V \rightarrow A_\infty$</p> $\mathcal{L}(u) = \begin{cases} a_d & \text{if } u = d \\ \infty & \text{if } \text{attrs}_{\mathcal{L}}(u) = \emptyset \\ a \in \text{attrs}_{\mathcal{L}}(u), \text{ minimal by } \prec & \text{if } \text{attrs}_{\mathcal{L}}(u) \neq \emptyset \end{cases}$ <p>$\text{attrs}_{\mathcal{L}}(u) = \{a \mid (e, a) \in \text{choices}_{\mathcal{L}}(u)\}$</p> <p>$\text{choices}_{\mathcal{L}}(u) = \{(e, a) \mid e = (v, u),$ $a = \text{trans}(e, \mathcal{L}(v)), a \neq \infty\}$</p>

Fig. 2: Cheat sheet for SRP [30].

Example 2 (Path-sensitivity). Figure 1b shows another BGP network (from Propane [29]), with seven routers and destination R_1 . We would like to verify that R_7 can reach R_1 .

In the real network, R_4 chooses the route from R_3 which has higher local preference (as shown by red/solid arrows). Under the least precise NRC abstraction, R_4 could choose the route from R_2 instead. Regardless of R_4 's choice, the community tags in the routes received by R_5 and R_6 are the same, and so R_7 will receive a route either way – our abstraction tracks this correlation and correctly concludes that R_7 can reach R_1 .

III. PRELIMINARIES

In this section we briefly cover the background on the key building blocks required to describe our technical contributions. Our NRC abstractions are formalized using the Stable Routing Problem (SRP) model [30], [13], a formal model of network routing for distributed routing protocols. We also briefly describe SMT-based verification using the SRP model (e.g., Minesweeper [11]) and support for graph-based reasoning in the SMT solver MonoSAT [21].

Definition 1 (Stable Routing Problem (SRP) [30]). An SRP is a tuple $(G, A, a_d, \prec, \text{trans})$ where $G = (V, E, d)$ is a graph representing the network topology with vertices V , directed edges E , and destination d ; A is a set of *attributes* representing route announcements; $a_d \in A$ denotes the initial route sent by d ; $\prec \subseteq A \times A$ is a partial order that models the route selection procedure (if $a_1 \prec a_2$ then a_1 is preferred); $\text{trans} : E \times A_\infty \rightarrow A_\infty$, where $A_\infty = A \cup \{\infty\}$ and ∞ denotes no route, is a *transfer function* that models the processing of route announcements sent from one router to another.

Figure 2 summarizes the important notions for the SRP model [30]. The main difference from routing algebras [31], [32] is that the SRP model includes a network topology graph G to reason about a given network and its configurations.

SRP solutions. A solution of an SRP is a labeling function $\mathcal{L} : V \rightarrow A_\infty$ which represents the final route (attribute) chosen by each node when the protocol converges. An SRP can have multiple solutions, or it may have none. Any SRP solution satisfies a *local stability condition*: each node selects the best among the route announcements received from its neighbors.

Example 3 (SRP example). The network in Figure 1a running a simplified version of BGP (simplified for pedagogic

reasons) is modeled using an SRP in which attributes are tuples comprising an integer (local preference), a set of bit vectors (community tags), and a list of vertices (the path). We use $a.lp$, $a.comms$, and $a.path$ to refer to the elements of an attribute a . The initial attribute at the destination, $a_d = (100, \emptyset, [])$. The preference relation \prec models the BGP route selection procedure which is used to select the best route. The attribute with highest local preference is preferred; to break ties, the attribute with minimum path length is preferred (more details are in Appendix A). The transfer function for edge (R_1, R_3) adds the tag $c1$ and prepends R_1 to the path, returning $(100, a.comms \cup \{c1\}, [R_1] + a.path)$. The transfer function for edge (R_3, R_4) sets the local preference to 200 if the tag $c1$ is present, i.e., if $c1 \in a.comms$ it returns $(200, a.comms, [R_3] + a.path)$; otherwise, it returns $(100, a.comms, [R_3] + a.path)$. The transfer function for other edges (u, v) prepends u to the path, sets the local preference to the default value (100), and propagates the community tags.

SMT-based verification using SRP. Minesweeper [11] encodes the SRP instance for the network using an SMT formula N , such that satisfying assignments of N correspond to SRP solutions. To verify if a property encoded as a formula P holds, the satisfiability of $F = N \wedge \neg P$ is checked. If F is satisfiable, a property violation is reported. Otherwise, the property holds over the network (assuming N is satisfiable; otherwise there are no stable paths).

SMT with theory solver for graphs. MonoSAT [21] is an SMT solver with support for *monotonic* predicates. A predicate p is (positive) monotonic in a variable u if whenever $p(\dots u = 0 \dots)$ is true, $p(\dots u = 1 \dots)$ is also true. Graph reachability is a monotonic predicate: if node v_1 can reach node v_2 in a graph with an edge removed, it can still reach v_2 when the edge is added. MonoSAT leverages predicate monotonicity to provide efficient theory support for graph-based reasoning using a *symbolic graph*, a graph with a Boolean variable per edge. Formulas can include these Boolean edge variables as well as monotonic predicates such as reachability and max-flow. MonoSAT has been used to check reachability in data planes in AWS networks [33], [34], but not in *control planes*, as we do in this work.

IV. NRC ABSTRACTIONS

We formalize our NRC abstractions as *abstract SRP instances*, which are parameterized by a partial order.

Definition 2 (Abstract SRP). For an SRP $S = (G, A, a_d, \prec, \text{trans})$, an abstract SRP $\widehat{S}_{\prec'}$ is a tuple $(G, A, a_d, \prec', \text{trans})$, where G, A, a_d , and trans are defined as in the SRP S , and $\prec' \subseteq A_\infty \times A_\infty$ is a partial order which satisfies

$$\forall B \subseteq A, \text{minimal}(B, \prec) \subseteq \text{minimal}(B, \prec') \quad (1)$$

where $\text{minimal}(B, \prec) = \{a \in B \mid \nexists a' \in B. a' \neq a \wedge a' \prec a\}$ denotes the set of minimal elements of B according to \prec . Condition (1) specifies that for any set of attributes B , the minimal elements of B by \prec are also minimal by \prec' .

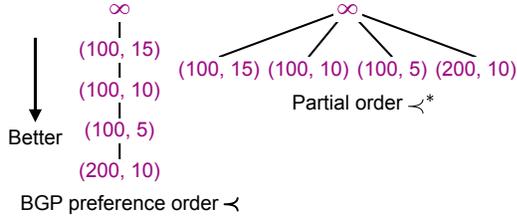


Fig. 3: Partial orders in concrete and abstract SRPs.

Note that condition (1) ensures that the solutions (*i.e.*, minimal elements) at any node in an SRP are also solutions at the same node in the abstract SRP, *i.e.*, the NRC abstractions over-approximate the behavior of an SRP. The precision of an NRC abstraction depends on the partial order used. Our least precise abstraction uses \prec^* , in which any two attributes are incomparable and ∞ is worse than all attributes, and corresponds to choosing any available route. The following example illustrates solutions of an abstract SRP \widehat{S}_{\prec^*} .

Example 4 (Abstract SRP \widehat{S}_{\prec^*}). Figure 3 shows Hasse diagrams for partially ordered sets comprising simplified BGP attributes (pairs with local preference and path length; ∞ denotes no route) at a node u and two partial orders: (1) \prec , the partial order in the standard (concrete) SRP (lifted to A_∞) that models BGP’s route selection procedure (shown on the left), and (2) \prec^* , the partial order corresponding to choosing any available route (shown on the right). Attributes appearing lower in the Hasse diagram are considered better. Hence, in the concrete SRP, u will select (200, 10). In the abstract SRP, any element that is minimal by \prec^* can be a solution for u so u nondeterministically selects an available route. Observe that (200, 10), the solution for u in the concrete SRP, is *guaranteed* to be one of the solutions for u in the abstract SRP. This over-approximation due to condition (1) ensures that our abstraction is sound, *i.e.*, it will not miss any property violations.

Verification with an NRC abstraction. To verify that a property holds in a network using an abstraction \prec' , we construct an SMT formula \widehat{N} such that satisfying assignments of \widehat{N} are solutions of the abstract SRP $\widehat{S}_{\prec'}$ for the network, and conjoin it with the negation of an encoding of the property P to get a formula $F = \widehat{N} \wedge \neg P$. If F is unsatisfiable, all solutions of $\widehat{S}_{\prec'}$ satisfy the property and verification is successful. Otherwise, we report a violation with a counterexample (a satisfying assignment), and a user can perform refinement (described later in this section). Our approach is sound for properties that hold for *all* stable states, *i.e.*, properties of the form $\forall \mathcal{L} \in \text{Sol}(S). P(\mathcal{L})$, where $\text{Sol}(S)$ denotes the SRP solutions for the network. Like Minesweeper [11], our approach only models the stable states of a network and cannot verify properties over transient states that arise before convergence.

Lemma 1. [Over-approximation] For an SRP S and corresponding abstract SRP $\widehat{S}_{\prec'}$ with solutions $\text{Sol}(S)$ and $\text{Sol}(\widehat{S}_{\prec'})$ respectively, $\text{Sol}(S) \subseteq \text{Sol}(\widehat{S}_{\prec'})$.

Protocol	Partial order	Best route
OSPF	\prec^*	Any
	$\prec_{(\text{pathcost})}$	min path cost
	\prec_{ospf}	min path cost, min router ID
BGP	\prec^*	Any
	$\prec_{(lp)}$	max lp (local preference)
	$\prec_{(lp,pl)}$	max lp, min path length
	$\prec_{(lp,pl,MED)}$	max lp, min path length, min MED (Multi-exit Discriminator)
	\prec_{bgp}	max lp, min path length, min MED, min router ID

Fig. 4: Hierarchy of NRC abstractions for OSPF and BGP.

The proof follows from the definition of SRP solutions and the over-approximation condition (1) (full proof in Appendix B).

Theorem 1. [Soundness] Given SMT formulas \widehat{N} and N modeling the abstract and concrete SRPs respectively and SMT formula P encoding the property to be verified, if $\widehat{N} \wedge \neg P$ is unsatisfiable, then $N \wedge \neg P$ is also unsatisfiable.

The proof follows from Lemma 1 and is shown in Appendix B.

Verification under failures. We model link failures using ∞ , which denotes no route (device failures are modeled as failures of all incident links). Let F denote a set of failed links. Given SRP $S = (G, A, a_d, \prec, \text{trans})$, we model network behavior under failures F using an SRP $S_F = (G, A, a_d, \prec, \text{trans}_F)$ where trans_F returns ∞ along edges in F and is the same as trans for other edges. We similarly define an abstract SRP for S_F , $\widehat{S}_{\prec'F} = (G, A, a_d, \prec', \text{trans}_F)$; it only differs from S_F in the partial order \prec' . Since Lemma 1 holds for an arbitrary concrete SRP S , it holds for S_F , *i.e.*, any solution of S_F is also a solution of $\widehat{S}_{\prec'F}$. Hence, the NRC abstractions are sound for verification under specified failures.

Hierarchy of NRC abstractions. The least precise NRC abstraction (using \prec^*) does not model the route selection procedure at all, and chooses any route. More precise abstractions can be obtained by modeling the route selection procedure *partially*. Figure 4 shows partial orders and corresponding route selection procedures (shown as steps in a ranking function) for OSPF and BGP, ordered from least precise (\prec^*) to most precise (\prec). For example, $\prec_{(lp,pl)}$ corresponds to the first two steps of BGP’s route selection procedure, *i.e.*, it first finds routes with maximum local preference, and from these, selects one with minimum path length. Appendix A has more details of BGP’s route selection procedure. Abstractions higher up in the hierarchy are more precise as they model more of the route selection procedure but are more expensive as their SMT encodings have more variables and constraints. This tradeoff between precision and performance is evident in our experiments: verification with $\prec_{(lp)}$ was successful for all networks for which verification with \prec^* gave false positives (§VI-B), but took up to 2.7x more time.

Abstraction refinement. If verification with an abstraction fails, we use an automated procedure to *validate* the returned

counterexample by checking if each node actually chose the best route. The selected routes in the counterexample may contain only some fields, depending on the abstraction used. We find the values of the other fields and the set of available routes by applying the transfer functions along the edges in the counterexample, starting from the destination router (*i.e.*, by effectively simulating the counterexample on the concrete SRP). We then check if all routers selected the best route that they received. If this is the case, we have found a *real* counterexample, *i.e.*, a stable solution in the real network that violates the property; if not, the counterexample is *spurious*. We can eliminate the spurious counterexample by adding a blocking clause that is the negation of the variable assignment corresponding to it and repeat verification with the same abstraction in a CEGAR [35] loop, but this could take many iterations to terminate. Instead, we suggest choosing a more precise abstraction which is higher up in the NRC hierarchy. We could potentially use a *local* refinement procedure that uses a higher-precision abstraction only at certain routers, based on the counterexample. We plan to explore this and other ways of counterexample-guided abstraction refinement in future work.

V. SMT ENCODINGS

In this section we present our SMT encodings for an abstract SRP based on symbolic graphs. SRPs [30] can model many distributed routing protocols (*e.g.*, RIP, BGP, etc.) where the protocol and configurations determine the partial order for route selection and the transfer function. We begin by providing definitions for a symbolic graph and its solutions.

Definition 3 (Symbolic graph [21]). A symbolic graph \mathcal{G}_{RE} is a tuple (G, RE) where $G = (V, E)$ is a graph and $RE = \{re_{uv} | (u, v) \in E\}$ is a set of Boolean routing edge variables.

Definition 4 (Symbolic graph solutions [21]). A symbolic graph $\mathcal{G}_{RE} = (G, RE)$ and a formula F over RE has solutions $Sol(\mathcal{G}_{RE}, F)$ which are subgraphs of G defined by assignments to RE that satisfy F , such that an edge (u, v) is in a solution subgraph iff $re_{uv} = 1$ in the corresponding satisfying assignment.

A. Routing Constraints on Symbolic Graphs

We now describe the constraints in our SMT formulation, \hat{N} , of the abstract SRP \hat{S} . The symbolic graph solutions $Sol(G_{RE}, \hat{N})$ correspond to solutions of \hat{S} . The complete formulation is summarized in Figure 5.

- **Routing choice constraints:** Each node other than the destination chooses a neighbor to route through or *None*, which denotes no route (eqn. 2). We use a variable $nChoice$ to denote a node's choice. The routing edge re_{vu} is true iff u chooses a route from v (eqn. 3).
- **Route availability constraints:** If a node u chooses to route through a neighbor v , then v must have a route to the destination (eqn. 5). If every neighbor v either has no route ($\neg hasRoute_v$) or the route is dropped ($routeDropped_{vu}$), then u must choose *None* (eqn. 6).

- **Attribute transfer and route filtering constraints:** If u chooses to route through neighbor v (*i.e.*, $re_{vu} = 1$), the transfer function relates their attributes and v 's route must not be dropped along edge (v, u) (eqns. 7 and 8). The attribute at the destination is the initial route a_d (eqn. 9).

Our formulation is parameterized by three placeholders: (1) $hasRoute_v$, which is true iff v receives a route from the destination; (2) $trans_{vu}$, the transfer function along edge (v, u) ; and (3) $routeDropped_{vu}$, which is true iff the route is filtered along the edge (v, u) . Of these, $trans_{vu}$ and $routeDropped_{vu}$ depend on the network protocol and configuration, and are shown in an example below. The encodings of $hasRoute$ are described in the next subsection.

Example 5 (Transfer constraints). The attribute transfer and route filtering constraints in the abstract SRP (with partial order \prec^*) are shown below for the network in Figure 1b.

We only model fields used in route filtering (*i.e.*, the community attribute) and ignore local preference and path length. We use a bit vector variable $comm_u$ to denote the community attribute at node R_u , and a Boolean routing edge variable re_{uv} for each edge (R_u, R_v) . We encode the presence of community tag $c1$ as 1, and its absence as 0.

Initial route at destination. We set the community attribute to 0 at the destination R_1 using the constraint $comm_1 = 0$.

Transfer constraints along edge (R_1, R_3) . The transfer function adds the community tag $c1$. The route is never dropped along this edge, so the placeholder $routeDropped_{13}$ is false.

$$re_{13} \rightarrow comm_3 = 1 \quad (15)$$

$$re_{13} \rightarrow \neg routeDropped_{13} \quad (16)$$

$$routeDropped_{13} \leftrightarrow False \quad (17)$$

Our implementation simplifies formulas when $routeDropped$ is a constant, and only asserts equation (15) above.

Transfer constraints along edges (R_5, R_7) and (R_6, R_7) . The transfer functions propagate the community attribute and filter routes based on whether tag $c1$ is present.

$$re_{57} \rightarrow comm_7 = comm_5 \quad (18)$$

$$re_{57} \rightarrow \neg routeDropped_{57} \quad (19)$$

$$routeDropped_{57} \leftrightarrow (comm_5 = 1) \quad (20)$$

$$re_{67} \rightarrow comm_7 = comm_6 \quad (21)$$

$$re_{67} \rightarrow \neg routeDropped_{67} \quad (22)$$

$$routeDropped_{67} \leftrightarrow (comm_6 = 0) \quad (23)$$

Transfer constraints along other edges. The transfer functions propagate the community attribute and do not filter routes.

$$re_{vu} \rightarrow comm_u = comm_v \quad (24)$$

$$re_{vu} \rightarrow \neg routeDropped_{vu} \quad (25)$$

$$routeDropped_{vu} \leftrightarrow False \quad (26)$$

Our implementation simplifies the formulas by substituting the value of $routeDropped$, and only asserts equation (24).

Abstract SRP $\widehat{S} = (G, A, a_d, \prec', trans)$, $G = (V, E, d)$
Symbolic graph $\mathcal{G}_{RE} = (G, RE)$

Variables

$attr_u$: bit vector *route announcement fields*,
 $\forall u \in V$
 $nChoice_u$: bit vector *neighbor choice*, $\forall u \in V \setminus \{d\}$
 $hasRoute_u$: Boolean *placeholder for route availability*, $\forall u \in V$
 $routeDropped_{uv}$: Boolean *route dropped along an edge*,
 $\forall (u, v) \in E$

Constants

$nID(u, v)$: integer *u's neighbor ID for v*, $\forall (u, v) \in E$
 $None_u$: integer *ID denoting no neighbor*, $\forall u \in V$

Routing choice constraints

$$\left(\bigvee_{(v,u) \in E} nChoice_u = nID(u, v) \right) \vee nChoice_u = None_u \quad (2)$$

$$nChoice_u = nID(u, v) \leftrightarrow re_{vu} \quad (3)$$

$$\neg re_{vd} \quad \forall (v, d) \in E \quad (4)$$

Route availability constraints

$$nChoice_u = nID(u, v) \rightarrow hasRoute_v \quad (5)$$

$$nChoice_u = None_u \leftrightarrow \bigwedge_{(v,u) \in E} \neg hasRoute_v \vee routeDropped_{vu} \quad (6)$$

Attribute transfer and route filtering constraints

$$re_{vu} \rightarrow attr_u = trans_{vu}(attr_v) \quad (7)$$

$$re_{vu} \rightarrow \neg routeDropped_{vu} \quad (8)$$

$$attr_d = a_d \quad (9)$$

Solver-specific constraints

(a) SMT solvers with graph theory support (e.g., MonoSAT):

$$\forall u \in V, hasRoute_u \leftrightarrow \mathcal{G}_{RE}.reaches(d, u) \quad (10)$$

(b) SMT solvers without graph theory support (e.g., Z3):

$$hasRoute_d \quad (11)$$

$$\forall u \neq d, hasRoute_u \leftrightarrow \bigvee_{v, (v,u) \in E} hasRoute_v \wedge re_{vu} \quad (12)$$

$$rank_d = 0 \quad (13)$$

$$\forall (v, u) \in E, re_{vu} \rightarrow rank_u = (rank_v + 1) \quad (14)$$

Fig. 5: Symbolic graph-based encoding for an abstract SRP.

B. Solver-specific Constraints

We have two encodings of *hasRoute*, depending on whether the SMT solver has graph theory support.

SMT solvers with graph theory support. We use the reachability predicate $\mathcal{G}_{RE}.reaches$ to encode *hasRoute*: $hasRoute_v$ is true iff $\mathcal{G}_{RE}.reaches(d, v)$ (i.e., there is a path from d to v in the symbolic graph \mathcal{G}_{RE}), where d is the destination (eqn. 10). Additionally, we use the reachability predicate to model regular expressions over paths, which most tools do not support. For example, the regular expression “ $.^*ab.^*c.^*d.^*$ ” (where ‘.’ matches any character and ‘*’ denotes 0 or more occurrences of the preceding character) matches any path that traverses edge (a, b), node c, and then node d, and is encoded as $re_{ab} \wedge \mathcal{G}_{RE}.reaches(b, c) \wedge \mathcal{G}_{RE}.reaches(c, d)$.

Standard SMT solvers. We interpret *hasRoute* as a reachability marker which indicates whether a route has been received and add constraints to propagate the marker in the symbolic graph (eqns. 11 and 12). To prevent solutions with loops, we use a variable, *rank*, at each node to track the path length along with additional constraints (eqns. 13 and 14).

Loop prevention. In BGP, routing loops are prevented using the AS path attribute, the list of autonomous systems (ASes) in the route; routers drop routes if the AS path contains their AS. To model BGP’s loop prevention mechanism exactly, Minesweeper’s [11] SMT encoding would require $O(N^2)$ additional variables (where N is the number of routers) to track for each router, the set of routers in the AS path. Since this is expensive, Minesweeper uses an optimization that relies on the route selection procedure to prevent loops when routers use default local preference: the shorter loop-free path will be selected. Our encodings for *hasRoute* model BGP’s loop prevention mechanism exactly with fewer additional variables: the MonoSAT encoding uses no additional variables and the Z3 encoding uses $O(N)$ additional variables (*rank*).

C. Benefits of the NRC Abstractions in SMT Solving

Fewer attributes. The most direct benefit is that with NRC abstractions many route announcement fields become irrelevant and can be removed from the network model, resulting in smaller SMT formulas. Specifically, all fields required to model *route filtering* (i.e., the dropping of route announcements) and the property of interest are retained, but fields used only for route selection (e.g., local preference) can be removed depending on the specific abstraction.

Expensive transfers can be avoided during SMT search. Once a neighbor is selected during the SMT search, then transfers of attributes from other neighbors become irrelevant. In contrast, without any abstraction, each node *must* consider transfers of attributes from *all* neighbors to pick the best route.

D. Encoding Properties for Verification

Reachability. We encode the property that a node u can reach destination d by asserting its negation: $nChoice_u = None_u$.

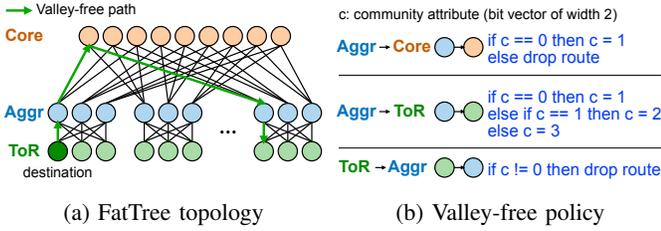


Fig. 6: Example data center network with a valley-free policy.

Non-reachability/Isolation. We encode the property that a node u can never reach d by asserting $nChoice_u \neq None_u$.

No-transit property. Routing policies between autonomous systems (ASes) are typically influenced by business relationships such as provider-customer or peer-peer [19], [36]. A provider AS is paid to carry traffic to and from its customers while peer ASes exchange traffic between themselves and their customers without any charge. The BGP policies (Gao-Rexford conditions [19]) between ASes usually ensure that an AS does not carry traffic from one peer or provider to another. This is called a *no-transit* property; its negation is encoded as $\bigvee_{u \in V} \bigvee_{v, w \in PeerProv(u), v \neq w} r_{evu} \wedge r_{euw}$, where $PeerProv(u)$ denotes neighbors of u that are its peers or providers.

Policy properties. BGP policies can be defined by assigning meaning to specific community tags. Policy properties can then be encoded using formulas over the communities at a node.

Example 6 (Valley-free Policy). The valley-free policy prevents paths that have valleys, *i.e.*, paths which go up, down, and up again between the layers of a FatTree network topology [23], [29]; a valley-free path is shown in Figure 6a. Figure 6b shows an implementation of the valley-free policy where c denotes the community attribute in BGP. A path between ToR routers with a valley between the Aggr and Core layers will cross an Aggr router at least three times, updating c to 3. Hence, the negation of the valley-free property at a node u is encoded as $comm_u = 3$.

VI. IMPLEMENTATION AND EVALUATION

We implemented our abstractions and SMT encodings in a prototype tool called ACORN, with backends to MonoSAT and Z3 solvers. (The SMT encoding for an abstract SRP (§V) is extended for a concrete SRP using additional constraints described in Appendix D.) ACORN’s input is an intermediate representation (IR) of a network topology and configurations (described in Appendix C) which represents routing policy using match-action rules, similar to route-maps in Cisco’s configuration language, and could serve as a target for frontends such as Batfish [14] or NV [13] in the future.

In our evaluation, we measure the effectiveness of the NRC abstractions and use two backend SMT solvers – MonoSAT and Z3 (with bitvector theory and bit-blasting enabled). We use four settings: (1) **abs_mono**: with NRC abstraction (\leftarrow^*), using MonoSAT; (2) **abs_z3**: with NRC abstraction (\leftarrow^*), using Z3; (3) **mono**: without abstraction, using MonoSAT; (4) **z3**:

without abstraction, using Z3. We evaluated ACORN on two types of benchmarks: (1) data center networks with FatTree topologies [23] (a commonly used topology), and (2) wide area networks from Topology Zoo [24] and BGPStream [25] (more details are in Appendix C). We also compared ACORN with two state-of-the-art control plane verifiers on the data center benchmarks. All experiments were run on a Mac laptop with a 2.3 GHz Intel i7 processor and 16 GB memory.

A. Data Center Networks

We generated data center network benchmarks with FatTree topologies [23], with 125 to 36,980 nodes running four policies: (1) shortest-path routing policy, (2) valley-free policy, (3) an extension of the valley-free policy with an isolation property – it uses regular expressions to enforce isolation between a FatTree pod and an external router connected to the core routers, and (4) a buggy valley-free policy in which routers in the last pod cannot reach routers in other pods. We checked reachability for all policies, and a policy-based property for (2) and (3). The results are shown in Figure 7, with each graph showing the number of nodes on the x-axis and the verification time (in seconds) on the y-axis.

Our results show that for *all* data center examples, and with *both* solvers, *using the NRC abstraction is uniformly better than using the no-abstraction setting*. With the MonoSAT solver, the NRC abstraction can achieve a relative speed-up of 52x for verifying reachability (when verification completes within a 1 hour timeout). Also, MonoSAT performed better than Z3 by up to 10x; leveraging graph-based reasoning was clearly beneficial for these examples. Our abstract settings successfully verified all properties without any false positives, showing that the NRC abstraction can handle realistic policies. For networks running the buggy valley-free policy, our tool correctly reports that the destination is unreachable (results are in Figure 7f). Furthermore, our abstraction is effective even in these cases: `abs_mono` finishes on 3,000 nodes within an hour, while both no-abstraction settings time out on 2,000 nodes.

In terms of scalability, for both solvers, the no-abstraction setting times out beyond 4,500 nodes for reachability verification, while the abstract setting scales up to about 37,000 nodes for the shortest-path and valley-free policies, and up to 18,000 nodes for the isolation policy. *To the best of our knowledge, no other control plane verifier has shown the correctness of benchmarks of such large sizes*; all prior related work has been shown on networks with up to 4,500 nodes (maximum), which are much smaller than large data centers in operation today.

B. Wide Area Networks

To evaluate ACORN on less regular network topologies than data centers, we considered wide area network benchmarks. These typically have small sizes and are not easily parameterized, unlike data center topologies. We evaluated ACORN on two sets of wide area networks: (1) 10 of the larger networks from Topology Zoo [24], with 22 to 79 routers, which we annotated with business relationships (since Topology Zoo only provides topologies), and (2) 10 example networks based

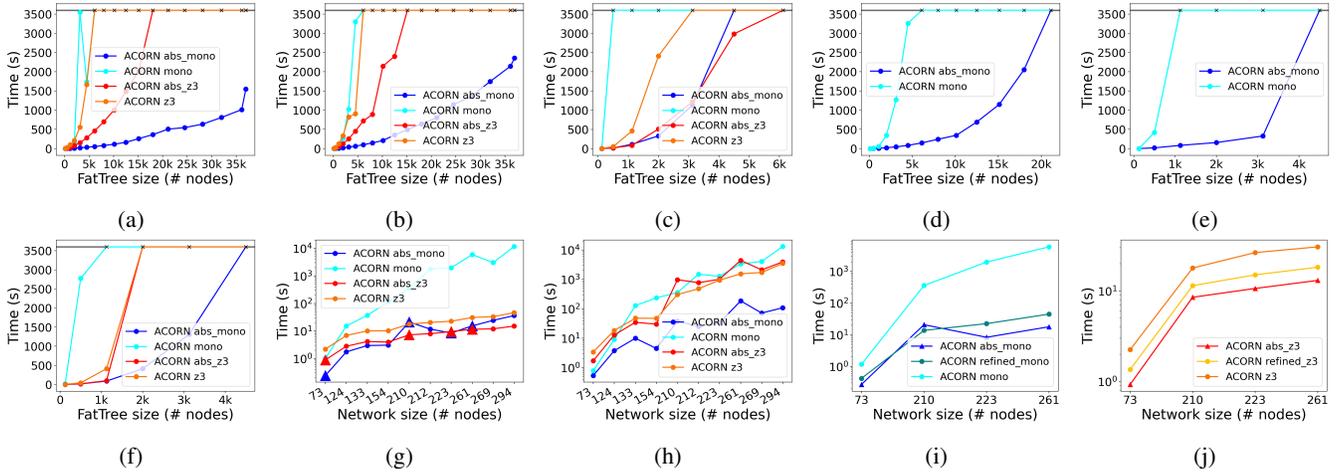


Fig. 7: Results for data center networks: (a) reachability with shortest-path routing, (b) reachability with valley-free policy, (c) valley-free property, (d) reachability with isolation policy, (e) isolation property, and (f) reachability with a buggy valley-free policy. Results for BGPStream examples: (g) reachability, (h) no-transit property, refinement using (i) MonoSAT and (j) Z3.

on parts of the Internet that were involved in misconfiguration incidents as reported on BGPStream [25], which we annotated with publicly available business relationships (CAIDA AS relationships dataset [37]). For all benchmarks, we used a BGP policy that implements the Gao-Rexford conditions [19]: (1) routes from peers and providers are not exported to other peers and providers, and (2) routes from customers are preferred over routes from peers, which are preferred over routes from providers. We then checked two properties: reachability of all nodes to a destination, and the no-transit property (§V-D).

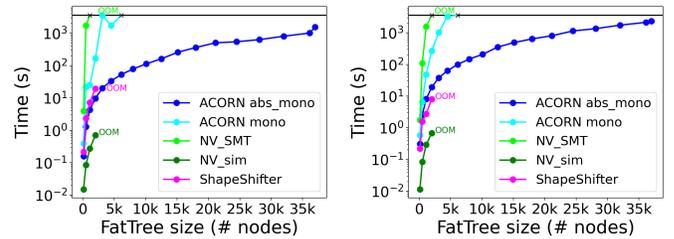
Topology Zoo benchmarks. The abstract settings successfully verify both properties and are up to 3x faster than the respective no-abstraction settings. All settings take less than 0.5s for both properties (detailed results are in Appendix C).

BGPStream benchmarks. The results are in Figures 7g to 7j, with the number of nodes (ASes) on the x-axis and verification time in seconds on the y-axis (log scale). The abstract settings successfully verified reachability in 6 networks and gave false positives (denoted by triangular markers) for 4; when successful, the abstract settings performed much better than the no-abstraction settings with relative speedups of up to 323x for MonoSAT and 3x for Z3. For the no-transit property, abs_mono is up to 120x faster than mono, while abs_z3 is faster than z3 for some networks but slower for others.

For the 4 benchmarks with false positives, we used a more precise abstraction, $\prec_{(lp)}$, which models local preference (results shown in Figures 7i and 7j). Our $\prec_{(lp)}$ abstraction is successful on all 4 networks, with relative speedups (over no abstraction) of up to 133x for MonoSAT and 1.8x for Z3, and relative slowdowns (over \prec^*) of up to 2.7x for MonoSAT and 1.5x for Z3. These results demonstrate the precision-cost tradeoff enabled by the NRC abstraction hierarchy.

C. Comparison with Existing Tools

We compared ACORN with two state-of-the-art control plane verifiers: ShapeShifter [16] and NV [13] (FastPlane [15]



(a) Shortest-path policy (b) Valley-free policy

Fig. 8: Comparison of tools on data center examples.

and Hoyan [18] are not publicly available). ShapeShifter uses simulation with abstract interpretation [38], with binary decision diagrams (BDDs) [39] representing sets of abstract routing messages. NV is a functional programming language for modeling and verifying network control planes. It provides a simulator (based on Multi-Terminal BDDs [40] but without abstraction of routing messages) and an SMT-based verifier that uses Z3. (NV’s SMT engine has been shown to perform better than Minesweeper [13].) NV uses a series of front-end transformations to generate an SMT formula (we only report NV’s SMT solving time), but its encoding is not based on symbolic graphs. A comparison of our no-abstraction settings against NV_SMT gives some indication of the effectiveness of our SMT encoding. We performed experiments on the data center benchmarks (§VI-A), where we generated corresponding inputs for ShapeShifter and NV with the same routing message fields. The results for the shortest-path routing and valley-free policies are shown in Figure 8, with the number of nodes shown on the x-axis, verification time in seconds on the y-axis (log scale), timeouts denoted by ‘x’, and out-of-memory denoted by ‘OOM’. (ShapeShifter and NV could not be run on the isolation benchmarks as they do not support regular expressions over AS paths.) Note that both NV and ShapeShifter run out of memory for networks with more than 3,000 nodes while ACORN’s mono and abs_mono settings can

verify larger networks with 4,500 nodes and 36,980 nodes, respectively. These results show that SMT-based methods for network control plane verification can scale to large networks with tens of thousands of nodes.

D. Discussion and Limitations

ACORN is sound for properties that hold for all *stable* states of a network, *i.e.*, properties of the form $\forall s P(s)$ where s is a stable state, such as reachability, policy-based properties, device equivalence, and way-pointing. Like many SMT-based tools, ACORN cannot verify properties over *transient* states that arise before convergence. For checking reachability, our least precise abstraction works well in practice; to verify a property about the path length between two routers, a user should use an abstraction that models path length (otherwise our verification procedure would give a false positive). We have shown that our abstractions are sound under specified failures; however, our tool does not yet model failures, which we plan to consider in future work.

VII. RELATED WORK

Our work is related to other efforts in network verification and the use of nondeterministic abstractions for verification.

Distributed control plane verification. These methods [41], [12], [42], [17], [11], [13] aim to verify all data planes that emerge from the control plane. Simulation-based tools [14], [43], [15] can scale to large networks, but can miss errors that are triggered only under certain environments. The FAST-PLANE [15] simulator scales to large data centers (results shown for ≈ 2000 nodes) but it requires the network policy to be monotonic [31] (a route announcement’s preference decreases along any edge in the network) while our approach does not. HOYAN [18] uses a hybrid simulation and SMT-based approach which tracks multiple routes received at each router to check reachability under failures, but in the context of the given simulation. The ShapeShifter [16] work is the closest to ours in terms of route abstractions, but it does not scale as well as our tool (§VI-C). Moreover, our SMT-based approach provides better precision by exploring multiple routing choices at each node and tracking correlations across different nodes, whereas ShapeShifter uses a conservative abstraction at each node, much as SMT-based program verification allows path-sensitivity for more precision than path-insensitive static analysis. For example, ShapeShifter’s ternary abstraction (which abstracts each community tag bit to $\{0, 1, *\}$) would result in a false positive on Example 2 (§II), while ACORN verifies it correctly. Bagpipe [12] verifies BGP policies using symbolic execution and uses a simplified BGP route selection procedure that chooses routes with maximum local preference, similar to our NRC abstraction using $\prec_{(lp)}$. Our abstraction hierarchy is more general and can be applied to any routing protocol.

ARC [44] and QARC [45] use a graph-based abstraction combined with graph algorithms and mixed-integer linear programming respectively, but do not support protocol features such as local preference and community tags. Tiramisu [46] uses a similar graph-based representation, but with multiple

layers to capture inter-protocol dependencies and was shown to scale to networks with a few hundred devices. Bonsai [30] compresses the network control plane to take advantage of symmetry in the network topology and policy; NRC abstractions can be applied even when the network is not symmetric.

Some recent approaches [47], [48], [20] use modular verification techniques to improve the scalability of verification; the core ideas in modular verification are orthogonal to our work. Among these efforts, LIGHTYEAR [20] also verifies BGP policies using an over-approximation that allows routers to choose any received route – this corresponds to our NRC abstraction with partial order \prec^* . However, unlike our approach, it requires a user to provide suitable invariants.

Data plane verification. These efforts [49], [50], [51], [52], [53], [54], [55], [56] model the data forwarding rules and check properties such as reachability, absence of routing loops, etc. Many such methods have been shown to successfully handle the scale and complexity of real-world networks. Similar to these methods, our least precise abstraction does not model the route selection procedure but we verify all data planes that emerge from the control plane, not just one snapshot.

Nondeterminism and abstractions. Nondeterministic abstractions have been used in many different settings in software and hardware verification. Examples include control flow nondeterminism in Boolean program abstractions in SLAM [57], a sequentialization technique [58] that converts control nondeterminism (*i.e.*, interleavings in a concurrent program) to data nondeterminism, and a localization abstraction [59] in hardware designs. Our NRC abstractions use route nondeterminism to soundly abstract network control plane behavior.

VIII. CONCLUSIONS AND FUTURE DIRECTIONS

The main motivation for our work is to improve the scalability of symbolic verification of network control planes. Our approach is centered around two core contributions: a hierarchy of nondeterministic routing choice abstractions, and a new SMT encoding that can leverage specialized SMT solvers with graph theory support. Our tool, ACORN, has verified reachability (an important property for network operators) on data center benchmarks (with FatTree topologies and commonly used policies) with $\approx 37,000$ routers, which far exceeds what has been shown by existing related tools. Our evaluation shows that our abstraction performs *uniformly better* than no abstraction for verifying reachability for different network topologies and policies, and with two different SMT solvers. In future work, we plan to consider verification under failures, and combine our abstractions with techniques based on modular verification of network control planes.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grants 1837030 and 2107138. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF. We would also like to thank Anish Athalye for permitting use of Basalt, a language for graphic design.

APPENDIX A
BGP OVERVIEW

BGP is the protocol used for routing between *autonomous systems* (ASes) in the Internet. An autonomous system (AS) is a network controlled by a single administrative entity, *e.g.*, the network of an Internet Service Provider (ISP) in a particular country, or a college campus network. A simplified version of the decision process used to select best routes in BGP is shown in Table I [36]. A router compares two route announcements by comparing the attributes in each row of the table, starting from the first row. A route announcement with higher local preference is preferred, regardless of the values of other attributes; if two route announcements have equal local preference, then their path lengths will be compared. BGP allows routes to be associated with additional state via the community attribute, a list of string tags. Decisions can be taken based on the tags present in a route announcement; for example, a route announcement containing a particular tag can be dropped or the route preference can be altered (*e.g.*, by increasing the local preference if a particular tag is present).

APPENDIX B

PROOF OF SOUNDNESS OF THE NRC ABSTRACTIONS

Lemma 1. [Over-approximation] For an SRP S and corresponding abstract SRP $\hat{S}_{\prec'}$ with solutions $Sol(S)$ and $Sol(\hat{S}_{\prec'})$ respectively, $Sol(S) \subseteq Sol(\hat{S}_{\prec'})$.

Proof. We need to show that for each labeling \mathcal{L} , if $\mathcal{L} \in Sol(S)$ then $\mathcal{L} \in Sol(\hat{S}_{\prec'})$. An SRP solution \mathcal{L} is defined by

$$\mathcal{L}(u) = \begin{cases} a_d & \text{if } u = d \\ \infty & \text{if } \text{attrs}_{\mathcal{L}}(u) = \emptyset \\ a \in \text{attrs}_{\mathcal{L}}(u), \text{ minimal by } \prec & \text{if } \text{attrs}_{\mathcal{L}}(u) \neq \emptyset \end{cases}$$

where $\text{attrs}_{\mathcal{L}}(u)$ is the set of attributes that u receives from its neighbors. The abstract SRP $\hat{S}_{\prec'}$ differs from the SRP S only in the partial order. Therefore, to show that \mathcal{L} is a solution of $\hat{S}_{\prec'}$, we need to show that if $\text{attrs}_{\mathcal{L}}(u) \neq \emptyset$, then $\mathcal{L}(u)$ is minimal by \prec' . By the definition of an abstract SRP, the set of minimal attributes according to \prec' is a superset of the set of minimal attributes according to \prec , which means $\mathcal{L}(u)$ is minimal by \prec' . Therefore, any SRP solution \mathcal{L} is a solution of the abstract SRP $\hat{S}_{\prec'}$. \square

Theorem 1. [Soundness] Given SMT formulas \hat{N} and N modeling the abstract and concrete SRPs respectively and SMT formula P encoding the property to be verified, if $\hat{N} \wedge \neg P$ is unsatisfiable, then $N \wedge \neg P$ is also unsatisfiable.

Proof. If $\hat{N} \wedge \neg P$ is unsatisfiable, every solution of the abstract SRP satisfies the given property. By Lemma 1, the property also holds for all solutions of the concrete SRP S , *i.e.*, there is no property violation in the real network. \square

APPENDIX C

ACORN INTERMEDIATE REPRESENTATION (IR) AND BENCHMARK EXAMPLES

Intermediate Representation (IR). Our IR represents a transfer function as a list of match-action rules, similar to

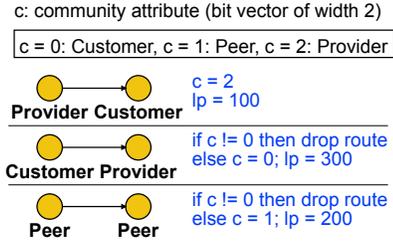


Fig. 9: BGP policy implementing Gao-Rexford conditions [19]

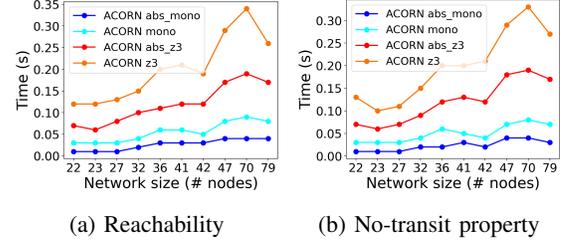


Fig. 10: Results for Topology Zoo examples.

route-maps in Cisco’s configuration language. We support matching on the community attribute and some types of regular expressions over the AS path. Our implementation currently supports regular expressions that check whether the path contains certain ASes or a particular sequence of ASes, and could be extended to support general regular expressions in the future. A match can be associated with multiple actions, which can update route announcement fields such as the community attribute, local preference, and AS path length.

Benchmark examples. The details of the wide area network examples we used (§VI-B) are described below.

Topology Zoo benchmarks. We used 10 topologies from the Topology Zoo [24], which we pre-processed, *e.g.*, by removing duplicate nodes and nodes with id “None”. The details of the resulting topologies are shown in Table II. We annotated the topologies with business relationships, considering each node as an AS, and used a BGP policy that implements the Gao-Rexford conditions [19] (Figure 9). The annotated benchmark files (in GML format) are included in our benchmark repository, along with the examples in our IR format.

BGPStream benchmarks. We created a set of 10 examples based on parts of the Internet involved in BGP hijacking incidents, as reported on BGPStream [25]. For each hijacking incident, we created a network with the ASes involved and used the CAIDA AS Relationships dataset [37] to add edges between ASes with the given business relationships (customer-provider or peer-peer). We then removed some ASes (if required) so that our no-abstraction setting could verify that all ASes in the resulting network can reach the destination (taken to be the possibly hijacked AS). We used a BGP policy (shown in Figure 9) that implements the Gao-Rexford conditions [19]. The details of the examples are shown in Table III.

Results for Topology Zoo examples. Detailed results for the Topology Zoo benchmark examples are shown in Figure 10.

Step	Attribute	Description	Preference (Lower/Higher)
1	Local preference	An integer set locally and not propagated	Higher
2	AS path length	The number of ASes the route has passed through	Lower
3	Multi-exit Discriminator (MED)	An integer influencing which link should be used between two ASes	Lower
4	Router ID	Unique identifier for a router used for tie breaking	Lower

TABLE I: Simplified BGP decision process to select the best route [36].

Benchmark	Topology name	Size
TZ1	VinaREN	22 nodes, 24 edges
TZ2	FCCN	23 nodes, 25 edges
TZ3	GTS Hungary	27 nodes, 28 edges
TZ4	GTS Slovakia	32 nodes, 34 edges
TZ5	GRnet	36 nodes, 41 edges
TZ6	RoEduNet	41 nodes, 45 edges
TZ7	LITNET	42 nodes, 42 edges
TZ8	Bell South	47 nodes, 62 edges
TZ9	Tecove	70 nodes, 70 edges
TZ10	ULAKNET	79 nodes, 79 edges

TABLE II: Topology Zoo examples.

Benchmark	Incident date	Size
B1	2021-06-14	261 nodes, 3325 edges
B2	2021-06-17	223 nodes, 2722 edges
B3	2021-06-18	133 nodes, 1205 edges
B4	2021-06-19	210 nodes, 2100 edges
B5	2021-06-21	269 nodes, 3351 edges
B6	2021-06-22	212 nodes, 2233 edges
B7	2021-06-22	294 nodes, 4108 edges
B8	2021-06-22	124 nodes, 860 edges
B9	2021-06-22	73 nodes, 270 edges
B10	2021-06-25	154 nodes, 1176 edges

TABLE III: BGPStream examples.

APPENDIX D

SMT CONSTRAINTS FOR CONCRETE SRP

We extend our abstract SRP formulation (Figure 5) to encode a concrete SRP by adding additional constraints ensuring that each node picks the best route, *i.e.*, for every edge $(v, u) \in E$, if u selects the route from v then v 's route must be the best route that u receives from its neighbors. This requires keeping track of the attribute fields used in route selection (such as path length) and possibly additional variables to track the minimum or maximum value of an attribute. The constraints required to model the first two steps in BGP's route selection procedure are shown in Example 7.

Example 7 (Encoding route selection in BGP). We keep track of local preference (denoted lp) and AS path length (denoted $path$) and encode transfer constraints over these attributes (*e.g.*, to increment path length). For each edge (v, u) , we use lp_{vu} to denote the local preference of the route sent from v to u after applying the transfer function. For each node u we use $maxLp_u$ to track the maximum local preference of routes node u receives, and $minPath_u$ to track the minimum path length among routes with the maximum local preference.

We define $maxLp_u$ below ($nValid_{vu} \leftrightarrow hasRoute_v \wedge \neg routeDropped_{vu}$ indicates whether v sends a route to u).

$$\bigwedge_{(v,u) \in E} nValid_{vu} \rightarrow maxLp_u \geq lp_{vu}$$

$$nChoice_u \neq None_u \rightarrow \bigvee_{(v,u) \in E} nValid_{vu} \wedge maxLp_u = lp_{vu}$$

We define $minPath_u$ using similar constraints:

$$\bigwedge_{(v,u) \in E} (nValid_{vu} \wedge lp_{vu} = maxLp_u) \rightarrow minPath_u \leq path_v$$

$$nChoice_u \neq None_u \rightarrow$$

$$\bigvee_{(v,u) \in E} nValid_{vu} \wedge lp_{vu} = maxLp_u \wedge minPath_u = path_v$$

We then add constraints to ensure that if u chooses a route from any neighbor v , then v 's route must be the best.

$$nChoice_u = nID(u, v) \rightarrow lp_{vu} = maxLp_u \wedge path_v = minPath_u$$

REFERENCES

- [1] N. Rockwell, "Summary of june 8 outage," <https://www.fastly.com/blog/summary-of-june-8-outage>, 2021.
- [2] J. Graham-Cumming, "Cloudflare outage on july 17, 2020," <https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/>, 2020.
- [3] M. Anderson, "Time warner cable says outages largely resolved," <http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved>, NY, NY, 2014.
- [4] S. Ragan, "Bgp errors are to blame for monday's twitter outage, not ddos attacks," <https://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html>, 2016.
- [5] D. Roberts, "It's been a week and customers are still mad at bb&t," <https://www.charlotteobserver.com/news/business/banking/article202616124.html>, 2018.
- [6] Y. Sverdlik, "United says it outage resolved, dozen flights canceled monday," <https://www.datacenterknowledge.com/archives/2017/01/23/united-says-it-outage-resolved-dozen-flights-canceled-monday>, 2017.
- [7] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma, "Validating datacenters at scale," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: ACM, 2019, pp. 200–213.
- [8] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang, D. Yu, C. Tian, H. Zheng, and B. Y. Zhao, "Safely and automatically updating in-network acl configurations with intent language," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. Association for Computing Machinery, 2019, p. 214–226.
- [9] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *NSDI 14*, 2014.
- [10] L. L. Peterson and B. S. Davie, *Computer Networks, Fifth Edition: A Systems Approach*, 5th ed., 2011.
- [11] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *SIGCOMM*, Aug. 2017.
- [12] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Formal semantics and automated verification for the border gateway protocol," in *NetPL*, 2016.
- [13] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker, "NV: An intermediate language for verification of network control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, 2020, p. 958–973.

- [14] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *NSDI*, 2015.
- [15] N. P. Lopes and A. Rybalchenko, "Fast BGP simulation of large data-centers," in *Proc. of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Jan. 2019.
- [16] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Abstract interpretation of distributed network control planes," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019.
- [17] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 953–967.
- [18] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, D. She, Q. Ma, B. Cheng, H. Xu, M. Zhang, Z. Wang, and R. Fonseca, "Accuracy, scalability, coverage: A practical configuration verifier on a global wan," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 599–614.
- [19] L. Gao and J. Rexford, "Stable internet routing without global coordination," in *SIGMETRICS*, 2000.
- [20] A. Tang, R. Beckett, K. Jayaraman, T. Millstein, and G. Varghese, "Lightyear: Using modularity to scale bgp control plane verification," *arXiv preprint arXiv:2204.09635*, 2022.
- [21] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu, "SAT modulo monotonic theories," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15, 2015, p. 3702–3709.
- [22] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.
- [23] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.
- [24] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.
- [25] "BGP Stream," <https://bgpstream.com>.
- [26] "ACORN benchmark repository," https://github.com/divya-urs/ACORN_benchmark.
- [27] A. Abhashkumar, K. Subramanian, A. Andreyev, H. Kim, N. K. Salem, J. Yang, P. Lapukhov, A. Akella, and H. Zeng, "Running BGP in data centers at scale," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 2021, pp. 65–81.
- [28] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Trans. Networking*, vol. 10, no. 2, 2002.
- [29] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *SIGCOMM*, 2016.
- [30] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, 2018, p. 476–489.
- [31] J. a. L. Sobrinho, "An algebraic theory of dynamic network routing," *IEEE/ACM Trans. Netw.*, vol. 13, no. 5, pp. 1160–1173, Oct. 2005.
- [32] T. G. Griffin and J. L. Sobrinho, "Metarouting," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Aug. 2005, pp. 1–12.
- [33] J. Backes, S. Bayless, B. Cook, C. Dodge, A. Gacek, A. J. Hu, T. Kahsai, B. Kocik, E. Kotelnikov, J. Kukovec, S. McLaughlin, J. Reed, N. Rungra, J. Sizemore, M. A. Stalzer, P. Srinivasan, P. Subotic, C. Varming, and B. Whaley, "Reachability analysis for aws-based networks," in *Computer Aided Verification (CAV), Proceedings, Part II*, 2019, pp. 231–241.
- [34] S. Bayless, J. Backes, D. DaCosta, B. Jones, N. Launchbury, P. Trentin, K. Jewell, S. Joshi, M. Zeng, and N. Mathews, "Debugging network reachability with blocked paths," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 851–862.
- [35] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification, 12th International Conference, CAV, Proceedings*, 2000, pp. 154–169.
- [36] M. Caesar and J. Rexford, "BGP routing policies in ISP networks," *Netw. Mag. of Global Internetwkg.*, vol. 19, no. 6, p. 5–11, Nov. 2005.
- [37] "The CAIDA AS Relationships Dataset, May 1 2021," <https://www.caida.org/catalog/datasets/as-relationships/>.
- [38] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77, 1977, p. 238–252.
- [39] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.
- [40] E. M. Clarke, M. Fujita, and X. Zhao, "Multi-terminal binary decision diagrams and hybrid decision diagrams," in *Representations of discrete functions*. Springer, 1996, pp. 93–108.
- [41] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. L. Talcott, "FSR: Formal analysis and implementation toolkit for safe inter-domain routing," *IEEE/ACM Trans. Networking*, vol. 20, no. 6, 2012.
- [42] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *OSDI*, 2016.
- [43] B. Quoitin and S. Uhlig, "Modeling the routing of an autonomous system with c-bgp," *Netw. Mag. of Global Internetwkg.*, vol. 19, no. 6, pp. 12–19, Nov. 2005.
- [44] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *SIGCOMM*, 2016.
- [45] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*, 2020, pp. 974–988.
- [46] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 201–219.
- [47] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker, "Kirigami, the verifiable art of network cutting," *arXiv preprint arXiv:2202.06098*, 2022.
- [48] —, "Modular control plane verification via temporal invariants," *arXiv preprint arXiv:2204.10303*, 2022.
- [49] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.
- [50] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with ant eater," in *SIGCOMM*, 2011.
- [51] E. Al-Shaer and S. Al-Haj, "FlowChecker: configuration analysis and verification of federated openflow infrastructures," in *3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig 2010*, 2010, pp. 37–44.
- [52] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *NSDI*, 2013.
- [53] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, Apr. 2013, pp. 99–112.
- [54] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *POPL*, 2014.
- [55] S. Zhang and S. Malik, "SAT based verification of network data planes," in *Automated Technology for Verification and Analysis (ATVA)*, 2013.
- [56] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *NSDI*, 2015.
- [57] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001, pp. 203–213.
- [58] A. Lal and T. W. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," in *Computer Aided Verification, 20th International Conference, CAV, Proceedings*, 2008, pp. 37–51.
- [59] E. M. Clarke, R. P. Kurshan, and H. Veith, "The localization reduction and counterexample-guided abstraction refinement," in *Time for Verification, Essays in Memory of Amir Pnueli*, 2010, pp. 61–71.

Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA⁺

William Schultz
 Northeastern University
 Boston, MA
 schultz.w@northeastern.edu

Ian Dardik
 Carnegie Mellon University
 Pittsburgh, PA
 idardik@andrew.cmu.edu

Stavros Tripakis
 Northeastern University
 Boston, MA
 stavros@northeastern.edu

Abstract—We present a new technique for automatically inferring inductive invariants of parameterized distributed protocols specified in TLA⁺. Ours is the first such invariant inference technique to work directly on TLA⁺, an expressive, high level specification language. To achieve this, we present a new algorithm for invariant inference that is based around a core procedure for generating *plain*, potentially non-inductive *lemma invariants* that are used as candidate conjuncts of an overall inductive invariant. We couple this with a greedy lemma invariant selection procedure that selects lemmas that eliminate the largest number of counterexamples to induction at each round of our inference procedure. We have implemented our algorithm in a tool, *endive*, and evaluate it on a diverse set of distributed protocol benchmarks, demonstrating competitive performance and ability to uniquely solve an industrial scale reconfiguration protocol.

I. INTRODUCTION

Automatically verifying the safety of distributed systems remains an important and difficult challenge. Distributed protocols such as Paxos [32] and Raft [39] serve as the foundation of modern fault tolerant systems, making the correctness of these protocols critical to the reliability of large scale database, cloud computing, and other decentralized systems [47], [8], [11], [38]. An effective approach for reasoning about the correctness of these protocols involves specifying system *invariants*, which are assertions that must hold in every reachable system state. Thus, a primary task of verification is proving that a candidate invariant holds in every reachable state of a given system. For adequately small, finite state systems, symbolic or explicit state model checking techniques [12], [26], [6] can be sufficient to automatically prove invariants. For verification of infinite state or *parameterized* protocols, however, model checking techniques may, in general, be incomplete [7]. Thus, the standard technique for proving that such a system satisfies a given invariant is to discover an *inductive invariant*, which is an invariant that is typically stronger than the desired system invariant, and is preserved by all protocol transitions. Discovering inductive invariants, however, is one of the most challenging aspects of verification and remains a non-trivial task with a large amount of human effort required [50], [13], [49], [44]. Thus, automating the inference of these invariants is a desirable goal.

In general, the problem of inferring inductive invariants for infinite state protocols is undecidable [40]. Even the verification of inductive invariants may require checking the validity of arbitrary first order formulas, which is undecidable [41]. Thus, this places fundamental limits on the development of fully general algorithmic techniques for discovering inductive invariants.

Significant progress towards automation of inductive invariant discovery for infinite state protocols has been made with the Ivy framework [42]. Ivy utilizes a restricted system modeling language that allows for efficient checking of verification goals via an SMT solver such as Z3 [17]. In particular, the EPR and extended EPR subsets of Ivy are decidable. Ivy also provides an interface for an interactive, counterexample guided invariant discovery process. The Ivy language, however, may place an additional burden on users when protocols or their invariants don't fall naturally into one of the decidable fragments of Ivy. Transforming a protocol into such a fragment is a manual and nontrivial task [41].

Subsequent work has attempted to fully automate the discovery of inductive invariants for distributed protocols. State of the art tools for inductive invariant inference for distributed protocols include I4 [35], fol-ic3 [29], IC3PO [24], SWISS [25], and DistAI [51]. All of these tools, however, accept only Ivy or an Ivy-like language [2] as input. Moreover, several of these tools work only within the restricted decidable fragments of Ivy.

In this paper, we present a new technique for automatic discovery of inductive invariants for protocols specified in TLA⁺, a high level, expressive specification language [33]. To our knowledge, this is the first inductive invariant discovery tool for distributed protocols in a language other than Ivy. Our technique is built around a core procedure for generating small, *plain* (potentially non-inductive) invariants. We search for these invariants on finite protocol instances, employing the so-called *small scope* hypothesis [27], [35], [4], circumventing undecidability concerns when reasoning over unbounded domains. We couple this invariant generation procedure with an invariant selection procedure based on a greedy counterexample elimination heuristic in order to incrementally construct an overall inductive invariant. By restricting our inference reasoning to finite instances, we avoid restrictions imposed by modeling approaches that try to maintain decidability of

This work was supported by the U.S. National Science Foundation under NSF SaTC award CNS-1801546.

SMT queries.

Our technique is partially inspired by prior observations [13], [44], [25], [10] that, for many practical protocols, an inductive invariant I is typically of the form $I = P \wedge A_1 \wedge \dots \wedge A_n$, where P is the main invariant (i.e. safety property) we are trying to establish, and A_1, \dots, A_n are a list of *lemma invariants*. Each lemma invariant A_i may not necessarily be inductive, but it is necessarily an invariant, and it is typically much smaller than I . These lemma invariants serve to strengthen P so as to make it inductive. Many prior approaches to inductive invariant inference have focused on searching for lemma invariants that are inductive, or inductive relative to previously discovered information [25], [10], [24], [29]. In contrast, our inference procedure searches for *plain* lemma invariants and uses them as candidates for conjuncts of an overall inductive invariant. To search for lemma invariants, we sample candidates using a syntax-guided approach [20], and verify the candidates using an off the shelf model checker.

We have implemented our invariant inference procedure in a tool, *endive*, and we evaluate its performance on a set of diverse protocol benchmarks, including 29 of the benchmarks reported in [24]. Our tool solves nearly all of these benchmarks, and compares favorably with other state of the art tools, despite the fact that all of these tools accept Ivy or decidable Ivy fragments as inputs. We also evaluate our tool and other state of the art tools on a more complex, industrial scale protocol, *MongoLoglessDynamicRaft (MLDR)* [44]. MLDR performs dynamic reconfiguration in a Raft based replication system. Our tool is the only one which manages to find a correct inductive invariant for MLDR.

To summarize, in this paper we make the following contributions:

- A new technique for inductive invariant inference that works for distributed protocols specified in TLA⁺.
- A tool, *endive*, which implements our inductive invariant inference algorithm. To our knowledge, this is the only existing tool that works directly on TLA⁺.
- An experimental evaluation of our tool on a diverse set of distributed protocol benchmarks.
- The first, to our knowledge, automatic inference of an inductive invariant for an industrial scale Raft-based reconfiguration protocol.

The rest of this paper is organized as follows. Section II presents preliminaries and a formal problem statement. Section III describes our algorithm for inductive invariant inference, along with more details on our technique. Section IV provides an experimental evaluation of our algorithm, as implemented in our tool, *endive*. Section V examines related work, and Section VI presents conclusions and goals for future work.

II. PRELIMINARIES AND PROBLEM STATEMENT

1) *TLA⁺*: Throughout the rest of this paper, we adopt the notation of TLA⁺ [33] for formally specifying systems and their correctness properties. TLA⁺ is an expressive, high level specification language for specifying distributed and

concurrent protocols. It has also been used effectively in industry for specifying and verifying correctness of protocol designs [5], [38]. Note that our tool accepts models written in TLA⁺. Figure 1 describes a simple lock server protocol [42], [49] in TLA⁺ which we will use as a running example.

2) *Symbolic Transition Systems*: The protocols considered in this paper can be modeled as parameterized *symbolic transition systems* (STSs), like the one shown in Figure 1. This STS is parameterized by two *sorts*, called *Server* and *Client* (Line 1). Each sort represents an uninterpreted constant symbol that can be interpreted as any set of values. In this paper we assume that sorts may only be interpreted over finite domains of distinct values e.g. $Server = \{a_1, \dots, a_k\}$ and $Client = \{c_1, \dots, c_k\}$.

In addition to types, a STS also has a set of *state variables*. A *state* is an assignment of values to all state variables. We use the notation $s \models P$ to denote that state s *satisfies* state predicate P , i.e., that P evaluates to true once we replace all state variables in P by their values as given by s .

The STS of Figure 1 has two state variables, called *locked* and *held* (Line 2). The state predicate *Init* specifies the possible values of the state variables at an *initial state* of the system (Lines 3-5). *Init* states that initially *locked*[i] is TRUE for all $i \in Server$, and that *held*[i] is $\{\}$ (the empty set) for all $i \in Client$. The predicate *Next* defines the *transition relation* of the STS (Lines 14-16). In TLA⁺, *Next* is typically written as a disjunction of *actions* i.e., possible symbolic transitions. In the example of Figure 1 there are two possible symbolic transitions: either some client c and some server s engage in a “connect” action defined by the *Connect*(c, s) predicate, or some client c and some server s engage in a “disconnect” action defined by the *Disconnect*(c, s) predicate.

Given two states, s and s' , we use the notation $s \rightarrow s'$ to denote that there exists a transition from s to s' , i.e., that the pair (s, s') satisfies the transition relation predicate *Next*. A *behavior* is an infinite sequence of states s_0, s_1, \dots , such that $s_0 \models Init$ and $s_i \rightarrow s_{i+1}$ (i.e., $(s_i, s_{i+1}) \models Next$) for all $i \geq 0$. A state s is *reachable* if there exists a behavior s_0, s_1, \dots , such that $s = s_i$ for some i . We use *Reach*(M) to denote the reachable states of a transition system M .

The entire set of behaviors of the system is defined as a single temporal logic formula *Spec* (Line 17). In TLA⁺, *Spec* is typically defined as the TLA⁺ formula $Init \wedge \square [Next]_{Vars}$, where \square is the “always” operator of linear temporal logic, and $[Next]_{Vars}$ represents a transition which either satisfies *Next* or is a *stuttering* step, i.e., where all state variables in *Vars* remain unchanged.

3) *Invariants*: In this paper we are interested in the verification of safety properties, and in particular *invariants*, which are state predicates that hold at all reachable states. Formally, a state predicate P is an *invariant* if $s \models P$ holds for every reachable state s . The model of Figure 1 contains one such candidate invariant, specified by the predicate *Safe* (Line 18). *Safe* states that there cannot be two different clients c_i and c_j which both hold locks to the same server.

```

1 CONSTANT Server, Client
2 VARIABLE locked, held

3 Init  $\triangleq$ 
4    $\wedge \text{locked} = [i \in \text{Server} \mapsto \text{TRUE}]$ 
5    $\wedge \text{held} = [i \in \text{Client} \mapsto \{\}]$ 

6 Connect(c, s)  $\triangleq$ 
7    $\wedge \text{locked}[s] = \text{TRUE}$ 
8    $\wedge \text{held}' = [\text{held EXCEPT } ![c] = \text{held}[c] \cup \{s\}]$ 
9    $\wedge \text{locked}' = [\text{locked EXCEPT } ![s] = \text{FALSE}]$ 

10 Disconnect(c, s)  $\triangleq$ 
11    $\wedge s \in \text{held}[c]$ 
12    $\wedge \text{held}' = [\text{held EXCEPT } ![c] = \text{held}[c] \setminus \{s\}]$ 
13    $\wedge \text{locked}' = [\text{locked EXCEPT } ![s] = \text{TRUE}]$ 

14 Next  $\triangleq$ 
15    $\vee \exists c \in \text{Client}, s \in \text{Server} : \text{Connect}(c, s)$ 
16    $\vee \exists c \in \text{Client}, s \in \text{Server} : \text{Disconnect}(c, s)$ 

17 Spec  $\triangleq \text{Init} \wedge \square[\text{Next}]_{\langle \text{locked}, \text{held} \rangle}$ 

18 Safe  $\triangleq$ 
19    $\forall c_i, c_j \in \text{Client} :$ 
20    $(\text{held}[c_i] \cap \text{held}[c_j] \neq \{\}) \Rightarrow (c_i = c_j)$ 

```

Fig. 1. A simple parameterized protocol defined in TLA⁺.

4) *Verification*: The verification problem consists in checking that a system satisfies its specification. In TLA⁺, both the system and the specification are written as temporal logic formulas. Therefore, expressed in TLA⁺, the safety verification problem we consider in this paper consists of checking that the temporal logic formula

$$\text{Spec} \Rightarrow \square \text{Safe} \quad (1)$$

is *valid* (i.e., true under all assignments). That is, establishing that *Safe* is an invariant of the system defined by *Spec*.

5) *Finite State Instances*: *Instantiating* a sort means fixing it to a *finite* domain of distinct elements. For example, we can instantiate *Server* to be the set $\{a_1, a_2\}$ (meaning there are only two servers, denoted a_1 and a_2), and *Client* to be the set $\{c_1, c_2\}$ (meaning there are only two clients, denoted c_1 and c_2). For the parameterized symbolic transition systems considered in this paper, when we instantiate all sorts of an STS, the system becomes finite-state, i.e., the set of all possible system states is finite.

6) *Inductive Invariants*: A standard technique for solving the safety verification problem (1) is to come up with an *inductive invariant* [36]. That is, a state predicate *Ind* which satisfies the following conditions:

$$\text{Init} \Rightarrow \text{Ind} \quad (2)$$

$$\text{Ind} \wedge \text{Next} \Rightarrow \text{Ind}' \quad (3)$$

$$\text{Ind} \Rightarrow \text{Safe} \quad (4)$$

where *Ind'* denotes the predicate *Ind* where state variables are replaced by their primed, next-state versions. Conditions (2) and (3) are, respectively, referred to as *initiation* and *consecution*. Condition (2) states that *Ind* holds at all initial states.

$$A_1 \triangleq \forall s \in \text{Server} : \forall c \in \text{Client} : \text{locked}[s] \Rightarrow (s \notin \text{held}[c])$$

$$\text{Ind} \triangleq \text{Safe} \wedge A_1$$

Fig. 2. A lemma invariant, A_1 , and an inductive invariant, *Ind*, for the protocol and safety property given in Figure 1.

Condition (3) states that *Ind* is *inductive*, i.e., if it holds at some state s then it also holds at any successor of s . Together these two conditions imply that *Ind* is also an invariant, i.e., that it holds at all reachable states. Condition (4) states that *Ind* is stronger than the invariant *Safe* that we are trying to prove. Therefore, if all reachable states satisfy *Ind*, they also satisfy *Safe*, which establishes (1). The difficulty is in coming up with an inductive invariant which satisfies the above conditions. The problem we consider in this paper is to infer such an inductive invariant automatically.

7) *Lemma Invariants*: An inductive invariant *Ind* typically has the form $\text{Ind} \triangleq \text{Safe} \wedge A_1 \wedge \dots \wedge A_k$, where the conjuncts A_1, \dots, A_k are state predicates and we refer to them as *lemma invariants*. Observe that each A_i must itself be an invariant. The reason is that *Ind* must be an invariant, i.e., must contain all reachable states, and since *Ind* is stronger than (i.e., contained in) each A_i , each A_i must itself contain all reachable states. Furthermore, although all lemma invariants must be invariants, they need not be individually inductive. However, the conjunction of all lemma invariants together with the safety property *Safe* must be inductive. Figure 2 provides an example of an inductive invariant, *Ind*, for the protocol and safety property given in Figure 1. *Ind* contains a single lemma invariant, A_1 .

8) *Counterexamples to Induction*: Given a state predicate P (which is typically a candidate inductive invariant), a *counterexample to induction (CTI)* is a state s such that: (1) $s \models P$; and (2) s can reach a state satisfying $\neg P$ in k steps, i.e. there exist transitions $s \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_k$ and $s_k \models \neg P$. That is, a CTI is a state s which proves that P is not inductive i.e., not “closed” under the transition relation. We denote the set of all CTIs of predicate P by $\text{CTIs}(P)$. Note that for any inductive invariant *Ind*, the set $\text{CTIs}(\text{Ind})$ is empty. Given another state predicate Q and a state $s \in \text{CTIs}(P)$, we say that Q *eliminates* s if $s \not\models Q$, i.e., if $s \models \neg Q$.

III. OUR APPROACH

At a high level, our inductive invariant inference method consists of the following steps:

- 1) Generate many candidate lemma invariants, and store them in a repository that we call *Invs*.
- 2) Generate counterexamples to induction for a current candidate inductive invariant, *Ind*. If we cannot find any such CTIs, return *Ind*.
- 3) Select lemma invariants from *Invs* so that all CTIs are eliminated. If we cannot eliminate all CTIs, either give up, or go to Step 1 and populate the repository with more

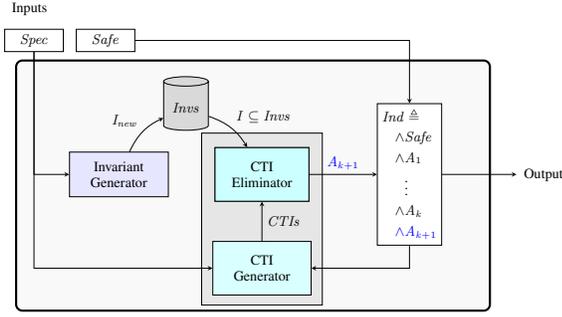


Fig. 3. Components of our technique for inductive invariant inference.

Algorithm 1 Our inductive invariant inference algorithm.

```

1: Inputs:
    $M$ : Finite instance of a parameterized STS
    $Safe$ : Candidate invariant
    $Invs$ : Lemma invariant repository (typically empty initially)
    $G$ : Grammar for invariant generation
2: procedure INFERINDUCTIVEINVARIANT( $M, Safe, G, Invs$ )
3:    $Ind \leftarrow Safe$ 
4:    $X \leftarrow GenerateCTIs(M, Ind)$ 
5:    $Invs \leftarrow GenerateLemmaInvariants(M, Invs, G)$ 
6:   while  $X \neq \emptyset$  do
7:     if  $\exists A \in Invs : A$  eliminates at least one CTI in  $X$  then
8:       pick  $A_{max} \in Invs$  that eliminates the most CTIs from  $X$ 
9:        $Ind \leftarrow Ind \wedge A_{max}$ 
10:       $X \leftarrow X \setminus \{s \in X : s \not\models A_{max}\}$ 
11:     else
12:       either goto Line 5
13:       or return ( $Ind$ , "Fail: couldn't eliminate all CTIs.")
14:     end if
15:      $X \leftarrow GenerateCTIs(M, Ind)$ 
16:   end while
17:   return ( $Ind$ , "Success: managed to eliminate all CTIs.")
18: end procedure

```

lemma invariants. Otherwise, add the selected lemma invariants to Ind and repeat from Step 2.

The conceptual approach is illustrated in Figure 3. Our detailed algorithm is described in Section III-A. Section III-B provides details on our lemma invariant generation procedure, Section III-C provides details on CTI generation, and Section III-D describes the selection of lemma invariants.

A. Inductive Invariant Inference Algorithm

Our inductive invariant inference algorithm is given in pseudocode in Algorithm 1. The algorithm takes as input: (1) a finite instance of a symbolic transition system M , (2) a candidate invariant (safety property) $Safe$, (3) a lemma invariant repository $Invs$, and (4) a grammar G for generating lemma invariant candidates. The use of the grammar is discussed further in Section III-B. $Invs$ may initially be empty, or be pre-populated from previous runs of the algorithm. The algorithm aims to discover an inductive invariant, Ind , of the form $Ind = Safe \wedge A_1 \wedge \dots \wedge A_n$.

The algorithm maintains a current inductive invariant candidate, Ind , which it initializes to $Safe$, the safety property that we are trying to prove (Line 3). It then generates a set X of CTIs of Ind (Line 4). The algorithm may also initialize

the repository of lemma invariants, $Invs$, or add more lemma invariants to $Invs$ if it is initially non-empty (Line 5). The procedures *GenerateLemmaInvariants* and *GenerateCTIs* are described in more detail below, in Sections III-B and III-C, respectively.

In its main loop, the algorithm tries to eliminate all currently known CTIs. As long as the set X of currently known CTIs is non-empty, the algorithm tries to find a lemma invariant in the $Invs$ repository that eliminates the maximal number of remaining CTIs possible. If such a lemma invariant exists, the algorithm adds it as a new conjunct to Ind (Line 9), removes from X the CTIs that were eliminated by the new conjunct (Line 10), and proceeds by attempting to generate more CTIs, since the updated Ind is not necessarily inductive (Line 15).

If no lemma invariant exists in the current repository $Invs$ that can eliminate any of the currently known CTIs (Line 11), then we may either (1) generate more lemma invariants in the repository, or (2) give up. The first choice is implemented by the **goto** statement in Line 12. The second choice represents a failure of the algorithm to find an inductive invariant (Line 13). However, in this case we still return Ind since, even though it is not inductive, it may contain several useful lemma invariants. These lemma invariants are useful in the sense that they might be part of an ultimate inductive invariant.

If all known CTIs have been eliminated, the algorithm terminates successfully and returns Ind (Line 17). Successful termination of the algorithm indicates that the returned Ind is likely to be inductive. However our method does not provide a formal inductiveness guarantee. Ind might not be inductive for a number of reasons. First, as we discuss further in Section III-C, our CTI generation procedure is probabilistic in nature, and therefore *GenerateCTIs* might miss some CTIs. Second, even if the finite-state instance M explored by the algorithm has no remaining CTIs, there might still exist CTIs in other instances of the STS, for larger parameter values.

Even though a candidate invariant returned by a successful termination of Algorithm 1 is not formally guaranteed to be inductive, we ensure soundness of our overall procedure by doing a final check that the discovered candidate inductive invariant is correct using the TLA^+ proof system (TLAPS) [16]. Validation of invariants in TLAPS is discussed further in Section III-E. In practice we found that all of the invariants generated in our evaluation (Section IV) are correct inductive invariants.

We also remark that in the current version of our algorithm and in the current implementation of our tool, we only explore the single finite-state instance of the STS provided by the user, and we do not attempt to automatically increase the bounds of the parameters within the algorithm, as is done for example in the approach described in [24]. This is, however, a relatively straightforward extension to our algorithm, and would like to explore this option in future work.

B. Lemma Invariant Generation

For a given finite instance M of a parameterized transition system, the goal of lemma invariant generation is to produce

$$\begin{aligned}
\langle seed \rangle &::= locked[s] \mid s \in held[c] \mid held[c] = \emptyset \\
\langle quant \rangle &::= \forall s \in Server : \forall c \in Client \\
\langle expr \rangle &::= \langle seed \rangle \mid \neg \langle expr \rangle \mid \langle expr \rangle \vee \langle expr \rangle \\
\langle pred \rangle &::= \langle quant \rangle : \langle expr \rangle
\end{aligned}$$

Fig. 4. Example of a grammar for lemma invariant generation for the *lockserver* protocol shown in Figure 1. The list of unquantified *seed* predicates and the quantifier template, *quant*, are provided as user inputs.

a set of state predicates that are invariants of M . To search for these invariants, we adopt an approach similar to other, *syntax-guided synthesis* based techniques [21], [20] for invariant discovery. We randomly sample invariant candidates from a defined *grammar*, which is generated from a given set of *seed* predicates. Each seed predicate is an atomic boolean predicate over the state variables of the system. Note that the parameterized distributed protocols that we consider in this paper typically have inductive invariants that are universally or existentially quantified over the parameters of the protocol or other values of the system state. So, our invariant generation technique assumes a fixed quantifier template that is provided as input. The provided seed predicates are unquantified predicates that can contain bound variables that appear in the given quantifier template. An example of a simple grammar for the protocol of Figure 1 is shown in Figure 4.

Candidate invariants are produced by generating random predicates over the space of seed predicates. Specifically, a candidate predicate is formed as a random disjunction of seed predicates, where each disjunct may be negated with probability $\frac{1}{2}$. The logical connectives $\{\vee, \neg\}$ are functionally complete [48], so they serve as a simple basis for generating candidate invariants, which we chose to reduce the invariant search space.

For a given set of candidate invariants, C , we check which of the predicates in C are invariants using an explicit state model checker. This can be done effectively due to our use of the small scope hypothesis i.e. the fact that we reason only about a finite instance M of a parameterized transition system. This largely reduces the invariant checking problem to a data processing task. Namely:

- (1) Generate $Reach(M)$, the set of reachable states of M .
- (2) Check that $s \models P$ for each predicate $P \in C$ and each $s \in Reach(M)$.

Note that after (1) has been completed once, the set of reachable states can be cached and only step (2) must be re-executed when searching for additional invariants.

In theory, the worst case cost of step (2) is proportional to $|C| \cdot |Reach(M)|$. In practice, however, it can often be much less costly than this, since once a state violates a predicate P , P need not be checked further. Furthermore, both of the above computation steps are highly parallelizable, a fact we make use of in our implementation, as discussed further in Section IV-A.

We also remark that, in practice, the *GenerateLemmaInvariants* procedure is configured to search for candidate invariants of a fixed term size i.e. with a fixed or maximal number of disjuncts. In our implementation, presented in Section IV-A, we utilize this to search for smaller invariants (fewer terms) first, before searching for larger ones. That is, we prefer to eliminate CTIs if possible with smaller invariants before searching for larger ones. This aims to bias our procedure towards discovery of compact inductive invariant lemmas.

Furthermore, since *GenerateLemmaInvariants* does not employ an exhaustive search for invariants over a given space of predicates, it accepts a numeric parameter, N_{lemmas} , which determines how many candidate predicates to sample. More details of how the concrete values of this parameter are configured are discussed in our evaluation, in Section IV.

C. CTI Generation

Each round of our algorithm relies on access to a set of multiple CTIs, as a means to prioritize between different choices of new lemma invariants. To generate these CTIs, we use a probabilistic technique proposed in [34] that utilizes the TLC explicit state model checker [52]. Given a finite instance of a STS M with system states S , transition relation predicate $Next$, and given candidate inductive invariant Ind , the procedure *GenerateCTIs*(M, Ind) works by calling the TLC model checker. TLC attempts to randomly sample states $s_0 \in S$ for which there exists a sequence of states $s_1, s_2, \dots, s_{k-1}, s_k \in S$, such that both of the following hold:

- $\forall i = 0, 1, \dots, k-1 : (s_i, s_{i+1}) \models Next \wedge s_i \models Ind$
- $s_k \not\models Ind$.

The model checker will report this behavior, and all states $s_0, s_1, s_2, \dots, s_{k-1}$ are recorded as counterexamples to induction.

Due to the randomized nature of this technique, the CTI generation procedure requires a given parameter, N_{ctis} , that effectively determines how many possible states TLC will attempt to sample before terminating the CTI generation procedure. This is required, since, for systems with sufficiently large state spaces, even if finite, sampling all possible states is infeasible. Generally, this parameter can be tuned based on the amount of compute power available to the tool, or a latency tolerance of the user. We discuss more details of this parameter and how it is tuned in our experiments in Section IV.

In practice, during our evaluation we found that TLC was able to effectively generate many thousands of CTIs at each round of the inference algorithm using the above technique. This provided an adequately diverse distribution of CTIs for effectively guiding our counterexample elimination procedure, which we describe in more detail in Section III-D. Section IV presents more detailed metrics on CTI generation as measured when testing our implementation on a variety of protocol benchmarks. In future we feel it would be valuable to explore and compare with other, SMT/SAT based techniques for this type of counterexample generation task [18], [30].

D. Lemma Invariant Selection by CTI Elimination

The task of selecting lemma invariants for use as inductive invariant conjuncts is based on a process of CTI elimination, as described briefly in Section III-A. That is, CTIs are used as guidance for which invariants to choose for new lemma invariants to append to the current inductive invariant candidate. Once a sufficiently large set of CTIs has been generated, as discussed in Section III-C, we select lemma invariants using a greedy heuristic of CTI elimination, which we describe below.

1) *CTI Elimination*: Recall that a CTI s is *eliminated* by a state predicate A if $s \not\models A$. When examining a current set of CTIs, X , our algorithm looks for the next lemma invariant $A \in Invs$ that eliminates the most CTIs in X . The algorithm will continue choosing additional lemma invariants according to this strategy until all counterexamples are eliminated, or until it cannot eliminate any further counterexamples. Each selected invariant $A_i \in Invs$ will be appended as a new conjunct to the current inductive invariant candidate i.e. $Ind \leftarrow Ind \wedge A_i$. Once all counterexamples have been eliminated, the tool will terminate and return a final candidate inductive invariant. This is a simple heuristic for choosing new invariant conjuncts that aims to bias the overall inductive invariant towards being relatively concise. That is, if we have a choice between two alternate lemma conjuncts to choose from, we prefer the conjunct that eliminates more CTIs.

More generally, lemma selection at each round of the algorithm can be viewed as a version of the set covering problem [15]. Ideally, we would like to find the smallest set of lemma invariants that eliminate (i.e. cover) the set of CTIs X . Solving this problem optimally is known to be NP-complete [28], but we have found a greedy heuristic [14] to work sufficiently well in our experiments, the results of which are presented in Section IV. In future we would like to explore more sophisticated heuristics for lemma selection that take into account additional metrics, like syntactic invariant size, quantifier depth, etc.

E. Validation of Inductive Invariant Candidates

If our inference algorithm terminates successfully, it will return a candidate inductive invariant. Since we look for invariants on finite protocol instances, though, this candidate may not be an inductive invariant for general (e.g. unbounded) protocol instances. So, upon termination, we check to see if the returned candidate invariant is truly inductive for all protocol instances by passing it to an SMT solver. Currently, we use the TLA⁺ proof system (TLAPS) [16] for this step, which generates an SMT encoding for TLA⁺ [37].

For many of the protocols we tested and the invariants discovered by our tool, we found that this step was fully automated (see Section IV and Table III in the Appendix). That is, no user assistance was required to establish validity of the discovered invariant. In cases where the underlying solver cannot automatically prove the candidate inductive invariant, some amount of human guidance can be provided by decomposing the proof into smaller SMT queries. We have completed this validation step for all of the inductive invariant

candidates discovered in our experiments, and we confirmed that all candidate invariants produced by our tool were indeed correct inductive invariants (see Section IV).

IV. IMPLEMENTATION AND EVALUATION

A. Implementation and Experimental Setup

Our invariant inference algorithm is implemented in a tool, *endive*, whose main implementation consists of approximately 2200 lines of Python code. There are also some optimized subroutines which consist of an additional few hundred lines of C++ code. Internally, *endive* makes use of version 2.15 of the TLC model checker [52], with some minor modifications to improve the efficiency of checking many invariants simultaneously. TLC is used by *endive* for most of the algorithm's compute intensive verification tasks, like checking candidate lemma invariants (Section III-B) and CTI elimination checking (Section III-D1).

For all of the experiments discussed below, *endive* is configured to use 24 parallel TLC worker threads for invariant checking, 4 parallel threads for CTI generation, and 4 threads for CTI elimination. CTI generation and CTI elimination can be parallelized further in a straightforward manner, but we limit these procedures to 4 parallel threads to simplify certain aspects of our current implementation.

For each benchmark run, we initialize *Invs* (as explained in Algorithm 1) as an empty set and configure the lemma invariant generation procedure discussed in Section III-B with a parameter value of $N_{lemmas} = 15000$. The grammars used for invariant generation were mined from predicates appearing in each protocol specification.

We configure our CTI generation procedure with a parameter value of $N_{ctis} = 50000$. N_{ctis} does not directly correspond to how many concrete CTI states will be generated, but a higher value indicates TLC will sample more states when searching for CTIs. We also limit the maximum number of CTIs returned by each call to the *GenerateCTIs* procedure to 10000 states. In theory, generating more CTIs provides better counterexample diversity, and is therefore better for our CTI elimination heuristics. We impose an upper limit, however, to avoid scalability issues in our tool's current implementation. In practice we found this limit sufficient to provide effective guidance for lemma invariant selection.

All of our experiments were run on a 48-core Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz machine with 196GB of RAM.

B. Benchmarks

To evaluate *endive*, we measured its performance on 29 protocols selected from an existing benchmark set published in [24]. We also evaluate *endive* on an additional, industrial scale protocol, *MongoLoglessDynamicRaft (MLDR)*, which is a recent protocol for distributed dynamic reconfiguration in a Raft based replication system [45], [44].

1) *Protocol Conversion*: The 29 benchmarks we used from [24] were originally specified in Ivy [42], but *endive* accepts protocols in TLA⁺, so it was necessary to manually translate the protocols from Ivy to TLA⁺. There are significant differences in how protocols are specified in Ivy and TLA⁺. The underlying approach to modeling systems as discrete transition systems, however, by specifying initial states and a transition relation, are common between them. In our manual translation, we aimed to emulate the original Ivy model as close as possible.

The formal specification for the *MongoLoglessDynamicRaft* protocol (*MLDR*) was originally written in TLA⁺ [45]. Thus, in order to compare with other invariant inference tools which accept Ivy as their input language, we had to translate *MLDR* from TLA⁺ into Ivy. This conversion process was highly nontrivial due to the significant differences between the Ivy and TLA⁺ languages. TLA⁺ is a very expressive language that includes integers, strings, sets, functions, records, and sequences as primitive data types along with their standard semantics. In contrast, the Ivy modeling language, RML [42], includes only basic, first order relations and functions. For more complex datatypes (e.g. arrays or sequences), their semantics must be defined and axiomatized manually.

An artifact containing all of our source code and instructions for reproducing our evaluation results can be found at [43]. A public, open-source version of our tool is also available at [1].

C. Results

Our overall results are shown in Table I. We compared *endive* with four recent, state of the art techniques for inferring invariants of distributed protocols: IC3PO [24], fol-ic3 [29], SWISS [25], and DistAI [51]. Note that *endive* accepts protocols in TLA⁺, whereas all other tools accept protocols in Ivy or mypyvy.

The numbers shown for both IC3PO and fol-ic3 in Table I are as reported in the evaluation presented in [24], with timeouts indicated by a *TO* entry. For the SWISS results in Table I, where possible, we show the runtime numbers reported in [25], indicated with a † mark. For the benchmarks in Table I that were not tested in [25], we present the results from our own runs of the tool, all using default SWISS configuration parameters. We ran SWISS both with an invariant template matching our own template for *endive* and also in automatic mode, and report the better of the two results. The results for DistAI are reported from our runs using the tool in its default configuration. For DistAI and SWISS, we report an *err* result in cases where the tool returned an error without producing a result. We report a *fail* result in cases where DistAI or SWISS terminated without error but did not discover an inductive invariant. In all cases where a benchmark protocol was not available in the required input language for the corresponding tool, we mark this with an *n/a* entry.

For each benchmark result in Table I, we report the total wall clock time to discover an inductive invariant in the *Time* column, along with the number of total lemma invariants contained in the discovered invariant, including the safety

No.	Protocol	<i>endive</i>		IC3PO		fol-ic3		SWISS		DistAI	
		Time	Inv	Time	Inv	Time	Inv	Time	Inv	Time	Inv
1	tla-consensus	1	1	0	1	1	1	1	2	2	1
2	tla-tcommit	2	1	1	2	2	3	2	8	2	7
3	i4-lock-server	7	2	1	2	1	2	†1	2	err	
4	ex-quorum-leader-election	11	2	3	5	24	8	11	5	3	8
5	pyv-toy-consensus-forall	19	3	3	5	11	5	†3	7	err	
6	tla-simple	8	2	6	3	TO		28	8	err	
7	ex-lockserv-automaton	23	9	7	12	10	12	fail		2	13
8	tla-simpleregular	10	4	8	4	57	9	65	21	err	
9	pyv-sharded-kv	312	6	10	8	22	10	†4024		2	16
10	pyv-lockserv	35	9	11	12	8	11	†3684		2	13
11	tla-twophase	43	10	14	9	9	12	33	24	29	306
12	i4-learning-switch	TO		14	10	TO		TO		21	32
13	ex-simple-decentralized-lock	44	4	19	15	4	8	1	2	26	17
14	i4-two-phase-commit	69	11	27	11	8	9	†6	15	17	67
15	pyv-consensus-wo-decide	127	8	50	9	168	26	†18	8	err	
16	pyv-consensus-forall	175	8	99	10	2461	27	†29	9	err	
17	pyv-learning-switch	TO		127	13	TO		†959		79	70
18	i4-chord-ring-maintenance	n/a		229	12	TO		†TO		53	164
19	pyv-sharded-kv-no-lost-keys	13	2	3	2	3	2	†1	4	fail	
20	ex-naive-consensus	40	4	6	4	73	18	18	5	fail	
21	pyv-client-server-ae	46	2	2	2	877	15	†3	5	err	
22	ex-simple-election	24	4	7	4	32	10	9	5	err	
23	pyv-toy-consensus-epr	19	4	9	4	70	14	†2	4	err	
24	ex-toy-consensus	7	2	10	3	21	8	6	4	err	
25	pyv-client-server-db-ae	4941	8	17	6	TO		†24	13	err	
26	pyv-hybrid-reliable-broadcast	n/a		587	4	1360	23	†TO		err	
27	pyv-firewall	38	5	2	3	7	8	75	5	err	
28	ex-majorityset-leader-election	53	4	72	7	TO		28	10	err	
29	pyv-consensus-epr	247	8	1300	9	1468	30	72	10	err	
30	mldr	2025	6	TO		n/a		err		err	

TABLE I
DISTRIBUTED PROTOCOL BENCHMARK RESULTS.

property, in the *Inv* column. Note that the number of total lemmas in the invariants discovered by SWISS was not reported in [25]. Thus, we report the number of lemmas discovered by SWISS in our own runs, for the cases where we were able to run SWISS successfully to produce an invariant.

More detailed statistics on the *endive* benchmark results are provided in Appendix A, specifically: the number of eliminated CTIs, runtime profiling information, finite instance sizes used, and automation level of the TLAPS proofs.

D. Comparison with Other Tools

Although Table I relates our approach to several others, we note that our tool is not directly comparable to other tools. The most fundamental difference is that our tool accepts TLA⁺ whereas all other tools in Table I accept Ivy or mypyvy. Furthermore, some tools work only with the restricted decidable EPR or extended EPR fragments of Ivy. To our knowledge, this is the case with SWISS and DistAI. As a result, our tool is a-priori less automated than other tools, following a standard tradeoff between expressivity and automation. In practice, however, and despite this theoretical limitation, our tool produces a result in most cases, while some of the a-priori more automated tools time out or fail.

Another important difference between the tools of Table I is what kind of inductive invariants can be produced by each tool. In our case, the user provides the grammar of possible lemma invariants as an input to the tool, allowing both universal and existentially quantified invariants (\forall and \exists). DistAI is limited to only universally quantified (\forall) invariants, and SWISS is

limited to invariants that fall into the extended EPR fragment, though it can learn both universal and existentially quantified invariants. Both fol-ic3 and IC3PO attempt to learn the quantifier structure itself during counterexample generalization, and can infer both universal and existentially quantified invariants. These tools do not always guarantee, however, that the discovered invariants will fall into a decidable logic fragment. Thus, they provide no explicit guarantee that the overall inference procedure will, in general, be fully automated.

E. Discussion

Our tool, *endive*, was able to successfully discover an inductive invariant for 25 of the 29 protocol benchmarks from [24], and all of the invariants it discovered were proven correct using TLAPS. For the two protocols out of these 29 that our tool did not solve, *pyv-learning-switch* and *i4-learning-switch*, this was due to scalability limitations of CTI generation, which we believe could be improved with a smarter CTI generation algorithm or by incorporating a symbolic model checker [30] for this task.

endive was also able to automatically discover an inductive invariant for a key safety property of *MLDR*, a Raft-based distributed dynamic reconfiguration protocol [45]. This protocol, reported in Table I as *mldr*, is a significantly more complex, industrial scale protocol [44]. IC3PO was not able to discover an invariant for our Ivy model of the *MLDR* protocol after a 1 hour timeout when given the same instance size used in the TLA^+ model given to *endive*. SWISS and DistAI both produced an error when run on our Ivy model of *MLDR*.

Generally, the wall clock time taken for *endive* to discover an inductive invariant is of a similar order of magnitude to IC3PO. *endive* even outperforms IC3PO in some cases, despite the fact that *endive* works with TLA^+ and IC3PO works with Ivy. Moreover, in several cases where *endive*'s runtime exceeds that of IC3PO, *endive* is able to discover a smaller inductive invariant (e.g. *pyv-lockserv*, *ex-simple-decentralized-lock*, *pyv-consensus-forall*). Additionally, *endive* is often able to discover a considerably smaller invariant than tools like DistAI and SWISS. For example, on *tlatwophase*, *endive* learns an invariant with 10 overall conjuncts, whereas SWISS learns a 24 conjunct invariant, and DistAI learns a much larger invariant, with over 300 conjuncts. *endive* performs similarly well for the *tlasimpleregular* and *i4-two-phase-commit* benchmarks. This demonstrates that *endive* compares favorably against other enumerative approaches for inductive invariant inference, both in terms of efficiency and compactness of invariants, while also working over TLA^+ , a much more expressive input language.

It is additionally worth noting that our current *endive* implementation is not highly optimized. In particular, the TLC model checker, used internally by *endive*, is implemented in Java and interprets TLA^+ specifications dynamically [31], rather than compiling models to a low level, native representation as done by tools like SPIN [26]. As a result, TLC may not be the most efficient for our inference procedure, and could likely be optimized further.

V. RELATED WORK

There are several recently published techniques that attempt to solve the problem of inductive invariant inference for distributed protocols. The IC3PO tool [24], which extended the earlier I4 tool [35], uses a technique based on IC3 [10] with a novel *symmetry boosting* technique that serves to accelerate IC3/PDR and also to infer the quantifier structure of lemma invariants. The fol-ic3 algorithm presented in [29] presents another IC3 based algorithm which uses a novel *separators* technique for discovering quantified formulas to separate positive and negative examples during invariant inference. SWISS [25] is another recent approach that uses an enumerative search for quantified invariants while using the Ivy tool to validate possible inductive candidates. It relies on SMT based reasoning over an unbounded domain, and does not reason directly about finite instances of distributed protocols. DistAI [51] uses a similar approach but additionally utilizes a technique of sampling reachable protocol states to filter invariants, which is similar to our approach of executing explicit state model checking as a means to quickly discover invariants. DistAI is limited, however, to learning only universally quantified invariants.

In addition to these inductive invariant inference techniques, there also exists prior work on alternative techniques for parameterized protocol verification. These include approaches based on cutoff detection [3], regular model checking [9], and symbolic backward reachability analysis [23].

More broadly, there exist many prior techniques for the automatic generation of program and protocol invariants that rely on data driven or grammar based approaches. Houdini [22] and Daikon [19] both use enumerative checking approaches to discover program invariants. FreqHorn [20] tries to discover quantified program invariants about arrays using an enumerative approach that discovers invariants in stages and also makes use of the program syntax. Other techniques have also tried to make invariant discovery more efficient by using improved search strategies based on MCMC sampling [46].

VI. CONCLUSIONS AND FUTURE WORK

We presented a new technique for inferring inductive invariants for distributed protocols specified in TLA^+ and evaluated it on a diverse set of protocol benchmarks. Our approach is novel in that: (1) it is the first, to our knowledge, to infer inductive invariants directly for protocols specified in TLA^+ and (2) it is based around a core procedure for generating plain, not necessarily inductive, lemma invariants. Our results show that our approach performs strongly on a diverse set of distributed protocol benchmarks. In addition, it is able to discover an inductive invariant for an industrial scale dynamic reconfiguration protocol.

In future, our tool can be extended to allow for automatic quantifier template search, and further optimizations can be made to the lemma invariant generation and selection procedures. It would be interesting to explore ways in which the invariant generation procedure can be guided more directly by the generated counterexamples to induction, as a means to

prune the search space of candidate invariants more efficiently, perhaps using techniques similar to those presented in [46]. We would also be interested to see if quantifier structures can be inferred from the protocol syntax itself. Improving the performance of TLC, or experimenting with other, more efficient model checkers [26] would be another avenue, since model checking performance is a main bottleneck of our current approach.

REFERENCES

- [1] endive invariant inference tool, Github repository. <https://github.com/will62794/endive>, 2022.
- [2] mypyvy tool, github repository. <https://github.com/wilcoxjay/mypyvy>, 2022.
- [3] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *Int. J. Softw. Tools Technol. Transf.*, 18(5):495–516, Oct 2016.
- [4] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions? In *International Conference on Computer Aided Verification*, pages 221–234. Springer, 2001.
- [5] Robert Beers. Pre-RTL formal verification: An Intel experience. In *2008 45th ACM/IEEE Design Automation Conference*, pages 806–811, 2008.
- [6] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. volume 58, pages 117 – 148, 12 2003.
- [7] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability of Parameterized Verification. *Synthesis Lectures on Distributed Computing Theory*, 6(1):1–170, 2015.
- [8] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slaton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP ’21, page 836–850. Association for Computing Machinery, 2021.
- [9] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract Regular Model Checking. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 372–386, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [10] Aaron R Bradley. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.
- [11] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol (Short Paper). In Bruno Bernardo and Diego Marmosoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASICs)*, pages 10:1–10:8, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- [12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, jun 1992.
- [13] Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal Verification of Multi-Paxos for Distributed Consensus. In *International Symposium on Formal Methods*, pages 119–136. Springer, 2016.
- [14] V. Chvatal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235, aug 1979.
- [15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.
- [16] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernan Vanzetto. TLA+ Proofs. *Proceedings of the 18th International Symposium on Formal Methods (FM 2012)*, Dimitra Giannakopoulou and Dominique Mery, editors. Springer-Verlag Lecture Notes in Computer Science, 7436:147–154, January 2012.
- [17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [18] Rafael Dutra, Kevin Laefer, Jonathan Bachrach, and Koushik Sen. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE ’18*, page 549–559. Association for Computing Machinery, 2018.
- [19] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.
- [20] Grigory Fedyukovich and Rastislav Bodík. Accelerating Syntax-Guided Invariant Synthesis. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 251–269, Cham, 2018. Springer International Publishing.
- [21] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified Invariants via Syntax-Guided Synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 259–277, Cham, 2019. Springer International Publishing.
- [22] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME ’01, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.
- [23] Silvio Ghilardi and Silvio Ranise. MCMT: A Model Checker modulo Theories. In *Proceedings of the 5th International Conference on Automated Reasoning, IJCAR’10*, page 22–29, Berlin, Heidelberg, 2010. Springer-Verlag.
- [24] Aman Goel and Karem Sakallah. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *NASA Formal Methods Symposium*, pages 131–150. Springer, 2021.
- [25] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding Invariants of Distributed Systems: It’s a Small (Enough) World After All. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.
- [26] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.
- [27] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.
- [28] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.
- [29] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-Order Quantified Separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 703–717. Association for Computing Machinery, 2020.
- [30] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct 2019.
- [31] Markus A Kuppe. A Verified and Scalable Hash Table for the TLC Model Checker: Towards an Order of Magnitude Speedup. Master’s thesis, University of Hamburg., 2017.
- [32] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [33] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Jun 2002.
- [34] Leslie Lamport. Using TLC to Check Inductive Invariance. <http://lamport.azurewebsites.net/tla/inductive-invariant.pdf>, 2018.
- [35] Haojun Ma, Aman Goel, Jean Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *SOSP 2019 - Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.
- [36] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, Heidelberg, 1995.
- [37] Stephan Merz and Hernán Vanzetto. Encoding TLA+ into Many-Sorted First-Order Logic. In Michael Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklos Biro, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 54–69, Cham, 2016. Springer International Publishing.
- [38] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.
- [39] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, pages 305–320, USA, 2014. USENIX Association.

- [40] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of Inferring Inductive Invariants. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 217–231. ACM, 2016.
- [41] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.
- [42] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.
- [43] William Schultz, Ian Dardik, and Stavros Tripakis. Artifact for FM-CAD 2022 paper: Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. <https://doi.org/10.5281/zenodo.6994922>, August 2022.
- [44] William Schultz, Ian Dardik, and Stavros Tripakis. Formal Verification of a Distributed Dynamic Reconfiguration Protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2022*, page 143–152, Philadelphia, PA, USA, 2022. Association for Computing Machinery.
- [45] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and Analysis of a Logless Dynamic Reconfiguration Protocol. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [46] Rahul Sharma and Alex Aiken. From Invariant Checking to Invariant Inference Using Randomized Search. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2014.
- [47] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, page 1493–1509. Association for Computing Machinery, 2020.
- [48] William Wernick. Complete sets of logical functions. *Transactions of the American Mathematical Society*, 51:117–132, 1942.
- [49] James R. Wilcox, Doug Woos, Pavel Panchevka, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015.
- [50] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016*, page 154–165. Association for Computing Machinery, 2016.
- [51] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.
- [52] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66. Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

APPENDIX A

DETAILED BENCHMARK RESULTS

Table II gives a more detailed breakdown of the results presented in Table I for our *endive* invariant inference tool. The *Check*, *Elim*, and *CTIGen* columns of Table II indicate, respectively, the wall clock time in seconds for (1) checking

candidate lemma invariants, (2) eliminating CTIs, and (3) generating CTIs. The *CTIs* column indicates the total number of eliminated CTIs.

Recall that we limit the maximum number of generated CTIs to 10000 per round, as mentioned in Section IV-A. This explains why some protocol results for the *endive* tool report elimination of exactly 10000 CTIs. For example, for the *tlaw* benchmark, an inductive invariant was discovered in a single round of the algorithm loop (starting at Line 6 of Algorithm 1), so no more than 10000 CTIs were generated in the entire run. If the benchmark run eliminated greater than 10000 CTIs, this indicates that it ran for more than 1 round.

Also, for protocols that eliminated 0 CTIs (e.g. *tlacensus*, *tlaccommit*), this indicates that the starting safety property was already inductive. Thus, no CTIs were ever generated and no lemma invariants were needed. Similarly, some protocols eliminated a nonzero amount of CTIs less than 10000 (e.g. *ex-quorum-leader-election*). This may be the case when no more than a single round of the algorithm was needed to discover an inductive invariant, or that the number of generated counterexamples at each round did not exceed 10000. Recall that, even within a single round of the algorithm, as shown in Algorithm 1, it is possible to discover multiple new lemma invariants.

Additional statistics on the instance sizes used during invariant inference and the degree of automation required for TLAPS proofs are shown in Table III. The *TLAPS Auto* column indicates whether the TLAPS proof of the inductive invariant discovered by *endive* was completely automatic (indicated with a \checkmark), or required some user assistance (indicated with a \times).

To provide more fine-grained detail on the level of automation for each TLAPS proof, the *TLAPS Auto* column also includes the number of verification conditions in the induction check that were proved fully automatically. For a protocol with a transition relation of the form $Next = T_1 \vee \dots \vee T_k$ and an inductive invariant candidate $Ind = A_1 \wedge \dots \wedge A_n$, the consecution check $Ind \wedge Next \Rightarrow Ind'$ is typically the most significant verification burden, and can be trivially decomposed into $k \cdot n$ verification conditions (VCs). That is, a verification condition $Ind \wedge T_j \Rightarrow A'_i$ is generated for each $j \in \{1, \dots, k\}$ and $i \in \{1, \dots, n\}$, giving $k \cdot n$ total VCs. We notate these statistics in the *TLAPS Auto* column as (# VCs proved automatically / $k \cdot n$ total VCs). Protocols that were proved fully automatically are shown as $(k \cdot n / k \cdot n)$. The *Check (s)* column also shows the total time in seconds needed to check each proof, as measured on a 2020 M1 Macbook Air using version 1.4.5 of the TLA+ proof manager.

TABLE II
DETAILED PROFILING RESULTS FOR THE *endive* RESULTS FROM TABLE I.

No.	Protocol	Time	CTIs	Check	Elim	CTIGen
1	tla-consensus	1	0	0	0	1
2	tla-tcommit	2	0	0	0	2
3	i4-lock-server	7	12	2	2	4
4	ex-quorum-leader-election	11	204	2	2	7
5	pyv-toy-consensus-forall	19	412	2	2	15
6	tla-simple	8	15	2	2	5
7	ex-lockserv-automaton	23	3624	6	8	9
8	tla-simpleregular	10	1972	3	3	5
9	pyv-sharded-kv	312	11715	17	46	249
10	pyv-lockserv	35	3654	11	11	13
11	tla-twophase	43	10000	10	22	12
12	i4-learning-switch	TO				
13	ex-simple-decentralized-lock	44	2035	13	18	14
14	i4-two-phase-commit	69	10408	18	19	33
15	pyv-consensus-wo-decide	127	12995	56	39	32
16	pyv-consensus-forall	175	10609	63	25	88
17	pyv-learning-switch	TO				
18	i4-chord-ring-maintenance	n/a				
19	pyv-sharded-kv-no-lost-keys	13	404	2	2	9
20	ex-naive-consensus	40	10000	10	15	16
21	pyv-client-server-ae	46	10000	2	4	40
22	ex-simple-election	24	551	10	7	8
23	pyv-toy-consensus-epr	19	384	8	6	6
24	ex-toy-consensus	7	14	2	2	4
25	pyv-client-server-db-ae	4941	12546	4657	46	239
26	pyv-hybrid-reliable-broadcast	n/a				
27	pyv-firewall	38	1740	11	22	7
28	ex-majorityset-leader-election	53	10000	12	15	26
29	pyv-consensus-epr	247	16269	80	38	129
30	mldr	2025	7751	1272	651	102

TABLE III
ADDITIONAL STATISTICS FOR *endive* RESULTS REPORTED IN TABLE I.

No.	Protocol	Instance Size	TLAPS Auto	Check (s)
1	tla-consensus	Value={v1,v2,v3}	✓ (1/1)	13
2	tla-tcommit	RM={rm1,rm2,rm3}	✓ (2/2)	1
3	i4-lock-server	Server={s1,s2} Client={c1,c2}	✓ (4/4)	1
4	ex-quorum-leader-election	Node={n1,n2,n3,n4}	✓ (4/4)	1
5	pyv-toy-consensus-forall	Node={n1,n2,n3} Value={v1,v2}	✓ (6/6)	1
6	tla-simple	N=4	✓ (4/4)	1
7	ex-lockserv-automaton	Node={n1,n2,n3}	✓ (45/45)	6
8	tla-simpleregular	N=3	✓ (12/12)	1
9	pyv-sharded-kv	Node={n1,n2,n3} Key={k1,k2} Value={v1,v2}	✓ (18/18)	15
10	pyv-lockserv	Node={n1,n2,n3}	✓ (45/45)	6
11	tla-twophase	RM={rm1,rm2,rm3}	✗ (68/70)	18
12	i4-learning-switch	TO		
13	ex-simple-decentralized-lock	Node={n1,n2,n3}	✓ (8/8)	17
14	i4-two-phase-commit	Node={n1,n2,n3}	✓ (77/77)	6
15	pyv-consensus-wo-decide	Node={n1,n2,n3}	✗ (35/40)	20
16	pyv-consensus-forall	Node={n1,n2,n3}	✗ (46/48)	25
17	pyv-learning-switch	TO		
18	i4-chord-ring-maintenance	n/a		
19	pyv-sharded-kv-no-lost-keys	Node={n1,n2} Key={k1,k2} Value={v1,v2}	✓ (6/6)	12
20	ex-naive-consensus	Node={n1,n2,n3} Value={v1,v2}	✗ (11/12)	6
21	pyv-client-server-ae	Node={n1,n2,n3} Request={r1,r2} Response={p1,p2}	✓ (6/6)	2
22	ex-simple-election	Acceptor={a1,a2,a3} Proposer={p1,p2}	✗ (11/12)	5
23	pyv-toy-consensus-epr	Node={n1,n2,n3} Value={v1,v2}	✗ (6/8)	9
24	ex-toy-consensus	Node={n1,n2,n3} Value={v1,v2}	✗ (1/4)	1
25	pyv-client-server-db-ae	Node={n1,n2,n3} Request = {r1,r2,r3} Response={p1,p2,p3} DbRequestId={i1,i2}	✓ (40/40)	20
26	pyv-hybrid-reliable-broadcast	n/a		
27	pyv-firewall	Node={n1,n2,n3}	✗ (4/10)	23
28	ex-majorityset-leader-election	Node={n1,n2,n3}	✗ (9/12)	9
29	pyv-consensus-epr	Node={n1,n2,n3} Value={v1,v2}	✗ (39/40)	21
30	mldr	MaxTerm=3 MaxConfigVersion=3 Server={n1,n2,n3,n4}	✗ (15/24)	226

Awaiting for Godot: Stateless Model Checking that Avoids Executions where Nothing Happens

Bengt Jonsson 
 Uppsala University, Sweden
 Email: bengt@it.uu.se

Magnus Lång 
 Uppsala University, Sweden
 Email: magnus.lang@it.uu.se

Konstantinos Sagonas 
 Uppsala University, Sweden and NTUA, Greece
 Email: kostis@it.uu.se

Abstract—Stateless Model Checking (SMC) is a verification technique for concurrent programs that checks for safety violations by exploring all possible thread schedulings. It is highly effective when coupled with Dynamic Partial Order Reduction (DPOR), which introduces an equivalence on schedulings and need explore only one in each equivalence class. Even with DPOR, SMC often spends unnecessary effort in exploring loop iterations that are *pure*, i.e., have no effect on the program state. We present techniques for making SMC with DPOR more effective on programs with pure loop iterations. The first is a static program analysis to detect loop purity and an associated program transformation, called *Partial Loop Purity Elimination*, that inserts assume statements to block pure loop iterations. Subsequently, some of these assumes are turned into *await* statements that completely remove many assume-blocked executions. Finally, we present an extension of the standard DPOR equivalence, obtained by weakening the conflict relation between events. All these techniques are incorporated into a new DPOR algorithm, *OPTIMAL-DPOR-AWAIT*, which can handle both *awaits* and the weaker conflict relation, is *optimal* in the sense that it explores exactly one execution in each equivalence class, and can also diagnose livelocks. Our implementation in *NIDHUGG* shows that these techniques can significantly speed up the analysis of concurrent programs that are currently challenging for SMC tools, both for exploring their complete set of interleavings, but even for detecting concurrency errors in them.

I. INTRODUCTION

Ensuring correctness of concurrent programs is difficult, since one must consider all the different ways in which actions of different threads can be interleaved. Stateless model checking (SMC) [9] is a fully automatic technique for finding concurrency bugs (i.e., defects that arise only under some thread schedulings) and for verifying their absence. Given a terminating program and fixed input data, SMC systematically explores the set of all thread schedulings that are possible during program runs. A special runtime scheduler drives the SMC exploration by making decisions on scheduling whenever such choices may affect the interaction between threads. SMC has been implemented in many tools (e.g., VeriSoft [10], CHES [20], Concuerror [6], NIDHUGG [2], rInspect [24], CDSHECKER [21], RCMC [14], and GENMC [18]), and successfully applied to realistic programs (e.g., [11] and [17]).

SMC tools typically employ *dynamic partial order reduction* (DPOR) [8, 1] to reduce the number of explored schedulings. DPOR defines an equivalence relation on executions, which preserves relevant correctness properties, such as reachability of local states and assertion violations. For correctness, DPOR needs to explore at least one execution in each equivalence

p	q
<pre> if(x[0] > x[1]) swap(x[0], x[1]); y := 1; do b := y while(b ≠ 2); if(x[0] > x[1]) swap(x[0], x[1]) </pre>	<pre> do a := y while(a ≠ 1); if(x[1] > x[2]) swap(x[1], x[2]); y := 2 </pre>

Figure 1: A concurrent program implementing a sorting network. *p* sorts $x[0]$ and $x[1]$, and then uses y to signal that $x[1]$ is ready. *q* waits for y to be 1 and then sorts $x[1]$ and $x[2]$, completing one round of bubble sort. In the second round, shown in blue, *q* signals that the next “generation” of $x[1]$ is ready by setting y to 2, upon which *p* finishes the sort by sorting $x[0]$ and $x[1]$ again. Initially $y = 0$.

class. We call a DPOR algorithm *optimal* if it guarantees the exploration of exactly one execution per equivalence class.

In SMC, loops have to be bounded if they do not already terminate in a bounded number of iterations. Loop bounding may in general not preserve assertion failures. Hence a fairly large loop bound should be used, but this is often practically infeasible, and thus loop bounding must strike a balance between these two concerns. However, for loops whose execution has no global effects, the number of equivalence classes that need be explored by SMC can be significantly reduced while still preserving correctness properties, using techniques that we will present in this paper.

Consider the first round of the program snippet in Fig. 1 (shown in black), where thread *q* executes a loop that waits for thread *p* to set the shared variable y to 1. A naïve application of SMC with DPOR will explore an unbounded number of executions, since (in the absence of loop bounding) there is an infinite number of equivalence classes, one for each number of performed loop iterations. All iterations of this loop, however, are *pure*, i.e., they have no effect on the program state. For such loops, a bound of one will preserve correctness properties. In our example, the **do-while** loop of thread *q* can be rewritten into the sequence of statements $a := y$; **assume**($a = 1$), which will cause the SMC exploration to permanently block thread *q* whenever the condition of the **assume** is violated.

Using **assume** statements to bound loops causes executions where the condition of the **assume** is violated and its corresponding thread is blocked to be explored. This happens even if the condition will eventually be satisfied, and the original loop will exit, under any fair thread scheduling. *Assume-blocking* of a thread can occur in many contexts, each generating an execution that need not be explored. (We will shortly see this for the example in Fig. 1.) Furthermore, and perhaps more

seriously, this use of **assumes** prevents SMC from diagnosing livelocks in which the loop never exits even under fair thread scheduling. This is because a blocked execution corresponding to a livelock can also result from a spurious execution in which the **assume** reads a shared variable before it has been written to by another thread.

Here is where **await** statements can lead to further reductions. An **await** loads from a shared variable, but only if the loaded value satisfies some condition, otherwise it blocks. In contrast to assume-blocking, *await-blocking* is not permanent but can be repealed if the condition is later satisfied. Thereby, executions where blocking occurs by reading “too early” are avoided. Moreover, such executions can be distinguished from livelocks, in which the condition is not satisfied after some bounded time. For our example, the rewrite of the **do-while** loop into an **await**($y = 1$) statement results in a program for which SMC would explore only a single execution in which the **await** reads the value written by thread p .

Consider now the full program in Fig. 1, which performs a concurrent sort of a three-element array using a sorting network. This program can be scaled to larger arrays for increased available parallelism. Since any network sorting an array of size n will have at least $\Omega(n \log n)$ occurrences of a code snippet which exchanges two values after exiting a spinloop, exploring such a program with SMC will explore $\Omega(2^{n \log n})$ executions, even after rewriting the spinloops using **assume** statements. On the other hand, when using **await** statements, all executions fall into the same equivalence class. Thus, an optimal SMC algorithm that can properly handle **awaits** will explore only one execution, thereby achieving exponential reduction.

In this paper, we present techniques to (i) automatically transform a program to an intermediate representation that uses **await** as a primitive, and (ii) explore its executions using a provably optimal DPOR algorithm that is **await** aware and also uses a conflict relation between statements which is weaker than the standard one. We first present a static program analysis technique to detect pure loop executions and an associated program transformation, called *Partial Loop Purity (PLP) Elimination*, that inserts **assume** statements which are then turned into **awaits** if preceded by the appropriate load. We prove that PLP is sound in the sense that it preserves relevant correctness properties, including local state reachability and assertion failures. We also present and prove conditions under which PLP is guaranteed to remove all pure executions of a loop. Finally, we prove that our new DPOR algorithm OPTIMAL-DPOR-AWAIT, which is an extension of the Optimal-DPOR algorithm of Abdulla et al. [1, 3], is correct and optimal, also with respect to our weaker conflict relation.

All these techniques are available in NIDHUGG, a state-of-the-art SMC tool, and in the paper’s replication package [13]. Our evaluation, using multi-threaded programs which are currently challenging for most tools, shows that our techniques can achieve significant (and sometimes exponential) reduction in the total number of executions that need to be explored. Moreover, they enable detection of concurrency bugs which were previously out-of-reach for most concurrency testing tools.

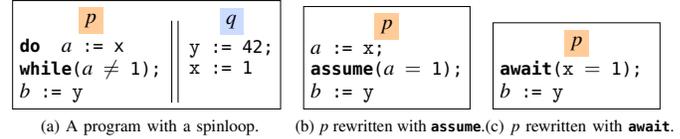


Figure 2: Multi-threaded program illustrating the rewrites; initially, $x = y = 0$. For (b) and (c), q is the same as in (a).

II. ILLUSTRATION THROUGH EXAMPLES

In this section, we illustrate our contributions through examples. First, in §II-A we show how **assume** and **await** statements are inserted. In §II-B we illustrate how our optimal DPOR algorithm handles **await** statements, and in §II-C how it handles the weaker conflict relation in which atomic fetch-and-adds on the same variable are not conflicting.

We consider programs consisting of a finite set of threads that share a finite set of shared variables (x, y, z). A thread has a finite set of local registers (a, b, c), and runs a deterministic code, built from expressions, atomic statements, and synchronisation operations, using standard control flow constructs. Atomic statements read or write to shared variables and local registers, including atomic read-modify-write operations, such as compare-and-swap and fetch-and-add. Synchronisation operations include locking a mutex and joining another thread. Executions of a program are defined by an interleaving of statements. We use sequential consistency in this paper, but we note that some weak memory models (e.g., TSO and PSO) can be modelled by an interleaving-based semantics, so our work can be extended to DPOR algorithms [2] that handle such memory models. Our loop transformations introduce **await** statements, that take a conditional expression over a global variable as a parameter and come in several forms: simple awaits (**await**($x = 0$)), load-await ($a := \mathbf{await}(x = 0)$), and exchange-await ($a := \mathbf{xchgawait}(x = 0, := 1)$). These operations block until their condition is satisfied.

A. Introducing Await Statements

Let us show an example of how loops are transformed by introducing **assume** and **await** statements. Consider the loop in Fig. 2a. There, thread p executes a spinloop, waiting for thread q to set the shared variable x . Each iteration of this loop, in which the value loaded into a is different from 1, is pure, i.e., it does not modify shared variables, nor any local register that may be used after the end of the loop. Therefore an **assume** statement is introduced at the point where the thread can distinguish pure executions from impure ones, i.e., after a has been loaded. The result of such a rewrite is shown in Fig. 2b. This program has two traces, one in which the **assume** succeeds, representing the executions in which the original loop terminates, and one where thread p gets *assume-blocked*. The latter trace will exist even in the case where the original loop is guaranteed to terminate under a fair scheduler. This problem is remedied by replacing the load into a and the following **assume** statement by an **await** with a test on the shared variable from which a reads. Such a rewrite results in the program in Fig. 2c. In this case, the **await** statement may permanently block only

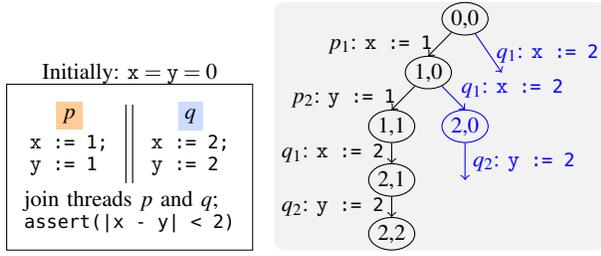


Figure 3: Program with a correctness assertion, and execution trees with the first scheduling of the program; nodes show the values of variables x and y .

if the original loop can livelock under fair scheduling. In our simple example, the rewritten program has only a single trace, since the original loop is guaranteed to terminate and can be replaced by the **await**. Programs with more complex loops (e.g., loops that are pure only along a subset of their paths) are also handled by our program transformation (§III), but the loop is not eliminated when **assumes** or **awaits** are introduced.

B. OPTIMAL-DPOR-AWAIT by Example

DPOR algorithms are based on regarding executions as equivalent if they induce the same ordering between executions of conflicting statements. The standard conflict relation regards two accesses to the same variable as conflicting if at least one is a write. We begin by illustrating the Optimal-DPOR algorithm [3] on the simple program in Fig. 3. There two threads, p and q , write to two shared variables x and y in sequence. Optimal-DPOR starts by exploring an arbitrary interleaved execution of the program. Assume it is $p_1.p_2.q_1.q_2$ as shown in Fig. 3 (we will denote executions by sequences of thread identifiers, possibly subscripted by sequence numbers). Each explored execution is then analysed to find *racess*, i.e., pairs of conflicting events that are adjacent in the happens-before order induced by the conflict relation. (An *event* is a particular execution step of a thread in an execution.) Our first execution contains two races, (p_1, q_1) and (p_2, q_2) . For each race, Optimal-DPOR creates a so-called *wakeup sequence*, i.e., a sequence which continues the analysed execution up to the first event in a way which reaches the second event instead of the first event. For the first race, the wakeup sequence is q_1 , and for the second race, it is $p_1.q_1.q_2$. The wakeup sequences are inserted as new branches just before the first event of the corresponding race, thereby gradually building a tree consisting of the explored executions and added wakeup sequences. The execution tree after the first execution is shown in Fig. 3.

After processing the first execution, Optimal-DPOR then picks the leftmost unexplored leaf in the tree, and extends it arbitrarily to a full execution, in which races are analysed, etc. As the algorithm backtracks, it deletes the nodes it backtracks from in the execution tree. The second execution has two races, (p_1, q_1) as well as (p_2, q_2) . However, the corresponding wakeup sequences will result in executions that are redundant, i.e., equivalent to already inserted ones, so no further insertion takes place. The algorithm proceeds in this way until there are no more unexplored leaves corresponding to wakeup sequences. In total, there are four executions explored by Optimal-DPOR, corresponding to the four possible final valuations of x and y .

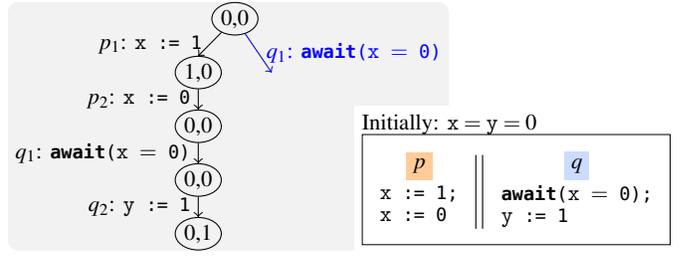


Figure 4: Exploration of a program with an **await** with two satisfying writes.

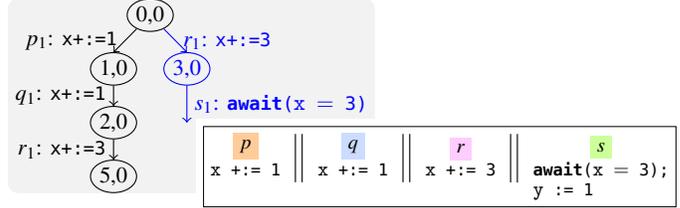


Figure 5: Exploration of a program with fetch-and-adds. Initially, $x = y = 0$.

Let us now look at how OPTIMAL-DPOR-AWAIT extends Optimal-DPOR to work for programs with **awaits**. Consider the program in Fig. 4. There, p writes to the global variable x , first updating it to 1, and then back to 0. Assume that the first execution is $p_1.p_2.q_1.q_2$. The analysis of races performed by Optimal-DPOR must now be extended to consider that **await** statements are sometimes blocked. First, the conflict between p_2 with q_1 will not be handled like a race, since q_1 is blocked just before p_2 . Therefore, we find the closest preceding point in the execution at which q_1 is not blocked, which in this case is at the beginning. We then construct the wakeup sequence q_1 and insert it at the beginning; cf. Fig. 4. Since this program only has two traces, OPTIMAL-DPOR-AWAIT will terminate after exploring the second execution.

C. Handling Atomic Fetch-and-Add Instructions in DPOR

To reduce the number of equivalence classes that need be explored by a DPOR algorithm, one can weaken the standard conflict relation between statements by considering two atomic fetch-and-add (FAA) statements on the same variable as non-conflicting if the loaded values are afterwards unused. In the absence of **await** statements, many existing DPOR algorithms like Optimal-DPOR handle this definition without modification. However, this weakening has a subtle interaction with **await** statements that must be handled by OPTIMAL-DPOR-AWAIT.

Consider the program in Fig. 5. In this program, three threads, p , q , and r , add atomically to the shared variable x , and a thread s awaits x having the value 3. We assume that DPOR considers the FAA statements p_1 , q_1 , and r_1 to be non-conflicting, but conflicting with the statement s_1 , should it execute.

Assume that the first explored execution is $p_1.q_1.r_1$. From this point, we cannot substitute s_1 for either of p_1 , q_1 , or r_1 , as s_1 is not enabled after any of $q_1.r_1$, $p_1.r_1$ or $p_1.q_1$, respectively. Yet, there is another execution in which s_1 is enabled. In order to construct this execution, we must not only schedule s_1 before one of the other events, but before two, both of p_1 and q_1 , so that only r_1 remains. Then, we could construct the wakeup sequence $r_1.s_1$. In general, OPTIMAL-DPOR-AWAIT

may need to reorder the sequence of independent FAAs that precede an `await` statement and select a subsequence of them, in order to unblock the `await` statement. This can be done in several ways, and OPTIMAL-DPOR-AWAIT is optimised to avoid enumerating all of them. In §IV-B, we will see how.

III. PARTIAL LOOP PURITY ELIMINATION

In this section, we describe *Partial Loop Purity Elimination*, a technique that prevents SMC from exploring executions with pure loop iterations. It consists of (1) a static analysis technique which annotates programs with conditions under which a loop will execute a pure iteration, and (2) a program transformation which inserts `assume` statements based on the analysis.

We consider loops consisting of a set of basic blocks, with a single header block. Each basic block contains a sequence of program statements. Blocks are connected via edges, labelled by conditions. We also consider program representations on Static Single Assignment (SSA) form, which means that each register is assigned by exactly one statement. Thus, a register uniquely identifies the statement that assigns to it. When the value of a register in one block depends on which predecessor block was executed, this is expressed using a *phi node*. For example, in a block C with predecessors A and B containing registers a and b , respectively, the statement $c := \phi(A : a, B : b)$ defines the register c to get the value of a when the previous basic block was A and of b when the previous block was B .

An execution of a loop iteration is *pure* if the execution starts and ends at the header of the loop, and during the iteration (i) no modification of a global variable is performed, (ii) nor of any local variable that may be used after the end of the iteration, and (iii) no *internal* (not to the header) backedge is taken. In SSA form, modification of local variables can be inferred from the phi nodes in the header. If such a phi node uses a different value on the backedge to the header than when first entering, then the loop iteration modified a local variable that is used on some path after the iteration, and we call the header *impure* along the backedge. Our definition considers executions that complete inner loop iterations to be non-pure. However, our PLP transformation will block inner loops from completing pure iterations.

A register a *reaches* a program point l if all paths to l pass a 's definition. During a loop execution, we say that an expression over registers is *defined-true* at some program point l in the loop, if the expression evaluates to true under (i) the current valuation of registers that were assigned either outside the loop or during the current loop iteration, and (ii) any valuation of all other registers. We now define a central concept; that of the Forward Purity Condition.

Definition 1 (Forward Purity Condition). *Let l be a program point in a loop. Then, a Forward Purity Condition (FPC) at l is an expression in Disjunctive Normal Form over the registers such that if an execution, without leaving the loop or taking an internal backedge, proceeds to a program point l' , at which the expression is defined-true, then*

- (i) *the execution from l' will reach the loop header without taking an internal backedge, and*

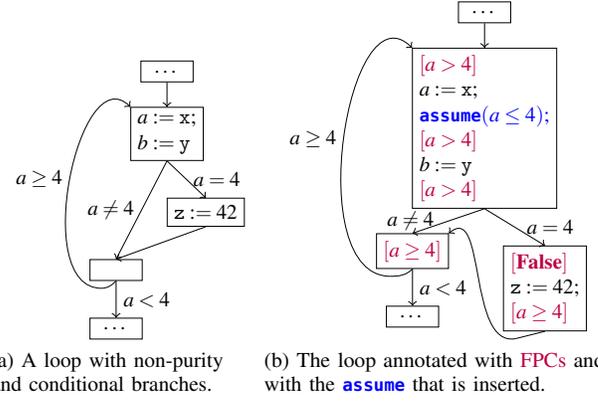


Figure 6: Program snippet illustrating the concepts of the PLP transformation.

- (ii) *the execution from l to the loop header will not modify any global variables nor any local variable that may be used after execution has reached the loop header.* \square

We will denote a FPC with brackets, for example $[c > 42]$ or `[False]`. A *purity condition* (PC) of a loop is a FPC of the loop at the beginning of its header. Thus, whenever a loop iteration passes a program point where the PC is defined-true, and has not taken an internal backedge, then that iteration is pure.

We illustrate these concepts for the program snippet in Fig. 6a. In it, the loop loads x and y into registers a and b , then branches on the value of a , and along the path where $a = 4$, there is a write to z . Since a write to a global variable is non-pure, the loop is not pure whenever $a = 4$. The two paths converge in a common block where a loop condition ($a \geq 4$) is checked. This loop is pure if (i) it takes the backedge, i.e., $a \geq 4$ holds, and (ii) the write to z is not performed, i.e., $a \neq 4$ also holds. The conjunction of these conditions, $a > 4$, becomes a purity condition for the entire loop. We thereafter insert an `assume` with the negation of a disjunct of the PC at the earliest point that it is defined-true, i.e., after the load of x , shown in blue in Fig. 6b.

Let us now describe the analysis stage for computing purity conditions. Its first step is to compute FPCs at all points in the loop. Intuitively, the FPC at a point l is a disjunction $c_1 \vee \dots \vee c_n$, where each c_i is a (forward) path condition for reaching the header via a pure execution from l . We compute FPCs by backwards propagation through statements and basic blocks. Let $\text{FPC}(s\bullet)$ be the FPC immediately after statement s , let $\text{FPC}(\bullet s)$ be the FPC immediately before statement s , let $\text{FPC}(\bullet B)$ be the FPC at the beginning of block B , and let $\text{FPC}(B\bullet)$ be the FPC at the end of block B .

For each statement s , we compute $\text{FPC}(\bullet s)$ as $\text{FPC}(s\bullet) \wedge g$, where g is the condition under which s does not update a global variable. For instance, g is `False` for stores, `True` for loads, $a = 0$ for an atomic add of form $x += a$, $a = b$ for an atomic exchange of form $b := \text{xchg}(x, a)$, and $c = 1$ for an atomic compare-exchange of form $c := \text{cmpxchg}(x, a, b)$.

FPCs for basic blocks are computed as follows. First, for an edge with condition g from a block A in the loop to a block B , let $\text{FPC}(A, B)$ be the FPC along that edge, defined as follows;

- if B is outside the loop, then $\text{FPC}(A, B) = [\mathbf{False}]$,
- if B is the header block, then if B is impure along (A, B) , then $\text{FPC}(A, B) = [\mathbf{False}]$, otherwise $\text{FPC}(A, B) = [g]$.
- if B is inside the loop, then $\text{FPC}(A, B) = [\mathbf{False}]$ if the edge from A to B is an internal backedge (A, B) , otherwise $\text{FPC}(A, B) = [\text{FPC}(\bullet B) \wedge g]$,

We propagate FPCs backwards through basic blocks by the above rules for statements. We then compute the FPC at the end of a block A with outgoing arcs to B_1, \dots, B_k as $\text{FPC}(A \bullet) = \bigvee_{i=1}^k \text{FPC}(A, B_i)$. We can thereafter calculate FPCs for basic blocks by starting from the edges that leave the loop or go back to its header. Cycles in the control flow graph are no issue, since the FPC of a backedge (A, B) does not depend on B . In Fig. 6b, we can see the FPCs computed by the analysis on the example.

After the analysis, we insert **assume** statements. Given a purity condition of form $c_1 \vee c_2 \vee \dots \vee c_n$, for each c_i we insert an **assume**($\neg c_i$) at the earliest point that is textually after the definitions of all registers in c_i . For registers that do not reach the insertion location, arbitrary values can be used when execution does not pass their definitions. Moreover, if any memory access along the path corresponding to c_i cannot be statically determined not to segfault, we must not insert c_i before that memory access. For this purpose, we associate an optional “earliest insertion point” with every c_i in each FPC computed by the analysis. Finally, to exclude paths that took some internal backedge, a “took internal backedge” boolean register is introduced, computed by phi-nodes, and included in the conjunction c_i .

Theorem 1, whose proof appears in the extended version [12] of this paper, states two essential properties of PLP. These properties intuitively say that PLP removes pure executions while preserving relevant correctness properties. If σ is a local state occurring in a loop \mathbb{L} of a thread p , we say that \mathbb{L} is *unavoidably pure* from σ to denote that whenever thread p is in local state σ during an execution, then p is in the process of completing a pure iteration of \mathbb{L} .

Theorem 1. *Let \mathbb{P}' be the program resulting from applying PLP to \mathbb{P} . Then \mathbb{P}' satisfies the following properties.*

- 1) Local State Preservation: *each local state σ of a thread p which is reachable in \mathbb{P} is also reachable in \mathbb{P}' , provided no loop of p is unavoidably pure from σ .*
- 2) Pure Loop Elimination: *no execution of \mathbb{P}' exhibits a completed pure loop iteration of some thread.*

We remark that in the definition of pure loop iterations, we assume possibly conservative characterisations of “global variable” and “local variable that may be used after the end of the iteration” that can be determined by a standard syntactical analysis of the program, and hence used in the PLP analysis.

IV. THE OPTIMAL-DPOR-AWAIT ALGORITHM

In this section, we present OPTIMAL-DPOR-AWAIT, a DPOR algorithm for programs with **await** statements, which is both correct and optimal. Given a terminating program on

given input, it explores exactly one maximal execution in each equivalence class induced by the equivalence relation \simeq .

A. Happens-Before Ordering and Equivalence

DPOR algorithms are based on a partial order on the events in each execution. Given an execution E of a program \mathbb{P} , an *event* of E is a particular execution step by a single thread; the i 'th event by thread p is identified by the tuple $\langle p, i \rangle$, and \hat{e} denotes the thread p of an event $e = \langle p, i \rangle$. Let $\text{dom}(E)$ denote the set of events in E . We define a *happens-before relation* on $\text{dom}(E)$, denoted $\xrightarrow{\text{hb}}_E$, as the smallest transitive relation such that $e \xrightarrow{\text{hb}}_E e'$ if e occurs before e' in E , and either

- (i) e and e' are performed by the same thread, e spawns the thread which performs e' , or e' joins the thread which performs e , or
- (ii) e and e' access a common shared variable x , at least one of them writes to x , and they are not both atomic fetch-and-add operations.

Note that the last condition makes atomic fetch-and-add operations on the same shared variable independent. It follows that $\xrightarrow{\text{hb}}_E$ is a partial order on $\text{dom}(E)$. We define two executions, E and E' , as equivalent, denoted $E \simeq E'$, if they induce the same happens-before relation on the same set of events, (i.e., $\text{dom}(E) = \text{dom}(E')$ and $\xrightarrow{\text{hb}}_E = \xrightarrow{\text{hb}}_{E'}$). If $E \simeq E'$, then all variables are modified by the same sequence of statements, implying that each thread runs through the same sequence of local states in E and E' .

B. The Working of the OPTIMAL-DPOR-AWAIT Algorithm

OPTIMAL-DPOR-AWAIT is shown in Algorithm 1. It performs a depth-first exploration of executions using the recursive procedure $\text{Explore}(E)$, where E is the currently explored execution, which can also be interpreted as the stack of the depth-first exploration. In addition, for each prefix E' of E , the algorithm maintains

- a *sleep set* $\text{sleep}(E')$, i.e., a set of threads that should not be explored from E' , for the reason that each extension of form $E'.p$ for $p \in \text{sleep}(E')$ is equivalent to a previously explored sequence,
- a *wakeup tree* $\text{wut}(E')$, i.e., an ordered tree $\langle B, \prec \rangle$, where B is a prefix-closed set of sequences, whose leaves are called *wakeup sequences*, and \prec is the order in which sequences were added to $\text{wut}(E')$. For each $w \in B$ the sequence $E'.w$ will be explored during the call $\text{Explore}(E')$ in the order given by \prec .

All previously explored sequences together with the current wakeup tree (i.e., all sequences of form $E'.w$ for $w \in \text{wut}(E')$) and a prefix E' of E form the current execution tree, denoted \mathcal{E} . The branches of \mathcal{E} are ordered by the order in which they were added to the tree. Note that the recursive call to $\text{Explore}(E)$ may insert into $\text{wut}(E')$ for prefixes E' of E .

Let $v \setminus p$ denote the sequence v with the first occurrence of an event by thread p (if any) removed. Let $\text{next}_{[E]}(p)$ denote the next event performed by thread p after E . Two important concepts are *races* and *weak initials*.

Definition 2 (Non-Blocking Races). Let e, e' be two events in different threads in an execution E , where e occurs before e' . Then e and e' are in a non-blocking race, denoted $e \lesssim_E e'$, if (i) e and e' are adjacent in \xrightarrow{hb}_E (i.e., $e \xrightarrow{hb}_E e'$, and for no other event e'' we have $e \xrightarrow{hb}_E e'' \xrightarrow{hb}_E e'$), and (ii) e' cannot be enabled or disabled by an event in another thread. \square

Definition 3 (Weak Initials). For an execution $E.w$, the set of weak initials of w (after E), denoted $WI_{[E]}(w)$, is the set of threads p such that $E.w \simeq E.p.(w \setminus p)$ if p is in w , and $E.w.p \simeq E.p.w$ if p is not in w . \square

Intuitively, $p \in WI_{[E]}(w)$ if $next_{[E]}(p)$ is independent with all events that precede it in w in the case that p is in w , otherwise with all events in w . If $p \in WI_{[E]}(w)$ we say that w is *redundant* wrt. $E.p$, since some extension of $E.w$ is equivalent to some extension of $E.p$. An important property of the execution tree \mathcal{E} that is maintained by the algorithm is that an extension w of an existing sequence E is added only if \mathcal{E} does not contain an execution of form $E'.p$ such that E' but not $E'.p$ is a prefix of E , and $w'.w$ is redundant wrt. $E'.p$, where E' is defined by $E = E'.w'$.

For the OPTIMAL-DPOR-AWAIT algorithm, we define

- $pre(E, e)$ as the prefix of E up to but not including e ,
- $notdep(e, E)$ as the subsequence of E of events that occur after e but do not happen-after e .
- $u \lesssim_{[E]} w$ to denote that $E.u.v \simeq E.w$ for some v ; intuitively u is a “happens-before prefix” of w .

The algorithm runs in two phases: race detection (lines 3–22) and exploration (lines 24–33). Exploration picks the next unexplored leaf of the exploration tree and extends it with arbitrary scheduling to a maximal execution. This leaf is reached step-by-step: at each step, the current execution E is extended by the leftmost child of the root of $wut(E)$ and used in a recursive call to *Explore* (lines 28–31) in order to perform the next step. If $wut(E)$ only contains the empty sequence, an arbitrary thread is chosen for the next step and added to $wut(E)$ (line 26). This step-by-step extension of the current execution is continued until a maximal execution is reached. At each step, the new sleep set after $E.p$ is constructed by taking the elements of $sleep(E)$ that are independent with p . After a recursive call to $E.p$, the subtree rooted at $E.p$ can be removed from the wakeup tree. To remember that we should not attempt to explore any sequences that are redundant wrt. $E.p$, we add p to $sleep(E)$.

The race detection phase is entered when the explored sequence E is maximal. There we examine E for races and construct new non-redundant executions. We distinguish between two types of races: non-blocking races, such as between a write and a read, handled on lines 3–6, and blocking races, such as involving an **await** event, handled on lines 7–22.

For each non-blocking race $e \lesssim_E e'$, we let E' be the prefix of E that precedes e , and construct a wakeup sequence v by appending \hat{e}' to the subsequence of events that occur after e in E but do not happen-after e (line 5). By construction, the sequence $E'.v$ is an execution. Moreover $\hat{e}' \notin WI_{[E']}(v)$ since

Algorithm 1: OPTIMAL-DPOR-AWAIT

Initial call: *Explore*($\langle \rangle$) with $wut(\langle \rangle) = \{\langle \rangle\}, \emptyset$, $sleep(\langle \rangle) = \emptyset$

```

1 Explore( $E$ )
2 if  $enabled(E) = \emptyset$  then
3   foreach  $e, e' \in dom(E)$  such that  $(e \lesssim_E e')$  do
4     let  $E' = pre(E, e)$ 
5     let  $v = (notdep(e, E), \hat{e}')$ 
6     if  $sleep(E') \cap WI_{[E']}(v) = \emptyset$  then  $insert(v, E')$ 
7     foreach  $\langle e', E' \rangle \in (\{\langle next_{[E]}(p), E \rangle \mid p \text{ is blocked after } E\}$ 
8        $\cup \{\langle e', pre(E, e') \rangle \mid e' \text{ is in } E \text{ and may block}\})$  do
9        $can-stop := \mathbf{False}$ 
10      foreach  $e$  in  $E'$  (starting from the end)
11        that may enable or disable  $e'$  do
12        let  $E'' = pre(E, e)$ 
13        let  $w = notdep(e, E)$ 
14        if  $e$  conflicts with all events that may
15          enable or disable  $e'$  then  $can-stop := \mathbf{True}$ 
16         $did-insert := \mathbf{False}$ 
17        foreach maximal subsequence  $u$  of  $w$  such that
18           $u \lesssim_{[E'']} w$  and  $e'$  is enabled after  $E''.u$  do
19           $did-insert := \mathbf{True}$ 
20          let  $v = u.\hat{e}'$ 
21          if  $sleep(E'') \cap WI_{[E'']}(v) = \emptyset$  then  $insert(v, E'')$ 
22          if  $can-stop$  and  $did-insert$  then break
23 else
24   if  $wut(E) = \{\langle \rangle\}, \emptyset$  then
25     choose  $p \in enabled(E)$ 
26      $wut(E) := \{\langle p \rangle\}, \emptyset$ 
27     while  $\exists p \in wut(E)$  do
28       let  $p = \min_{\prec} \{p \in wut(E)\}$ 
29        $sleep(E.p) := \{q \in sleep(E) \mid p, q \text{ independent after } E\}$ 
30        $wut(E.p) := subtree(wut(E), p)$ 
31       Explore( $E.p$ )
32       add  $p$  to  $sleep(E)$ 
33       remove all sequences of form  $p.w$  from  $wut(E)$ 
34  $insert(v, E')$ 
35  $u := \langle \rangle$ 
36 let  $c$  be the list of children of  $u$  in  $wut(E')$  from left to right
37 foreach sequence  $u.p$  in  $c$  do
38   if  $p \in WI_{[E'.u]}(v)$  then
39     if  $p \notin v$  or  $(v := v \setminus p) = \langle \rangle$  then return
40      $u := u.p$ 
41     if  $u$  is a leaf of  $wut(E')$  then return
42     goto line 36
43 add  $v$  as a new rightmost descendant of  $u$  in  $wut(E')$ 
44 return

```

the occurrence of e' in v does not happen-after e . Thus, v is non-redundant wrt. $E'.\hat{e}'$. If v is also non-redundant wrt. $E'.p$ for each $p \in sleep(E')$, then v is inserted into the wakeup tree at E' , extending $wut(E')$ with a new leaf if necessary.

Races involving events that can be blocked are handled at lines 7–22. For each such event e' , we extract the prefix E' that precedes e' . Then, for each e in E' that potentially conflicts with e' , we extract the prefix E'' preceding e and the sequence w of events that does not happen-after e . For each maximal happens-before prefix u of w after which e' is enabled, we construct a wakeup sequence v as $u.\hat{e}'$ (line 20), which is checked for redundancy and possibly inserted into the wakeup tree in the same way as for a nonblocking race. Such prefixes can be enumerated by recursively removing the

suffix of one event that may enable or disable e' at a time, stopping whenever e' is enabled by the current prefix. As an optimisation, implemented by the flags *can-stop* and *did-insert*, once the algorithm has found a wakeup sequence that enables e' before some event that conflicts with every event that may enable or disable e' , it needs not consider reversing e' with even earlier events e , as those reversals will be considered in a later recursive call.

The function $insert(v, E)$ for inserting a sequence v into a wakeup tree $wut(E')$ is shown in lines 34–44. Starting from the root, represented by the empty sequence, it traverses $wut(E')$ downwards (the current point being u), always descending (line 40) to the leftmost child $u.p$ such that p is a weak initial of the remainder of v until either (i) arriving at a leaf indicating that v was redundant to begin with and $wut(E')$ can be left unchanged (line 41), (ii) encountering a p which is not in v , or exhausting v (line 39), or (iii) arriving at a node with no child passing the test at line 38, and then adding the remainder of v as a new leaf (line 43), since it was shown to be non-redundant.

Algorithm OPTIMAL-DPOR-AWAIT is correct and optimal in the sense that it explores exactly one maximal execution in each equivalence class, as stated in the following theorem whose proof is in the extended version of this paper [12].

Theorem 2. *For a terminating program \mathbb{P} , OPTIMAL-DPOR-AWAIT has the properties that (i) for each maximal execution E of \mathbb{P} , it explores some execution E' with $E' \simeq E$, and (ii) it never explores two different but equivalent maximal executions.*

V. IMPLEMENTATION AND EVALUATION

We have implemented our techniques on top of the NIDHUGG tool. NIDHUGG is a state-of-the-art stateless model checker for C/C++ programs with Pthreads, which works at the level of LLVM Intermediate Representation (IR), typically produced by the Clang compiler. We have added our PLP analysis and transformations, as well as the rewrite from load-assume, exchange-assume, and compare-exchange-assume pairs into load-await and exchange-await, as passes over LLVM IR. NIDHUGG comes with a selection of SMC algorithms. One of them is Optimal-DPOR, which we have used as a basis for our implementation of OPTIMAL-DPOR-AWAIT including IFAA, the optimisation of treating fetch-and-add instructions to the same memory location as independent. All the techniques in this paper are now included in upstream NIDHUGG and are enabled when giving the `-optimal` flag.

A. Overall Performance

First, we evaluate our technique and compare its performance against baseline NIDHUGG and the SAVER [16] technique, implemented in a recent version of GENMC [18]. SAVER has a similar goal to our PLP transformation, but tries to identify pure loop iterations dynamically, aborting threads if they perform a pure loop iteration. SAVER’s approach does not allow further rewrite with **awaits**.

For our evaluation, we used a set of real-world benchmarks similar to those used by the SAVER [16] paper. We note that

all atomic memory accesses in these benchmarks have been converted to SC, as this is the only common memory model that both tools support. Where relevant, benchmarks are ran with the same loop bound as in the SAVER paper. For most benchmarks, this is one greater than the number of threads. After the benchmark name, the number of threads are shown in parentheses. Benchmarks `mcslock`, `qspinlock` and `seqlock` are tests of data structures from the Linux kernel. Benchmarks `ttaslock` and `twalock` are mockups based on, but not the same as, the benchmarks in the SAVER paper, because its authors were not at liberty to share the original benchmark sources. Both are tests of locking algorithms. Benchmark `mpmc-queue` tests a multiproducer-multiconsumer queue algorithm, `linuxrwlocks` tests a readers-writers lock algorithm, `treiber-stack` tests a lock-free stack algorithm, and `ms-queue` tests a lock-free queue. Benchmarks `mutex` and `mutex-musl` test two mutex algorithms, the second one used in the musl C standard library implementation. Benchmark `sortnet` is an extended version of the concurrent sort program from Fig. 1. In this version, the sorting networks are generated using Batcher’s odd-even mergesort. The number of elements sorted is twice the number of threads, so `sortnet(6)` sorts 12 elements. In our replication package [13], all the tools and benchmarks are provided, as well as scripts that can replicate the tables in this section.

We evaluate all techniques based on the number of executions they explore. In fact, we show this number using an addition of form $T + B$, where T is the number of explored completed executions and B is the number of executions that are blocked in the sense that either an **await** is deadlocked or some thread is blocked for executing **assume(false)** (in NIDHUGG) or a pure loop iteration (in SAVER). We remark that the SAVER paper reports only the T part, but, as we will see, often the number of blocked executions is significant and outnumbers the number of explored completed executions. Obviously, both numbers contribute to the time an SMC tool takes to explore these programs. The evaluation was performed on a Ryzen 5950X running a July 2022 Arch Linux system.

In Table I, there are four sets of NIDHUGG columns. Baseline shows the performance of unmodified NIDHUGG/Optimal. The PLP columns shows the performance of using unmodified NIDHUGG/Optimal together with Partial Loop Purity Elimination. Pure loops are bounded with **assumes**. The PLP+Await columns shows the result of PLP and transforming **assumes** into **awaits**, where possible. Finally, the `...+IFAA` columns report results from when OPTIMAL-DPOR-AWAIT treats atomic fetch-and-add operations as independent. For the two sets of GENMC columns, the SAVER columns show the performance of GENMC v0.6, which implements the SAVER technique, and the Baseline columns show the performance of GENMC v0.5.3, which does not. The timeout we have used for these benchmarks is 1 hour.

Starting at the top of Table I, `qspinlock` is a benchmark that does not benefit from SAVER nor PLP, but establishes that the baseline algorithms of both tools are very similar but GENMC is faster. In the next four benchmarks (`mcslock`, `twalock`, `mutex`, and `mutex-musl`), both PLP and SAVER are ineffective, but **awaits** eliminate most of the blocked traces (in `mcslock`) or

Table I: Number of (complete-blocked) executions explored by algorithms implemented in GENMC and NIDHUGG on a set of challenging benchmarks, as well as the execution time (in seconds) taken. The \odot symbol means that the exploration did not finish in 1h, and \dagger means that the tool crashed.

Benchmark	GENMC				NIDHUGG							
	Baseline		SAVER		Baseline		PLP		PLP+Await		...+IFAA	
	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time	Execs	Time
qspinlock(2)	6+2	0.02	6+2	0.02	6+2	0.06	6+2	0.08	6+2	0.08	6+2	0.09
qspinlock(3)	564+462	0.06	564+462	0.06	564+462	0.20	564+462	0.20	564+456	0.21	564+456	0.20
mcslock(3)	336+426	0.09	336+426	0.09	336+426	0.20	336+426	0.23	336+72	0.18	336+72	0.18
mcslock(4)	26232+33432	42.06	26232+33432	3.95	26232+33432	16.59	26232+33432	16.95	26232+4824	9.53	26232+4824	9.43
twalock(3)	96+90	0.02	96+90	0.02	96+90	0.09	96+90	0.09	96	0.08	96	0.08
twalock(4)	6144+7224	0.35	6144+7224	0.36	6144+7224	1.40	6144+7224	1.45	6144	0.80	6144	0.81
mutex-musl(2)	20+2	0.02	20+2	0.01	20+2	0.07	20+2	0.07	20	0.06	20	0.07
mutex-musl(3)	136728+12834	4.74	136728+12834	5.03	25146+93000	11.89	25146+93000	12.04	25146+81972	10.90	14736+36846	5.29
mutex(2)	12+2	0.02	12+2	0.02	12+2	0.07	12+2	0.07	12	0.07	10	0.07
mutex(3)	9486+1236	0.35	6582+1188	0.25	9486+1236	1.07	6582+1188	0.84	6582+336	0.76	3618+312	0.44
ms-queue(3)	925+350	0.13	75+284	0.06	901+374	0.58	901+374	0.58	901+374	0.59	901+374	0.60
ms-queue(4)	11696504+8399226	2388.57	10662+192438	18.35	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot
linuxrwlocks(3)	38033+31993	3.03	24+59	0.02	38033+31993	6.95	38033+31993	7.24	38033	4.36	3840	0.54
linuxrwlocks(4)	\odot	\odot	1060+5518	0.22	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot
ttaslock(3)	162+183	0.02	162+183	0.03	162+183	0.10	36+81	0.08	36	0.07	36	0.07
ttaslock(4)	20760+29440	1.34	20760+29440	1.46	20760+29440	4.94	576+2308	0.30	576	0.15	576	0.15
seqlock(3)	147+230	0.04	9+83	0.02	147+230	0.14	9+83	0.10	9+36	0.08	9+36	0.09
seqlock(4)	87980+105123	19.68	88+2805	0.17	87980+104583	41.58	88+2769	0.44	88+729	0.20	88+729	0.20
mPMC-queue(3)	11206+11612	1.35	166+987	0.09	11206+8188	3.35	166+840	0.24	166+517	0.20	76+421	0.17
mPMC-queue(4)	\odot	\odot	39706+1277783	87.18	\odot	\odot	39706+1123234	226.45	39706+360426	88.29	5410+114208	24.15
treiber-stack(3)	426	0.04	274+80	0.04	426	0.16	274+80	0.14	274+60	0.15	274+60	0.15
treiber-stack(4)	1546168+9216	217.44	250088+167916	33.17	1546168+9216	403.58	250088+167916	98.24	250088+90896	87.92	250088+90896	88.20
sortnet(4)	\dagger	\dagger	1+728	0.33	1+312	0.48	1+312	0.45	1	0.08	1	0.08
sortnet(5)	\dagger	\dagger	1+15231	10.87	1+4517	9.38	1+4517	9.47	1	0.08	1	0.08
sortnet(6)	\dagger	\dagger	1+163292	140.83	1+38285	100.18	1+38285	98.82	1	0.08	1	0.08

all of them (in the remaining three). Moreover, we see that IFAA is effective in mutex and mutex-musl, and manages to almost halve the total number of executions explored.

PLP fails to identify the loop purity in ms-queue. The restriction on the form of purity conditions imposed by our implementation in NIDHUGG is underapproximating the purity condition to **[False]**. This demonstrates a downside with doing purity analysis statically, as SAVER never needs to represent purity conditions in order to eliminate pure loop iterations.

In linuxrwlocks, PLP is ineffective, because this benchmark does not contain pure loop iterations as we have defined them. Rather, the loop contains a pair of fetch-and-add and fetch-and-sub that cancel out, which is called a “zero-net-effect” loop in the SAVER paper [16]. These are out of scope for a static analysis, as SAVER has to dynamically undo the elimination if a read appears to have observed the intermediate effect. Despite the lack of PLP, OPTIMAL-DPOR-AWAIT significantly speeds up linuxrwlocks.

In ttaslock, we believe some implementation issue is preventing SAVER from eliminating pure loop iterations. PLP does work, however, and **awaits** eliminate all the blocked executions.

In the next three benchmarks (seqlock, mPMC-queue and treiber-stack), PLP discovers the same pure loop iterations as SAVER, and permits a rewrite to **awaits** that significantly reduces the search space, even by an order of magnitude for seqlock, and on mPMC-queue IFAA further halves it.

Finally, OPTIMAL-DPOR-AWAIT really shines on sortnet. GENMC cannot take advantage of **awaits**, and so has to explore

an exponential number of (assume-blocked) traces, where NIDHUGG can explore the program in just one. Unfortunately, GENMC v0.5.3 crashes on this benchmark, but we believe it would yield the same numbers as SAVER, which also explores a significant number of redundant executions.

B. Effectiveness on SafeStack

Next, we evaluate the ability of OPTIMAL-DPOR-AWAIT to expose difficult-to-find bugs in real-world code bases. The benchmark we will use is called safestack. It was first posted to the CHES forum, and subsequently included in the SCTBench [23] and SVComp benchmark suites. The original safestack code attempts to implement a lock-free stack but contains an ABA bug which is quite challenging for concurrency testing and SMC tools to find, in the sense that exposing the bug requires at least five context switches. The test harness is also quite big, containing three threads each performing four operations on the stack. Let us refer to this original harness as safestack-444 to indicate that each of its three threads performs four operations (pop, push, pop, push). We will also use shortened versions of this harness: four versions with just two threads, and four versions where each of the three threads performs fewer operations. The smallest harness that exposes the bug is safestack-331.

We first compare the two SMC tools and their algorithms on versions of safestack that do not exhibit the bug and thus require exhaustive exploration of all traces. Table II shows the results. First, notice that the dynamic technique that SAVER implements

Table II: Number of (complete+blocked) executions that SMC algorithms in GENMC and NIDHUGG explore on shortened, bug-free versions of safestack.

Benchmark	GENMC		NIDHUGG			
	Baseline	SAVER	Baseline	PLP	PLP+Await	...+IFAA
safestack-21(2)	119+6	119+6	119+6	34+2	34+1	19+1
safestack-31(2)	928+107	928+107	928+107	103+27	103+25	56+25
safestack-32(2)	7189+296	7189+296	7189+296	1073+27	1073+12	463+12
safestack-33(2)	121334+12652	121334+12652	121334+12652	6434+1636	6434+1584	2600+1160
safestack-211(3)	1267120+325932	995224+325932	1259280+324382	2690+1126	2690+928	962+686
safestack-311(3)	0+286818740	0+275399108	⊙	0+26536	0+24078	0+14960
safestack-321(3)	⊙	⊙	⊙	906529+388117	906529+331337	288057+216830

is completely or mostly ineffective in these programs; compare it to the baseline numbers. In contrast, PLP achieves significant reduction of the set of executions that NIDHUGG explores. Finally, both the transformation of **assumes** to **awaits** and the IFAA optimisation are applicable and result in further reductions in the number of explored executions. The number of complete traces is 0 on *safestack-311* since the code does not allow popping the last element, so all traces end up with one thread livelocking in *pop* with the queue containing only one element. For Table II, the timeout used is 10 hours.

With our next and last experiment, using *safestack-331*, we can evaluate the tools’ abilities to expose the bug. Neither GENMC, with or without SAVER, nor baseline NIDHUGG find anything after running for more than 2000 hours! On the other hand, if we run NIDHUGG with PLP, awaits, and IFAA, it discovers the bug in just 8 minutes, after exploring $2 + 2453474$ traces. How much of its search space an SMC tool has to search before it encounters a bug can be up to “luck”, so to ensure that this result is not due to luck we “fix” the bug by commenting out all the assertions in the benchmark and run NIDHUGG again. This gives us an upper bound on the size of the search space, i.e., how much would need to be searched to find the bug in the worst case, and also provides an indication of how long it might take to verify the program after fixing the bug. On the fixed *safestack-331*, NIDHUGG terminates in only 24 minutes after exploring $5772 + 8521721$ traces. This demonstrates how the techniques we presented in this paper substantially reduce the search space on *safestack*, allowing the bug to be found or its absence verified by an exhaustive SMC technique. To our knowledge, no other exhaustive technique has ever been able to discover the bug in *safestack*.

VI. RELATED WORK

Since SMC tools assume the analysed program to terminate, they must first bound unbounded loops. Several tools [2, 21, 14, 15] have an automatic loop unroller that is parameterised by a chosen loop bound. Several SMC tools, including NIDHUGG [2], RCMC [14] and GENMC [15], transform simple forms of spinloops, such as the one shown in Fig. 2a, to **assume** statements, but only transform simple polling loops that can be recognised syntactically. We are not aware of any tool that transforms loops into **await** statements, meaning existing tools are susceptible to scalability problems for programs like the sorting networks shown in Fig. 1. An SMC technique that can diagnose livelocks of spinloops under fair scheduling is VSYNC [22]. However, to do so it enforces fairness, and cannot

bound the loop even with an **assume**, thus exploring many more traces than tools which transform spinloops to **assumes**.

SAVER [16] also aims to block pure loop iterations by introducing **assume** statements. It identifies pure loop iterations dynamically, instead of by static analysis as in our approach. SAVER’s approach allows to detect a larger class of pure loop iterations, but it does not allow further rewrite with **awaits**. Furthermore, our PLP transformation can block a looping thread at any point in the loop, not just at the back edge. SAVER also employs several smaller program transformations, such as loop rotation and merging of bisimilar control flow graph nodes, that can increase the number of loops that may qualify as pure. These transformations are orthogonal to the detection of pure loop iterations, and could also be used in our framework.

Checking for purity of loop iterations is an idea that has appeared in other contexts, such as to verify atomicity for concurrent data structures [7, 19] and to reduce complexity for model checking them (e.g., [4]).

The Optimal-DPOR algorithm implemented in NIDHUGG, handles mutex locks but not **await** statements. In the journal article of the Optimal-DPOR algorithm [3], principles for handling other blocking statements are presented. Our OPTIMAL-DPOR-AWAIT develops these principles into a practical and efficient algorithm, which we have also implemented in NIDHUGG. As future work, the Optimal-DPOR with Observers [5] algorithm, which allows two statements to only conflict in the presence of a third event, could also be extended (potentially at higher cost) to handle **awaits**.

VII. CONCLUDING REMARKS

We have presented techniques for making SMC with DPOR more effective on loops that perform pure iterations, including a static program analysis technique to detect pure loop executions, a program transformation to block and also remove them, a weakening of the standard conflict relation, and an optimal DPOR algorithm which handles the so introduced concepts. We have implemented the techniques in NIDHUGG, showing that they can significantly speed up the analysis of concurrent programs with pure loops, and also detect concurrency errors.

ACKNOWLEDGEMENTS

This work was partially supported by the Swedish Research Council through grants #621-2017-04812 and 2019-05466, and by the Swedish Foundation for Strategic Research through project aSSiST. We thank the anonymous FMCAD reviewers for detailed comments and suggestions which have improved the presentation aspects of our work.

REFERENCES

- [1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, “Optimal dynamic partial order reduction,” in *Symposium on Principles of Programming Languages*, ser. POPL 2014. New York, NY, USA: ACM, 2014, pp. 373–384. [Online]. Available: <http://doi.acm.org/10.1145/2535838.2535845>
- [2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, “Stateless model checking for TSO and PSO,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 9035. Berlin, Heidelberg: Springer, 2015, pp. 353–367. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46681-0_28
- [3] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, “Source sets: A foundation for optimal dynamic partial order reduction,” *Journal of the ACM*, vol. 64, no. 4, pp. 25:1–25:49, Sep. 2017. [Online]. Available: <http://doi.acm.org/10.1145/3073408>
- [4] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezne, “An integrated specification and verification technique for highly concurrent data structures,” *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 5, pp. 549–563, 2017. [Online]. Available: <https://doi.org/10.1007/s10009-016-0415-4>
- [5] S. Aronis, B. Jonsson, M. Lång, and K. Sagonas, “Optimal dynamic partial order reduction with observers,” in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference*, ser. LNCS, vol. 10806. Cham: Springer, Apr. 2018, pp. 229–248. [Online]. Available: https://doi.org/10.1007/978-3-319-89963-3_14
- [6] M. Christakis, A. Gotovos, and K. Sagonas, “Systematic testing for detecting concurrency errors in Erlang programs,” in *Sixth IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST 2013. Los Alamitos, CA, USA: IEEE, Mar. 2013, pp. 154–163. [Online]. Available: <https://doi.org/10.1109/ICST.2013.50>
- [7] C. Flanagan, S. Freund, and S. Qadeer, “Exploiting purity for atomicity,” *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 275–291, Apr. 2005. [Online]. Available: <https://doi.org/10.1109/TSE.2005.47>
- [8] C. Flanagan and P. Godefroid, “Dynamic partial-order reduction for model checking software,” in *Principles of Programming Languages, (POPL)*. New York, NY, USA: ACM, Jan. 2005, pp. 110–121. [Online]. Available: <http://doi.acm.org/10.1145/1040305.1040315>
- [9] P. Godefroid, “Model checking for programming languages using VeriSoft,” in *Principles of Programming Languages, (POPL)*. New York, NY, USA: ACM Press, Jan. 1997, pp. 174–186. [Online]. Available: <http://doi.acm.org/10.1145/263699.263717>
- [10] —, “Software model checking: The VeriSoft approach,” *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, Mar. 2005. [Online]. Available: <http://dx.doi.org/10.1007/s10703-005-1489-x>
- [11] P. Godefroid, R. S. Hanmer, and L. Jagadeesan, “Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft,” in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. New York, NY, USA: ACM, Mar. 1998, pp. 124–133. [Online]. Available: <https://doi.org/10.1145/271771.271800>
- [12] B. Jonsson, M. Lång, and K. Sagonas, “Awaiting for Godot: Stateless model checking that avoids executions where nothing happens,” arXiv CoRR, Aug. 2022, Extended Version with Proofs. [Online]. Available: <https://arxiv.org/abs/2208.09259>
- [13] —, “Replication Package for Awaiting for Godot: Stateless Model Checking that Avoids Executions where Nothing Happens,” Aug. 2022, artifact for the FMCAD 2022 paper with the same title. [Online]. Available: <https://doi.org/10.5281/zenodo.6979940>
- [14] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, “Effective stateless model checking for C/C++ concurrency,” *Proc. ACM on Program. Lang.*, vol. 2, no. POPL, pp. 17:1–17:32, Jan. 2018. [Online]. Available: <https://doi.org/10.1145/3158105>
- [15] M. Kokologiannakis, A. Raad, and V. Vafeiadis, “Model checking for weakly consistent libraries,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, Jun. 2019, pp. 96–110. [Online]. Available: <https://doi.org/10.1145/3314221.3314609>
- [16] M. Kokologiannakis, X. Ren, and V. Vafeiadis, “Dynamic partial order reductions for spinloops,” in *Formal Methods in Computer Aided Design*, ser. FMCAD 2021. IEEE, Oct. 2021, pp. 163–172. [Online]. Available: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_25
- [17] M. Kokologiannakis and K. Sagonas, “Stateless model checking of the Linux kernel’s hierarchical read-copy-update (tree RCU),” in *Proceedings of International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: ACM, 2017, pp. 172–181. [Online]. Available: <https://doi.org/10.1145/3092282.3092287>
- [18] M. Kokologiannakis and V. Vafeiadis, “GenMC: A model checker for weak memory models,” in *Computer Aided Verification - 33rd International Conference, CAV 2021, Proceedings, Part I*, ser. LNCS, vol. 12759. Springer, Jul. 2021, pp. 427–440. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_20
- [19] M. Lesani, T. D. Millstein, and J. Palsberg, “Automatic atomicity verification for clients of concurrent data structures,” in *Computer Aided Verification, CAV 2014*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Cham: Springer, Jul. 2014, pp. 550–567. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_37
- [20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, “Finding and reproducing heisenbugs in concurrent programs,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI ’08. Berkeley, CA, USA: USENIX Association, Dec. 2008, pp. 267–280. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1855741.1855760>
- [21] B. Norris and B. Demsky, “A practical approach for model checking C/C++11 code,” *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, pp. 10:1–10:51, May 2016. [Online]. Available: <http://doi.acm.org/10.1145/2806886>
- [22] J. Oberhauser, R. L. d. L. Chehab, D. Behrens, M. Fu, A. Paolillo, L. Oberhauser, K. Bhat, Y. Wen, H. Chen, J. Kim, and V. Vafeiadis, “Vsync: Push-button verification and optimization for synchronization primitives on weak memory models,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: ACM, 2021, p. 530–545. [Online]. Available: <https://doi.org/10.1145/3445814.3446748>
- [23] P. Thomson, A. F. Donaldson, and A. Betts, “Concurrency testing using controlled schedulers: An empirical study,” *ACM Trans. Parallel Comput.*, vol. 2, no. 4, pp. 23:1–23:37, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2858651>
- [24] N. Zhang, M. Kusano, and C. Wang, “Dynamic partial order reduction for relaxed memory models,” in *Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, Jun. 2015, pp. 250–259. [Online]. Available: <http://doi.acm.org/10.1145/2737924.2737956>

Synthesizing Transducers from Complex Specifications

Anvay Grover
 The University of Wisconsin-Madison
 Madison, USA
 anvayg@cs.wisc.edu

Ruediger Ehlers
 Clausthal University of Technology
 Clausthal, Germany
 ruediger.ehlers@tu-clausthal.de

Loris D’Antoni
 The University of Wisconsin-Madison
 Madison, USA
 loris@cs.wisc.edu

Abstract—Automating string transformations has been a driving application of program synthesis. Existing synthesizers that solve this problem produce programs in domain-specific languages (DSL) that are designed to simplify synthesis and therefore lack nice formal properties. This limitation prevents the synthesized programs from being used in verification applications (e.g., to check complex pre-post conditions) and makes the synthesizers hard to modify due to their reliance on the given DSL.

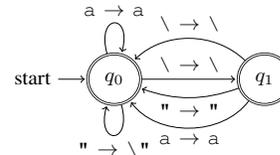
We present a constraint-based approach to synthesizing transducers, a model with strong closure and decidability properties. Our approach handles three types of specifications: input-output (i) examples, (ii) types expressed as regular languages, and (iii) distances that bound how many characters the transducer can modify when processing an input string. Our work is the first to support such complex specifications and it does so by using the algorithmic properties of transducers to generate constraints that can be solved using off-the-shelf SMT solvers. Our synthesis approach can be extended to many transducer models and it can be used, thanks to closure properties of transducers, to compute repairs for partially correct transducers.

I. INTRODUCTION

String transformations are used in data transformations [1], sanitization of untrusted inputs [2], [3], and many other domains [4]. Because in these domains bugs may cause serious security vulnerabilities [2], there has been increased interest in building tools that can help programmers verify [2], [3] and synthesize [1], [5], [6] string transformations.

Techniques for *verifying* string transformations rely on automata-theoretic approaches that provide powerful decidability properties [2]. On the other hand, techniques for *synthesizing* string transformations rely on domain-specific languages (DSLs) [1], [5]. These DSLs are designed to make synthesis practical and have to give up the closure and decidability properties enabled by automata-theoretic models. The disconnect between the two approaches raises a natural question: *Can one synthesize automata-based models and therefore retain and leverage their elegant properties?*

A *finite state transducer* (FT) is an automaton where each transition reads an input character and outputs a string of output characters. For instance, Figure 1 shows a transducer that ‘escapes’ instances of the " character. So, on input a"\a, the transducer outputs the string a\"\"a. FTs have found wide adoption in a variety of domains [3], [7] because of their many desirable properties (e.g., decidable equivalence check and closure under composition [8]). There has been



(a) Transducer *EscapeQuotes*

Examples: {a"a → a\"a, a\\a → a\\a, a\aa → a\aa, a\"a → a\"a, \ → \}
 Types: {a"}*?|[([a"]*\[a"\\][a"]*)* → a*?|[a*\[a"\\]a"}*
 Distance: At most 1 edit per input character

(b) Specification to synthesize *EscapeQuotes*

Fig. 1: Simplified version of *EscapeQuotes* from [2].

increasing work on building SMT solvers for strings that support transducers; the Ostrich tool [9] allows a user to write programs in SMT where string-transformations are modelled using transducers. One can then write constraints over such programs and use an SMT solver to automatically check for satisfiability or prove unsatisfiability of those constraints. For example, given a program like the following:

```
y = escapeQuotes(x)
z = escapeQuotes(y)
assert(y==z) //Checking idempotence
```

one can use Ostrich to write a set of constraints and use them to prove whether the assertion holds. However, to do so, one needs to first write a transducer *T* that implements the function *escapeQuotes*. However, writing transducers by hand is a cumbersome and error-prone task and what we present in this paper is an approach for synthesizing such transducers.

In this paper, we present a technique for synthesizing transducers from high-level specifications. We use three different specification mechanisms to quickly yield desirable transducers: input-output examples, input-output types, and input-output distances. When provided with the specification in Figure 1b, our approach yields the transducer in Figure 1a. While none of the three specification mechanisms are effective in isolation, they work well altogether. Input-output examples are easy to provide, but only capture finitely many inputs. Similarly, input-output types are a natural way to prevent a transducer from generating undesired strings and can often be

obtained from function/API specifications. Last, input-output distances are a natural way to specify how much of the input string should be preserved by the transformation.

We show that if the size of the transducers is fixed, all such specifications can be encoded as a set of constraints whose solution directly provides a transducer. While the constraints for examples are fairly straightforward, to encode types and distances, we show that one can use constraints to “guess” the simulation relation and the invariants necessary to prove that the transducer has the given type and respects the given distance constraint.

Because our constraint-based approach is based on decision procedures and is modular, it can support more complex models of transducers: (i) Symbolic Finite Transducers (s-FTs), which support large alphabets [10], and (ii) FTs with lookahead, which can express functions that otherwise require non-determinism. In addition, closure properties of transducers allow us to reduce repair problems for string transformations to our synthesis problem.

Contributions: We make the following contributions.

- A constraint-based synthesis algorithm for synthesizing transducers from complex specifications (Sec. III).
- Extensions of our synthesis algorithm to more complex models—e.g., symbolic transducers and transducers with lookahead—and problems—e.g., transducer repair—that showcase the flexibility of our approach and the power of working with transducers, which enjoy strong theoretical properties—unlike domain-specific languages (Sec. IV).
- ASTRA: a tool that can synthesize and repair transducers and compares well with a state-of-the-art tool for synthesizing string transformations (Sec. V).

Proofs and additional results are available at [11].

II. TRANSDUCER SYNTHESIS PROBLEM

In this section, we define the transducer synthesis problem.

A *deterministic finite automaton* (DFA) over an alphabet Σ is a tuple $D = (Q_D, \delta_D, q_D^{init}, F_D)$: Q_D is the set of states, $\delta_D : Q_D \times \Sigma \rightarrow Q_D$ is the transition function, q_D^{init} is the initial state, and F_D is the set of final states. The extended transition function $\delta_D^* : Q_D \times \Sigma^* \rightarrow Q_D$ is defined as $\delta_D^*(q, \varepsilon) = q$ and $\delta_D^*(q, au) = \delta_D^*(\delta_D(q, a), u)$. We say that D accepts a string w if $\delta_D^*(q_D^{init}, w) \in F_D$. The *regular language* $\mathcal{L}(D)$ is the set of strings accepted by a DFA D .

A total *finite state transducer* (FT) is a tuple $T = (Q_T, \delta_T^{st}, \delta_T^{out}, q_T^{init})$, where Q_T are states and q_T^{init} is the initial state. Transducers have two transition functions: $\delta_T^{st} : Q_T \times \Sigma \rightarrow Q_T$ defines the target state, while $\delta_T^{out} : Q_T \times \Sigma \rightarrow \Sigma^*$ defines the output string of each transition. The extended function for states δ_T^{st*} is defined analogously to the extended transition function for a DFA. The extended function for output strings is defined as $\delta_T^{out*}(q, \varepsilon) = \varepsilon$ and $\delta_T^{out*}(q, au) = \delta_T^{out*}(q, a) \cdot \delta_T^{out}(\delta_T^{st*}(q, a), u)$. Given a string w we use $T(w)$ to denote $\delta_T^{out*}(q_T^{init}, w)$, i.e., the output string generated by T on w . Given two DFAs P and Q , we write $\{P\}T\{Q\}$ for a transducer T iff for every string s in $\mathcal{L}(P)$, the output string $T(s)$ belongs to $\mathcal{L}(Q)$.

An *edit operation* on a string is either an insertion/deletion of a character, or a replacement of a character with a different one. For example, editing the string ab to the string acb requires one edit operation, which is inserting a c after the a . The *edit distance* $ed_dist(s, t)$ between two strings s and t is the number of edit-operations required to reach t from s . We use $len(w)$ to denote the length of a string w . The *mean edit distance* $mean_ed_dist(s, t)$ between two strings s and t is defined as $ed_dist(s, t)/len(s)$. For example, the mean edit distance from ab to acb is $1/2 = .5$.

We can now formulate the transducer synthesis problem. We assume a fixed alphabet Σ . If the specification requires that s is translated to t , we write that as $s \mapsto t$.

Problem Statement 1 (Transducer Synthesis). *The transducer synthesis problem has the following inputs and output:*

Inputs

- Number of states k and upper bound l on the length of the output of each transition.
- Set of input-output examples $E = \{s \mapsto t\}$.
- Input-output types P and Q , given as DFAs.
- A positive upper bound $d \in \mathbb{Q}$ on the mean edit distance.

Output A total transducer $T = (Q_T, \delta_T^{st}, \delta_T^{out}, q_T^{init})$ with k states such that:

- Every transition of T has an output with length at most l , i.e., $\forall q_T \in Q_T, a \in \Sigma. len(\delta_T^{out}(q, a)) \leq l$.
- T is consistent with the examples: $\forall s \mapsto t \in E. T(s) = t$.
- T is consistent with input-output types, i.e., $\{P\}T\{Q\}$.
- For every string $w \in P$, $mean_ed_dist(w, T(w)) \leq d$.

The synthesis problem that we present here is for FTs, and in Section III, we provide a sound algorithm to solve it using a system of constraints. One of our key contributions is that our encoding can be easily adapted to synthesizing richer models than FTs (e.g., symbolic transducers [8] and transducers with regular lookahead), while still using the same encoding building blocks (Section IV).

III. CONSTRAINT-BASED TRANSDUCER SYNTHESIS

In this section, we present a way to generate constraints to solve the transducer synthesis problem defined in Section II. The synthesis problem can then be solved by invoking a *Satisfiability Modulo Theories* (SMT) solver on the constraints.

We use a constraint encoding, rather than a direct algorithmic approach because of the multiple objectives to be satisfied. Synthesizing a transducer that translates a set of input-output examples is already an NP-Complete problem [12]. On top of that, we also need to handle input-output types and distances. Our encoding is divided into three parts, one for each objective, which are presented in the following subsections. This division makes our encoding very modular and programmable. In Section IV we show how it can be adapted to different transducer models and problems. We include a brief description of the size of the constraint encoding in the extended version.

The transducer we are synthesizing has k (part of the problem input) states $Q_T = \{q_0, \dots, q_{k-1}\}$. We often use q_T^{init} as an alternative for q_0 , the initial state of T .

We illustrate how our encoding represents a transition $q_1 \xrightarrow{a/bc} q_2$. The target state is captured using an uninterpreted function $d^{st} : Q_T \times \Sigma \rightarrow Q_T$, e.g., $d^{st}(q_1, a) = q_2$. Representing the output of the transition is trickier because its length is not known a priori. The output bound l allows us to limit the number of characters that may appear in the output. We use an uninterpreted function $d_{ch}^{out} : Q_T \times \Sigma \times \{0, \dots, l-1\} \rightarrow \Sigma$ to represent each character in the output string; in our example, $d_{ch}^{out}(q_1, a, 0) = b$ and $d_{ch}^{out}(q_1, a, 1) = c$. Since an output string's length can be smaller than l , we use an additional uninterpreted function $d_{len}^{out} : Q_T \times \Sigma \rightarrow \{0, \dots, l\}$ to model the length of a transition's output; in our example $d_{len}^{out}(q_1, a) = 2$. We say an assignment to the above variables extends to a transducer T for the transducer T obtained by instantiating δ^{st} and δ^{out} as described above.

A. Input-output Examples

Goal: For each input output-example $s \mapsto t \in E$, T should translate s to t .

Translating s to the correct output string means that $\delta_T^{out*}(q_T^{init}, s) = t$. Generating constraints that capture this behavior of T on an example is challenging because we do not know a priori what parts of t are produced by what steps of the transducer's run. Suppose that we need to translate $s = a_0a_1$ to $t = b_0b_1b_2$. A possible solution is for the transducer to have the run $q_0 \xrightarrow{a_0/b_0} q_1 \xrightarrow{a_1/b_1b_2} q_2$. Another possible solution might be to instead have $q_0 \xrightarrow{a_0/b_0b_1} q_1 \xrightarrow{a_1/b_2} q_2$. Notice that the two runs traverse the same states but produce different parts of the output strings at each step. Intuitively, we need a way to “track” how much output the transducer has produced before processing the i -th character in the input and what state it has landed in. For every input example $s \mapsto t$ such that $s = a_0 \dots a_n$ and $t = b_0 \dots b_m$, we introduce an uninterpreted function $config_s : \{0, \dots, n\} \rightarrow \{0, \dots, m\} \times Q_T$ such that $config_s(i) = (j, q_T)$ iff after reading $a_0 \dots a_{i-1}$, the transducer T has produced the output $b_0 \dots b_{j-1}$ and reached state q_T —i.e., $\delta_T^{out*}(q_0, a_0 \dots a_{i-1}) = b_0 \dots b_{j-1}$ and $\delta_T^{st*}(q_0, a_0 \dots a_{i-1}) = q_T$.

We describe the constraints that describe the behavior of $config_s$. Constraint 1 states that a configuration must start at the initial state and be at position 0 in the output.

$$config_s(0) = (0, q_T^{init}) \quad (1)$$

Constraint 2 captures how the configuration is updated when reading the i -th character of the input. For every $0 \leq i < n$, $0 \leq j < m$, $c \in \Sigma$, and $q_T \in Q_T$:

$$\begin{aligned} config_s(i) &= (j, q_T) \wedge a_i = c \Rightarrow \\ & \left[\bigwedge_{0 \leq z < l} (d_{ch}^{out}(q_T, c, z) = b_{j+z} \vee z \geq d_{len}^{out}(q_T, c)) \wedge \right. \\ & \left. config_s(i+1) = (j + d_{len}^{out}(q_T, c), d^{st}(q_T, c)) \right] \quad (2) \end{aligned}$$

Informally, if the i -th character is c and the transducer has reached state q_T and produced the characters $b_0 \dots b_{j-1}$ so far, the transition reading c from state q_T outputs characters

$b_j \dots b_{j+f-1}$, where f is the output length of the transition. The next configuration is then $(j + f, d^{st}(q_T, c))$.

Finally, Constraint 3 forces T to be completely done with generating t when s has been entirely read. Recall that $len(s) = n$ and $len(t) = m$.

$$\bigvee_{q_T \in Q_T} config_s(n) = (m, q_T) \quad (3)$$

The encoding for examples is sound and complete [11].

B. Input-Output Types

Goal: T should satisfy the property $\{P\}T\{Q\}$.

Encoding this property using constraints is challenging because it requires enforcing that when T reads one of the (potentially) infinitely many strings in P it always outputs a string in Q . To solve this problem, we draw inspiration from how one proves that the property $\{P\}T\{Q\}$ holds—i.e., using a simulation relation that relates runs over P , T and Q . Intuitively, if P has read some string w , we need to be able to encode the behavior of T in terms of w , i.e., what state of T this transducer is in after reading w and what output string w' it produced. Further, we also need to be able to encode in which state Q would be after reading the output string w' . We do this by introducing a function $sim : Q_P \times Q_T \times Q_Q \rightarrow \{0, 1\}$, which preserves the following invariant: $sim(q_P, q_T, q_Q)$ holds if there exist strings w, w' such that $\delta_P^*(q_P^{init}, w) = q_P$, $\delta_T^{st*}(q_T^{init}, w) = q_T$, $\delta_T^{out*}(q_T^{init}, w) = w'$, and $\delta_Q^*(q_Q^{init}, w') = q_Q$.

Constraint 4 states the initial condition of the simulation—i.e., P , T , and Q are in their initial states.

$$sim(q_P^{init}, q_T^{init}, q_Q^{init}) \quad (4)$$

Constraint 5 encodes how we advance the simulation relation for states q_P, q_T, q_Q and for a character $c \in \Sigma$, using free variables c_0, \dots, c_{l-1} and q_Q^0, \dots, q_Q^l that are separate for each combination of q_P, q_T, q_Q , and c :

$$\begin{aligned} sim(q_P, q_T, q_Q) \Rightarrow & \bigwedge_{0 \leq z < l} (d_{len}^{out}(q_T, c) = z \Rightarrow \\ & \left[\bigwedge_{0 \leq x < z} d_{ch}^{out}(q_T, c, x) = c_x \right] \wedge \\ & [q_Q^0 = q_Q \wedge \bigwedge_{1 \leq x < z} q_Q^x = d_Q(q_Q^{x-1}, c_{x-1})] \wedge \\ & sim(\delta_P(q_P, c), d^{st}(q_T, c), q_Q^z)) \quad (5) \end{aligned}$$

Intuitively, if $sim(q_P, q_T, q_Q)$ and we read a character c , P moves to $\delta_P(q_P, c)$ and T moves to $d^{st}(q_T, c)$. However, we also need to advance Q and the d_{len}^{out} symbols produced by d_{ch}^{out} . We hard-code the transition relation δ_Q in an uninterpreted function $d_Q : Q_Q \times \Sigma \rightarrow Q_Q$, and apply it to compute the output state reached when reading the output string. E.g., if $d_{len}^{out}(q_T, c) = 2$ and $d_{ch}^{out}(q_T, c, 0) = c_0$ and $d_{ch}^{out}(q_T, c, 1) = c_1$, the next state in Q is $d_Q(d_Q(q_Q, c_0), c_1)$.

Lastly, Constraint 6 states that if we encounter a string in $\mathcal{L}(P)$ —i.e., P is in a state $q_P \in F_P$ —the relation does not

contain a state $q_Q \notin F_Q$. Since Q is deterministic, this means that Q accepts T 's output.

$$\bigwedge_{q_P \in F_P} \bigwedge_{q_Q \notin F_Q} \neg \text{sim}(q_P, q_T, q_Q) \quad (6)$$

The constraint encoding for types is sound and complete [11].

C. Input-output Distance

Goal: The mean edit distance between any input string w in $\mathcal{L}(P)$ and the output string $T(w)$ should not exceed d .

Capturing the edit distance for all the possible inputs in the language of P and the corresponding outputs produced by the transducer is challenging because these sets can be infinite. Furthermore, exactly computing the edit distance between an input and an output string may involve comparing characters appearing on different transitions in the transducer run. For example, consider the transducer shown in Figure 2a and suppose that we are only interested in strings in the input type $P = a(ba)^*a$. The first transition from q_0 deletes the a , therefore making 1 edit. This transducer has a cycle between states q_1 and q_2 , which can be taken any number of times. Each iteration, locally, would require that we make 2 edits: one to change the b to a , and the other to change the a to b . However, the total number of edits made over any string in the input type $P = a(ab)^*a$ by this transducer is 1, because the transducer changes strings of the form $a(ba)^n a$ to be of the form $(ab)^n a$. Looking at the transitions in isolation, we are prevented from deducing that the edit distance is always 1 because the first transition delays outputting a character. If there was no such delay, as is the case for the transducer in Figure 2b, which is equivalent on the relevant input type to the one in Figure 2a, then this issue would not arise.

We take inspiration from Benedikt et al. [13] and focus on the simpler problem of synthesizing a transducer that has ‘aggregate cost’ that satisfies the given objective.¹ For a transducer T and string $s = a_0 \dots a_n$, let $q_T^{init} \xrightarrow{a_0/y_0} q_T^1 \dots q_T^n \xrightarrow{a_n/y_n} q_T^{n+1}$ be the run of s on T . Then, the *aggregate cost* of T on s is the sum of the edit distances $ed_{\text{dist}}(a_i, y_i)$ over all indices $0 \leq i \leq n$. The *mean aggregate cost* of T on s is the aggregate cost divided by $\text{len}(s)$, the length of s . It follows that if T has a mean aggregate cost lower than some specified d for every string, then it also has a mean edit distance lower than d for every string.

However, the mean aggregate cost overapproximates the edit distance, e.g., the transducer in Figure 2a has mean aggregate cost 1, while the mean edit distance when considering only strings in $P = a(ab)^*a$ is less than 1/2. For this reason, if the mean edit distance objective was set to 1/2, our constraint encoding can only synthesize the transducer in Figure 2b, and not the equivalent one in Figure 2a.

¹Benedikt et al. [13] studied a variant of the problem where the distance is bounded by some finite constant. Their work shows that when there is a transducer between two languages that has some bounded global edit distance, then there is also a transducer that is bounded (but with a different bound) under a local method of computing the edit distance—i.e., one where the computation of the edit distance is done transition by transition.

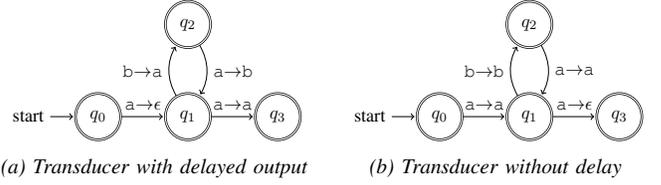


Fig. 2: Transducers with and without delay.

Our encoding is complete for transducers in which the aggregate cost coincides with the actual edit distance. We leave the problem of being complete with regards to global edit distance as an open problem. In fact, we are not even aware of an algorithm for *checking* (instead of synthesizing) whether a transducer satisfies a mean edit distance objective.² In Section IV-B, we present transducers with lookahead, which can mitigate this source of incompleteness. Furthermore, our evaluation shows that using the aggregate cost and enabling lookahead are both effective techniques in practice.

We can now present our constraints. First, we provide constraints for the edit distance of individual transitions (recall that transitions are being synthesized and we therefore need to compute their edit distances separately). Secondly, we provide constraints that implicitly compute state invariants to capture the aggregate cost between input and output strings at various points in the computation. We are given a rational number d as an input to the problem, which is the allowed distance bound.

Edit Distance of Individual Transitions. To compute the edit distance between the input and the output of each transition, we introduce a function $ed: Q_T \times \Sigma \rightarrow \mathbb{Z}$. For a transition from state q_T reading a character c , $ed(q_T, c)$ represents the edit distance between c and $\delta_T^{out}(q_T, c)$. Notice that this quantity is bounded by the output bound l . The constraints to encode the value of this function are divided into two cases: i) the output of the transition contains the input character c (Constraint 7), ii) the output of the transition *does not* contain the input character c (Constraint 8). In both cases, the values are set via a simple case analysis on whether the length of the output is 0 (edit distance is 1) or not (the edit distance is related to the length of the output).

$$\begin{aligned} & [\bigvee_{0 \leq z < d_{\text{len}}^{\text{out}}(q_T, c)} d_{\text{ch}}^{\text{out}}(q_T, c, z) = c] \Rightarrow \\ & [d_{\text{len}}^{\text{out}}(q_T, c) = 0 \Rightarrow ed(q_T, c) = 1 \wedge \\ & d_{\text{len}}^{\text{out}}(q_T, c) \neq 0 \Rightarrow ed(q_T, c) = d_{\text{len}}^{\text{out}}(q_T, c) - 1] \end{aligned} \quad (7)$$

$$\begin{aligned} & [\bigwedge_{0 \leq z < d_{\text{len}}^{\text{out}}(q_T, c)} d_{\text{ch}}^{\text{out}}(q_T, c, z) \neq c] \Rightarrow \\ & [d_{\text{len}}^{\text{out}}(q_T, c) = 0 \Rightarrow ed(q_T, c) = 1 \wedge \\ & d_{\text{len}}^{\text{out}}(q_T, c) \neq 0 \Rightarrow ed(q_T, c) = d_{\text{len}}^{\text{out}}(q_T, c)] \end{aligned} \quad (8)$$

²The mean edit distance is similar to mean payoff [14], which discounts a cost by the length of a string and looks at the behavior of a transducer in the limit. Our distance is different because 1) it looks at finite-length strings, and 2) it requires computing the edit distance, which cannot be done one transition at a time.

Edit Distance of Arbitrary Strings. Suppose that T has the transitions $q_0 \xrightarrow{a/a} q_1 \xrightarrow{a/bc} q_2$, and the specified mean edit distance is $d = 0.5$. The edit distance is 0 for the first transition and 2 for the second one. For the input string aa , the mean aggregate cost is $2/2$, which means that the specification is not satisfied. In general, we cannot keep track of every input string in the input type and look at its length and the number of edits that were made over it. So, how can we compute the mean aggregate cost over any input string? The first part of our solution is to scale the edit distance over a single transition depending on the specified mean edit distance. This operation makes it such that an input string is under the edit distance bound if the sum of the weighted edit distances of its transitions is ≥ 0 . The invariant we need to maintain is that the sum of the weights at any stage of the run gives us where we are with regard to the mean aggregate cost. For each transition we compute the difference between the edit distance over the transition and the specified mean edit distance d . We introduce the uninterpreted function $\text{wed} : Q_T \times \Sigma \rightarrow \mathbb{Q}$, which stands for weighted edit distance. For a transition at q_T reading a character c , the weighted edit distance is given by $\text{wed}(q_T, c) = d - \text{ed}(q_T, c)$. The sum of the weights of all transitions tells us the cumulative difference. Going back to our example, the weighted edit distances of the two transitions are $\text{wed}(q_0, a) = 0.5$ and $\text{wed}(q_1, a) = -1.5$, making the cumulative distance -1 and implying that the specification is violated. We can now compute the mean edit distance over a run without keeping track of the length of the run and the number of edits performed over it.

We still need to compute the weighted edit distance for every string in the possibly infinite language $\mathcal{L}(P)$. Building on the idea of simulation from the previous section, we introduce a new function called $\text{en} : Q_P \times Q_T \times Q_Q \rightarrow \mathbb{Q}$, which tracks an upper bound on the sum of the distances so far at that point in the simulation. This function is similar to a *progress measure*, which is a type of invariant used to solve *energy games* [15], a connection we expand on in Section VI. In particular, we already know that if there exist strings w, w' such that $\delta_P^*(q_P^{\text{init}}, w) = q_P$, $\delta_T^{st*}(q_T^{\text{init}}, w) = q_T$, $\delta_T^{\text{out}*}(q_T^{\text{init}}, w) = w'$, and $\delta_Q^*(q_Q^{\text{init}}, w') = q_Q$, then we have $\text{sim}(q_P, q_T, q_Q)$. Let this run over T be denoted by $q_T^{\text{init}} \xrightarrow{a_0/y_0} q_T^1 \dots q_T^{n-1} \xrightarrow{a_{n-1}/y_{n-1}} q_T$, where $w = a_0 \dots a_{n-1}$, $w' = y_0 \dots y_{n-1}$, and $q_T = q_T^n$. We have that $\text{en}(q_P, q_T, q_Q) \geq \sum_{i=0}^{n-1} \text{wed}(q_T^i, a_i)$.

The en function is a budget on the number of edits we can still perform. At the initial states, we start with no ‘initial credit’ and the energy is 0.

$$\text{en}(q_P^{\text{init}}, q_T^{\text{init}}, q_Q^{\text{init}}) = 0 \quad (9)$$

Constraint 10 bounds the energy budget according to the weighted edit distance of a transition by computing the minimum budget required at any point to still satisfy the distance bound. For each combination of q_P, q_T, q_Q , and $c \in \Sigma$, the

constraint uses free variables c_0, \dots, c_l and q_Q^0, \dots, q_Q^{l-1} :

$$\bigwedge_{0 \leq z < l} (\text{d}_{1\text{en}}^{\text{out}}(q_T, c) = z \Rightarrow [\bigwedge_{0 \leq x < z} \text{d}_{\text{ch}}^{\text{out}}(q_T, c, x) = c_x] \wedge [q_Q^0 = q_Q \wedge \bigwedge_{1 \leq x < z} q_Q^x = \text{d}_Q(q_Q^{x-1}, c_{x-1})]) \wedge \text{en}(q_P, q_T, q_Q) \geq \text{en}(\delta_P(q_P, c), \text{d}^{\text{st}}(q_T, c), q_Q^z) - \text{wed}(q_T, c)) \quad (10)$$

In our example, Constraint 10 encodes that the energy at q_0 can be 1 less than that at q_1 , but that the energy at q_1 needs to be 3 greater than at q_2 since we need to spend 3 edit operations over the second transition.

At any point during a run, the transducer is allowed to go below the mean edit distance and then ‘catch up’ later because we only care about the edit distance when the transducer has finished reading a string in $\mathcal{L}(P)$. Therefore, when we reach a final state of P , the transducer should not be in ‘energy debt’.

$$\bigwedge_{q_P \in F_P} \text{sim}(q_P, q_T, q_Q) \Rightarrow \text{en}(q_P, q_T, q_Q) \geq 0 \quad (11)$$

The encoding presented in this section is sound [11].

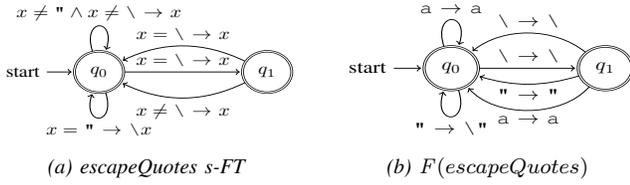
IV. RICHER MODELS AND SPECIFICATIONS

We extend our technique to more expressive models (Sections IV-A and IV-B) and show how our synthesis approach can be used not only to synthesize transducers, but also to repair them (Section IV-C). In the extended version, we describe an encoding of an alternative distance measure [11].

A. Symbolic Transducers

Symbolic finite automata (s-FA) and transducers (s-FT) extend their non-symbolic counterparts by allowing transitions to carry predicates and functions to represent (potentially infinite) sets of input characters and output strings. Figure 3a shows an s-FT that extends the `escapeQuotes` transducer from Figure 1a to handle alphabetic characters. The bottom transition from q_0 reads a character $"$ (bound to the variable x) and outputs the string $\backslash"$ (i.e., a \backslash followed by the character stored in x). Symbolic finite automata (s-FA) are s-FTs with no outputs. To simplify our exposition, we focus on s-FAs and s-FTs that only operate over ASCII characters that are ordered by their codes. In particular, all of our predicates are unions of intervals over characters (i.e., $x \neq \backslash$ is really the union of intervals $[\text{NUL}[\]$ and $[\]\text{-DEL}$]); we often use the predicate notation instead of explicitly writing the intervals for ease of presentation. Furthermore, we only consider two types of output functions: constant characters and offset functions of the form $x + k$ that output the character obtained by taking the input x and adding a constant k to it—e.g., applying $x + (-32)$ to a lowercase alphabetic letter gives the corresponding uppercase letter.

In the rest of the section, we show how we can solve the transducer synthesis problem in the case where P and Q are s-FAs and the goal is to synthesize an s-FT (instead of an FT) that meets the given specification. Intuitively, we do this by ‘finitizing’ the alphabet of the now symbolic input-output types, synthesizing a finite transducer over this alphabet using



(c) Set of minterms and their witness elements

Fig. 3: Example of Finitization

the technique presented in Section III, and then extracting an s-FT from the solution.

Finitizing the Alphabet. The idea of finitizing the alphabet of s-FAs is a known one [8] and is based on the concept of *minterms*, which is the set of maximal satisfiable Boolean combinations of the predicates appearing in the s-FAs. For an s-FA M , we can define its set of predicates as: $Predicates(M) = \{\phi \mid q \xrightarrow{\phi} q' \in \delta_M\}$. The set of minterms $mterms(M)$ is the set of satisfiable Boolean combinations of all the predicates in $Predicates(M)$. For example, for the set of predicates over the s-FT `escapeQuotes` in Figure 3a, we have that $mterms(escapeQuotes) = \{x \neq " \wedge x \neq \backslash, x = ", x = \backslash\}$. The reader can learn more about minterms in [8]. We assign each minterm a representative character, as indicated in Figure 3c, and then construct a finite automaton from the resulting finite alphabet Σ . For a character $c \in \Sigma$, we refer to its corresponding minterm by $mt(c)$. In the other direction, for each minterm $\psi \in minterms(M)$, we refer to its uniquely determined representative character by $wit(\psi)$.

For an s-FA M , we denote its corresponding FA over the alphabet $mterms(M)$ with $F(M)$. Given an s-FA M , the set of transitions of $F(M)$ is defined as follows:

$$\delta_{F(M)} = \{q \xrightarrow{wit(\psi)} q' \mid q \xrightarrow{\phi} q' \wedge \psi \in mterms(M) \wedge \text{IsSat}(\psi \wedge \phi)\}$$

This algorithm replaces a transition guarded by a predicate ϕ in the given s-FA with a set of transitions consisting of the witnesses of the minterms where ϕ is satisfiable. In interval arithmetic this is the set of intervals that intersect with the interval specified by ϕ . The transition from q_1 guarded by the predicate $[x \neq \backslash]$ in Figure 3a intersects with 2 minterms $[x \neq " \wedge x \neq \backslash]$ and $[x = "]$. As a result, we see that this transition is replaced by two transitions in Figure 3b, one that reads `"` and another that reads `a`.

From FTs to s-FTs. Once we have synthesized an FT T , we need to extract an s-FT from it. There are many s-FTs equivalent to a given FT and here we present one way of doing this conversion which is used in our implementation. Let the size of an interval I (the number of characters it contains) be given by $size(I)$, and the offset between 2 intervals I_1 and I_2 (i.e. the difference between the least elements of I_1 and I_2) be given by $offset(I_1, I_2)$. Suppose we have a transition $q \xrightarrow{c/y_0 \dots y_n} q'$, where $c, y_i \in \Sigma$. Then, we construct a transition

$q \xrightarrow{mt(c)/f_0 \dots f_n} q'$, where for each y_i , the corresponding function f_i is determined by the following rules (x always indicates variable bound to the input predicate):

- 1) If $c = y_i$, then $f_i = (x)$, i.e. the identity function.
- 2) If $mt(c)$ and $mt(y_i)$ consist of single intervals I_1 and I_2 , respectively, such that $size(I_1) = size(I_2)$, then $f_i = (x + offset(I_1, I_2))$. For instance, if the input interval is $[a-z]$ and the output interval is $[A-Z]$, then the output function is $(x + (-32))$, which maps lowercase letters to uppercase ones.
- 3) Otherwise $f_i = y_i$ —i.e., the output is a character in the output minterm.

While our s-FT recovery algorithm is sound, it may apply case 3 more often than necessary and introduce many constants, therefore yielding a transducer that does not generalize well to unseen examples. Our evaluation shows that our technique works well in practice. The proof of soundness of this algorithm in the extended version [11].

B. Synthesizing Transducers with Lookahead

Deterministic transducers cannot express functions where the output at a certain transition depends on future characters in the input. Consider the problem of extracting all substrings of the form `<x>` (where `x ≠ <`) from an input string. This is the `getTags` problem from [16]. A deterministic transducer cannot express this transformation because when it reads `<` followed by `x` it has to output `<x` if the next character is a `>` and nothing otherwise. However, the transducer does not have access to the next character!

Instead, we extend our technique to handle deterministic transducers with lookahead, i.e., the ability to look at the string suffix when reading a symbol. Formally, a *Transducer with Regular Lookahead* is a pair (T, R) where T is an FT with $\Sigma_T = Q_R \times \Sigma$, and R is a total DFA with $\Sigma_R = \Sigma$. The transducer T now has another input in its transition function, although it still only outputs characters from Σ , i.e., $\delta_T^{out} : Q_T \times (Q_R \times \Sigma) \rightarrow \Sigma$, and $\delta_T^{st} : Q_T \times (Q_R \times \Sigma) \rightarrow Q_T$. The semantics is defined as follows. Given a string $w = a_0 \dots a_n$, we define a function r_w such that $r_w(i) = \delta_R(q_R^{init}, a_n \dots a_{i+1})$. In other words, $r_w(i)$ gives the state reached by R on the reversed suffix starting at $i+1$. At each step i , the transducer T reads the symbol $(a_i, r_w(i))$. The extended transition functions now take as input a lookahead word, which is a sequence of pairs of lookahead states and characters, i.e., from $(Q_R \times \Sigma)^*$.

To synthesize transducers with lookahead, we introduce uninterpreted functions d_R for the transition function of R , and $look_w$ for the r -values of w on R . We also introduce a bound k_R on the number of states in the lookahead automaton R (our algorithm has to synthesize both T and R). The modified constraints needed to encode input-output types and input-output examples to use lookahead are described in the extended version of the paper [11]. Part of the transducer with lookahead we synthesize for the `getTags` problem is shown in Figure 4. Notice that there are 2 transitions out of q_1 for the same input but different lookahead state: the string `<x` is outputted when the lookahead state is r_1 .

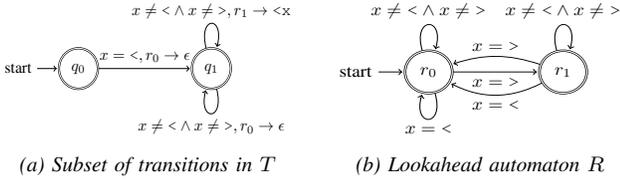


Fig. 4: Regular lookahead for getTags

Lookahead and aggregate cost: Lookahead can help representing transducers, even deterministic ones, in a way that has lower aggregate cost—i.e., the aggregate cost better approximates the actual edit distance. Suppose that we want to synthesize a transducer that translates the string `abc` to `ab` and the string `abd` to `bd`. This translation can be done using a deterministic transducer with transitions $q_0 \xrightarrow{a/\epsilon} q_1 \xrightarrow{b/\epsilon} q_2$, followed by two transitions from q_2 that choose the correct output based on the next character. Such a transducer would have a high aggregate cost of 4, even though the actual edit distance is 1. In contrast, using lookahead we can obtain a transducer that can output each character when reading it; this transducer will have aggregate cost 1 for either string. We conjecture that for every transducer T , there always exists an equivalent transducer with regular lookahead (T', R) for which the edit distance computation for aggregate cost coincides with the actual edit distance of T .

C. Transducer Repair

In this section, we show how our synthesis technique can also be used to “repair” buggy transducers. The key idea is to use the closure properties of automata and transducers—e.g., closure under union and sequential compositions [8]—to reduce repair problems to synthesis ones. The ability to algebraically manipulate transducers and automata is one of the key aspects that distinguishes our work from other synthesis works that use domain-specific languages [1], [5].

We describe two settings in which we can repair an incorrect transducer T_{bad} . **1.** Let $\{P\}T_{bad}\{Q\}$ be an input-output type violated by T_{bad} and let $Out_P(T_{bad})$ be the finite automaton describing the set of strings T_{bad} can output when fed inputs in P (this is computable thanks to closure properties of transducers). We are interested in the case where $Out_P(T_{bad}) \setminus Q \neq \emptyset$ —i.e., T_{bad} can produce strings that are not in the output type.

2. Let $\overline{[s \mapsto t]}$ be a set of input-output examples. We are interested in the case where there is some example $s \mapsto t$ such that $T_{bad}(s) \neq t$.

Repairing from the Input Language. This approach synthesizes a new transducer for the inputs on which T_{bad} is incorrect. Using properties of transducers, we can compute an automaton describing the exact set of inputs $P_{bad} \subseteq P$ for which T_{bad} does not produce an output in Q (see pre-image computation in [10]). Let $restrict(T, L)$ be the transducer that behaves as T if the input is in L and does not produce an output otherwise (closure under restriction [10]). If we synthesize a transducer T_1 with type $\{P_{bad}\}T_1\{Q\}$, then the

transducer $restrict(T_1, P_{bad}) \cup restrict(T_{bad}, P \setminus P_{bad})$ satisfies the desired input-output type (closure under union).

Fault Localization from Examples. We use this technique when T_{bad} is incorrect on an example. We can compute a set of “suspicious” transitions by taking all the transitions traversed when $T(s) \neq t$ for some $s \mapsto t \in E$ (i.e., one of these transitions is wrong) and removing all the transitions traversed when $T(s) = t$ for some $s \mapsto t \in E$ (i.e., transitions that are likely correct). Essentially, this is a way of identifying P_{bad} when T_{bad} is wrong on some examples. We can also use this technique to limit the transitions we need to synthesize when performing repair.

V. EVALUATION

We implemented our technique in a Java tool ASTRA (Automatic Synthesis of TRANsducers), which uses Z3 [17] to solve the generated constraints. We evaluate using a 2.7 GHz Intel Core i5, RAM 8 GB, with a 300s timeout.

Q1: Can ASTRA synthesize practical transformations?

Benchmarks. Our first set of benchmarks is obtained from Optician [5], [6], a tool for synthesizing lenses, which are bidirectional programs used for keeping files in different data formats synchronized. We adapted 11 of these benchmarks to work with ASTRA (note that we only synthesize one-directional transformations), and added one additional benchmark `extrAcronym2`, which is a harder variation (with a larger input type) of `extrAcronym`. We excluded benchmarks that require some memory, e.g., swapping words in a sentence, as they cannot be modeled with transducers. Our second set of benchmarks (Miscellaneous) consists of 6 problems we created based on file transformation tasks (`unixToDos`, `dosToUnix` and `CSVSeparator`), and s-FTs from the literature—`escapeQuotes` from [18], `getTags` and `quicktimeMerger` from [16]. All of the benchmarks require synthesizing s-FTs and `getTags` requires synthesizing an s-FT with lookahead (details in Table I).

To generate the examples, we started with the examples that were used in the original source when available. In 5 cases, ASTRA synthesized a transducer that was not equivalent to the one synthesized by Optician. In these cases, we used ASTRA to synthesize two different transducers that met the specification, computed a string on which the two transducers differed, and added the desired output for that string as an example. We repeated this task until ASTRA yielded the desired transducer and we report the time for such sets of examples. The ability to check equivalence of two transducers is yet another reason why synthesizing transducers is useful. For each benchmark we chose a mean edit distance of 0.5 when the transformation could be synthesized with this distance and of 1 otherwise.

Effectiveness of ASTRA. ASTRA can solve 15/18 benchmarks (13 in <1s and 2 under a minute) and times out on 3 benchmarks where both P and Q are big.

While the synthesized transducers have at most 3 states, we note that this is because ASTRA synthesizes total transducers and then restricts their domains to the input type P . This is advantageous because synthesizing small total transducers is

TABLE I: ASTRA’s performance on the synthesis benchmarks. The right-most set of columns gives the synthesis time for ASTRA and Optician (under 2 different configurations). The middle set of columns gives the sizes of the parameters to the synthesis problem: Q_P and Q_Q denote the number of input and output states, and δ_P and δ_Q denote the number of transitions in the input and output types, respectively. A \times represents a benchmark that failed. — stands in for data that is not available; this is because we only re-ran Optician on the benchmarks that were already encoded in its benchmark set, plus a few additional ones for comparing between the tools that we wrote ourselves.

	Benchmark	Q_P	Q_Q	δ_P	δ_Q	Σ	E	k	l	d	ASTRA (s)	Optician (s)	Optician-re (s)
Optician	extrAcronym	6	3	10	3	3	2	1	1	.5	0.11	0.05	\times
	extrAcronym2	6	3	16	3	3	3	2	1	1	0.42	—	—
	extrNum	15	13	17	12	3	1	1	1	1	0.93	0.05	0.07
	extrQuant	4	3	8	5	2	1	2	1	1	0.19	0.09	\times
	normalizeSpaces	7	6	19	10	2	2	2	1	1	0.46	16.64	\times
	extrOdds	15	9	29	13	5	3	3	2	1	15.87	0.12	\times
	capProb	3	3	3	3	2	2	2	1	1	0.05	0.05	\times
	removeLast	6	3	8	3	3	3	2	1	.5	0.21	0.15	0.07
	sourceToViews	18	7	26	15	5	3	3	2	1	50.92	0.06	\times
	normalizeNamePos	19	7	35	24	13	1	6	2	1	\times	0.05	0.10
	titleConverter	22	13	41	41	15	1	3	1	1	\times	0.07	\times
bibtexToReadable	14	11	41	35	12	1	5	1	1	\times	0.64	0.15	
Miscellaneous	unixToDos	5	7	17	19	4	4	2	2	.5	1.24	—	—
	dosToUnix	7	5	19	17	4	4	2	1	.5	0.41	—	—
	CSVSeparator	5	5	9	9	4	1	1	1	1	0.142	—	—
	escapeQuotes	2	2	6	5	3	5	2	2	1	0.188	\times	\times
	quicktimeMerger	7	3	9	3	2	2	1	1	.5	0.075	—	—
	getTags	3	3	9	4	3	5	2	2	1	0.95	\times	\times

easier than synthesizing transducers that require more states to define the domain. For instance, when we restrict the solution of extrAcronym2 to its input type, the resulting transducer has 11 states instead of the 2 required by the original solution!

Comparison with Optician. We do not compare ASTRA to tools that only support input-output examples. Instead, we compare ASTRA to Optician on the set of benchmarks common to both tools. Like ASTRA, Optician supports input-output examples and types, but the types are expressed as regular expressions. Furthermore, Optician also attempts to produce a program that minimizes a fixed information theoretical distance between the input and output types [5].

Optician is faster when the number of variables in the constraint encoding increases, while ASTRA is faster on the normalizeSpaces benchmark. Optician, which uses regular expressions to express the input and output types, does not work so well with unstructured data. To confirm this trend, we wrote synthesis tasks for the escapeQuotes and getTags benchmarks in Optician and it was unable to synthesize those—e.g., escapeQuotes requires replacing every " character with \ ". To further look at the reliance of Optician on regular expressions, we converted the regular expressions used in the lens synthesis benchmarks to automata and then back to regular expressions using a variant of the state elimination algorithm that acts on character intervals. This results in regular expressions that are not very concise and might have redundancies. Optician could only solve 4/11 benchmarks that it was previously synthesizing (Optician-re in Table I).

Answer to Q1: ASTRA can solve real-world benchmarks and has performance comparable to that of Optician for similar tasks. Unlike Optician, ASTRA does not suffer from variations in how the input and output types are specified.

Q2: Can ASTRA repair transducers in practice?

Benchmarks. We considered the benchmarks in Table II. The only pre-existing benchmark that we found was escapeQuotes, through the interface of the Bek programming language used for verifying transducers [18]. We generated 11 additional faulty transducers to repair in the following two ways: (i) Introducing faults in our synthesis benchmarks: We either replaced the output string of a transition with a constant character, inserted an extra character, or deleted a transition altogether. (ii) Incorrect transducers: We intentionally provided fewer input-output examples and used only example-based constraints on some of our synthesis benchmarks.

All the benchmarks involve s-FTs. Three benchmarks are wrong on input-output types and examples, and the rest are only wrong on examples. Additionally, we note that to repair a transducer, we need the “right” set of minterms. Typically, the set of minterms extracted from the transducer predicates is the right one, but in the case of the escapeBrackets problems, ASTRA needs a set of custom minterms we provide manually. We are not aware of another tool that solves transducer repair problems and so do not show any comparisons.

Effectiveness of ASTRA. We indicate the number of suspicious transitions identified by our fault localization procedure (Section IV-C) in the column labeled $\delta_{T_{bad}}$. In many cases, ASTRA can detect 50% of the transitions or more as being likely correct, therefore reducing the space of unknowns.

We compare 2 different ways of solving repair problems in ASTRA. One uses the repair-from-input approach described in Section IV-C (Default in Table II). The second approach involves using a ‘template’, where we supply the constraint solver with a partial solution to the synthesis problem, based on the transitions that were localized as potentially buggy

TABLE II: ASTRA’s performance on the repair benchmarks. Default is the case where a new transducer is synthesized for P_{bad} and Template is the case where a partial solution to the solver is provided. The $\delta_{T_{bad}}$ column gives the number of transitions that were localized by the fault-localization procedure as a fraction of the total number of transitions in the transducer. The other columns that describe the parameters of the synthesis problem in the default case are the same as for Table I.

	Benchmark	Q_P	Q_Q	δ_P	δ_Q	Σ	E	k	l	d	$\delta_{T_{bad}}$	Default (s)	Template (s)
Fault injected	swapCase1	2	1	6	3	3	2	1	1	1	3/3	0.04	0.02
	swapCase2	2	1	4	3	3	2	1	1	1	1/2	✗	✗
	swapCase3	2	1	6	3	3	2	1	1	1	1/3	0.06	0.05
	escapeBrackets1	2	6	16	36	8	4	1	4	4	1/3	0.69	0.42
	escapeBrackets2	1	6	1	7	6	5	1	4	4	1/2	✗	✗
	escapeBrackets3	2	7	8	36	9	5	1	4	4	2/3	1.12	0.34
	caesarCipher	2	1	4	2	3	1	1	1	1	1/1	✗	✗
Synth.	extrAcronym2	11	3	30	3	3	3	2	1	1	12/30	0.59	10.15
	capProb	3	3	3	3	2	2	2	1	1	3/3	0.04	0.04
	extrQuant	8	3	16	5	2	1	2	1	1	5/10	0.37	0.51
	removeLast	6	3	8	3	3	2	2	1	.5	7/8	0.40	1.08
	escapeQuotes	3	2	9	5	3	5	2	1	1	3/5	0.17	0.10

(Template in Table II).

ASTRA can solve 9/12 repair benchmarks (all in less than 1 second). The times using either approach are comparable in most cases. While one might expect templates to be faster, this is not always the case because the input-output specification for the repair transducer is small, but providing a template requires actually providing a partial solution, which in some cases happens to involve many constraints.

Answer to Q2: ASTRA can repair transducers with varying types of bugs.

VI. RELATED WORK

Synthesis of string transformations. String transformations are one of the main targets of program synthesis. Gulwani showed they could be synthesized from input-output examples [1] and introduced the idea of using a DSL to aid synthesis. Optician extended the DSL-based idea to synthesizing lenses [5], [6], which are programs that transform between two formats. Optician supports not only examples but also input-output types. While DSL-based approaches provide good performance, they are also monolithic as they rely on the structure of the DSL to search efficiently. ASTRA does not rely on a DSL and can synthesize string transformations from complex specifications that cannot be handled by DSL-based tools. Moreover, transducers allow applying verification techniques to the synthesized programs (e.g., checking whether two solutions are equivalent). One limitation of transducers is that they do not have ‘memory’, and consequently ASTRA cannot be used for data-transformation tasks where this is required—e.g., mapping the string `Firstname Lastname` to `Lastname, Firstname`—something Optician can do. We remark that there exist transducer models with such capabilities [19] and our work lays the foundations to handle complex models in the future.

Synthesis of transducers. Benedikt et al. studied the ‘bounded repair problem’, where the goal is to determine whether there exists a transducer that maps strings from an input to an output type using a bounded number of edits [13]. Their

work was the first to identify the relation between solving such a problem and solving games, an idea we leverage in this paper. However, their work is not implemented, cannot handle input-output examples, and therefore shies away from the source of NP-Completeness. Hamza et al. studied the problem of synthesizing minimal non-deterministic Mealy machines (transducers where every transition outputs exactly one character), from examples [12]. They prove that the problem of synthesizing such transducers is NP-complete and provide an algorithm for computing minimal Mealy machines that are consistent with the input-output examples. ASTRA is a more general framework that incorporates new specification mechanisms, e.g., input-output types and distances, and uses them all together. Mealy machines are also synthesized from temporal specifications in reactive synthesis and regular model checking, where they are used to represent parameterized systems [20], [21]. This setting is orthogonal to ours as the specification is different and the transducer is again only a Mealy machine.

The constraint encoding used in ASTRA is inspired by the encoding presented by Daniel Neider for computing minimal separating DFA, i.e. a DFA that separates two disjoint regular languages [22]. ASTRA’s use of weights and energy to specify a mean edit distance is based on energy games [23], a kind of 2-player infinite game that captures the need for a player to not exceed some available resource. One way of solving such games is by defining a *progress measure* [15]. To determine whether a game has a winning strategy for one of the players, it can be checked whether such a progress measure exists in the game. We showed that the search for such a progress measure can be encoded as an SMT problem.

VII. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under grants 1763871, 1750965, 1918211, and 2023222, Facebook and a Microsoft Research Faculty Fellowship.

REFERENCES

- [1] S. Gulwani, “Automating string processing in spreadsheets using input-output examples,” in *PoPL’11, January 26-28, 2011, Austin, Texas, USA*, January 2011.
- [2] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, “Fast and precise sanitizer analysis with bek,” in *USENIX Security Symposium*, vol. 58. USENIX, 2012.
- [3] L. D’Antoni and M. Veanes, “Static analysis of string encoders and decoders,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2013, pp. 209–228.
- [4] Y. Zhang, A. Albarghouthi, and L. D’Antoni, “Robustness to programmable string transformations via augmented abstract training,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 11 023–11 032.
- [5] A. Miltner, S. Maina, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic, “Synthesizing symmetric lenses,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–28, 2019.
- [6] A. Miltner, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic, “Synthesizing bijective lenses,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2017.
- [7] M. Mohri, “Finite-state transducers in language and speech processing,” *Computational linguistics*, vol. 23, no. 2, pp. 269–311, 1997.
- [8] L. D’Antoni and M. Veanes, “Automata modulo theories,” *Communications of the ACM*, vol. 64, no. 5, pp. 86–95, 2021.
- [9] T. Chen, M. Hague, J. He, D. Hu, A. W. Lin, P. Rümmer, and Z. Wu, “A decision procedure for path feasibility of string manipulating programs with integer data type,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 325–342.
- [10] L. D’Antoni and M. Veanes, “The power of symbolic automata and transducers,” in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 47–67.
- [11] A. Grover, R. Ehlers, and L. D’Antoni, “Synthesizing transducers from complex specifications,” 2022. [Online]. Available: <https://arxiv.org/abs/2208.05131>
- [12] J. Hamza and V. Kunčák, “Minimal synthesis of string to string functions from examples,” in *Verification, Model Checking, and Abstract Interpretation*, C. Enea and R. Piskac, Eds. Cham: Springer International Publishing, 2019, pp. 48–69.
- [13] M. Benedikt, G. Puppis, and C. Riveros, “Regular repair of specifications,” in *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, 2011, pp. 335–344.
- [14] R. Bloem, K. Chatterjee, and B. Jobstmann, “Graph games and reactive synthesis,” in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 921–962. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_27
- [15] L. Brim, J. Chaloupka, L. Doyen, R. Gentilini, and J.-F. Raskin, “Faster algorithms for mean-payoff games,” *Formal methods in system design*, vol. 38, no. 2, pp. 97–118, 2011.
- [16] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Björner, “Symbolic finite state transducers: Algorithms and applications,” in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’12. New York, NY, USA: Association for Computing Machinery, 2012, p. 137–150. [Online]. Available: <https://doi.org/10.1145/2103656.2103674>
- [17] L. De Moura and N. Björner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [18] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, “Fast and precise sanitizer analysis with bek,” <http://rise4fun.com/Bek/>, 2012.
- [19] R. Alur, “Streaming string transducers,” in *Logic, Language, Information and Computation*, L. D. Beklemishev and R. de Queiroz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–1.
- [20] O. Markgraf, C.-D. Hong, A. W. Lin, M. Najib, and D. Neider, “Parameterized synthesis with safety properties,” in *Asian Symposium on Programming Languages and Systems*. Springer, 2020, pp. 273–292.
- [21] A. W. Lin and P. Rümmer, “Liveness of randomised parameterised systems under arbitrary schedulers,” in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 112–133.
- [22] D. Neider, “Computing minimal separating dfas and regular invariants using sat and smt solvers,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2012, pp. 354–369.
- [23] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga, “Resource interfaces,” in *Embedded Software, Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003, Proceedings*, 2003, pp. 117–133. [Online]. Available: https://doi.org/10.1007/978-3-540-45212-6_9

Synthesis of Semantic Actions in Attribute Grammars

Pankaj Kumar Kalita 
 Computer Science and Engineering
 Indian Institute of Technology Kanpur
 Kanpur, India
 pkalita@cse.iitk.ac.in

Miriyala Jeevan Kumar
 Fortanix Tech. India Pvt. Ltd.
 Bengaluru, India
 g1.miriyala@gmail.com

Subhajit Roy 
 Computer Science and Engineering
 Indian Institute of Technology Kanpur
 Kanpur, India
 subhajit@iitk.ac.in

Abstract—Attribute grammars allow the association of semantic actions to the production rules in context-free grammars, providing a simple yet effective formalism to define the semantics of a language. However, drafting the semantic actions can be tricky and a large drain on developer time. In this work, we propose a synthesis methodology to automatically infer the semantic actions from a set of examples associating strings to their meanings. We also propose a new coverage metric, *derivation coverage*. We use it to build a sampler to effectively and automatically draw strings to drive the synthesis engine. We build our ideas into our tool, PĀṆINI, and empirically evaluate it on twelve benchmarks, including a forward differentiation engine, an interpreter over a subset of Java bytecode, and a mini-compiler for C language to two-address code. Our results show that PĀṆINI scales well with the number of actions to be synthesized and the size of the context-free grammar, significantly outperforming simple baselines.

Index Terms—Program synthesis, Attribute grammar, Semantic actions, Syntax directed definition

I. INTRODUCTION

Attribute grammars [1] provide an effective formalism to supplement a language syntax (in the form of a context-free grammar) with semantic information. The semantics of the language is described using *semantic actions* associated with the grammar productions. The semantic actions are defined in terms of *semantic attributes* associated with the non-terminal symbols in the grammar.

Almost no modern applications use hand-written parsers anymore; instead, most language interpretation engines today use automatic parser generators (like YACC [2], BISON [3], ANTLR [4] etc.). These parser generators employ the simple, yet powerful formalism of attribute grammars to couple parsing with semantic analysis to build an efficient frontend for language understanding. This mechanism drives many applications like model checkers (eg. SPIN [5]), automatic theorem provers (eg. Q3B [6], CVC5 [7]), compilers (eg. CIL [8]), database engines (eg. MYSQL [9]) etc.

However, defining appropriate semantic actions is often not easy: they are tricky to express in terms of the inherited and synthesized attributes over the grammar symbols in the respective productions. Drafting these actions for large grammars requires a significant investment of developer time.

In this work, we propose an algorithm to automatically synthesize semantic actions from *sketches* of attribute grammars.

$S \mapsto E$ ^[1] $E \mapsto E + F$ ^[2] $E - F$ ^[3] F ^[4] $F \mapsto F * K$ ^[5] K ^[6] $K \mapsto K \wedge \text{num}$ ^[7] $\text{SIN}(K)$ ^[8] $\text{COS}(K)$ ^[9] num ^[10] var ^[11]	<div style="background-color: #f8d7da; padding: 2px;">output(E.val);</div> <div style="background-color: #fff3cd; padding: 2px;">$E.\text{val} \leftarrow h_1^*(E.\text{val}, F.\text{val});$</div> <div style="background-color: #fff3cd; padding: 2px;">$E.\text{val} \leftarrow h_2^*(E.\text{val}, F.\text{val});$</div> <div style="background-color: #fff3cd; padding: 2px;">$E.\text{val} \leftarrow F.\text{val};$</div> <div style="background-color: #fff3cd; padding: 2px;">$F.\text{val} \leftarrow h_3^*(F.\text{val}, K.\text{val});$</div> <div style="background-color: #fff3cd; padding: 2px;">$F.\text{val} \leftarrow K.\text{val};$</div> <div style="background-color: #fff3cd; padding: 2px;">$K.\text{val} \leftarrow h_4^*(K.\text{val}, \text{num});$</div> <div style="background-color: #fff3cd; padding: 2px;">$K.\text{val} \leftarrow h_5^*(K.\text{val});$</div> <div style="background-color: #fff3cd; padding: 2px;">$K.\text{val} \leftarrow h_6^*(K.\text{val});$</div> <div style="background-color: #fff3cd; padding: 2px;">$K.\text{val} \leftarrow \text{getVal}(\text{num}) + 0\epsilon;$</div> <div style="background-color: #fff3cd; padding: 2px;">$K.\text{val} \leftarrow \text{lookUp}(\Omega, \text{var}) + 1\epsilon;$</div>
--	--

Fig. 1: Attribute grammar for automatic forward differentiation (Ω is the symbol table)

Fig. 1 shows a sketch of an attribute grammar for automatic forward differentiation using dual numbers (we explain the notion of dual numbers and the example in detail in §III-A). The production rules are shown in green color while the semantic actions are shown in the blue color. Our synthesizer attempts to infer the definitions of the holes in this sketch (the function calls h_1^* , h_2^* , h_3^* , h_4^* , h_5^* , h_6^*); we show these holes in yellow background. As an attribute grammar attempts to assign “meanings” to language strings, the meaning of a string in this language is captured by the output construct.

This is a novel synthesis task: the current program synthesis tools synthesize a program such that a desired specification is met. In our present problem, we attempt to synthesize semantic actions within an attribute grammar: the synthesizer is required to *infer* definitions of the holes such that for *all* strings in the language described by the grammar, the computed semantic value (captured by the output construct) matches the intended semantics of the respective string—this is a new problem that cannot be trivially mapped to a program synthesis task.

Our core observation to solve this problem is the follows: for any string in the language, the sequence of semantic actions executed for the syntax-directed evaluation of any string is a loop-free program. This observation allows us to reduce attribute grammar synthesis to a *set program synthesis tasks*. Unlike a regular program synthesis task where we are interested in synthesizing a single program, the above reduction requires us to solve a set of *dependent* program

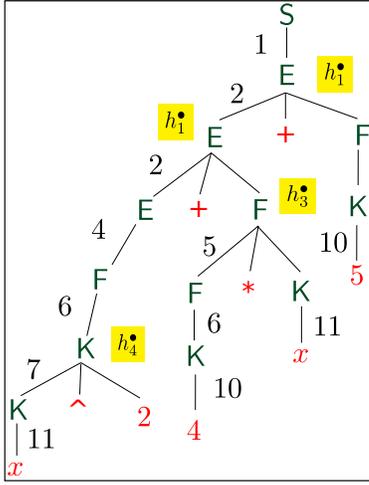


Fig. 2: Parse tree for input $x^2+4*x+5$

synthesis instances simultaneously. These program synthesis tasks are dependent as they contain common components (to be synthesized) shared by multiple programs, and hence, they cannot be solved in isolation—as the synthesis solution from one instance can influence others.

Given a set of examples E (strings that can be derived in the provided grammar) and their expected semantics O (semantic values in `output`); for each example $e_i \in E$, we apply the reduction by sequencing the set of semantic actions for the productions that occur in the derivation of e_i . This sequence of actions forms a loop-free program P_i , with the expected semantic output $o_i \in O$ as the specification. We collect such programs-specification pairs, $\langle P_i, o_i \rangle$, to create a set of dependent synthesis tasks. An attempt at *simultaneous synthesis* of all these set of tasks by simply conjoining the synthesis constraints does not scale.

Our algorithm adopts an incremental, counterexample guided inductive synthesis (CEGIS) strategy, that attempts to handle only a “small” set of programs simultaneously—those that violate the current set of examples. Starting with only a single example, the set of satisfied examples are expanded incrementally till the specifications are satisfied over all the programs in the set.

Furthermore, to relieve the developer from providing examples, we also propose an example generation strategy for attribute-grammars based on a new coverage metric. Our coverage metric, *derivation coverage*, attempts to capture distinct behaviors due to the presence or absence of each of the semantic actions corresponding to the syntax-directed evaluation of different strings.

We build an implementation, PĀṆINI¹, that is capable of automatically synthesizing semantic actions (across both synthesized and inherited attributes) in attribute grammars. For the attribute grammar sketch in Fig. 1, PĀṆINI automatically synthesizes the definitions of holes as shown in Fig. 3 in a mere 39.2 seconds. We evaluate our algorithm on a set of attribute grammars, including a Java bytecode interpreter and

¹PĀṆINI (पानिनि) was a Sanskrit grammarian and scholar in ancient India.

a mini-compiler frontend. Our synthesizer takes a few seconds on these examples.

To the best of our knowledge, ours is the first work on automatic synthesis of semantic actions on attribute grammars. The following are our contributions in this work:

- We propose a new algorithm for synthesizing semantic actions in attribute grammars;
- We define a new coverage metric, *derivation coverage*, to generate effective examples for this synthesis task;
- We build our algorithms into an implementation, PĀṆINI, to synthesize semantic actions for attribute grammars;
- We evaluate PĀṆINI on a set of attribute grammars to demonstrate the efficacy of our algorithm. We also undertake a case-study on the attribute grammar of the parser of the SPIN model-checker to automatically infer the constant-folding optimization and abstract syntax tree construction.

An extended version of this article is available [10]. The implementation and benchmarks of PĀṆINI are available at <https://github.com/pkalita595/Panini>.

II. PRELIMINARIES

Attribute grammars [1] provide a formal mechanism to capture language semantics by extending a context-free grammar with *attributes*. An attribute grammar \mathcal{G} is specified by $\langle S, P, T, N, F, \Gamma \rangle$, where

- T and N are the set of *terminal* and *non-terminal* symbols (resp.), and $S \in N$ is the *start symbol*;
- A set of (context-free) productions, $p_i \in P$, where $p_i : X_i \mapsto Y_{i1}Y_{i2} \dots Y_{in}$; a production consists of a *head* $X_i \in N$ and *body* $Y_{i1} \dots Y_{in}$, such that each $Y_{ik} \in T \cup N$.
- A set of *semantic actions* $f_i \in F$;
- $\Gamma : P \rightarrow F$ is a map from the set of productions P to the set of *semantic actions* $f_i \in F$.

The set of productions in \mathcal{G} describes a *language* (denoted as $\mathcal{L}(\mathcal{G})$) to capture the set of *strings* that can be *derived* from S . A *derivation* is a sequence of applications of productions $p_i \in P$ that transforms S to a string, $w \in \mathcal{L}(\mathcal{G})$; unless specified, we will refer to the *leftmost* derivation where we always select the leftmost non-terminal for expansion in a sentential form. As we are only concerned with parseable grammars, we constrain our discussion in this paper to *unambiguous* grammars.

The semantic actions associated with the grammar productions are defined in terms of *semantic attributes* attached to the non-terminal symbols in the grammar. Attributes can be *synthesized* or *inherited*: while a synthesized attributes are computed from the children of a node in a parse tree, an inherited attribute is defined by the attributes of the parents or siblings.

Fig. 1 shows an attribute grammar with *context-free productions* and the associated *semantic actions*. Fig. 2 shows the parse tree of the string $x^2 + 4x + 5$ on the provided grammar; each internal node of the parse tree have associated semantic actions (we have only shown the “unknown” actions that need to be inferred).

$$\begin{array}{lll}
h_1^\bullet (a_1 + a_2\varepsilon, b_1 + b_2\varepsilon): & h_2^\bullet (a_1 + a_2\varepsilon, b_1 + b_2\varepsilon): & h_3^\bullet (a_1 + a_2\varepsilon, b_1 + b_2\varepsilon): \\
r \leftarrow a_1 + b_1 & r \leftarrow a_1 - b_1 & r \leftarrow a_1 * b_1 \\
d \leftarrow a_2 + b_2 & d \leftarrow a_2 - b_2 & d \leftarrow a_2 * b_1 + a_1 * b_2 \\
\mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon \\
\text{(a)} & \text{(b)} & \text{(c)} \\
h_5^\bullet (a_1 + a_2\varepsilon): & h_6^\bullet (a_1 + a_2\varepsilon): & h_4^\bullet (a_1 + a_2\varepsilon, c): \\
r \leftarrow \sin(a_1) & r \leftarrow \cos(a_1) & r \leftarrow \text{pow}(a_1, c) \\
d \leftarrow a_2 * \cos(a_1) & d \leftarrow a_2 * \sin(a_1) * -1 & d \leftarrow a_2 * \text{pow}(a_1, c - 1) \\
\mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon \\
\text{(d)} & \text{(e)} & \text{(f)}
\end{array}$$

Fig. 3: Synthesized holes for holes in Fig. 1

Parser generators [2] accept an attribute grammar and automatically generate parsers that perform a *syntax-directed evaluation* of the semantic actions. For ease of discussion, we assume that the semantic actions are *pure* (i.e. do not cause side-effects like printing values or modifying global variables) and generate a deterministic *output value* as a consequence of applying the actions.

An attribute grammar is *non-circular* if the dependencies between the attributes in every syntax tree are acyclic. Non-circularity is a sufficient condition that all strings have unique evaluations [11].

Notations. We notate production symbols by serif fonts, non-terminal symbols (or placeholders) by capital letters (eg. X) and terminal symbols by small letters (eg. a). Sets are denoted in capital letters. We use arrows with tails (\rightarrow) in productions and string derivations to distinguish it from function maps. We use the notation $e[g_1/g_2]$ to imply that all instances of the subexpression g_2 are to be substituted by g_1 within the expression e . We use the notation of Hoare logic [12] to capture program semantics: $\{P\}Q\{R\}$ implies that if the program Q is executed with a *precondition* P , it can only produce an output state in R ; P and R are expressed in some base logic (like first-order logic).

III. OVERVIEW

Sketch of an attribute grammar. We allow the *sketch* \mathcal{G}^\bullet of an attribute grammar (as syntax directed definition (SDD)), $\mathcal{G}^\bullet = \langle S, P, T, N, H^\bullet, \Gamma \rangle$, to contain *holes* for unspecified functionality within the semantic actions $h_i^\bullet \in H^\bullet$. For example, in Fig. 1, the set of holes comprises of the functions $H = \{h_1^\bullet, h_2^\bullet, h_3^\bullet, h_4^\bullet, h_5^\bullet, h_6^\bullet\}$. If the semantic action corresponding to a production p contains hole(s), we refer to the production p as a *sketchy* production; when the definitions for all the holes in a sketchy production are resolved, we say that the production is *ready*. The *completion* (denoted $\mathcal{G}^{\{f_1, \dots, f_n\}}$) of a grammar sketch \mathcal{G}^\bullet denotes the attribute grammar where a set of functions f_1, \dots, f_n replace the holes $h_1^\bullet, \dots, h_n^\bullet$. We denote the syntax-directed evaluation of a string w on an attribute grammar \mathcal{G} as $\llbracket w \rrbracket^{\mathcal{G}}$; we consider that any such evaluation results in a *value* (or \perp if $w \notin \mathcal{G}$).

Example Suite. An *example* (or *test*) for an attribute grammar \mathcal{G} can be captured by a tuple $\langle w, v \rangle$ such that $w \in \mathcal{L}(\mathcal{G})$ and

$\llbracket w \rrbracket^{\mathcal{G}} = v$. A set of such examples constitutes an *example suite* (or *test suite*).

If the language described by the grammar \mathcal{G} supports *variables*, then any evaluation of \mathcal{G} needs a *context*, β , that binds the free variables to *input values*. We denote such examples as $\llbracket w \rrbracket_\beta^{\mathcal{G}} = v$. When the grammar used is clear from the context, we drop the superscript and simplify the notation to $\llbracket w \rrbracket_\beta = v$. Consider the example $\llbracket [x^3]_{x=2} = 8 + 12\varepsilon \rrbracket$, where “ x^3 ” is a string from the grammar shown in Fig. 1 and the input string evaluates to $8 + 12\varepsilon$ under the binding $x = 2$. Clearly, if the language does not support variables, the context β is always empty.

Problem Statement. Given a sketch of an attribute grammar, \mathcal{G}^\bullet , an example set E and a domain-specific language (DSL) D , synthesize instantiations of the holes by strings, $w \in D$, such that the resulting attribute grammar agrees with all examples in E .

In other words, PĀṆINI synthesizes functions f_1, \dots, f_n in the domain-specific language D such that the completion $\mathcal{G}^{\{f_1, \dots, f_n\}}$ satisfies all examples in E .

A. Motivating example: Automated Synthesis of a Forward Differentiation Engine

We will use synthesis of an automatic forward differentiation engine using dual numbers [13] as our motivating example. We start with a short tutorial on how dual numbers are used for forward differentiation.

1) *Forward Differentiation using Dual numbers:* Dual numbers, written as $a + b\varepsilon$, captures both the value of a function $f(x)$ (in the *real* part, a), and that its differentiation with respect to the variable x , $f'(x)$, (in the *dual* part, b)—within the same number. Clearly, $a, b \in \mathbb{R}$ and we assume $\varepsilon^2 = 0$ (as it refers to the second-order differential, that we are not interested to track). The reader may draw parallels to complex numbers that are written as $a + ib$, where ‘ i ’ identifies the imaginary part, and $i^2 = -1$.

Let us understand forward differentiation by calculating $f'(x)$ at $x = 3$ for the function $f(x) = x^2 + 4x + 5$.

First, the term x needs to be converted to a dual number at $x = 3$. For $x = 3$, the real part is clearly 3. To find the dual

part, we differentiate the term with respect to variable x , i.e. $\frac{dx}{dx}$ that evaluates to 1. Hence, the dual number representation of the term x at $x = 3$ is $3 + 1\varepsilon$.

Now, dual value of the term x^2 can be computed simply by taking a square of the dual representation of x :

$$\overbrace{(3 + 1\varepsilon)^2}^{x^2} = \overbrace{(3 + 1\varepsilon)}^x * \overbrace{(3 + 1\varepsilon)}^x = 3^2 + (2*3*\varepsilon) + \varepsilon^2 = 9 + 6\varepsilon + 0$$

Finally, the dual number representation for the constant 4 is $4 + 0\varepsilon$ (as differentiation of constant is 0). Similarly, the dual value for $4x$: $(4 + 0\varepsilon) * (3 + 1\varepsilon) = 12 + 4\varepsilon + 0$. So, we can compute the dual number for $f(x) = x^2 + 4x + 5$ as:

$$\overbrace{(9 + 6\varepsilon)}^{x^2} + \overbrace{(12 + 4\varepsilon)}^{4x} + \overbrace{(5 + 0\varepsilon)}^5 = 26 + 10\varepsilon$$

Hence, the value of $f(x)$ at $x = 3$ is $f(3) = 26$ (real part of the dual number above) and that of its derivative, $f'(x) = 2x + 4$ is $f'(3) = 10$, which is indeed given by the the dual part for the dual number above.

2) *Synthesizing a forward differentiation engine*: The attribute grammar in Fig. 1 (adapted from [14]) implements forward differentiation for expressions in the associated context-free grammar; we will use this attribute grammar to illustrate our synthesis algorithm. $\text{lookUp}(\Omega, \text{var})$ returns the value of the variable var from symbol table Ω .

We synthesize programs for the required functionalities for the holes from the domain-specific language (DSL) shown in Equation 1. Function $\text{pow}(a, c)$ calculates a raised to the power of c . We assume the availability of an input-output oracle, $\text{Oracle}(w(\beta))$, that returns the expected semantic value for string w under the context β .

$$\begin{aligned} \text{Fun} &::= C + C\varepsilon \\ C &::= \text{var} \mid \text{num} \mid 1 \mid 0 \mid -C \mid C + C \mid C - C \mid C * C \\ &\quad \mid \sin(C) \mid \cos(C) \mid \text{pow}(C, \text{num}) \end{aligned} \quad (1)$$

B. Synthesis of semantic actions

Our core insight towards solving this synthesis problem is that the sequence of semantic actions corresponding to the syntax-directed evaluation of any string on the attribute grammar constitutes a loop-free program.

Fig. 5 shows the loop free program from the semantic evaluation of the example

$\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3} = 26 + 10\varepsilon$; the Hoare triple captures the synthesis constraints over the holes.

Similarly, our algorithm constructs constraints (as Hoare triples) over the set of all examples E (e.g.

$$\begin{aligned} \llbracket x + x \rrbracket_{x=13} = 26 + 2\varepsilon, \quad \llbracket 3 - x \rrbracket_{x=7} = -4 - 1\varepsilon, \\ \llbracket x * x \rrbracket_{x=4} = 16 + 8\varepsilon, \\ \llbracket \sin(x^2) \rrbracket_{x=3} = 0.41 - 5.47\varepsilon, \end{aligned}$$

$$\begin{aligned} \llbracket \cos(x^2) \rrbracket_{x=2} = -0.65 + 3.02\varepsilon, \\ \llbracket x * \cos(x) \rrbracket_{x=4} = -2.61 + 2.37\varepsilon. \end{aligned}$$

Synthesizing definitions for holes that satisfy Hoare triple constraints of all the above examples yields a valid completion of the sketch of the attribute grammar (see Fig. 3). As the above queries are “standard” program synthesis queries, they can be answered by off-the-shelf program synthesis tools [15], [16]. Hence, our algorithm reduces the problem of synthesizing semantic actions for attribute grammars to solving a conjunction of program synthesis problems.

While the above conjunction can be easily folded into a single program synthesis query and offloaded to a program synthesis tool, quite understandably, it will not scale. To scale the above problem, we employ a refutation-guided inductive synthesis procedure: we sort the set of examples by increasing complexity, completing the holes for the easier instances first. The synthesized definitions are

```

{x = 3}
K1.val ← 3 + 1ε;
K2.val ← h4*(K1.val, 2);
K3.val ← 4 + 0ε;
F1.val ← h3*(K3.val, K1.val);
E1.val ← h1*(K2.val, F1.val);
K4.val ← 5 + 0ε;
output ← h1*(E1.val, K4.val);
{output = 26 + 10ε}

```

Fig. 5: Hoare triple constraint for $x^2 + 4x + 5$ at $x = 3$

frozen while handling new examples; however, unsatisfiability of a synthesis call with frozen procedures *refutes* the prior synthesized definitions. Say we need to synthesize definitions for $\{h_0^*, \dots, h_9^*\}$ and examples $\{e_1, \dots, e_{i-1}\}$ have already been handled, with definitions $\{h_1^* = f_1, \dots, h_5^* = f_5\}$ already synthesized. To handle a new example, e_i , we issue a synthesis call for procedures $\{h_6^*, \dots, h_9^*\}$ with definitions $\{h_1^* = f_1, \dots, h_5^* = f_5\}$ frozen. Say, the constraint corresponding to e_i only includes calls $\{h_2^*, h_4^*, h_6^*, h_8^*\}$ and the synthesis query is unsatisfiable. In this case, we unfreeze *only* the participating frozen definitions (i.e. $\{h_2^*, h_4^*\}$) and make a new synthesis query. As new query only attempts to synthesize a few new calls (with many participating calls frozen to previously synthesized definitions), this algorithm scales well.

For example, consider the grammar in Fig. 1: the loop-free program resulting from the semantic evaluation of the input $\llbracket x + x \rrbracket_{x=13} = 26 + 2\varepsilon$ (say trace t_1) includes only one h_1^* . Hence, we synthesize h_1^* with only the constraint $\{x = 13\}t_1\{output = 26 + 2\varepsilon\}$, that results in the definition shown in Fig. 3a. Next, we consider the input $\llbracket x * x \rrbracket_{x=4} = 16 + 8\varepsilon$; its constraint includes the holes h_3^* , which is synthesized as the function definition shown in Fig. 4. Now, with $\{h_1^*, h_3^*\}$ frozen to their respective synthesized definitions, we attempt to handle $\llbracket x * \cos(x) \rrbracket_{x=4} = -2.61 + 2.37\varepsilon$. Its constraint includes the holes $\{h_3^*, h_6^*\}$; now we *only* attempt to synthesize h_6^* while constraining h_3^* to use the definition in Fig. 4.

$$\begin{aligned} h_3^*(a_1 + a_2\varepsilon, b_1 + b_2\varepsilon): \\ r \leftarrow a_1 * b_1 \\ d \leftarrow b_1 + b_2 + 3 * a_2 \\ \mathbf{return} \ r + d\varepsilon \end{aligned}$$

Fig. 4: Wrong definition of h_3^*

In this case the synthesizer fails to synthesize h_6^\bullet since the synthesized definition for h_3^\bullet is incorrect, thereby *refuting* the synthesized definition of h_3^\bullet . Hence, we now unfreeze h_3^\bullet and call the synthesis engine again to synthesize both h_3^\bullet and h_6^\bullet together. This time we succeed in inferring correct synthesized definition as shown in Fig. 3.

C. Example Generation

We propose a new coverage metric, *derivation coverage*, to generate good samples to drive synthesis. Let us explain derivation coverage with an example, $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3}$ from the grammar in Fig. 1. The leftmost derivation of this string covers eight productions, $\{1, 2, 4, 5, 6, 7, 10, 11\}$ out of a total of 11 productions. Intuitively, it implies that the Hoare triple constraint from its semantic evaluation will *test* the semantic actions corresponding to these productions.

Similarly, the Hoare logic constraint from the example $\llbracket x^2 + 7 * x + \sin(x) \rrbracket_{x=2}$ will cover 9 of the productions, $\{1, 2, 4, 5, 6, 7, 8, 10, 11\}$. As it *also* covers the semantic action for the production 8, it tests an additional behavior of the attribute grammar. On the other hand, the example $\llbracket x^3 + 5x \rrbracket_{x=5}$ invokes the productions, $\{1, 2, 4, 5, 6, 7, 10, 11\}$. As all these semantic actions have already been *covered* by the example $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3}$, it does not include the semantic action of any new set of productions.

In summary, derivation coverage attempts to abstract the derivation of a string as the *set of productions in its leftmost derivation*. It provides an effective metric for quantifying the quality of an example suite and also for building an effective example generation system.

Validation. Our example generation strategy can start off by *sampling* strings w from the grammar (that improve derivation coverage), and context β ; next, it can query the oracle for the intended semantic value $v = Oracle(w \langle \beta \rangle)$ to create an example $\llbracket w \rrbracket_\beta^{\mathcal{G}} = v$.

Consider that the algorithm finds automatically an example $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3}$. Now, there are two possible, semantically distinct definitions that satisfy the above constraint (see Fig. 3c and Fig. 4), indicating that the problem is underconstrained. Hence, our system needs to select additional examples to resolve this. One solution is to sample multiple contexts on the same string to create multiple constraints:

- $\{x = 2\} K_1.val \leftarrow 2 + 1\epsilon; \dots \{output = 13 + 6\epsilon\}$
- $\{x = 4\} K_1.val \leftarrow 4 + 1\epsilon; \dots \{output = 29 + 10\epsilon\}$

The above constraints resolve the ambiguity and allows the induction of a semantically unique definitions. The check for semantic uniqueness can be framed as a check for *distinguishing inputs*: given a set of synthesized completion $\mathcal{G}^{\{f_1, \dots, f_n\}}$, we attempt to synthesize an alternate completion $\mathcal{G}^{\{g_1, \dots, g_n\}}$ and an example string w (and context β) such that $\llbracket w \rrbracket_\beta^{\mathcal{G}^{\{f_1, \dots, f_n\}}} \neq \llbracket w \rrbracket_\beta^{\mathcal{G}^{\{g_1, \dots, g_n\}}}$. In other words, for the same string (and context), the attribute grammar returns different evaluations corresponding to the two completions. For example, $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=2}$ is a *distinguishing inputs*

Algorithm 1: SYNTHHOLES($\mathcal{G}^\bullet, T, R, D$)

```

1  $\varphi \leftarrow \top$ ;
2  $\mathcal{G}_1^\bullet \leftarrow \mathcal{G}^\bullet[R]$ ;
3 for  $\langle w, v \rangle \in T$  do
4    $t \leftarrow \text{GENTRACE}(\llbracket w \rrbracket_{\mathcal{G}_1^\bullet})$ ;
5    $\varphi \leftarrow \varphi \wedge (out(t) = v)$ ;
6  $B \leftarrow \text{SYNTHESIZE}(\varphi, D)$ ;
7 return  $B$ ;
```

witnessing the ambiguity between the definitions shown in Fig. 3c and Fig. 4.

On the other hand, the algorithm could have sampled other strings (instead of contexts) for additional constraints. PĀÑINI prefers the latter; that is, it first generates a *good* example suite (in terms of derivation coverage) and only uses distinguishing input as a validation (post) pass. If such inputs are found, additional contexts are added to resolve the ambiguity.

We provide the detailed algorithm of example generation in the extended version [10].

IV. ALGORITHM

Given an attribute grammar \mathcal{G}^\bullet , a set of holes $h_i \in H$, a domain-specific language D , an example suite E and a context β , PĀÑINI attempts to find instantiations g_i for h_i such that,

Find $\{g_1, \dots, g_{|H|}\} \in D$ **such that** $\forall \langle s, \beta, v \rangle \in E. \llbracket s \rrbracket_\beta^{\mathcal{G}} = v$ (2)

where the attribute grammar $\mathcal{G} = \mathcal{G}^\bullet[g_1/h_1, \dots, g_{|H|}/h_{|H|}]$ and variable bindings β maps variables in s to values.

A. Basic Scheme: ALLATONCE

Our core synthesis procedure (Algorithm 1), SYNTHHOLES($\mathcal{G}^\bullet, E, R, D$), accepts a sketch \mathcal{G}^\bullet , an example (or test) suite E , a set of *ready* functions R and a DSL D ; all holes whose definitions are available are referred to as *ready* functions. When SYNTHHOLES is used as a top-level procedure (as in the current case), $R = \emptyset$; if not empty, the definitions of the ready functions are substituted in the sketch \mathcal{G}^\bullet to create a new sketch \mathcal{G}_1^\bullet on the remaining holes (Line 2). We refer to the algorithm where $R = \emptyset$ at initialization as the ALLATONCE algorithm.

Our algorithm exploits the fact that a syntax-directed semantic evaluation of a string w on an attribute grammar \mathcal{G} produces a loop-free program. It attempts to compute a symbolic encoding of this program trace in the formula φ (initialized to true in Line 1). GENTRACE() instruments the semantic evaluation on the string w to collect a symbolic trace (the loop-free program) consisting of the set of instructions encountered during the syntax-directed execution of the attribute grammar (Line 4); an output from an operation that is currently a *hole* is appended as a symbolic variable. The assertion that the expected output v matches the final symbolic output $out(t)$ from the trace t is appended to the list of constraints (Line 5). Finally, we use a program synthesis procedure, *Synthesize* with the constraints φ in an attempt to synthesize suitable function

definitions for the holes in φ (Line 6). Given a constraint in terms of a set of input vector \vec{x} and function symbols (corresponding to holes) \vec{h} ,

$$\text{SYNTHESIZE}(\varphi(\vec{x}, \vec{h})) := \vec{h} \text{ such that } \exists \vec{h}. \forall \vec{x}. \varphi(\vec{x}, \vec{h}) \quad (3)$$

We will use an example from forward differentiation (Fig. 1) to illustrate this. Let us consider two inputs, $\llbracket x^2 - 2 * x \rrbracket_{x=3} = 3 + 4\varepsilon$ and $\llbracket 3 * x + 6 \rrbracket_{x=2} = 12 + 3\varepsilon$. For the first input $\llbracket 3 * x + 6 \rrbracket_{x=2} = 12 + 3\varepsilon$, the procedure `GENTRACE()` (Line 4) generates a symbolic trace (denoted t_1):

$$\{x_1 = 2 + 1\varepsilon; \alpha_1 = h_3^\bullet(3 + 0\varepsilon, x_1); out_{t_1} = h_1^\bullet(\alpha_1, 6 + 0\varepsilon)\}$$

The following symbolic constraint is generated from above trace:

$$\varphi_{t_1} \equiv (x_1 = 2 + 1\varepsilon \wedge \alpha_1 = h_3^\bullet(3 + 0\varepsilon, x_1) \wedge out_{t_1} = h_1^\bullet(\alpha_1, 6))$$

In the trace t_1 , operations h_1^\bullet and h_3^\bullet are holes and variables, i.e., α_1, out_{t_1} are the fresh symbolic variables. In the next step (line 5), the constraints generated from trace t_1 is added,

$$\varphi \equiv \top \wedge (\varphi_{t_1} \wedge out_{t_1} = 12 + 3\varepsilon)$$

In the next iteration of the loop at line 3, the algorithm will take the second input, (i.e., $\llbracket x^2 - 2 * x \rrbracket_{x=3} = 3 + 4\varepsilon$). In this case, `GENTRACE()` will generate following trace (t_2),

$$\{x_2 = 3 + 1\varepsilon; \alpha_2 = h_4^\bullet(x_2, 2); \alpha_3 = h_3^\bullet(2 + 0\varepsilon, x_2); out_{t_2} = h_2^\bullet(\alpha_3, \alpha_4)\}$$

The generated constraints from t_2 will be,

$$\varphi_{t_2} \equiv (x_2 = 3 + 1\varepsilon \wedge \alpha_2 = h_4^\bullet(x_2, 2) \wedge \alpha_3 = h_3^\bullet(2 + 0\varepsilon, x_2) \wedge out_{t_2} = h_2^\bullet(\alpha_3, \alpha_4))$$

At line 5, new constraints will be,

$$\varphi \leftarrow \top \wedge (\varphi_{t_1} \wedge out_{t_1} = 12 + 3\varepsilon) \wedge (\varphi_{t_2} \wedge out_{t_2} = 3 + 4\varepsilon)$$

At line 6, with φ as constraints, the algorithm will attempt to synthesize definitions for the holes (i.e., $h_1^\bullet, h_2^\bullet, h_3^\bullet$ and h_4^\bullet).

B. Incremental Synthesis

The `ALLATONCE` algorithm exhibits poor scalability with respect to the size of the grammar and the number of examples. The route to a scalable algorithm could be to incrementally learn the definitions corresponding to the holes and make use of the functions synthesized in the previous steps to discover new ones in the subsequent steps.

However, driving synthesis one example at a time will lead to overfitting. We handle this complexity with a two-pronged strategy: (1) we partition the set of examples by the holes for which they need to synthesize actions, (2) we solve the synthesis problems by their difficulty (in terms of the number of functions to be synthesized) that allows us to memoize their results for the more challenging examples. We refer to this example as the `INCREMENTALSYNTHESIS` algorithm.

Algorithm 2: SYNTHATTRGRAMMAR($\mathcal{G}^\bullet, E, D$)

```

1  $T \leftarrow \emptyset$ ;
2  $R \leftarrow \emptyset$ ;
3 while  $T \neq E$  do
4    $\langle w, v \rangle \leftarrow \text{SELECTEXAMPLE}(E \setminus T)$ ;
5    $Z \leftarrow \text{GETSKETCHYPRODS}(\mathcal{G}^\bullet, w)$ ;
6   if  $Z \subseteq R$  then
7      $\mathcal{G}_1^\bullet \leftarrow \mathcal{G}^\bullet[R]$ ;
8     if  $\llbracket w \rrbracket_{\mathcal{G}_1^\bullet} = v$  then
9        $T \leftarrow T \cup \{\langle w, v \rangle\}$ ;
10      continue;
11    else
12       $R \leftarrow R \cup Z$ ;
13       $T_e \leftarrow T \cup \{\langle w, v \rangle\}$ ;
14    else
15       $T_e \leftarrow \{\langle w_i, v_i \rangle \mid w \cong w_i, \langle w_i, v_i \rangle \in E\}$ ;
16     $B \leftarrow \text{SYNTHHOLES}(\mathcal{G}^\bullet, T_e, R, D)$ ;
17    if  $B = \emptyset$  then
18      if  $R \cap Z \neq \emptyset$  then
19         $R \leftarrow R \setminus Z$ ;
20         $B \leftarrow \text{SYNTHHOLES}(\mathcal{G}^\bullet, T \cup T_e, R, D)$ ;
21      if  $B = \emptyset$  then
22        return  $\emptyset$ ;
23     $R_f \leftarrow \{(p_i : \{\dots, h_i \rightarrow B[h_i], \dots\}) \mid p_i \in Z \setminus R, h_i \in \text{holes}(\Gamma(p_i))\}$ ;
24     $R \leftarrow R \cup R_f$ ;
25     $T \leftarrow T \cup T_e$ ;
26 return  $R$ ;
```

Derivation Congruence. We define an equivalence relation, *derivation congruence*, on the set of strings $w \in \mathcal{L}(\mathcal{G})$: strings $w_1, w_2 \in \mathcal{L}(\mathcal{G})$ are said to be *derivation congruent*, $w_1 \cong_G w_2$ w.r.t. the grammar G , if and only if both the strings w_1 and w_2 contain the same set of productions in their respective derivations. For example, $w_1 : \llbracket 3 * x + 5 \rrbracket_{x=2}$,

$$w_2 : \llbracket 5 * x + 12 \rrbracket_{x=3} \text{ and } w_3 : \llbracket 4 * x + 7 * x \rrbracket_{x=3}.$$

Note that though the strings w_1, w_2 and w_3 are derivation congruent to each other, while w_1 and w_2 have similar parse trees, w_3 has a quite different parse tree. So, intuitively, all these strings are definition congruent to each other, as, even with different parse trees, they involve the same set of productions ($\{1, 2, 4, 5, 6, 10, 11\}$) in their leftmost derivation.

Algorithm 2 shows our incremental synthesis strategy. Our algorithm maintains a set of examples (or tests) T (line 2) that are consistent with the current set of synthesized functions for the holes; the currently synthesized functions (referred to as *ready* functions), along with the respective ready productions, are recorded in R (line 1). The algorithm starts off by selecting the *easiest* example $\langle w, v \rangle$ at line 4 such that the cardinality of the set of sketchy production, Z , in the derivation of w is the minimal among all examples not in T . The set R maintains a map from the set of sketchy productions to a set of assignments to functions synthesized (instantiations) for each hole contained in the respective semantic actions.

If all sketchy productions, Z , in the derivation of w are now *ready*, we simply *test* (line 6) to check if a syntax-guided evaluation with the currently synthesized functions in

R yield the expected value v : if the test passes, we add the new example to the set of passing examples in T (line 9). Otherwise, as the current hole instantiations in R is not consistent for the derivation of w , at line 12 we remove all the synthesized functions participating in syntax-directed evaluation of w (which is exactly Z). Furthermore, removal of some functions from R requires us to re-assert the new functions on all the past examples (contained in T) in addition to the present example (line 13).

If all the sketchy productions (Z) in the derivation of w are not ready, we attempt to synthesize functions for the *missing* holes, *with the set of current definitions in R provided in the synthesis constraint*.

The synthesis procedure (line 16), if successful, yields a set of function instantiation for the holes. In this case, the solution set from B is accumulated in R , and the set of *passing* tests extended to contain the new examples in T_e .

However, synthesis may fail as some of the current definitions in R that were assumed to be correct and included in the synthesis constraint is not consistent with the new examples (T_e). In this case, we remove the instantiations of all such holes occurring in the syntax-directed evaluation of w (line 19) and re-attempt synthesis (line 20). If this attempt fails too, it implies that no instantiation of the holes exist in the provided domain-specific language D (line 22).

We provide a detailed example on the run of this algorithm in the extended version [10].

Theorem. If the algorithm terminates with a non-empty set of functions, \mathcal{G}^\bullet instantiated with the synthesized functions will satisfy the examples in E ; that is, the synthesized functions satisfy Equation 2.

The proof is a straightforward argument with the inductive invariant that at each iteration of the loop, the \mathcal{G}^\bullet instantiated with the functions in R satisfy the examples in T .

V. EXPERIMENTS

Our experiments were conducted in Intel(R) Xeon(R) CPU E5-2620 @ 2.00GHz with 32 GB RAM and 24 cores on a set of benchmarks shown in Table I. PĀÑINI uses FLEX [17] and BISON [3] for performing a syntax-directed semantic evaluation over the language strings. PĀÑINI uses SKETCH [15] to synthesize function definitions over loop-free programs, and the symbolic execution engine CREST [18] for generating example-suites guided by derivation coverage.

We attempt to answer the following research questions:

- Can PĀÑINI synthesize attribute grammars from a variety of sketches?
- How do INCREMENTALSYNTHESIS and ALLATONCE algorithms compare?
- How does PĀÑINI scale with the number of holes?
- How does PĀÑINI scale with the size of the grammar?

The default algorithm for PĀÑINI is the INCREMENTALSYNTHESIS algorithm; unless otherwise mentioned, PĀÑINI refers to the implementation of INCREMENTALSYNTHESIS (Algorithm 2) for synthesis using examples generation guided by derivation coverage (detailed explanation available in the

extended version [10]). While ALLATONCE works well for small grammars with few examples, INCREMENTALSYNTHESIS scales well even for larger grammars, both with the number of holes and size of the grammar.

PĀÑINI can synthesize semantic-actions across both synthesized and inherited attributes. Some of our benchmarks contain inherited-attributes: for example, benchmark b8 uses inherited-attributes to pass the type information of the variables. Inherited-attributes pose no additional challenge; they are handled by the standard trick of introducing “marker” non-terminals [19].

A. Attribute Grammar Synthesis

We evaluated PĀÑINI on a set of attribute grammars adapted from software in open-source repositories [14], [19]–[24]. Table I shows the benchmarks, number of productions (**#P**), number of holes (**#H**), input example, solving time (**Time**, **AAO** for ALLATONCE and **IS** for INCREMENTALSYNTHESIS) and number of times a defined function was refuted (**#R**). Please recall that ALLATONCE refers to Algorithm 1 (§IV-A) and INCREMENTALSYNTHESIS refers to Algorithm 2 (§IV-B).

We provide more detailed descriptions of the benchmarks b1 to b9 in the extended version [10]. The benchmark b10 is the forward differentiation example described in §III-A. Benchmarks b11 and b12 are quite complex benchmarks that interpret a (subset) of Java bytecode and compile C code:

b11 Bytecode interpreter. Interpreter for a subset of Java bytecode; it supports around 36 instructions [25] of different type, i.e., load-store, arithmetic, logic and control transfer instructions.

b12 Mini-compiler. Fig. 6 shows the different steps of synthesizing semantic actions in mini-compiler. Fig. 6b is a sample input for the mini-compiler. Fig. 6a shows snippet of the attribute grammar for mini-compiler. Fig. 6c shows the two-address code generated from the input code shown in Fig. 6b, where h_a^\bullet and h_b^\bullet are two holes in the two-address code. Finally, in Fig. 6d shows the synthesized definition for h_a^\bullet and h_b^\bullet in the target language for two-address code.

Fig. 8 attempts to capture the fraction of time taken by the different phases of PĀÑINI: example generation and synthesis. Not surprisingly, the synthesis phase dominates the cost as it requires several invocation of the synthesis engines, whereas, the example generation phase does not invoke synthesis engines or smt solvers. Further, the difference in time spend in these two phases increases as the benchmarks get more challenging.

B. ALLATONCE *v/s* INCREMENTALSYNTHESIS

1) Scaling with holes: Fig. 9a and Fig. 9b show PĀÑINI scales with the sketches with increasingly more holes. We do this study for forward differentiation (b10) and bytecode interpreter (b11). As can be seen, PĀÑINI scales very well. On the other hand, ALLATONCE works well for small instances but soon blows up, timing out on all further instances. The interesting jump in b10 (at #Holes=8) was seen when we

S \rightarrow MAIN B
 | MAIN B

...
 A \rightarrow T

| E + T A.val = $h_a^*(E.val, T.val)$;
 | E - T A.val = $h_b^*(E.val, T.val)$;

T \rightarrow F

| T * F T.val = $h_c^*(T.val, F.val)$;
 | T / F T.val = $h_d^*(T.val, F.val)$;

...

(a) Attribute grammar sketch for mini-compiler

```
int main{
  int a, b, c;
  a = 4;
  b = a + 3; //  $h_a^*$ 
  c = a - b; //  $h_b^*$ 
  return c;
}
```

(b) A simple C code

op	arg1	arg2	dst
assign	4		a
h_a^*	a	9	T0
assign	T0		b
h_b^*	a	b	T1
assign	T1		T2
ret			T2

(c) Three-address code generated from C code in Fig. 6b

h_a^* a b:
 emit("load r1 a")
 emit("load r2 b")
 emit("plus r1 r2")

h_b^* a b:
 emit("load r1 a")
 emit("load r2 b")
 emit("sub r1 r2")

(d) Synthesized definition for h_a^*, h_b^*

Fig. 6: Synthesis of mini-compiler (b12)

TABLE I: Description of benchmarks

Id	Benchmark	#P	#H	Example	#R	Time (s)	
						AAO	IS
b1	Count ones	5	1	11001	0	3.2	3.1
b2	Binary to integer	5	1	01110	0	3.6	2.9
b3	Prefix evaluator	7	4	+ 3 4	0	TO	10.1
b4	Postfix evaluator	7	4	2 3 4 * +	0	TO	10.5
b5	Arithmetic calculator	8	4	5 * 2 + 8	0	TO	12.8
b6	Currency calculator	10	4	USD 3 + INR 8	0	TO	13.6
b7	if-else calculator	10	4	if(3+4 == 3) then 44; else 73;	1	TO	21.7
b8	Activation record layout	10	3	int a , b;	0	TO	13.8
b9	Type checker	11	5	(5 - 2) == 3	1	TO	15.4
b10	Forward differentiation	20	12	x*pow(x,3)	2	TO	39.2
b11	Bytecode interpreter	39	36	bipush 3; bipush 4; iadd;	3	TO	141.4
b12	Mini-compiler	43	6	int main(){ return 2+3;}	0	TO	9.2

```
init {
  run Foo(8+(6-7));
}
```

(e) PROMELA source code

```
node n1 = node(val=8);
node n2 = node(val=6);
node n3 = node(val=7);
node n4 =  $h_a^*$ (n2, n3);
node n5 =  $h_b^*$ (n1, n4);
node n6 =  $h_c^*$ ('Foo', n5);
```

(f) Trace generated

Fig. 7: Trace generation for AST construction

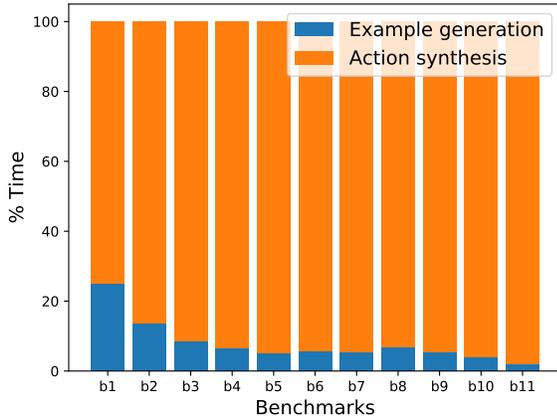


Fig. 8: Stacked bar graph for the % time spent in example creation and synthesis

started adding holes for the definitions of the more complex operators like $\sin()$ and $\cos()$.

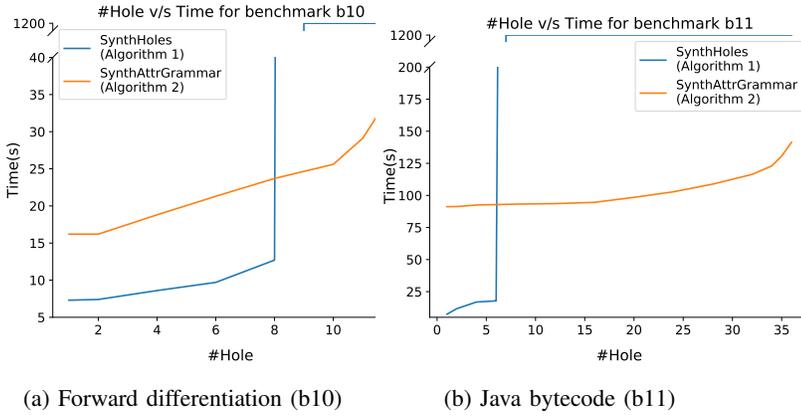
2) *Scaling with size of grammar*: Table I shows that INCREMENTALSYNTHESIS scales well with the size of the grammar (by the number of productions). On the other hand, ALLATONCE works well for the benchmarks b1 and b2 as they have only one hole while it times out for the rest.

The complexity of INCREMENTALSYNTHESIS is independent of the size of the attribute-grammar but dependent on the length of derivations and the size of the semantic actions. The current state of synthesis-technology allows PĀṆINI to synthesize practical attribute grammars that have a large number of productions but mostly “small” semantic actions and where short derivations can “cover” all productions. Further, any improvement in program-synthesis technology automatically improves the scalability of PĀṆINI.

VI. CASE STUDY

We undertook a case-study on the parser specification of the SPIN [5] model-checker. SPIN is an industrial-strength model-checker that verifies models written in the PROMELA [26] language against linear temporal logic (LTL) specifications. SPIN uses YACC [2] to build its parser for PROMELA. The modelling language, PROMELA, is quite rich, supporting variable assignments, branches, loops, arrays, structures, procedures etc. The attribute grammar specification in the YACC language is more than 1000 lines of code (ignoring newlines) having 280 production rules.

The semantic actions within the attribute grammar in the YACC description handle multiple responsibilities. We selected two of its operations:



(a) Forward differentiation (b10) (b) Java bytecode (b11)

Fig. 9: #Hole v/s Time for benchmarks b10 and b11

```
init {
  int flags[(5 * 25) - 42];
  int v = flags[10 - 4 + (9 / 3)];
}
```

(a) PROMELA source

```
init {
  int flags[83];
  int v = flags[9];
}
```

(b) PROMELA optimized

Fig. 10: Constant folding in PROMELA

a) *Constant folding array indices:* As the PROMELA code is parsed, semantic actions automatically constant-fold array indices (see Fig. 10). We removed all the actions corresponding to constant-folding by inserting 8 holes in the relevant production rules (these correspond to the non-terminal `const_expr`). The examples to drive the synthesis consisted of PROMELA code with arrays with complex expressions and the target output was the optimized PROMELA code. PĀÑINI was able to automatically synthesize this constant-folding optimization within less than 4 seconds.

b) *AST construction:* A primary responsibility of the semantic analysis phase is to construct the abstract syntax tree (AST) of the source PROMELA code. We, next, attempted to enquire if PĀÑINI is capable of this complex task.

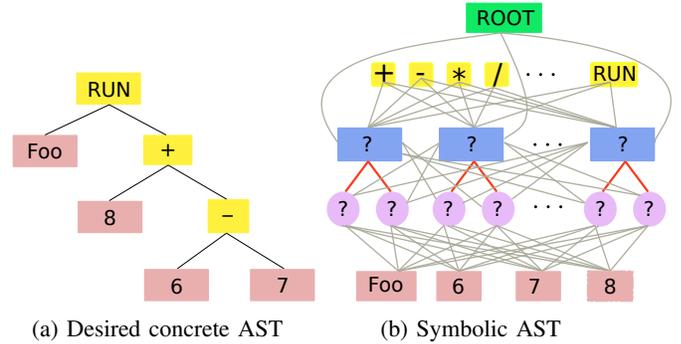
In this case, each example includes a PROMELA code as input and a tree (i.e. the AST) as the output value. We removed the existing actions via 23 holes. These holes had to synthesize the end-to-end functionality for a production rule with respect to building the AST: that, the synthesized code would decide the type of AST node to be created and the correct order of inserting the children sub-trees.

Run of the example suite on the sketchy productions generates a set of programs (one such program shown in Fig. 7); these programs produce *symbolic* ASTs that non-deterministically assigns type to nodes and assigns the children nodes. We leverage the support of references in Sketch to define self-referential nodes.

We insert constraints that establish tree isomorphism by recursively matching the symbolic ASTs with the respective output ASTs (available in example suite); for example, in Fig. 11 isomorphism constraints are enforced on the concrete and the symbolic ASTs. Sketch resolves the non-determinism en route to synthesizing the relevant semantic actions. In this case-study, PĀÑINI was able to synthesize the actions corresponding to the 23 holes within 20 seconds.

VII. RELATED WORK

Program synthesis is a rich area with proposals in varying domains: bitvectors [27], [28], heap manipulations [29]–[33], bug synthesis [34], differential privacy [35], [36], invariant



(a) Desired concrete AST (b) Symbolic AST

Fig. 11: Desired AST for code in Fig. 7 and symbolic AST. Grey lines (in Fig. 11b) denote symbolic choices.

generation [37], Skolem functions [38]–[40], synthesis of fences and atomic blocks [41] and even in hardware security [42]. However, to the best of our knowledge, ours is the first work on automatically synthesizing semantics actions for attribute grammars.

There has some work on automatically synthesizing parsers: PARSIFY [43] provides an interactive environments to automatically infer grammar rules to parse strings; it is been shown to synthesize grammars for Verilog, Tiger, Apache Logs, and SQL. CYCLOPS [44] builds an encoding for *Parse Conditions*, a formalism akin to *Verification Conditions* but for parseable languages. Given a set of positive and negative examples, CYCLOPS, automatically generates an LL(1) grammar that accepts all positive examples and rejects all negative examples. Though none of them handle attribute grammars, it may be possible to integrate them with PĀÑINI to synthesize *both* the context-free grammar and the semantic actions. We plan to pursue this direction in the future.

We are not aware of much work on testing attribute grammars. We believe that our *derivation coverage* metric can also be potent for finding bugs in attribute grammars, and can have further applications in dynamic analysis [45]–[47] and statistical testing [48], [49] of grammars. However, the effectiveness of this metric for bug-hunting needs to be evaluated and seems to be a good direction for the future.

REFERENCES

- [1] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, Jun 1968. [Online]. Available: <https://doi.org/10.1007/BF01692511>
- [2] S. C. Johnson and M. Hill, "YACC: Yet Another Compiler Compiler," *UNIX Programmer's Manual*, vol. 2, pp. 353–387, 1978.
- [3] GNU Bison. (last accessed 29 Jun 2021). [Online]. Available: <https://www.gnu.org/software/bison/>
- [4] ANTLR. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/antlr/antlr4>
- [5] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.
- [6] Q3B SMT solver. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/martijnjonas/Q3B>
- [7] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24
- [8] C Intermediate Language (CIL). (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/cil-project/cil>
- [9] MySQL. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/mysql/mysql-server>
- [10] P. K. Kalita, M. J. Kumar, and S. Roy, "Synthesis of semantic actions in attribute grammars," 2022. [Online]. Available: <https://arxiv.org/abs/2208.06916>
- [11] K. J. Rähä and M. Saarinen, "Testing attribute grammars for circularity," *Acta Informatica*, vol. 17, no. 2, pp. 185–192, Jun 1982. [Online]. Available: <https://doi.org/10.1007/BF00288969>
- [12] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, Oct. 1969. [Online]. Available: <https://doi.org/10.1145/363235.363259>
- [13] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018. [Online]. Available: <http://jmlr.org/papers/v18/17-468.html>
- [14] Automating differentiation using dual numbers. (last accessed 29 Jun 2021). [Online]. Available: <https://blog.demofox.org/2014/12/30/dual-numbers-automatic-differentiation/>
- [15] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5–6, p. 475–495, oct 2013. [Online]. Available: <https://doi.org/10.1007/s10009-012-0249-7>
- [16] E. Torlak and R. Bodik, "Growing solver-aided languages with Rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–152. [Online]. Available: <https://doi.org/10.1145/2509578.2509586>
- [17] Flex. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/westes/flex>
- [18] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 443–446. [Online]. Available: <https://doi.org/10.1109/ASE.2008.69>
- [19] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [20] A simple calculator. (last accessed 29 Jun 2021). [Online]. Available: <https://www.dabeaz.com/ply/example.html>
- [21] Evaluate postfix expression using YACC. (last accessed 29 Jun 2021). [Online]. Available: <https://prashantkulkarni17.wordpress.com/2011/09/20/evaluate-postfix-expression-using-yacc/>
- [22] Mini-compiler. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/SandyaSivakumar/Mini-Compiler/>
- [23] Conversion from binary to decimal. (last accessed 29 Jun 2021). [Online]. Available: <https://myprogworld.wordpress.com/2016/04/30/conversion-from-binary-to-decimal/>
- [24] Type check. (last accessed 29 Jun 2021). [Online]. Available: <http://pages.cs.wisc.edu/~fischer/cs536.s06/course.hold/html/NOTES/4.SYNTAX-DIRECTED-TRANSLATION.html#ex2>
- [25] Java Byte Code. (last accessed 29 Jun 2021). [Online]. Available: https://en.wikibooks.org/wiki/Java_Programming/Byte_Code
- [26] Concise Promela Reference. (last accessed 29 Jun 2021). [Online]. Available: <http://spinroot.com/spin/Man/Quick.html>
- [27] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 62–73. [Online]. Available: <https://doi.org/10.1145/1993498.1993506>
- [28] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 281–294. [Online]. Available: <https://doi.org/10.1145/1065010.1065045>
- [29] S. Roy, "From concrete examples to heap manipulating programs," in *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, ser. Lecture Notes in Computer Science, F. Logozzo and M. Fähndrich, Eds., vol. 7935. Springer, 2013, pp. 126–149. [Online]. Available: https://doi.org/10.1007/978-3-642-38856-9_9
- [30] A. Garg and S. Roy, "Synthesizing heap manipulations via integer linear programming," in *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015. Proceedings*, ser. Lecture Notes in Computer Science, S. Blazy and T. P. Jensen, Eds., vol. 9291. Springer, 2015, pp. 109–127. [Online]. Available: https://doi.org/10.1007/978-3-662-48288-9_7
- [31] S. Verma and S. Roy, "Synergistic debug-repair of heap manipulations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 163–173. [Online]. Available: <https://doi.org/10.1145/3106237.3106263>
- [32] S. Verma and Subhajit Roy, "Debug-localize-repair: A symbiotic construction for heap manipulations," *Formal Methods Syst. Des.*, vol. 58, no. 3, pp. 399–439, 2021. [Online]. Available: <https://doi.org/10.1007/s10703-021-00387-z>
- [33] N. Polikarpova and I. Sergey, "Structuring the synthesis of heap-manipulating programs," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3290385>
- [34] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 224–234. [Online]. Available: <https://doi.org/10.1145/3236024.3236084>
- [35] S. Roy, J. Hsu, and A. Albarghouthi, "Learning differentially private mechanisms," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 852–865. [Online]. Available: <https://doi.org/10.1109/SP40001.2021.00060>
- [36] Y. Wang, Z. Ding, Y. Xiao, D. Kifer, and D. Zhang, "DPGen: Automated program synthesis for differential privacy," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 393–411. [Online]. Available: <https://doi.org/10.1145/3460120.3484781>
- [37] S. Lahiri and S. Roy, "Almost correct invariants: Synthesizing inductive invariants by fuzzing proofs," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 352–364. [Online]. Available: <https://doi.org/10.1145/3533767.3534381>
- [38] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data-driven approach for boolean function synthesis," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020. Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 611–633. [Online]. Available: https://doi.org/10.1007/978-3-030-53291-8_31
- [39] P. Golia, S. Roy, and Kuldeep S. Meel, "Program synthesis as

- dependency quantified formula modulo theory,” in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Z. Zhou, Ed. ijcai.org, 2021, pp. 1894–1900. [Online]. Available: <https://doi.org/10.24963/ijcai.2021/261>
- [40] P. Golia, F. Slivovsky, S. Roy, and K. S. Meel, “Engineering an efficient boolean functional synthesis engine,” in *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*. IEEE, 2021, pp. 1–9. [Online]. Available: <https://doi.org/10.1109/ICCAD51958.2021.9643583>
- [41] A. Verma, P. K. Kalita, A. Pandey, and S. Roy, “Interactive debugging of concurrent programs under relaxed memory models,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 68–80. [Online]. Available: <https://doi.org/10.1145/3368826.3377910>
- [42] G. Takhar, R. Karri, C. Pilato, and S. Roy, “HOLL: Program synthesis for higher order logic locking,” in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 3–24. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_1
- [43] A. Leung, J. Sarracino, and S. Lerner, “Interactive parser synthesis by example,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 565–574. [Online]. Available: <https://doi.org/10.1145/2737924.2738002>
- [44] D. Singal, P. Agarwal, S. Jhunjhunwala, and S. Roy, “Parse condition: Symbolic encoding of LL(1) parsing,” in *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, G. Barthe, G. Sutcliffe, and M. Veanes, Eds., vol. 57. EasyChair, 2018, pp. 637–655. [Online]. Available: <https://easychair.org/publications/paper/DtjZ>
- [45] S. Roy and Y. N. Srikant, “Profiling k-iteration paths: A generalization of the Ball-Larus profiling algorithm,” in *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*. IEEE Computer Society, 2009, pp. 70–80. [Online]. Available: <https://doi.org/10.1109/CGO.2009.11>
- [46] R. Chouhan, S. Roy, and S. Baswana, “Pertinent path profiling: Tracking interactions among relevant statements,” in *Proceedings of the 13th IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 2013, pp. 16:1–16:12. [Online]. Available: <https://doi.org/10.1109/CGO.2013.6494983>
- [47] G. Kumar and S. Roy, “Online identification of frequently executed acyclic paths by leveraging data stream algorithms,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC ’13, Coimbra, Portugal, March 18-22, 2013*, S. Y. Shin and J. C. Maldonado, Eds. ACM, 2013, pp. 1694–1695. [Online]. Available: <https://doi.org/10.1145/2480362.2480680>
- [48] P. Chatterjee, A. Chatterjee, J. Campos, R. Abreu, and S. Roy, “Diagnosing software faults using multiverse analysis,” in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, C. Bessiere, Ed. ijcai.org, 2020, pp. 1629–1635. [Online]. Available: <https://doi.org/10.24963/ijcai.2020/226>
- [49] V. Modi, S. Roy, and S. K. Aggarwal, “Exploring program phases for statistical bug localization,” in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE ’13, Seattle, WA, USA, June 20, 2013*, S. N. Freund and C. S. Pasareanu, Eds. ACM, 2013, pp. 33–40. [Online]. Available: <https://doi.org/10.1145/2462029.2462034>

Reactive Synthesis Modulo Theories using Abstraction Refinement

Benedikt Maderbacher 
Graz University of Technology
Graz, Austria
benedikt.maderbacher@iaik.tugraz.at

Roderick Bloem 
Graz University of Technology
Graz, Austria
roderick.bloem@iaik.tugraz.at

Abstract—Reactive synthesis builds a system from a specification given as a temporal logic formula. Traditionally, reactive synthesis is defined for systems with Boolean input and output variables. Recently, new techniques have been proposed to extend reactive synthesis to data domains, which are required for more sophisticated programs. In particular, Temporal stream logic (TSL) extends LTL with state variables, updates, and uninterpreted functions and was created for use in synthesis. We present a new synthesis procedure for TSL(T), an extension of TSL with theories. Our approach is also able to find predicates, not present in the specification, that are required to synthesize some programs. Synthesis is performed using two nested counterexample guided synthesis loops and an LTL synthesis procedure. Our method translates TSL(T) specifications to LTL and extracts a system if synthesis is successful. Otherwise, it analyzes the counterstrategy for inconsistencies with the theory, these are then ruled out by adding temporal assumptions, and the next iteration of the loop is started. If no inconsistencies are found the outer refinement loop tries to identify new predicates and reruns the inner loop. A system can be extracted if the LTL synthesis returns realizable at any point, if no more predicates can be added the problem is unrealizable. The general synthesis problem for TSL is known to be undecidable. We identify a new decidable fragment and demonstrate that our method can successfully synthesize or show unrealizability of several non-Boolean examples.

I. INTRODUCTION

Reactive synthesis [1] is the problem of automatically constructing a system from a specification. The user provides a specification in temporal logic and the synthesis procedure constructs a system that satisfies it if one exists. Traditionally this only works for systems with Boolean input and output variables. However, real-world systems often use more sophisticated data like integers, reals, or structured data. For finite domains, it is possible to use bit-blasting to obtain an equivalent Boolean specification. However, in general, bit-blasting techniques do not work for infinite domains, bit-blasted specifications are hard to read, and a large number of variables make the specifications very hard to solve.

In recent years multiple theories have been proposed to perform reactive synthesis with non-Boolean inputs and outputs. There have been decidability results for synthesis using register automata [2], [3], [4] and variable automata [5].

Our work builds on temporal stream logic (TSL). TSL, proposed by Finkbeiner et al. [6], uses a logic based on

This work was supported by the Graz University of Technology through the LEAD Project “Dependable Internet of Things in Adverse Environments”.

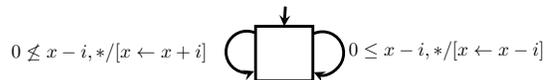


Fig. 1. Synthesized system for the running example.

linear temporal logic (LTL) with state variables, uninterpreted functions and predicates, and update expressions. TSL allows for an elegant and efficient synthesis method that separates control from data. However, the ability to specify how data is handled is limited because functions and predicates remain uninterpreted. Finkbeiner et al. [7] describe an extension to TSL modulo theories, but consider only satisfiability and not synthesis.

In this paper, we propose a new synthesis algorithm for temporal stream logic modulo theories that can be applied to arbitrary decidable theories in which quantifier elimination is possible. Let us consider a concrete example using the theory of linear integer arithmetic (LIA).

Example 1. We want to build a system with one integer state variable x and one integer input i . The objective is to keep the value of the state variable between 0 and 100. At any time step the system can select one of two updates: increase or decrease x by i , where i is chosen by the environment in the interval $0 \leq i < 5$. We assume that the initial state is any value inside the boundaries. These requirements can be written as the TSL formula

$$\phi \triangleq (0 \leq x \wedge x < 100 \wedge \square(0 \leq i \wedge i < 5)) \rightarrow \square(0 \leq x \wedge x < 100 \wedge ([x \leftarrow x - i] \vee [x \leftarrow x + i])),$$

where the propositions $[x \leftarrow x - i]$ and $[x \leftarrow x + i]$ describe updates to x . Figure 1 shows a mealy machine that realizes the specification above. It is impossible to write a correct system using only the predicates from the specification: if the environment chooses initially $x = 0$ the system has to first perform addition, however, if the environment chooses $x = 99$ the system has to perform subtraction. The predicates in the specification cannot distinguish these cases.

Inspired by this example we want our synthesis algorithm to function with expressions from theories, as well as identify new predicates where necessary. Figure 2 shows an overview of our approach. We use a different refinement loop than the

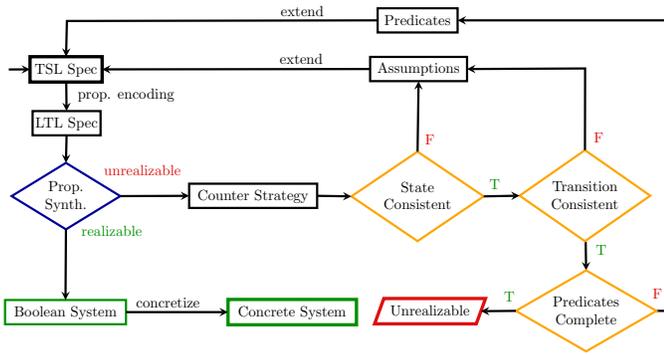


Fig. 2. Overview of the synthesis procedure.

original TSL synthesis approach [6], our approach, which is depicted in Figure 2, relies on checking local properties and also is able to create new predicates.

First, the TSL specification is encoded into an LTL formula that contains a Boolean variable for each theory predicate in the TSL formula. These variables are seen as inputs, which means that the environment determines their truth values. As a result, realizability of the LTL formula implies realizability of the TSL formula, but not vice versa, because the environment can choose values for the variables that are not consistent with the theory. The LTL formula is then given to a propositional LTL synthesis tool [8], [9]. If the Boolean synthesis is successful we obtain a Boolean system that can be concretized into a system that operates on the original value domain. If synthesis of the LTL formula is not successful, we get a Boolean counterstrategy that we analyze for inconsistencies with respect to the theory.

The central part of our algorithm is the theory consistency analysis of the counterstrategy. In contrast to Finkbeiner et al. [6] we treat the counterstrategy as a Moore machine instead of a tree. The output of a state in a counterstrategy is a valuation of the predicates and the transitions perform updates on the register values. Whereas the original approach analyzes potentially long traces in the tree, we perform a local analysis of individual states and transitions. We use an SMT solver to check whether the predicate valuation in each state is consistent and whether the valuations in two consecutive states are consistent with the updates on the transition between them. We show that if all states and transitions are (locally) consistent, then the counterstrategy is globally theory-consistent as well and thus the TSL specification is unrealizable.

If an inconsistency is found, the counterstrategy is spurious. We thus need to refine the LTL formula and start a new iteration. We refine the LTL specification by adding new assumptions and possibly new predicates. The assumptions refer to the relation between predicates (for inconsistent outputs of states) or between predicates and updates (for inconsistent transitions). If a transition is consistent for some values but inconsistent for others, we create a new predicate that distinguishes whether the update is valid or not.

The procedure can be likened to a CEGAR loop [10] or

to the DPLL(T) in which LTL synthesis plays the role of the propositional SAT solver and the consistency check is performed by the theory solver. The main difference is that in our case inconsistencies can span multiple time steps. Our approach has multiple advantages over the original TSL synthesis approach: it can easily be extended to new theories, it can find new predicate needed to realize certain specifications, and it can show unrealizability (without bound on the size of the considered systems).

The main contributions of our paper are as follows:

- A new synthesis procedure for TSL that works with theories and can generate additional predicates that are necessary to realize certain specifications.
- Synthesis for TSL, in general, is known to be undecidable [6], we show that the problem is decidable for equality logic.
- Our algorithm can prove that certain specifications are unrealizable.

The remaining paper is structured as follows: Section II summarizes required definitions from TSL. Section III formalizes the synthesis problem for TSL modulo theory. We describe the Boolean abstraction and the theory consistency analysis in detail in Section IV. The main synthesis procedure is described in Section V. An experimental evaluation was performed for multiple examples using the theories of linear integer arithmetic and linear real arithmetic (Section VI). We discuss related work in Section VII and conclude our work in Section VIII.

II. PRELIMINARIES

We use Temporal Stream Logic (TSL) [6], with the addition of decidable theories [7]. This section repeats the definitions from Finkbeiner et al. [6], [7] with some changes in notation and with a more general treatment of theories.

A. Theories and Updates

In contrast to [7], where axiomatic semantics of the used theories is used, we rely on the definitions built into an SMT solver. A theory \mathcal{T} consists of a signature (symbols for constants, functions, and predicates) and semantics as defined by the SMT solver (which can be axiomatic). The domain is called \mathbb{T} . In case of variables of different sorts (i.e. types) we use \mathbb{T} for the union of all domains and assume that all variables take values from their domain. In the following, we will use $E_{\mathcal{T}}(V)$ to denote the set of expressions in \mathcal{T} with the set of variables V denoting a superset of the free variables in the expression. The set $E_{\mathcal{T}}(V)$ is partitioned into a set $E_{\mathcal{T}}^{\mathbb{T}}(V)$ of terms (denoting values in \mathbb{T}) and a set $E_{\mathcal{T}}^{\mathbb{B}}(V)$ of formulas (denoting truth values). We assume that the theories used have decidable procedures for satisfiability checking and quantifier elimination. We assume that we are given a procedure `sat` that returns true iff a formula ϕ is satisfiable and a function `quantelim` that takes a formula and returns a theory-equivalent formula that does not contain quantifiers.

B. Temporal Stream Logic Modulo Theories

TSL(T) is based on linear temporal logic, but instead of Boolean variables it uses updates and Boolean theory expressions. Key concepts of TSL and TSL(T) are state variables R and input variables I which both hold values from the theory domain and updates that access state variables and input variables and write new values to the state variables. The grammar for TSL(T) formulas is

$$\begin{aligned}
\langle \text{ap} \rangle &:= E_{\mathcal{T}}^{\mathbb{B}}(R \cup I) \\
\langle \text{term} \rangle &:= E_{\mathcal{T}}^{\mathbb{T}}(R \cup I) \\
\langle \text{bconst} \rangle &:= \text{true} \mid \text{false} \\
\langle \text{upd} \rangle &:= [\langle \text{var} \rangle \leftarrow \langle \text{term} \rangle] \\
\langle \text{tsl} \rangle &:= \langle \text{ap} \rangle \mid \langle \text{upd} \rangle \mid \langle \text{bconst} \rangle \mid \neg \langle \text{tsl} \rangle \mid \langle \text{tsl} \rangle \wedge \langle \text{tsl} \rangle \\
&\mid \langle \text{tsl} \rangle \mathcal{U} \langle \text{tsl} \rangle \mid \mathcal{X} \langle \text{tsl} \rangle.
\end{aligned}$$

To define the semantics of TSL(T) we need some additional notation that will be used throughout the paper. An update $[r \leftarrow e]$ assigns a state variable r an expression $e \in E_{\mathcal{T}}^{\mathbb{T}}(R \cup I)$. We use U for the finite set of all updates that occur in the formula under consideration as well as the updates that assign each state variable to itself. An update function \mathbf{u} is a function that associates each state variable $r \in R$ with an expression $e \in E_{\mathcal{T}}^{\mathbb{T}}(R \cup I)$. We refer to the set of update functions where all pairs (r, e) are updates in U as \mathbf{U} . Equivalently, \mathbf{U} can be seen as the set of all subsets of U that contain exactly one update for each state variable in R .

We introduce the notations $\mathbf{R} \triangleq 2^{R \rightarrow \mathbb{T}}$ and $\mathbf{I} \triangleq 2^{I \rightarrow \mathbb{T}}$ for the sets of valuations of variables. We write R/\mathbf{r} (I/\mathbf{i}) to denote the replacement of all variables in R (I , resp.) by their corresponding values in $\mathbf{r} \in \mathbf{R}$ ($\mathbf{i} \in \mathbf{I}$, resp.). With slight abuse of notation, we identify $e[R/\mathbf{r}, I/\mathbf{i}]$ with the corresponding value in the domain. To apply an update function $\mathbf{u} \in \mathbf{U}$ to valuations $\mathbf{r} \in \mathbf{R}$ and $\mathbf{i} \in \mathbf{I}$ we write $\mathbf{u}[\mathbf{r}, \mathbf{i}]$ which is defined as $\mathbf{u}[\mathbf{r}, \mathbf{i}](r) = \mathbf{u}(r)[R/\mathbf{r}, I/\mathbf{i}]$ for each r .

The semantics of TSL(T) is defined with respect to a trace $\rho \in (\mathbf{I} \times \mathbf{R})^\omega$ of inputs and state variable valuations as follows. We assume that $\rho = \rho_0, \rho_1, \dots$ and that $\rho_j = (\mathbf{r}_j, \mathbf{i}_j)$ and we define

$$\begin{aligned}
\rho \models p &\text{ iff } \rho_0 \models p \text{ for } p \in \langle \text{ap} \rangle, \\
\rho \models [r \leftarrow e] &\text{ iff } \mathbf{r}_1(r) = e[R/\mathbf{r}_0, I/\mathbf{i}_0], \\
\rho \models \text{true}, \\
\rho \not\models \text{false}, \\
\rho \models \neg \phi &\text{ iff } \rho \not\models \phi, \\
\rho \models \phi \wedge \psi &\text{ iff } \rho \models \phi \text{ and } \rho \models \psi, \\
\rho \models \phi \mathcal{U} \psi &\text{ iff } \exists j. \rho_j, \rho_{j+1}, \dots \models \psi \text{ and} \\
&\forall i < j. \rho_i, \rho_{i+1}, \dots \models \phi \\
\rho \models \mathcal{X} \phi &\text{ iff } \rho_1, \rho_2, \dots \models \phi.
\end{aligned}$$

The unary temporal operators eventually (\diamond) and globally (\square) can be added using their usual definitions: $\diamond \varphi \equiv \text{true} \mathcal{U} \varphi$ and $\square \varphi \equiv \neg \diamond \neg \varphi$.

C. LTL Synthesis

Our algorithm relies on existing solvers for the linear temporal logic (LTL) synthesis problem which we also refer to as propositional synthesis. For completeness, we provide a brief description of the problem. A formal treatment is available in [1].

Given an LTL formula ϕ containing Boolean variables X separated into the two disjoint sets X_I and X_O . Consider a game between two players (environment and system) where at each point in time both players pick the values of their Boolean variables (first X_I by the environment followed by X_O by the system). The game is won by the system if the resulting infinite trace satisfies ϕ and is won by the environment otherwise. The synthesis problem is: does there exist a Mealy machine strategy for the system that wins against every environment? If no such strategy exists there exists a Moore machine strategy for the environment that wins against every system. An LTL synthesis tool such as Strix [8] can determine who wins and construct a (Mealy or Moore) strategy for the winning player.

III. SYNTHESIS PROBLEM FOR TSL(T)

We want to synthesize systems from a TSL(T) specification, which we defined in the previous section. Before giving a formal definition of synthesis we need to define the systems we want to build.

Our constructed systems differ from those considered by Finkbeiner et al.[6]. They synthesize control flow models, which consist of a circuit of logic gates and vertices of uninterpreted functions that determines the values of outputs and new cells based on inputs and old cells. We instead target an extension of Mealy machines.

A. Theory Mealy and Moore Machines

A system using state variables of an unbounded domain can be hard to represent finitely. To create actual programs our systems need to have a finite structure. This is achieved by restricting all operations on state and input variables to a finite set of symbolic operations. The values of state variables need to be determined by an update chosen from a finite set U . The set of all update functions using updates from U is denoted by \mathbf{U} . To make decisions based on the values of variables we use a finite set of predicates $P \subseteq E_{\mathcal{T}}^{\mathbb{B}}(R \cup I)$. For a given valuation $\mathbf{v} = (\mathbf{r}, \mathbf{i})$, let $P_{\mathbf{v}} \subseteq P$ be the subset of predicates that is true in \mathbf{v} : $P_{\mathbf{v}} = \{p \in P \mid \mathbf{v} \models p\}$. Using these we can define an *extended trace* as an infinite sequence $\rho_E = (\mathbf{r}_0, \mathbf{i}_0, \mathbf{u}_0, P_0), \dots$ over $(\mathbf{R} \times \mathbf{I} \times \mathbf{U} \times 2^P)$ such that for all j , $\mathbf{r}_{j+1} = \mathbf{u}_j[\mathbf{r}_j, \mathbf{i}_j]$ and $P_j = P_{(\mathbf{r}_j, \mathbf{i}_j)}$. The corresponding *theory trace* is the trace $(\mathbf{r}_0, \mathbf{i}_0), \dots$ over $\mathbf{R} \times \mathbf{I}$.

We introduce the new concept of *Theory Mealy Machines* that are state machines with inputs and state variables that range over the theory domain. A *Theory Mealy Machine* $M_{\mathcal{T}} = (Q, q_0, U, P, \mathbf{r}_0, \delta, \mu)$ consists of a finite set of states Q , an initial state $q_0 \in Q$, a finite set of updates U , a finite set of predicates $P \subseteq E_{\mathcal{T}}^{\mathbb{B}}(R \cup I)$, an initial valuation $\mathbf{r}_0 \in \mathbf{R}$, a transition function $\delta \in (Q \times 2^P) \rightarrow Q$ and an update selection function $\mu \in (Q \times 2^P) \rightarrow \mathbf{U}$.

A run σ of a theory Mealy machine induced by a sequence of input valuations $\bar{\mathbf{i}} = \mathbf{i}_0, \mathbf{i}_1, \dots \in \mathbf{I}^\omega$ is an infinite sequence of states Q and valuations \mathbf{R} $(q_0, \mathbf{r}_0), (q_1, \mathbf{r}_1), \dots$. Any two consecutive configurations (q_i, \mathbf{r}_i) and $(q_{i+1}, \mathbf{r}_{i+1})$ must be related by $q_{i+1} = \delta(q_i, P_{(\mathbf{r}_i, \mathbf{i}_i)})$ and $\mathbf{r}_{i+1} = \mathbf{u}_i[\mathbf{r}_i, \mathbf{i}_i]$ where $\mathbf{u}_i = \mu(q_i, P_{(\mathbf{r}_i, \mathbf{i}_i)})$.

An extended trace is obtained from a run as the infinite sequence $(\mathbf{r}_0, \mathbf{i}_0, \mathbf{u}_0, P_{(\mathbf{r}_0, \mathbf{i}_0)}), \dots$. A *Theory Mealy Machine* $M_{\mathcal{T}}$ realizes a TSL(T) formula ϕ if for all inputs sequences $\bar{\mathbf{i}} \in \mathbf{I}^\omega$ the resulting theory trace $\rho \equiv (\bar{\mathbf{i}}, \bar{\mathbf{r}})$ satisfies ϕ .

We also define Theory Moore machines, which read the updates produced by a Mealy machine and produce the inputs read by a Mealy machine. Intuitively, Mealy machines are used to show realizability of a TSL(T) specification, while Moore machines are used to show their unrealizability. A *Theory Moore Machine* $M_{\mathcal{T}} = (Q, q_0, U, P, \mathbf{r}_0, \delta, \iota)$ consists of a finite set of states Q , an initial state $q_0 \in Q$, a finite set of updates U , a finite set of predicates $P \subseteq E_{\mathcal{T}}^{\mathbb{B}}(R \cup I)$, an initial valuation $\mathbf{r}_0 \in \mathbf{R}$, and a transition function $\delta \in (Q \times U) \rightarrow Q$, and $\iota: Q \times \mathbf{R} \rightarrow \mathbf{I}$ is the output function.

A run σ of a *Theory Moore Machine* induced by an infinite sequence of update functions $\bar{\mathbf{u}} \in U^\omega$ is a sequence of states and valuations $(q_0, \mathbf{r}_0), (q_1, \mathbf{r}_1), \dots$. Any two consecutive entries (q_i, \mathbf{r}_i) and $(q_{i+1}, \mathbf{r}_{i+1})$ must be related by $q_{i+1} = \delta(q_i, \mathbf{u}_i)$ and $\mathbf{r}_{i+1} = \mathbf{u}_i[\mathbf{r}_i, \iota(q_i, \mathbf{r}_i)]$.

B. Problem Statement

Given a TSL(T) formula ϕ , the inputs I , the state variables R , and the updates U , the synthesis problem asks whether there exists a Theory Mealy machine $M_{\mathcal{T}}$ over R, I , and U such that for all input sequences $\bar{\mathbf{i}}$ the trace generated by $M_{\mathcal{T}}$ satisfies ϕ . Note that the created machine $M_{\mathcal{T}}$ must use the same variables I and R as well as the updates U as ϕ , but it may use predicates P that are not present in ϕ .

IV. BOOLEAN ABSTRACTION

A. Propositional Encoding of TSL(T)

This subsection describes the propositional encoding of TSL(T) into LTL as proposed by Finkbeiner et al. [6]. The fact that the functions and predicates in our terms have an interpretation does not affect this translation and it is equivalent to the one for TSL.

A TSL(T) formula ϕ is encoded to an LTL formula $\phi_{\mathcal{B}}$. Formula $\phi_{\mathcal{B}}$ is obtained by replacing each update u in ϕ by a Boolean output variable p_u and each atomic proposition ap by a Boolean input variable p_{ap} . Additionally, the formula ensures that for each variable exactly one update is active at any point in time. This results in: $\phi_{\mathcal{B}} \triangleq \square \left(\bigwedge_r \bigvee_i \left(p_{[r \leftarrow e_i]} \wedge \bigwedge_{j \neq i} \neg p_{[r \leftarrow e_j]} \right) \right) \wedge \phi[\text{ap}/p_{\text{ap}}, \dots, u/p_u, \dots]$.

Example 2. The TSL(T) formula from Example 1 is encoded as the LTL formula

$$\begin{aligned} \phi_{\mathcal{B}} \triangleq & \square (p_{[x \leftarrow x-i]} \wedge \neg p_{[x \leftarrow x+i]} \vee p_{[x \leftarrow x+i]} \wedge \neg p_{[x \leftarrow x-i]}) \wedge \\ & ((p_{0 \leq x} \wedge p_{x < 100} \wedge \square (p_{0 \leq i} \wedge p_{i < 5})) \rightarrow \\ & \square (p_{0 \leq x} \wedge p_{x < 100} \wedge (p_{[x \leftarrow x-i]} \vee p_{[x \leftarrow x+i]}))), \end{aligned}$$

where $p_{0 \leq x}$, $p_{x < 100}$, $p_{0 \leq i}$, and $p_{i < 5}$ are input variables and $p_{[x \leftarrow x-i]}$ and $p_{[x \leftarrow x+i]}$ are output variables.

B. Boolean Mealy and Moore Machines

Given a set of Boolean variables $V = \{p_u \mid u \in U\} \cup \{p_{\text{ap}} \mid \text{ap} \in P\}$, we say that a Boolean trace $\rho_{\mathcal{B}} = v_0, v_1, \dots$ over 2^V corresponds to an extended trace ρ_E iff for all j , $p_{\text{ap}} \in \rho_{\mathcal{B}}(j)$ iff $\mathbf{r}_j \cup \mathbf{i}_j \models \text{ap}$ and $p_u \in \rho_{\mathcal{B}}(j)$ iff $u \in \mathbf{u}_j$. Clearly, every extended trace corresponds to a Boolean trace, but the opposite is not true, for instance, because two predicates contradict each other, or because the updates and the predicates do not match.

The LTL specifications obtained from the propositional encoding can be realized by standard Mealy machines or shown unrealizable by standard Moore machines. To make the meaning of the input and output variables clearer we will call them predicates P and updates U and refer to the machines as Boolean Mealy and Moore Machines. We use \mathbf{U} and 2^P and leave the translation into vectors of Boolean variables implicit.

A *Boolean Mealy machine* is a tuple $(Q, P, U, q_0, \delta_{\mathcal{B}}, \mu_{\mathcal{B}})$, where Q is a set of states, U is a set of updates, P is a set of predicates, $q_0 \in Q$ is the initial state, $\delta_{\mathcal{B}} \in Q \times 2^P \rightarrow Q$ is the transition function, and $\mu_{\mathcal{B}} \in Q \times 2^P \rightarrow \mathbf{U}$ is the update selection function.

A run $\sigma_{\mathcal{B}}$ of a Boolean Mealy machine induced by a sequence of predicate sets $\bar{P} = P_0, P_1, \dots$ is an infinite sequence of states, updates, and predicate sets $(q_0, \mathbf{u}_0, P_0), (q_1, \mathbf{u}_1, P_1), \dots$ where $q_{i+1} = \delta_{\mathcal{B}}(q_i, P_i)$ and $\mathbf{u}_{i+1} = \mu_{\mathcal{B}}(q_i, P_i)$. The corresponding Boolean trace $\rho_{\mathcal{B}}$ is $(\mathbf{u}_0, P_0), (\mathbf{u}_1, P_1), \dots$.

A Boolean Mealy machine $M_{\mathcal{B}}$ is theory consistent with respect to theory \mathcal{T} iff every trace $\rho_{\mathcal{B}}$ induced by a consistent sequence \bar{P} has a corresponding extended trace ρ_E .

A *Boolean Moore machine* is a tuple $M_{\mathcal{B}} = (Q, P, U, q_0, \delta, o)$ where Q is a set of states, P is a set of predicates, $q_0 \in Q$ is the initial state, $\delta \in Q \times \mathbf{U} \rightarrow Q$ is the transition function, and $o \in Q \rightarrow 2^P$ is the output function.

A run $\sigma_{\mathcal{B}}$ of a Boolean Moore machine induced by a sequence $\bar{\mathbf{u}} = \mathbf{u}_0, \mathbf{u}_1, \dots$ is an infinite sequence of states and predicate sets $(q_0, P_0), (q_1, P_1), \dots$ where $q_{i+1} = \delta(q_i, \mathbf{u}_i)$ and $P_i = o(q_i)$. We call a Boolean Moore machine theory consistent with respect to theory \mathcal{T} iff every trace $\rho_{\mathcal{B}} = P_0, P_1, \dots$ can be extended to an extended trace ρ_E .

C. Theory Consistency Analysis

We propose three criteria to locally analyze Boolean Moore machines $M_{\mathcal{B}} = (Q, P, U, q_0, \delta, o)$ and sets of theory inputs variables I for theory consistency.

- Every state must be inhabited by at least one concrete state i.e. $\text{sat}(o(q))$ for every $q \in Q$.
- Every transition must be valid for at least one pair of concrete pre and post states i.e. $\text{sat}(o(q_i) \wedge \mathbf{u} \wedge o(q_j)')$ for every $(q_i, \mathbf{u}, q_j) \in \delta$.
- Every transition must be valid for all concrete pre-states i.e. $o(q_i) \rightarrow wp(u, \exists i'. o(q_j))$ for every $(q_i, \mathbf{u}, q_j) \in \delta$.

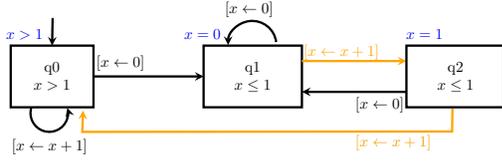


Fig. 3. Theory consistent machine that is not locally consistent.

Lemma 1. *If these local consistency criteria are satisfied for a Boolean Moore machine M_B , it is (globally) theory consistent and there exists a theory Moore machine M_T whose extended traces are consistent with the traces of M_B .*

Proof. Assuming the criteria are all satisfied. Every output function is satisfiable, this includes the initial state which contains an initial value r_0 of a theory machine. For every transition $(\varphi_i, \mathbf{u}, \varphi_j)$ where there exists a model of φ_i all of the models map to a model of φ_j , by the transition property checked by the algorithm. Therefore by induction, all paths starting in the initial state and only using transitions from δ have a corresponding extended trace $\rho_E = (r_0, \mathbf{i}_0, \mathbf{u}_0, P_0), (r_1, \mathbf{i}_1, \mathbf{u}_1, P_1), \dots$ where $r_{i+1} = \mathbf{u}_i(r_0, \mathbf{i}_0)$ and \mathbf{i}_i chosen such that $r_i \cup \mathbf{i}_i \models P_i$. A theory Moore machine can be obtained by providing a function o that chooses the values \mathbf{i}_i based on q_i and r_i . \square

Example 3. Our approach checks consistency on a local level, the environment strategy can still be theory consistent if the third criterion is violated. The Boolean Moore machine in Figure 3 has two transitions (orange) that do not satisfy this criterion even though the machine is theory consistent. The blue annotations are not part of the machine but are used to argue its (global) consistency. The transition $(q1, [x \leftarrow x + 1], q2)$ would be invalid for $x = 1$ in $q1$, but for every execution $x = 0$ in $q1$ and the problem does not appear. A similar situation occurs for the transition $(q2, [x \leftarrow x + 1], q0)$, where x will always be 1 and the transition is only invalid for $x < 1$. The blue annotations show the possible values of x in every state, demonstrating that all transitions are consistent.

We propose Algorithm 1 to locally analyze Boolean Moore machines for theory consistency, based on the criteria above. To be usable in our synthesis refinement loop the algorithm also creates additional assumptions and predicates that block inconsistent counter strategies. The counterstrategy analysis is performed in three stages. The first checks for consistency of outputs in a single state the second and third check consistency of transitions. The third check also creates new predicates. We will use some shorthand notation to define formulas: The output function $o(q)$ will be used to refer to the expression consisting of the conjunction of the elements in P , negated if their corresponding Boolean variable is false. We use $o(q)'$ with the same meaning as $o(q)$, except all free variables are renamed to their primed version. Similarly, \mathbf{u} is to be read as the conjunction of $r' = e$ for each update in \mathbf{u} .

a) *State Consistency:* To check state consistency we look at every (reachable) state in the counterstrategy and use an

def *isconsistent*(m):

Data: Boolean Moore machine

$m = (Q, P, U, q_0, \delta, o)$, set of variables I

Result: \top or (possibly) \perp with additional assumptions and predicates.

foreach $q \in \text{reachable}(Q)$; // Case 1

if $\neg \text{sat}(o(q))$ **then**

yield $\perp, \square \neg o(q)$;

foreach $(q_i, \mathbf{u}, q_j) \in \text{reachable}(\delta)$; // Case 2

if $\neg \text{sat}(o(q_i) \wedge \mathbf{u} \wedge o(q_j)')$ **then**

yield $\perp, \square(o(q_i) \wedge \mathbf{u} \rightarrow \mathcal{X} \neg o(q_j))$;

foreach $(q_i, \mathbf{u}, q_j) \in \text{reachable}(\delta)$; // Case 3

$wp := \text{weakest precondition}(\mathbf{u}, \exists i'. o(q_j)')$;

$wp := \text{quantelim}(wp)$;

if $\text{sat}(o(q_i) \wedge \neg wp)$ **then**

yield $\perp, \square(\neg wp \wedge \mathbf{u} \rightarrow \mathcal{X} \neg o(q_j))$;

return \top ;

Algorithm 1: Check Theory Consistency.

SMT solver to check if the output assignment is consistent with the theory. For example, the two variables $p_{x \geq 5}$ and $p_{x < 0}$ cannot be true in the same state. If such a problem is found we generate a new assumption that rules out this assignment in every state. In the previous example, this would generate the assumption $\square(\neg p_{x \geq 5} \vee \neg p_{x < 0})$.

b) *Transition Consistency:* Once all states produce consistent outputs and there still exists a counterstrategy, we turn towards transitions. As of now, there are no assumptions that link the state before an update was performed to the state afterward. This step checks if there are impossible transitions. We again use an SMT solver to perform this analysis. Let's look at the transition $\{p_{x \geq 5}\}[x \leftarrow x + 1]\{\neg p_{x \geq 5}\}$. To check if the following SMT problem is generated $x \geq 5 \wedge x' = x + 1 \wedge \neg(x' \geq 5)$, this is unsatisfiable and we can generate an assumption to eliminate it $\square(p_{x \geq 5} \wedge [x \leftarrow x + 1] \rightarrow \mathcal{X} p_{x \geq 5})$.

c) *New Predicates:* Another case is that a transition is possible for some, but not all of the values. For instance, the triple $\{p_{x < 0}\}[x \leftarrow x + 1]\{p_{x \geq 0}\}$ does not hold for all values of x . This shows that our current abstraction might not be precise enough to correctly describe this transition and we need an additional predicate. We calculate the weakest precondition of the post state given the updates of the transition. This gives us the predicate $x \geq -1$. If states in the pre-state are not included in the weakest precondition they can not take the transition. The weakest precondition can also be used as the new predicate to distinguish which concrete states may take a transition. In case there are input variables the future inputs are existentially quantified in the post-condition. This results in a natural extension of the weakest precondition, it contains all states that can reach one of the valid post-conditions.

Lemma 2. *If for a Boolean Moore machine M_B Algorithm 1 returns inconsistent together with assumptions ψ every Theory Moore machine M_T — where M_B and M_T share the inputs, state variables, and updates — satisfies ψ .*

Proof. Algorithm 1 can produce three different types of assumptions ψ_s, ψ_t, ψ_p corresponding to the three cases of the algorithm. Let $M_{\mathcal{T}}$ be an arbitrary theory Moore machine. Let ψ_s be $\Box \neg o(q)$ for an unsatisfiable $o(q)$. $M_{\mathcal{T}}$ must define an output valuation for every state because $o(q)$ is empty no state in any $M_{\mathcal{T}}$ can produce such an output. Therefore all $M_{\mathcal{T}}$ satisfy ψ_s .

Let ψ_t be $\Box(o(q_i) \wedge \mathbf{u} \rightarrow \mathcal{X} \neg o(q_j))$ where $\neg \text{sat}(o(q_i) \wedge \mathbf{u} \wedge o(q_j)')$ and $\text{sat}(o(q_i))$. None of the values satisfying $o(q_i)$ have a successor in $o(q_j)$ after performing \mathbf{u} . The added constraint is equivalent to $\neg(o(q_i) \wedge \mathbf{u} \wedge o(q_j)') \Leftrightarrow \neg o(q_i) \vee \neg \mathbf{u} \vee \neg o(q_j)'$ $\Leftrightarrow (o(q_i) \wedge \mathbf{u}) \rightarrow \neg o(q_j)' \Leftrightarrow o(q_i) \wedge \mathbf{u} \rightarrow \mathcal{X} \neg o(q_j)$. All transitions in all $M_{\mathcal{T}}$ satisfy this property at all points in time.

Let ψ_p be $\Box(\neg p \wedge \mathbf{u} \rightarrow \mathcal{X} \neg o(q_j))$ where p is the weakest precondition of $o(q_j)$ under \mathbf{u} and $\text{sat}(o(q_i) \wedge \mathbf{u} \wedge \neg o(q_j))$. By the definition of weakest precondition, no value in $\neg p$ leads to $o(q_j)$ when performing \mathbf{u} . This also holds in the presence of inputs. The quantifier elimination procedure leads to the weakest precondition for unknown inputs at the next time step. All transitions in all $M_{\mathcal{T}}$ will lead from $\neg p$ to $\neg o(q_j)$ when performing \mathbf{u} .

All added constraints are satisfied by all states and transitions in all $M_{\mathcal{T}}$. The constraints only talk about individual states and transitions therefore also all traces in $M_{\mathcal{T}}$ satisfy these constraints and $\forall M_{\mathcal{T}}. M_{\mathcal{T}} \models \psi$. \square

D. Generalizing Counterexamples

The counterexamples generated by Algorithm 1 only block the exact state or transition present in the counterstrategy. To achieve faster and better convergence it is necessary to generalize these counterexamples. Generalization of counterexamples is done using an algorithm to find an unsatisfiable core, i.e., a small (not necessarily minimal) subset of clauses such that their conjunction is unsatisfiable.

This is done for each assumption returned from Algorithm 1 as follows. For $\Box \neg o$ we compute $o_{usc} = \text{unsatcore}(o)$ and produce the generalized assumption $\Box \neg o_{usc}$. For $\Box(o1 \wedge \mathbf{u} \rightarrow \mathcal{X} \neg o2)$ we compute $o1_{usc}, \mathbf{u}_{usc}, o2'_{usc} = \text{unsatcore}(o1 \wedge \mathbf{u} \wedge o2')$ by keeping track of where each conjunct originated. This is then turned back into the generalized assumption $\Box(o1_{usc} \wedge \mathbf{u}_{usc} \rightarrow \mathcal{X} \neg o2_{usc})$.

Using unsat cores in this way allows us to find smaller counterexamples that do not depend on superficial information. Therefore, the counterexamples also block situations where unrelated predicates or updates are different.

V. SYNTHESIS ALGORITHM

A. Synthesis

Our synthesis procedure is shown in Algorithm 2. The procedure starts with a specification in TSL(T) that is translated to an LTL specification as described in Section IV-A. The LTL specification is given to a synthesis tool for propositional LTL. If the synthesis tool finds a realizing system, this system encodes a solution for the TSL(T) synthesis problem. If not, the LTL synthesizer gives us a counterstrategy, which we

Data: TSL(T) specification: ϕ

Result: Satisfying Mealy machine or unrealizable or non-termination

```

while true do
   $\phi_B := \text{prop\_encode}(\phi)$ ;
   $(r, m) := \text{synth}(\phi_B)$ ;
  if  $r$  is UNREALIZABLE then
     $c, \psi := \text{isconsistent}(m)$ ;
    if  $c$  is  $\perp$  then  $\phi := \psi \rightarrow \phi$ ;
    else return UNREALIZABLE;
  else
    return concretize(m);

```

Algorithm 2: Synthesis using abstraction refinement.

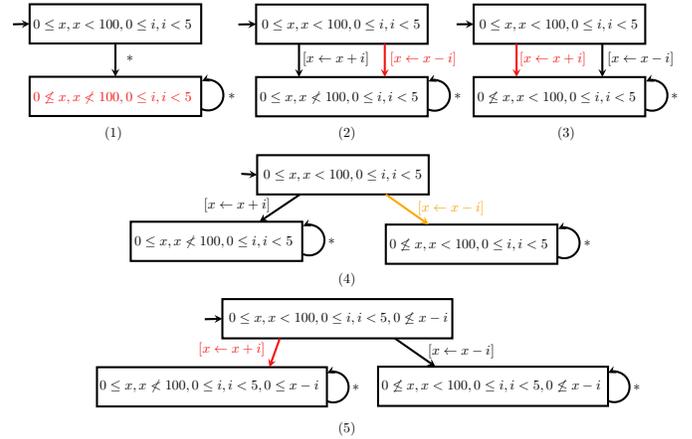


Fig. 4. Synthesis steps for the running example.

analyze to find any inconsistencies with the theory. If the counterstrategy is theory-consistent, it gives us a counterexample to the realizability of the TSL(T) formula. If the theory solver shows that the counterexample is theory-inconsistent, we refine the specification, strengthening it to exclude the inconsistency observed. We illustrate the approach using an example and then prove its correctness.

Example 4. We apply Algorithm 2 to synthesize a system from the specification ϕ given in Example 1. To better illustrate the algorithm we will only introduce one new assumption per iteration. The machines created during execution (5 counterstrategies in the form of Moore machines and one strategy in the form of a Mealy machine) are depicted in Figure 4. We refer to the specification in step n as $\phi_n \triangleq \Box \bigwedge_{k=1}^n \psi_k \rightarrow \phi$ where ψ_k are the assumptions added in step k .

The LTL encoding of $\phi_1 = \phi$ is ϕ_B . (See Example 2) Propositional synthesis results in the counterstrategy shown in Figure 4.1. Algorithm 1 reveals that the second state of this counterexample (shown in red) is inconsistent, because $\neg(0 \leq x) \wedge \neg(x < 100)$ is unsatisfiable. We obtain the new assumption $\psi_1 \triangleq \Box(0 \leq x \vee x < 100)$. Note that this is a more general assumption than just the negated state formula.

Attempting to synthesize a system for ϕ_1 (we will omit the LTL encoding step from now on) results in the counterstrat-

egy shown in Figure 4.2. This time, all outputs are consistent, but the transition $[x \leftarrow x - 1]$ is inconsistent: If $x < 100$ and $0 \geq i$ and we apply the update $[x \leftarrow x - i]$, it cannot be that $x \not< 100$ in the following state. We obtain the assumption $\psi_2 \triangleq \square(x < 100 \wedge 0 \leq i \wedge [x \leftarrow x - i] \rightarrow \mathcal{X} x < 100)$.

Boolean synthesis for ϕ_2 results in a similar counterstrategy (Figure 4.3), this time the transition $[x \leftarrow x + i]$ is inconsistent and $\psi_3 \triangleq \square(0 \leq x \wedge 0 \leq i \wedge [x \leftarrow x + i] \rightarrow \mathcal{X} 0 \leq x)$.

Synthesis for ϕ_3 leads to the counterstrategy shown in Figure 4.4. The first two consistency checks pass, but case 3 reports that the transition $[x \leftarrow x - i]$ is only valid for some of the possible values in the first state. We learn the new predicate $0 \leq x - i$ and the assumption $\psi_4 \triangleq \square(0 \leq x - i \wedge [x \leftarrow x - i] \rightarrow \mathcal{X} 0 \leq x) \wedge \square \neg(0 \leq i \wedge 0 \leq x - i \wedge 0 \not\leq x)$. This includes the assumption obtained from the transition as well as one to prevent state inconsistencies with the new predicate¹.

Running the Boolean synthesis algorithm again results in the counterstrategy in 4.5. The transition $[x \leftarrow x + i]$ is inconsistent and we add $\psi_5 \triangleq \square(0 \not\leq x - i \wedge i < 5 \wedge [x \leftarrow x + i] \rightarrow \mathcal{X} x < 100)$.

Boolean synthesis is executed for the last time on ϕ_5 . This time, the synthesis tool produces a Boolean system satisfying the specification this corresponds to the system shown in Figure 1.

Theorem 1. *Any Boolean system M_B returned by Algorithm 2 can be converted to a theory system M_T that satisfies ϕ .*

Proof. To obtain M_T an initial valuation \mathbf{r}_0 that satisfies ϕ at point 0 has to be chosen. All other components are the same as in M_B . Let $\phi' = \psi \rightarrow \phi$ be the last specification used in the algorithm and ψ all the added assumptions. We know $M_B \models \phi'_B$ and that M_B and M_T share the same extended traces. Therefore, M_T satisfies ϕ' . From Lemma 2 follows that $M_T \models \psi$. Thus M_T satisfies ϕ . \square

Even though our algorithm is not guaranteed to terminate it can prove unrealizability in certain cases. For $x = 0 \rightarrow \square([x \leftarrow x + 1] \wedge x < 3)$ we can perform two refinement steps and learn the new predicates $x \geq 2$ and $x \geq 1$. Using these the propositional synthesis tool can build a consistent environment strategy. There are no conflicts that could be used to further refine the specification. This shows that the specification is unrealizable.

Theorem 2. *If Algorithm 2 returns unrealizable there is no $M_T \models \phi$ and ϕ is unrealizable by machines using the updates U .*

Proof. If there exists a machine $M'_T \models \neg\phi$ there is no machine $M_T \models \phi$. The propositional synthesis tool provides us with a machine $M_B \models \neg\phi'$ where $\phi' = \psi \rightarrow \phi$ for assumptions ψ . The consistency check results in consistent, so by Lemma 1 there exists a $M'_T \models \neg\phi'$. According to Lemma 2 ψ is satisfied by M'_T . Therefore, $M'_T \models \neg\phi$ and no $M_T \models \phi$. \square

¹We include this here instead of in its own step to simplify the example.

B. Limitations

Our algorithm is not guaranteed to terminate. In Section VI we discuss multiple specifications that can be successfully synthesized or where unrealizability can be shown. In this section, we show two exemplar cases for which our algorithm will not terminate.

Our algorithm cannot handle reachability properties where the number of required steps depends on the concrete value of a state variable and is unbounded. The specification $0 \leq x \rightarrow (\diamond(x < 0) \wedge \square([x \leftarrow x + 1] \vee [x \leftarrow x - 1]))$ with the state variable x is an example of this. The specification is obviously realized by a system always using the update $[x \leftarrow x - 1]$. However, we would add the new predicates $x \geq 1, x \geq 2, \dots$ without terminating.

A similar problem can occur for unrealizability. For $x = 1 \rightarrow (\diamond(x = 0) \wedge \square[x \leftarrow x + 1])$ we learn the predicates $x = -1, x = -2, \dots$ without terminating. However, the predicate $x > 1$ would allow us to prove unrealizability.

C. Decidable fragment

Theorem 3. *The TSL synthesis problem for the theory of equality is decidable.*

Algorithm 2 will always terminate if the set of predicates that Algorithm 1 can generate is finite. For a finite set of predicates the assumptions that can be added by cases, one and two are also finite. Since the assumptions block the counterstrategy from reappearing this means there can not be infinitely many counter strategies and the synthesis algorithm will terminate with the correct answer. The theory of equality only allows updates to move values. Iterating the weakest precondition can only create finitely many predicates, for the equality theory also the quantifier elimination cannot introduce new constants. Thus the number of possible predicates is finite and the problem is decidable.

VI. EXPERIMENTAL EVALUATION

We implemented our algorithm in our tool Raboniel². Our implementation relies on several external tools: ts-tools [6] is used for parsing TSL and to perform the propositional encoding, strix [8] is used for LTL synthesis, and Z3 [11] is used as the SMT solver. When performing counterexample analysis using Algorithm 1 we add all assumptions from the same case before we start the next iteration. The obtained theory Mealy machine can be compiled into a Python program.

A. Extended running example

The first experiment is an extension of Example 4. We change two parameters in the specification. The system is no longer allowed to change between the two updates at every step. Instead after changing the update, it has to use the new update for the next c steps. This shows how our algorithm deals with more complex temporal properties. We also varied the size of the intervals for x and i demonstrating that our algorithm is independent of the size of the concrete state

²<https://doi.org/10.5281/zenodo.5647461>

TABLE I
RESULTS FOR THE EXTENDED RUNNING EXAMPLE.

c	x_{max}	i_{max}	# refine.	# states	# new pred.	time [s]
1	100	5	4	1	2	1.0
2	100	5	5	2	2	1.3
2	100 000	50	5	2	2	1.3
3	100	5	9	2	4	2.9
3	100 000	50	10	2	4	3.1
1	100	110	6	unreal.	3	1.0
2	100	60	4	unreal.	2	0.8

space. The results are listed in table I including the used parameters (c , x_{max} , i_{max}), the number of refinements, the number of states in the minimized system, the number of learned predicates during the whole execution and the total run time in seconds. The table includes realizable as well as unrealizable configurations. The different ranges for x and i show how our approach can handle state spaces symbolically, it behaves the same whether there are 100 or 100000 concrete states, these number of concrete states is also far above what can be solved with explicit states in LTL. The larger values of c require the system to plan further ahead by limiting how often it can switch its output, this requires a second state and a few additional predicates.

B. Elevator

A classic example for reactive synthesis is a controller for an elevator. The single state variable $floor$ represents the current position of the elevator. It can start anywhere between the first floor and the maximum floor and is not allowed to leave this interval. The controller has three options: move the elevator up or down or stay in the same position. Every floor has to be visited infinitely often. The results are shown in Table II as type simple. We varied the number of floors of the building to show how our algorithm scales with more complex specifications. No new predicates are learned as a sufficient number of predicates is already included in the specification (equality tests for every floor are part of the liveness properties). The required time seems to grow exponentially with the growing number of floors. This leads to growing propositional synthesis problems (which is worst case double-exponential). The number of states stays constant because most of the complexity is part of the predicates e.g. the position of the elevator. The overall time is still reasonable even for a large number of floors.

A different version of this specification is shown in Table II as type signal. In this version, the environment controls a variable $signal$ to select the floor the elevator has to reach which is stored in the state variable $target$. This results in a more complex specification with worse run time, which is dominated by propositional synthesis.

C. Cyber-Physical Systems

The previous examples all used linear integer arithmetic. We can also use other SMT theories like linear real arithmetic (LRA). Using reals allows us to model linear cyber-physical systems. This example is inspired by Belta et al. [12] chapter

TABLE II
RESULTS FOR THE ELEVATOR.

type	# floors	# refinements	# states	time [s]
simple	3	13	2	3.1
simple	4	11	3	3.7
simple	5	15	4	8.2
simple	8	21	4	45
simple	10	24	4	185
signal	3	5	1	35
signal	4	5	1	217
signal	5	6	1	1424

9, a system of two coupled water tanks with linear dynamics; one water tank drains (x_2) and the other one (x_1) is refilled by the controller. An illustration is depicted in Figure 5. We discretize the inputs (refill tank x_1) with two values (0 and 0.0003), represented as different updates. We created two variants of the system. The first one is a safety specification where the water level of both tanks has to be kept between 0.1 and 0.7.

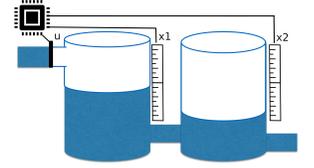


Fig. 5. Water tanks system

Synthesis of this system takes 31 seconds and 4 refinements. The resulting system only has a single state, but 13 new predicates where required.

The second version consists of only one water tank, but requires the liveness property: whenever the water level falls below 0.1 it has to eventually exceed 0.4. A system realizing that specification can be synthesized using 18 refinements in 95 seconds (9 new predicates), it consists of 2 states. These examples demonstrate that our tool can handle updates with more complex operations. This leads to a large number of new predicates, but can still be synthesized in less than two minutes.

D. Comparison with Related Work

The TSL paper by Finkbeiner et al. [6] contains various examples of TSL specifications. However, most of them do not require any refinement and the first LTL approximation is already realizable. For these examples, our tool would perform the same, because no theory refinement is used. A small number of examples required refinement. We converted two of them by replacing the uninterpreted functions for increment and decrement with native integer operations. The implementation of their refinement approach is not publicly available, we thus compare our results to the numbers reported in their paper. The experiment “TwoCountersInRange” took our tool 8 refinements and 1677s compared to their 173s. The experiment “OneCounterGUI” took our tool only 4 refinements and 17.2s, this is a factor 100 faster than their 1767s. These results suggest that both tools can outperform the other by an order of magnitude, depending on the example.

We also compared our tool using a benchmark set of safety games on infinite grid worlds that was introduced by Neider and Topcu [13]. We compare with the following tools: the

TABLE III

INFINITE GRID WORLD BENCHMARKS FROM [13]. ALL TIMES IN SECONDS
 — DENOTES A TIME-OUT AFTER 900S. TOOL ABBREVIATIONS: C
 (CONSYNTH), J (JSYN-VG), D (DT-SYNTH), S (SAT-SYNTH), R
 (RPNI-SYNTH), G (GENSYS)

Benchmark	C	J	D	S	R	G	Raboniel
Box	3.7	0.6	0.3	0.3	0.1	0.3	1.9
Box Limited	0.4	1.7	0.1	0.4	0.5	0.2	0.6
Diagonal	1.9	4.0	2.4	1.3	0.5	0.2	10.9
Evasion	1.5	0.5	0.2	81	0.1	0.7	5.4
Follow	—	1.2	0.3	88.9	—	0.7	—
Solitary Box	0.4	0.9	0.1	0.3	0.1	0.3	0.5
Square 5x5	—	6.5	2.5	0.6	0.2	0.3	75.1

logic-based synthesis tools ConSynth [14], JSyn-VG [15], and GenSys [16]; the automata-learning-based tools SAT-Synth and RPNI-Synth [13]; as well as the decision-tree-learning tool DT-Synth [17]. These experiments are shown in Table III our tool is listed as Raboniel, the results for the other tools are reproduced from [16]. Our tool is able to solve 6 out of the 7 benchmarks within 15 minutes. The execution time of our tool is on the lower end of the spectrum. However, TSL(T) allows us to express and handle more sophisticated specifications. Most other tools (except ConSynth and JSyn-VG) only support safety properties and would not be able to handle the other examples shown in this paper.

VII. RELATED WORK

The first paper on TSL [6] introduces this logic as a way to do synthesis while separating control flow and data processing. Reactive synthesis is used to build a control flow model which describes how the uninterpreted functions are combined and which of them is used when based on a logic circuit. This model can then be instantiated and translated to a functional reactive program (FRP)[18]. Our approach has less separation of data and control, by supporting theories we can reason about a lot of operations and construct systems that would not be possible using uninterpreted functions. We also directly create executable code without the intermediary FRP. Another major difference is the analysis of counter strategies. Finkbeiner et al. use an algorithm specific to uninterpreted functions that checks all possible traces up to a certain length. We have shown that consistency checking can also be done by local checks in a theory-independent way. That way we can also learn new predicates which allow for the synthesis of otherwise impossible specifications and prove unrealizability.

The extension to TSL(T) was first done by Finkbeiner et al. [7] they study uninterpreted functions and Presburger arithmetic and provide a search-based algorithm to check satisfiability. However, they did not look into synthesis.

Another recent extension of TSL is by Choi et al. [19]. They describe a different approach to adding arithmetic to TSL using syntax-guided synthesis (SyGuS) [20]. The TSL formula is translated into sequential SyGuS problems and the solutions are used to create assumptions. This technique cannot create new predicates and thus will not be able to solve problems

such as our running example. Their solution was developed independently and in parallel to our approach.

Other techniques for reactive synthesis beyond Booleans are: Reactive synthesis from register automata specifications has been studied[2], [4], [3], [21]. These models allow comparison (equality/inequalities) of data values, but no operations. Multiple decidable fragments have been identified. Another approach uses variable automata [5], [22] specifications these can perform arithmetic the authors also identified a decidable fragment. While for both register and variable automata strong theoretical results have been achieved we are not aware of any empirical evaluations. There are also synthesis tools that specifically target cyber-physical systems [23], [24] these often rely on finite or receding horizons instead of infinite traces. counterexample guided methods have also been used for program synthesis [25] and model synthesis [26].

VIII. CONCLUSION AND FUTURE WORK

The algorithm presented in this paper performs specification refinement in a pure lazy way. That is new assumptions are only added when they are encountered in a counterexample. Performing some analysis upfront and after learning new predicates has the potential to significantly improve the run time. Testing for incompatibilities between predicates would be an obvious target for this. Another extension would be new strategies for learning predicates and heuristics to prevent learning unnecessary predicates (slowing down propositional synthesis).

We presented a synthesis procedure for temporal stream logic modulo theories. Our algorithm is based on a CEGAR [10] loop and translation to propositional LTL synthesis. The synthesis problem for TSL modulo theories, in general, is undecidable. However, we can synthesize systems or prove unrealizability in many cases. Huge state spaces can be handled by using a symbolic representation during synthesis. Some specifications require new predicates, in many cases, we are able to automatically find these.

REFERENCES

- [1] R. Bloem, K. Chatterjee, and B. Jobstmann, “Graph games and reactive synthesis,” in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 921–962.
- [2] R. Ehlers, S. A. Seshia, and H. Kress-Gazit, “Synthesis with identifiers,” in *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, ser. Lecture Notes in Computer Science, K. L. McMillan and X. Rival, Eds., vol. 8318. Springer, 2014, pp. 415–433.
- [3] A. Khalimov, B. Maderbacher, and R. Bloem, “Bounded synthesis of register transducers,” in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11138. Springer, 2018, pp. 494–510.
- [4] L. Exibard, E. Filiot, and A. Khalimov, “Church synthesis on register automata over linearly ordered data domains,” in *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, ser. LIPIcs, vol. 187. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 28:1–28:16.

- [5] R. Faran and O. Kupferman, "On synthesis of specifications with arithmetic," in *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*, ser. Lecture Notes in Computer Science, A. Chatzigeorgiou, R. Dondi, H. Herodotou, C. A. Kapoutsis, Y. Manolopoulos, G. A. Papadopoulos, and F. Sikora, Eds., vol. 12011. Springer, 2020, pp. 161–173.
- [6] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, "Temporal stream logic: Synthesis beyond the booleans," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 11561. Springer, 2019, pp. 609–629.
- [7] B. Finkbeiner, P. Heim, and N. Passing, "Temporal stream logic modulo theories," in *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, ser. Lecture Notes in Computer Science, P. Bouyer and L. Schröder, Eds., vol. 13242. Springer, 2022, pp. 325–346.
- [8] P. J. Meyer, S. Sickert, and M. Luttenberger, "Strix: Explicit reactive synthesis strikes back!" in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 578–586.
- [9] S. Schewe and B. Finkbeiner, "Bounded synthesis," in *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*, ser. Lecture Notes in Computer Science, K. S. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, Eds., vol. 4762. Springer, 2007, pp. 474–488.
- [10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1855. Springer, 2000, pp. 154–169.
- [11] L. De Moura and N. Björner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.
- [12] C. Belta, B. Yordanov, and E. A. Gol, *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.
- [13] D. Neider and U. Topcu, "An automaton learning approach to solving safety games over infinite graphs," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 9636. Springer, 2016, pp. 204–221.
- [14] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko, "A constraint-based approach to solving games on infinite graphs," in *POPL*. ACM, 2014, pp. 221–234.
- [15] A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen, "Validity-guided synthesis of reactive systems from assume-guarantee contracts," in *TACAS (2)*, ser. Lecture Notes in Computer Science, vol. 10806. Springer, 2018, pp. 176–193.
- [16] S. Samuel, D. D'Souza, and R. Komondoor, "Gensys: a scalable fixed-point engine for maximal controller synthesis over infinite state spaces," in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 1585–1589.
- [17] D. Neider and O. Markgraf, "Learning-based synthesis of safety controllers," in *FMCAD*. IEEE, 2019, pp. 120–128.
- [18] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, "Synthesizing functional reactive programs," in *Haskell@ICFP*. ACM, 2019, pp. 162–175.
- [19] W. Choi, B. Finkbeiner, R. Piskac, and M. Santolucito, "Can reactive synthesis and syntax-guided synthesis be friends?" in *PLDI*. ACM, 2022, pp. 229–243.
- [20] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*. IEEE, 2013, pp. 1–8.
- [21] L. Exibard, E. Filiot, and P. Reynier, "Synthesis of data word transducers," *Log. Methods Comput. Sci.*, vol. 17, no. 1, 2021.
- [22] R. Faran and O. Kupferman, "LTL with arithmetic and its applications in reasoning about hierarchical systems," in *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, ser. EPIC Series in Computing, G. Barthe, G. Sutcliffe, and M. Veanes, Eds., vol. 57. EasyChair, 2018, pp. 343–362.
- [23] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*, A. Girard and S. Sankaranarayanan, Eds. ACM, 2015, pp. 239–248.
- [24] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "Tulip: a software toolbox for receding horizon temporal logic planning," in *HSCC*. ACM, 2011, pp. 313–314.
- [25] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett, "Counterexample-guided quantifier instantiation for synthesis in SMT," in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 9207. Springer, 2015, pp. 198–216.
- [26] M. Preiner, A. Niemetz, and A. Biere, "Counterexample-guided model synthesis," in *TACAS (1)*, ser. Lecture Notes in Computer Science, vol. 10205, 2017, pp. 264–280.

Learning Deterministic Finite Automata Decompositions from Examples and Demonstrations

Niklas Lauffer^{*} , Beyazit Yalcinkaya^{*} , Marcell Vazquez-Chanlatte , Ameesh Shah, and Sanjit A. Seshia 

University of California, Berkeley, CA, USA

{nlauffer, beyazit, marcell.vc, ameesh, ssesia}@berkeley.edu

Abstract—The identification of a *deterministic finite automaton* (DFA) from labeled examples is a well-studied problem in the literature; however, prior work focuses on the identification of monolithic DFAs. Although monolithic DFAs provide accurate descriptions of systems’ behavior, they lack simplicity and interpretability; moreover, they fail to capture sub-tasks realized by the system and introduce inductive biases away from the inherent decomposition of the overall task. In this paper, we present an algorithm for learning conjunctions of DFAs from labeled examples. Our approach extends an existing SAT-based method to systematically enumerate Pareto-optimal candidate solutions. We highlight the utility of our approach by integrating it with a state-of-the-art algorithm for learning DFAs from demonstrations. Our experiments show that the algorithm learns sub-tasks realized by the labeled examples, and it is scalable in the domains of interest.

I. INTRODUCTION

Grammatical inference is a mature and well-studied field with many application domains ranging from machine learning to computational biology [1]. The identification of a minimum size *deterministic finite automaton* (DFA) from labeled examples is one of the most well-investigated problems in this field. Furthermore, with the increase in computational power in recent years, the problem can be solved efficiently by various tools available in the literature (e.g., [2], [3]).

Existing work on DFA identification primarily focuses on the monolithic case, i.e., learning a single DFA from examples. Although such DFAs capture a language consistent with the examples, they may lack simplicity and interpretability. Furthermore, complex tasks often decompose into independent sub-tasks. However, monolithic DFA identification fails to capture the natural decomposition of the system behavior, introducing an inductive bias away from the inherent decomposition of the overall task. In this paper, we present an algorithm for learning *DFA decompositions* from examples by reducing the problem to graph coloring in SAT and a Pareto-optimal solution search over candidate solutions. A DFA decomposition is a set of DFAs such that the *intersection* of their languages is the language of the system, which implicitly defines a conjunction of simpler specifications realized by the overall system.¹We present an application of our algorithm to a state-of-the-art method for learning task specifications from

unlabeled demonstrations [4] to showcase a domain of interest for DFA decompositions.

Related Work. Existing work considers the problem of minimal DFA identification from labeled examples [1]. It is shown that the DFA identification problem with a given upper bound on the number of states is an NP-complete problem [5]. Another work shows that this problem cannot be efficiently approximated [6]. Fortunately, practical methods exist in the literature. A common approach is to apply the evidence driven state-merging algorithm [7], [8], [9], which is a greedy algorithm that aims to find a good local optimum. Other works for learning DFAs use evolutionary computation [10], [11], later improved by multi-start random hill climbing [12].

A different approach to the monolithic DFA identification is to leverage highly-optimized modern SAT solvers by encoding the problem in SAT [13]. In follow up works, several symmetry breaking predicates are proposed for the SAT encoding to reduce the search space [3], [14], [15], [16]. However, to the best of our knowledge, no work considers directly learning DFA decompositions from examples and demonstrations.

This work also relates to the problem of decomposing a known automaton. Ashar et al. [17] explore computing cascade and general decomposition of finite state machines. The Krohn–Rhodes theorem [18] reduces a finite automaton into a cascade of irreducible automata. Kupferman & Mosheiff [19] present various complexity results for DFA decomposability.

Finally, the problem of learning objectives from demonstrations of an expert dates back to the problem of Inverse Optimal Control [20] and, more recently in the artificial intelligence community, the problem of Inverse Reinforcement Learning (IRL) [21]. The goal in IRL is to recover the unknown reward function that an expert agent is trying to maximize based on observations of that expert. Recently, several works have considered a version of the IRL problem in which the expert agent is trying to maximize the satisfaction of a Boolean task specification [22], [23], [4]. However, no work considers learning *decompositions* of specifications from demonstrations.

II. PROBLEM FORMULATION

Let \mathcal{D} denote the set of DFAs over some fixed alphabet Σ . An (m_1, \dots, m_n) -DFA decomposition is a tuple of n DFAs $(\mathcal{A}_1, \dots, \mathcal{A}_n) \in \mathcal{D}^n$ where \mathcal{A}_i has m_i states and

¹Our algorithm and SAT encoding can easily be generalized to unions or even arbitrary Boolean combinations of DFAs.

^{*} Equal contribution

This work was partially supported by NSF grants 1545126 (VeHiCaL) and 1837132, by the DARPA contracts FA8750-18-C-0101 (Assured Autonomy) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, by Toyota under the iCyPhy center, and by Toyota Research Institute.

$m_1 \leq m_2 \leq \dots \leq m_n$. We associate a partial order \prec on DFA decompositions using the standard product order on the number of states. That is, $(\mathcal{A}'_1, \dots, \mathcal{A}'_n) \prec (\mathcal{A}_1, \dots, \mathcal{A}_n)$, if $m'_i \leq m_i$ for all $i \in [n]$ and $m'_j < m_j$ for some $j \in [n]$. In this case, we say $(\mathcal{A}'_1, \dots, \mathcal{A}'_n)$ *dominates* $(\mathcal{A}_1, \dots, \mathcal{A}_n)$. A DFA decomposition $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ *accepts* a string w iff all \mathcal{A}_i accept w . A string that is not accepted is *rejected*. The *language* of a decomposition, $\mathcal{L}(\mathcal{A}_1, \dots, \mathcal{A}_n)$, is the set of accepting strings, i.e., the intersection of all DFA languages.

In order to bias towards “simpler” solutions, we further extend the partial order \prec over equally sized (i.e., if $m'_i = m_i$ for all $i \in [n]$) decompositions by letting $(\mathcal{A}'_1, \dots, \mathcal{A}'_n) \prec (\mathcal{A}_1, \dots, \mathcal{A}_n)$ if $(\mathcal{A}'_1, \dots, \mathcal{A}'_n)$ has fewer total non-stuttering edges than $(\mathcal{A}_1, \dots, \mathcal{A}_n)$.

We study the problem of finding a DFA decomposition from a set of positive and negative labeled examples such that the decomposition accepts the positive examples and rejects the negative examples. We start by formally defining *the DFA decomposition identification problem* (DFA-DIP), and then presenting an overview of the proposed approach.

The Deterministic Finite Automaton Decomposition Identification Problem (DFA-DIP). Given positive examples, D_+ and negative examples, D_- , and a natural number $n \in \mathbb{N}$, find a (m_1, \dots, m_n) -DFA decomposition $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ satisfying the following conditions.

(C1) The decomposition is consistent with (D_+, D_-) :

$$\begin{aligned} D_+ &\subseteq \mathcal{L}(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n), \\ D_- &\subseteq \Sigma^* \setminus \mathcal{L}(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n). \end{aligned}$$

(C2) There does not exist a DFA decomposition that *dominates* $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ and satisfies (C1).

We refer to the set of DFA decompositions that solve an instance of DFA-DIP as the Pareto-optimal frontier of solutions. Note that for $n = 1$, DFA-DIP reduces to monolithic DFA identification. We propose finding the set of DFA decompositions that solve DFA-DIP by reduction to graph coloring in SAT and a breadth first search in solution space. Specifically, we extend the existing work on SAT-based monolithic DFA identification [13], [15] to finding n DFAs with m_1, \dots, m_n states and q non-stuttering edges such that the intersection of their languages is consistent with the given examples. On top of this SAT-based approach, we develop a search strategy over the numbers of states and edges passed to the SAT solver as these values are not known a priori.

III. LEARNING DFAS FROM EXAMPLES²

In this section, we present the proposed approach. We start with the SAT encoding of the DFA decomposition problem and continue with the Pareto frontier search in the solution space. We then showcase an example of learning conjunctions of DFAs from labeled examples. Finally, we present experimental results and evaluate the scalability of our method.

²Our MIT licensed code is freely available at [24].

A. Encoding DFA-DIP in SAT

We extend the SAT encoding for monolithic DFA identification presented in [13], [15], which solves a graph coloring problem, to finding n DFAs with m_1, m_2, \dots, m_n states. The extension relies on the observation that for conjunctions of DFAs, we need to enforce that a positive example must be accepted by *all* DFAs, and a negative example must be rejected by *at least one* of the DFAs. Due to space limitations, we only present the modified clauses of the encoding, and invite reader to Appendix A of the extended version of the paper [25] for further details.

The encoding works on an *augmented prefix tree acceptor* (APTA), a tree-shaped automaton with nodes corresponding to prefixes and edges to appending letters, constructed from given examples, which has paths for each example leading to accepting or rejecting states based on the example’s label; therefore, an APTA defines D_+ and D_- which then constrains the accepting states, rejecting states, and the transition function of the unknown DFAs. For each DFA, \mathcal{A}_i , the encoding will associate the APTA states with one of the m_i colors for DFA \mathcal{A}_i , subject to the constraints imposed by D_+ and D_- . APTA states with the same (DFA-indexed) color will be the same state in the corresponding DFA. We refer to states of an APTA as V , its accepting states as V_+ , and its rejecting states as V_- . Given n for the number of DFAs, m_1, \dots, m_n for the number of states of DFAs, and q for the number of non-stuttering edges, the SAT encoding uses three types of variables:

- 1) *color* variables $x_{v,i}^k \equiv 1$ ($k \in [n]$; $v \in V$; $i \in [m_k]$) iff APTA state v has color i in DFA k ,
- 2) *parent relation* variables $y_{l,i,j}^k \equiv 1$ ($k \in [n]$; $l \in \Sigma$, where Σ is the alphabet; $i, j \in [m_k]$) iff DFA k transitions with symbol l from state i to state j , and
- 3) *accepting color* variables $z_i^k \equiv 1$ ($k \in [n]$; $i \in [m_k]$) iff state i of DFA k is an accepting state.

The encoding for the monolithic DFA identification also uses the same variable types; however, in our encoding, we also index variables over n DFAs instead of a single DFA. With this extension, one can trivially instantiate the encoding presented in [13], [15]. Below, we list the new rules we define for our problem. For the complete list of rules, see Appendix A of the extended version of the paper [25].

(R1) A negative example must be rejected by *at least one* DFA:

$$\bigwedge_{v \in V_-} \bigvee_{k \in [n]} \bigwedge_{i \in [m_k]} x_{v,i}^k \implies \neg z_i^k.$$

(R2) Accepting and rejecting states of APTA cannot be merged:

$$\bigwedge_{v_- \in V_-} \bigwedge_{v_+ \in V_+} \bigwedge_{k \in [n]} \bigwedge_{i \in [m_k]} (x_{v_-,i}^k \wedge \neg z_i^k) \implies \neg x_{v_+,i}^k.$$

(R3) Upperbound on the number of non-stuttering edges:

$$\sum_{k \in [n]} \sum_{l \in \Sigma} \sum_{i, j \in [m_k], i \neq j} y_{l,i,j}^k \leq q.$$

In the encoding of [13], [15], we replace the rule stating that the resulting DFA must reject all negative examples with **(R1)**, and **(R2)** is used instead of the original rule stating that accepting and rejecting states of APTA cannot be merged. Notice that since a rejecting state of APTA is not necessarily a rejecting state of a DFA k , we need to use the new rule **(R2)**. Finally, **(R3)** enables controlling the maximum number of non-stuttering transitions. As we shall see, this will enable us to satisfy **(C2)**.

Theorem 1. *Given labeled examples, n for the number of DFAs, m_1, \dots, m_n for the number of states of DFAs, and q for the number non-stuttering edges, a solution to the above SAT encoding satisfies **(C1)** of DFA-DIP.*

Proof: We assume that the SAT-based reduction to graph coloring for monolithic DFA identification given in [13] is correct. Next, observe that **(R3)** can only remove solutions and thus does not effect **(C1)**. Constraint **(R1)** and **(R2)** replace similar constraint in the monolithic encoding given in [13]:

(R1') a negative example must be rejected by the DFA:

$$\bigwedge_{v \in V_-} \bigwedge_{i \in [m_k]} x_{v,i} \implies \neg z_i, \text{ and}$$

(R2') accepting and rejecting states of the APTA cannot be merged:

$$\bigwedge_{v_- \in V_-} \bigwedge_{v_+ \in V_+} \bigwedge_{i \in [m_k]} x_{v_-,i} \implies \neg x_{v_+,i}.$$

In the monolithic DFA case, there is only a single DFA so for ease of notation, we drop the index k . First notice that constraints **(R1')** and **(R2')** have no bearing on whether the DFA accepts each positive example. Therefore, our encoding automatically requires that each DFA in the DFA decomposition accepts all of the positive examples and is not constrained to unnecessarily accept any unspecified examples.

Constraint **(R1')** ensures that the resulting monolithic DFA rejects every negative example by making the color of the node in the APTA associated with the negative example rejecting. Constraint **(R1)** replaces this and ensures that at least one of the DFAs in the DFA decomposition rejects a negative example by making the color of the node in the APTA associated with the negative example rejecting in at least one of the n DFAs in the decomposition. Thus, the language intersection of the resulting decomposition correctly rejects negative examples.

Constraint **(R2')** ensures that all pairs of rejecting and accepting nodes of the APTA cannot be assigned the same color (i.e., merged) in the resulting DFAs. Constraint **(R2)**, which replaces **(R2')**, ensures that for each DFA in the decomposition, the pair $(x_{v_-,i}^k, x_{v_+,i}^k)$ of accepting and rejecting nodes of the APTA cannot be assigned the same color only if DFA k is rejecting the negative example associated with $x_{v_-,i}^k$ (which is handled by constraint **(R1)**). This allows all but one DFA in the DFA decomposition to accept negative examples. Therefore, the language of the decomposition is not constrained to reject any unspecified examples.

Algorithm 1 Pareto frontier enumeration algorithm.

Require: Positive D_+ and negative D_- labeled examples and positive integer n .

- 1: $(P^*, Q) \leftarrow \{(1, \dots, 1)\}$ ▷ Initial Pareto front and queue.
- 2: **while** $Q \neq \emptyset$ **do**
- 3: $m \leftarrow Q.dequeue()$
- 4: **if** $\nexists \hat{m} \in P^*$ s.t. $\hat{m} \prec m$ **then**
- 5: $SAT, \mathcal{A} \leftarrow SOLVE(n, m, D_+, D_-)$ ▷ Omits **(R3)**.
- 6: **if** SAT **then**
- 7: $P^* = P^* \cup \mathcal{A}$ ▷ Add to the Pareto frontier.
- 8: **else**
- 9: **for** $k = 1, \dots, n$ **do**
- 10: $(m', m'_k) \leftarrow (m, m'_k + 1)$
- 11: **if** $ordered(m')$ **then** $Q.enqueue(m')$
- 12: **return** $minimize_stutter(P^*)$ ▷ Binary search using **(R3)**.

B. Pareto Frontier Search

The SAT encoding detailed in section III-A produces a DFA decomposition that satisfies **(C1)**, but not necessarily **(C2)**. In this section, we provide the details of the Pareto frontier enumeration algorithm that uses the SAT encoding as an inner loop to find a DFA decomposition that solves DFA-DIP.

Our proposed Pareto frontier enumeration algorithm is a breadth first search (BFS) over DFA decomposition size tuples that skips tuples that are dominated by an existing solution. This BFS is over a directed, acyclic graph $G = (V, E)$ formed in the following way. There is a vertex in the graph for every ordered tuple of states sizes. There is an edge from (m_1, m_2, \dots, m_n) to $(m'_1, m'_2, \dots, m'_n)$ if there exists some $j \in [n]$ such that:

$$m'_i = \begin{cases} m_i + 1 & \text{if } i = j; \\ m_i & \text{otherwise.} \end{cases}$$

A size tuple (m_1, \dots, m_n) is a sink, i.e., the search does not continue past this vertex, if there exists a (m_1, \dots, m_n) -decomposition that solves DFA-DIP or the size tuple is dominated by a previously traversed solution. In the prior case, the associated DFA decomposition is also returned as a solution on the Pareto-optimal frontier. The BFS starts from $m_1 = m_2 = \dots = m_n = 1$, and performs the search as explained. Algorithm 1 presents the details of the BFS performed in the solution space for finding the Pareto frontier.

After finding a minimal number of states m_1, m_2, \dots, m_n that solve the problem, there still might exist multiple DFA decompositions of that size that solve **(C1)**. These ties are broken in favor of DFA decompositions that have the fewest total non-stuttering edges, q . For each minimal dfa this is done by a binary search over q and denoted: $minimize_stutter(\bullet)$.

Theorem 2. *Algorithm 1 is sound and complete; it outputs the full Pareto-optimal frontier of solutions without returning any dominated solutions, therefore satisfying **(C2)** of DFA-DIP.*

■ *Proof:* See the extended version of the paper [25]. ■

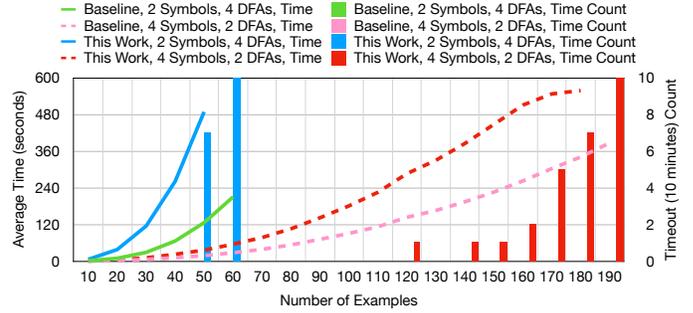
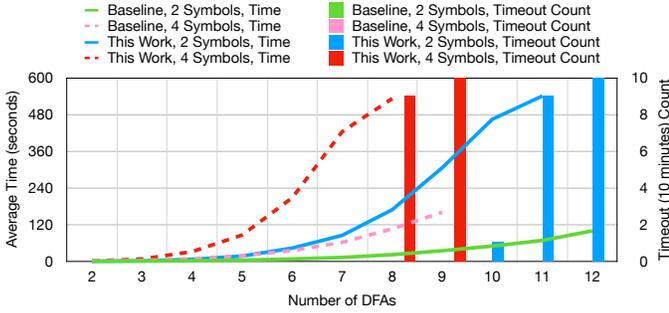


Fig. 1. Experiment results evaluating the scalability of our algorithm w.r.t. (a) number of DFAs implied by the examples and (b) number of labeled examples.

C. Example: Learning Partially-Ordered Tasks

We continue with a toy example showcasing the capabilities of the proposed approach. Later, we use the same class of decompositions to evaluate the scalability of our algorithm.

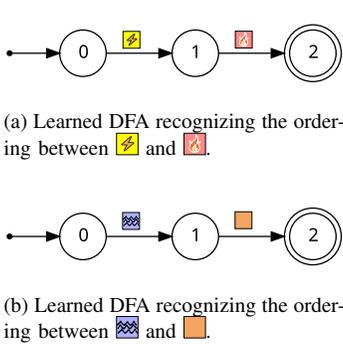


Fig. 2. Learned DFA decomposition.

Inspired from the multi-task reinforcement learning literature [26], our example focuses on partially-ordered temporal tasks executed in parallel. Specifically, consider a case where an agent is performing two ordering tasks in parallel: (i) observe yellow before red, and (ii) observe blue before orange. A positive example of such behavior is simply any sequence of observations ensuring both of the given orderings, e.g. yellow red blue orange, and a negative example is any sequence that fails to satisfy both orderings, e.g. yellow orange blue red. We generate such positive and negative examples and feed them to our algorithm. Figure 2 presents the learned DFAs recognizing ordering sub-tasks of the example. The intersection of their languages is consistent with the given observations, and their conjunction is the overall task realized by the system generating the traces. The monolithic DFA recognizing the same language has nine states, and is more complicated (see Figure 4 in Appendix C of the extended version of the paper [25]).

D. Experimental Evaluation

We evaluate the scalability of our algorithm through experiments with changing sizes of partially-ordered tasks introduced in Section III-C. In our evaluation, we aim to answer two questions: (Q1) “How does solving time scale with the number of ordering tasks?”, and (Q2) “How does solving time scale with the number of labeled examples?”. We implement our algorithm in Python with PySAT [27], and we use Glucose4 [28] as the SAT solver. Our baseline is an implementation of the monolithic DFA identification encoding from [13], [15] with the same software as our implementation.

Experiments are performed on a Quad-Core Intel i7 processor clocked at 2.3 GHz and a 32 GB main memory.

To evaluate the scalability, we randomly generate positive and negative examples with varying problem sizes. For (Q1), we generate 10 (half of which are positive and half of which are negative) partially-ordered task examples with (i) 2 symbols, and (ii) 4 symbols, and we vary the number of DFAs from 2 to 12. For (Q2), we generate 10 to 20 partially-ordered task examples with (i) 2 symbols and 4 DFAs, and (ii) 4 symbols and 2 DFAs. Half of these examples are positive and the other half is negative. Since the examples are generated randomly, we run the experiments for 10 different random seeds and report the average. We set the timeout limit to 10 minutes, and stop when our algorithm timeouts for all random seeds.

Figure 1a presents the experiment results answering (Q1), where we vary the number of DFAs implied by the given examples. For partially-ordered tasks with 2 symbols, green solid line is the (monolithic DFA) baseline and the blue solid is our algorithm. Similarly, for partially-ordered tasks with 4 symbols, pink dashed line is the baseline and the red dashed line is our algorithm. Figure 1b presents the experiment results answering (Q2), where we vary the number of examples. For partially-ordered tasks with 2 symbols and 4 DFAs, green solid line is the baseline and the blue solid is our algorithm; for partially-ordered tasks with 4 symbols and 2 DFAs, pink dashed line is the baseline and the red dashed line is our algorithm. As expected, the baseline scales better than our algorithm as we also search for the Pareto frontier and solve an inherently harder problem. Notice that given 10 examples, our algorithm is able to scale up to 11 DFAs for tasks with 2 symbols, and 8 DFAs for tasks with 4 symbols; for 2 symbols and 4 DFAs, it is able to scale up to 60 examples, and for 4 symbols and 2 DFAs, it is able to scale up to 190 examples. As we demonstrate in the next section, these limits for scalability are practically useful in certain domains.

IV. LEARNING DFAS FROM DEMONSTRATIONS

Next, we show how our algorithm can be incorporated into Demonstration Informed Specification Search (DISS) - a framework for learning languages from expert demonstra-

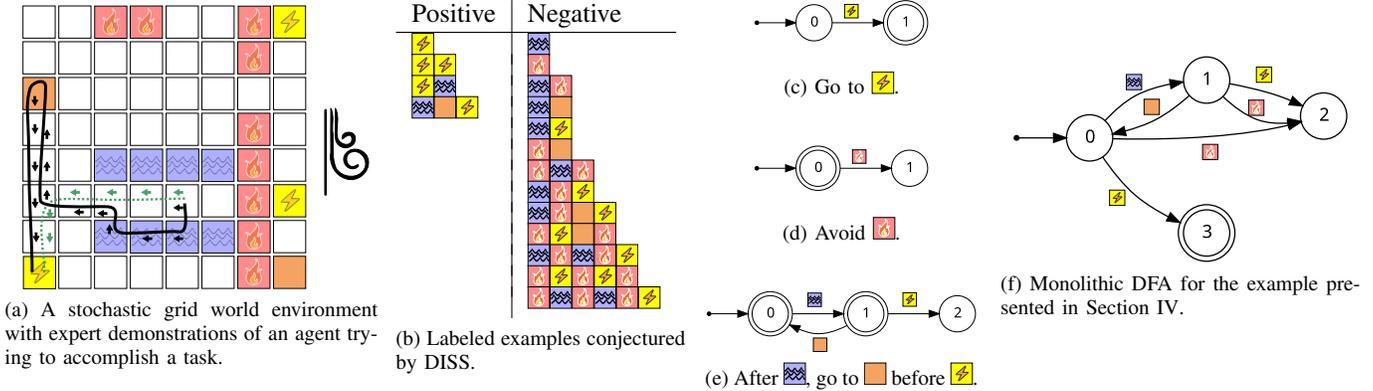


Fig. 3. Figure 3a shows the stochastic grid world environment. Figure 3b shows the positive and negative examples of the expert’s behavior conjectured by DISS and Figures 3c to 3e showcases the associated DFA decomposition identified by our algorithm. Figure 3f shows the monolithic DFA learned in [4].

tions [4]. For our purposes a *demonstration* is an unlabeled path through a workspace that maps to a string and is biased towards being accepting by some unknown language. For example, we ran our implementation of DISS using demonstrations produced by an expert attempting to accomplish a task in a stochastic grid world environment, the same example used in [4] and shown in Figure 3a. At each step, the agent can move in any of the four cardinal directions, but because of wind blowing from the north to the south, with some probability, the agent will transition to the space south of it in spite of its chosen action. Two demonstrations of the task “Reach \blacksquare while avoiding \blacktriangle . If it ever touches \blacksquare , it must then touch \blacksquare before reaching \blacksquare .” are shown in Figure 3a.

In order to efficiently search for tasks, DISS reduces the learning from demonstrations problem into a series of identification problems to be solved by a black-box identification algorithm. The goal of DISS is to find a task that minimizes the joint description length, called the energy, of the task and the demonstrations assuming the agent were performing said task. The energy is measured in bits to encode an object.

Below, we reproduce the results from [4], but using our algorithm as the task identifier rather than the monolithic DFA identifier provided³. The use of DFA decompositions biases DISS to conjecture concepts that are *simpler* to express in terms of a DFA decomposition. To define the description length of DFA decompositions, we adapt the DFA encoding used in [4] by expressing a decomposition as the concatenation of the encodings of the individual DFAs. To remove unnecessary redundancy two optimizations were performed. First common headers, e.g. indicating the alphabet size, were combined. Second, as the DFAs in a decomposition are ordered by size, we expressed changes in size rather than absolute size, see Appendix B in the extended version of the paper [25] for details.

³To allow exploring more decompositions, with some probability, the number of DFAs in the decomposition was randomly incremented or decremented during identification.

A. Experimental Evaluation

In Figures 3c to 3e we present the learned DFA decomposition along with the corresponding Figure 3b labeled examples conjectured by DISS to explain the expert behavior. Importantly, this decomposition exactly captures the demonstrated task. We note that this is in contrast to the DFA learned in [4], shown in Figure 3f, which allows visiting \blacktriangle after visiting \blacksquare . Further, we remark that the time required to learn the monolithic and decomposed DFAs was comparable. In particular, the number of labeled examples was less than 60 and as with the monolithic baseline, most of the time is not spent in task identification, but instead conjecturing the labeled examples. As we saw with in Section III-D, this number of examples is easily handled by our SAT-based identification algorithm. Finally, the number of labeled examples that needed to be conjectured to find low energy tasks was similar for both implementations (see Figures 5 and 6 in Appendix C of the extended version of the paper [25]). Thus, our variant of DISS performed similar to the monolithic variant, while finding DFAs that exactly represented the task.

V. CONCLUSION

To the best of our knowledge, this work presents the first approach for solving DFA-DIP. Our algorithm works by reducing the problem to a Pareto-optimal search of the space of the number of states in a DFA decomposition with a SAT call in the inner loop. The SAT-based encoding is based on an efficient reduction to graph coloring. We demonstrated the scalability of our algorithm on a class of problems inspired by the multi-task reinforcement learning literature and show that the additional computational cost for identifying DFA decompositions over monolithic DFAs is not prohibitive. Finally, we showed how identifying DFA decompositions can provide a useful inductive bias while learning from demonstrations.

REFERENCES

[1] C. De La Higuera, “A bibliographical study of grammatical inference,” *Pattern recognition*, vol. 38, no. 9, pp. 1332–1348, 2005.

- [2] S. Verwer and C. A. Hammerschmidt, “Flexfringe: a passive automaton learning package,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 638–642.
- [3] I. Zakirzyanov, A. Morgado, A. Ignatiev, V. Ulyantsev, and J. Marques-Silva, “Efficient symmetry breaking for sat-based minimum dfa inference,” in *International Conference on Language and Automata Theory and Applications*. Springer, 2019, pp. 159–173.
- [4] M. Vazquez-Chanlatte, A. Shah, G. Lederman, and S. A. Seshia, “Demonstration informed specification search,” *CoRR*, vol. abs/2112.10807, 2021. [Online]. Available: <https://arxiv.org/abs/2112.10807>
- [5] E. M. Gold, “Complexity of automaton identification from given data,” *Information and control*, vol. 37, no. 3, pp. 302–320, 1978.
- [6] L. Pitt and M. K. Warmuth, “The minimum consistent dfa problem cannot be approximated within any polynomial,” *Journal of the ACM (JACM)*, vol. 40, no. 1, pp. 95–142, 1993.
- [7] K. J. Lang, B. A. Pearlmutter, and R. A. Price, “Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm,” in *International Colloquium on Grammatical Inference*. Springer, 1998, pp. 1–12.
- [8] K. J. Lang, “Faster algorithms for finding minimal consistent dfas,” *NEC Research Institute, Tech. Rep*, 1999.
- [9] M. Bugalho and A. L. Oliveira, “Inference of regular languages using state merging algorithms with search,” *Pattern Recognition*, vol. 38, no. 9, pp. 1457–1467, 2005.
- [10] P. Dupont, “Regular grammatical inference from positive and negative samples by genetic search: the gig method,” in *International Colloquium on Grammatical Inference*. Springer, 1994, pp. 236–245.
- [11] S. Luke, S. Hamahashi, and H. Kitano, ““ genetic” programming,” in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*, 1999, pp. 1098–1105.
- [12] S. M. Lucas and T. J. Reynolds, “Learning dfa: evolution versus evidence driven state merging,” in *The 2003 Congress on Evolutionary Computation, 2003. CEC’03.*, vol. 1. IEEE, 2003, pp. 351–358.
- [13] M. J. Heule and S. Verwer, “Exact dfa identification using sat solvers,” in *International Colloquium on Grammatical Inference*. Springer, 2010, pp. 66–79.
- [14] V. Ulyantsev, I. Zakirzyanov, and A. Shalyto, “Bfs-based symmetry breaking predicates for dfa identification,” in *International Conference on Language and Automata Theory and Applications*. Springer, 2015, pp. 611–622.
- [15] —, “Symmetry breaking predicates for sat-based dfa identification,” *arXiv preprint arXiv:1602.05028*, 2016.
- [16] I. Zakirzyanov, A. Shalyto, and V. Ulyantsev, “Finding all minimum-size dfa consistent with given examples: Sat-based approach,” in *International Conference on Software Engineering and Formal Methods*. Springer, 2017, pp. 117–131.
- [17] P. Ashar, S. Devadas, and A. R. Newton, “Finite state machine decomposition,” in *Sequential Logic Synthesis*. Springer, 1992, pp. 117–168.
- [18] J. Rhodes, *Applications of automata theory and algebra : via the mathematical theory of complexity to biology, physics, psychology, philosophy, and games*. Singapore Hackensack, NJ: World Scientific, 2010.
- [19] O. Kupferman and J. Mosheiff, “Prime languages,” *Information and Computation*, vol. 240, pp. 90–107, 2015.
- [20] R. E. Kalman, “When is a linear control system optimal,” 1964.
- [21] A. Y. Ng and S. J. Russell, “Algorithms for inverse reinforcement learning,” in *ICML*. Morgan Kaufmann, 2000, pp. 663–670.
- [22] D. Kasenberg and M. Scheutz, “Interpretable apprenticeship learning with temporal logic specifications,” in *CDC*. IEEE, 2017, pp. 4914–4921.
- [23] G. Chou, N. Ozay, and D. Berenson, “Explaining multi-stage tasks by learning temporal logic formulas from suboptimal demonstrations,” in *Robotics: Science and Systems*, 2020.
- [24] M. Vazquez-Chanlatte, V. Lee, A. Shah, N. Lauffer, and B. Yalcinkaya, 2022. [Online]. Available: <https://github.com/mvcisback/dfa-identify/tree/decomposition>
- [25] N. Lauffer, B. Yalcinkaya, M. Vazquez-Chanlatte, A. Shah, and S. A. Seshia, “Learning deterministic finite automata decompositions from examples and demonstrations,” 2022. [Online]. Available: <https://arxiv.org/abs/2205.13013>
- [26] P. Vaezipoor, A. C. Li, R. A. T. Icarte, and S. A. Mcilraith, “Ltl2action: Generalizing ltl instructions for multi-task rl,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 10497–10508.
- [27] A. Ignatiev, A. Morgado, and J. Marques-Silva, “PySAT: A Python toolkit for prototyping with SAT oracles,” in *SAT*, 2018, pp. 428–437. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26
- [28] N. Eén and N. Sörensson, “An extensible sat-solver,” in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

Automated Conversion of Axiomatic to Operational Models: Theory and Practice

Adwait Godbole^{*} , Yatin A. Manerkar[†], and Sanjit A. Seshia[†] 

^{*}[†]University of California Berkeley, Berkeley, USA [†]University of Michigan, Ann Arbor, USA

Abstract—A system may be modelled as an *operational model* (which has explicit notions of state and transitions between states) or an *axiomatic model* (which is specified entirely as a set of invariants). Most formal methods (e.g., IC3, invariant synthesis, etc) are designed for operational models and are largely inaccessible to axiomatic models. Furthermore, no prior method exists to automatically convert axiomatic models to operational ones, so operational equivalents to axiomatic models had to be manually created and proven equivalent.

In this paper, we advance the state-of-the-art in axiomatic to operational model conversion. We show that general axioms in the μspec axiomatic modelling framework cannot be translated to equivalent finite-state operational models. We also derive restrictions on the space of μspec axioms that enable the feasible generation of equivalent finite-state operational models for them. As for practical results, we develop a methodology for automatically translating μspec axioms to equivalent finite-state automata-based operational models. We demonstrate the efficacy of our method by using the models generated by our procedure to prove the correctness of ordering properties on three register-transfer-level (RTL) designs.

I. INTRODUCTION

When modelling hardware or software systems using formal methods, one traditionally uses *operational* models (e.g. Kripke structures [1]), which have explicit notions of state and transitions. However, one may also model a system *axiomatically*, where instead of a state-transition relation, the system is specified entirely by a set of axioms (e.g., invariants) that it maintains. Executions that obey the axioms are allowed, and those that violate one or more axioms are forbidden. The vast majority of formal methods works use the operational modelling style. However, axiomatic models have been used to great effect in certain domains such as memory models, where they have shown order-of-magnitude improvements in verification performance over equivalent operational models [2].

Operational and axiomatic models each have their own advantages and disadvantages [3]. Operational models can be more intuitive as they typically resemble the system that they are modelling. Hence one is not required to reason about invariants to write the model. On the other hand, axiomatic models tend to be more concise and potentially offer faster verification [2].

Many formal methods (e.g., refinement procedures [4], invariant synthesis, IC3/PDR [5], [6]) are set up to use operational models. Axiomatic models are largely or completely incompatible with these techniques, as the axioms constrain full traces rather than a step of the transition relation. One way to take advantage of these techniques when using axiomatic

models is to create and use operational models equivalent to the axiomatic models. The only prior method of doing this was to first manually create the operational model and then manually prove it equivalent to the axiomatic model. There have been several works doing so [2], [7], [8], [9], [10].

Manually creating an operational model and proving equivalence is cumbersome and error-prone. The ability to automatically generate operational models equivalent to a given axiomatic model would be beneficial, eliminating both the time spent creating the operational model as well as the need for tedious manual equivalence proofs. Generated models can then be fed into techniques currently requiring operational models (e.g. IC3/PDR).

To this end, we make advances in this paper towards the automatic conversion of axiomatic models to equivalent operational models, on both theoretical and practical fronts. In our work, we focus specifically on μspec [11], a well-known axiomatic framework for modelling microarchitectural orderings, which has been used in a wide range of contexts [12], [13], [14], [15], [16] including memory consistency, cache coherence and hardware security.

On the theoretical front, we show that it is impossible to convert general μspec axioms to equivalent finite-state operational models. However, we show that it is feasible to generate equivalent operational models for a specific subset of μspec (henceforth referred to as μspecRE). On the practical side, we develop a method to automatically translate universal axioms¹ in μspecRE into equivalent finite-state operational models comprised of building blocks we term as *axiom automata* (finite automata that monitor whether an axiom has been violated). Furthermore, for arbitrary μspec axioms, our method can generate operational models that are equivalent to the axioms up to a program-size bound.

To evaluate our technique, we convert axioms for three RTL designs to their corresponding operational models: an in-order multicore processor (`multi_vsacle`), a memory-controller (`sdram_ctrl`), and an out-of-order single-core processor (`tomasulo`). We showcase how the generated models can be used with procedures like BMC and IC3/PDR which are usually inaccessible for axiomatic models, and we produce both bounded and unbounded proofs of correctness.

Overall, the contributions of this work are as follows:

- We prove that generation of equivalent finite-state operational models for arbitrary μspec axioms is impossible.

¹Axioms that do not contain \exists quantifiers.

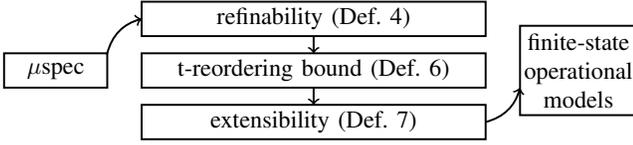


Fig. 1: Roadmap to obtain finite-state operational models.

- We provide a procedure for generating equivalent finite-state operational models for universal axioms in μspecRE .
- We propose the *axiom-automata* formulation to generate equivalent finite-state operational models from universal axioms in μspecRE (or from arbitrary μspec axioms if only guaranteeing equivalence up to a bounded program size).
- We evaluate our method for operational model generation by using our generated models to prove the (bounded/unbounded) correctness of ordering properties on three RTL designs: `multi_vscale`, `tomasulo`, and `sdram_ctrl`.

Generality. While axiomatic models enforce constraints over complete executions, operational models do this local to each transition. Ensuring that behaviours generated by the latter are also allowed by the former requires performing non-local consistency checks which are hard to reason about, especially for unbounded executions. This has been observed in manual operationalization works as well. Taking the example of [7], (which operationalizes C11), we address issues of eliminating consistent executions too early [7, §3] and repeatedly checking consistency [7, §4] by developing concepts such as *t-reordering boundedness* (Def. 6) and *extensibility* (Def. 7). Though we focus on μspec , we believe many of the underlying challenges and concepts carry over to frameworks such as Cat [2].

Outline. §II covers the syntax and semantics of μspec used in this paper. §III covers the formulation of the space of operational models we consider. They have finite control-state and read-only input tapes for the instruction streams (programs) executed by each core. §IV defines our notions of soundness, completeness, and equivalence when comparing operational and axiomatic models. In §V, we show that it is impossible to synthesise equivalent finite-state operational models from arbitrary axiomatic models. We develop an underapproximation, called *t-reordering boundedness*, that addresses this by bounding the depth of reorderings possible. In §VI we restrict μspec further by requiring *extensibility* (preventing current events from influencing orderings between previous events). Restricting μspec by *t-reordering boundedness* and *extensibility* is sufficient to enable the automatic generation of equivalent finite-state operational models (Thm. 2). §VII describes our conversion procedure based on axiom automata. §VIII evaluates our technique by using it to generate operational models, which are then used for checking properties of RTL designs. §IX covers related work, and §X concludes, with §XI suggesting avenues for future work. This paper is accompanied by an extended version which contains supplementary material and proofs [17].

II. μSPEC SYNTAX AND SEMANTICS

A. μspec Syntax

$$\begin{aligned}
 \langle \text{AX} \rangle &:= \forall i \text{ AX} \mid \exists i \text{ AX} \mid \phi(i_1, \dots, i_m) \\
 \langle \phi \rangle &:= \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \langle \text{atom} \rangle \\
 \langle \text{atom} \rangle &:= i_1 <_r i_2 \mid \text{hb}(i_1.\text{st}, i_2.\text{st}) \mid \text{P}(i_1, \dots) \\
 \langle \text{st} \rangle &:= \text{Fet} \mid \text{Dec} \mid \text{Exe} \mid \text{WB} \mid \dots
 \end{aligned}$$

Fig. 2: μspec Syntax.

μspec [11] is a domain-specific language used for specifying microarchitectural orderings. A μspec model consists of *axioms* that enforce first-order constraints over execution graphs; each axiom quantifies over instructions and is required to be a sentence (i.e. not have any free variables). Execution graphs that satisfy the axioms and are acyclic are deemed as valid executions. While ISA-level models [2], [18], [19] treat single instructions as atomic entities, μspec decomposes the execution of an instruction into a set of atomic *events*. Each instruction i and stage st is associated with an event $i.\text{st}$. A program execution is viewed as a directed acyclic graph called a micro-architectural happens-before graph (μhb graph) [12]. Such a graph for a given program has nodes corresponding to events of form $i.\text{st}$ for each instruction i in the program and each stage st prescribed by the model. Edges in the graph correspond to the *happens-before* (**hb**) relation: $\text{hb}(e_1, e_2)$ says that e_1 happened before e_2 . Thus, a cyclic μhb graph corresponds to an impossible scenario where an event happens before itself, and thus represents an execution that cannot occur on the microarchitecture.

Fig. 2 specifies μspec syntax. It has three types of atoms:

- (i) $\text{hb}(i_1.\text{st}, i_2.\text{st})$: happens-before predicate
- (ii) $i_1 <_r i_2$: the reference order (typically the program order)
- (iii) $\text{P}(i_1, \dots)$: instruction predicate atoms

Atoms of type (ii) capture the order in which instructions appear in a given program thread. Atoms of type (iii) are predicates over instructions which capture instruction properties, e.g. opcode, source/destination registers. We note that μspec models in literature [11] also make use of the `NodeExists` predicate which identifies event nodes that occur in the execution. We do not model `NodeExists` in this paper, but our approach can be augmented to incorporate it (see [17]).

We identify two types of axioms of interest: *Universal axioms* are of the form: $\forall i_1 \dots \forall i_k \phi(i_1, \dots, i_k)$, and represent constraints applied symmetrically over all tuples of instructions in a program. *Predicate-free axioms* are axioms that do not have occurrences of predicate (P) atoms. We extend these terms to an axiomatic semantics if all axioms are of that type. In this work, our theoretical treatment focuses on universal semantics. Practically though, some underlying ideas carry over to arbitrary axioms as we discuss in §VII, §VIII.

```

ax0:  $\forall i_1. \text{hb}(i_1.\text{Exe}, i_1.\text{Com})$ 
ax1:  $\forall i_1, i_2. (i_1 <_r i_2 \wedge \text{DepOn}(i_1, i_2))$ 
 $\implies \text{hb}(i_1.\text{Exe}, i_2.\text{Exe})$ 
ax2:  $\forall i_1, i_2. \text{SameCore}(i_1, i_2) \implies$ 
 $(\text{hb}(i_1.\text{Exe}, i_2.\text{Exe}) \vee \text{hb}(i_2.\text{Exe}, i_1.\text{Exe}))$ 
ax3:  $\forall i_1, i_2. i_1 <_r i_2 \implies \text{hb}(i_1.\text{Com}, i_2.\text{Com})$ 

```

Fig. 3: An example axiomatic model.

B. Illustrative μspec Example

Consider the four axioms in Fig. 3. In the axioms, **i1**, **i2** are instruction variables and **Exe**, **Com** are stage names (short for execute and commit respectively). The axiom **ax0** requires that for each instruction, the execute stage (**Exe**) of that instruction must happen before the commit stage (**Com**). Intuitively, **ax1** says that when **i2** depends on **i1** (captured by the predicate **DepOn**), **i1** should be executed before **i2**; **ax2** says that the execute events of instructions on the same core should be *totally ordered* by **hb**. The third axiom **ax3** says that when **i1** and **i2** are in program order (denoted by $<_r$), **i1** must be committed before **i2**.

Fig. 4 shows valid and invalid execution graphs for the program snippet in Fig. 5. The snippet is of a 2-core program, with two instructions per core. Instruction i_1 is dependent on the result of i_0 (since its source register is the same as the destination of i_0). In the example axiomatic semantics, **ax1** requires that the execute event of instruction i_0 be before that of i_1 . The execution in Fig. 4b is invalid w.r.t. **ax1** since $i_1.\text{Exe}$ is executed before $i_0.\text{Exe}$. The execution in Fig. 4a is valid even though the $i_2.\text{Exe}$ and $i_3.\text{Exe}$ events are reordered since i_3 does not depend on i_2 . Both executions are valid w.r.t. **ax0**, **ax2** and **ax3**.

C. Programming Model

We consider multi-core systems with each core executing a straight-line program over a finite domain of operations. This is common in memory models [2], [12], [16], [20] and distributed systems [21] literature.

1) *Cores*: The system consists of n processor cores: $\text{Cores} = [n]$. Each core executes operations from a finite set \mathbb{O} . The axiomatic model \mathcal{A} assigns predicates from \mathbb{P} an interpretation over the universe \mathbb{O} . We denote this interpretation as $\mathbb{P}^{\mathcal{A}} \subseteq \mathbb{O}^k$ for an arity- k predicate.

2) *Instruction streams*: An *instruction stream* \mathcal{I} is a word over \mathbb{O} : $\mathcal{I} \in \mathbb{O}^*$. A program \mathcal{P} is a set of *per-core* instruction streams: $\{\mathcal{I}_c\}_{c \in \text{Cores}}$. For a core c and label $0 \leq j < |\mathcal{I}_c|$, we call the triple $(c, j, \mathcal{I}_c[j])$ an *instruction*². We denote components of instruction $i = (c, j, \mathcal{I}_c[j])$, as: $c(i) = c$, label $\lambda(i) = j$ and operation $\text{op}(i) = \mathcal{I}_c[j]$. The set of instructions occurring in \mathcal{P} is: $\text{instrsOf}(\mathcal{P}) = \{(c, j, \mathcal{I}_c[j]) \mid c \in$

²Note the terminology: operations are commands that the core can execute. Since we interpret predicates over \mathbb{O} we require $|\mathbb{O}|$ to be a finite set for computability reasons. Instructions are operations combined with the label and core identifier (and hence form an infinite set).

$\text{Cores}, 0 \leq j < |\mathcal{I}_c|$) and the set of all possible instructions as $\mathbb{I} = \text{Cores} \times \mathbb{Z}^{\geq 0} \times \mathbb{O}$.

3) *Instruction stages*: Instruction execution in μspec is decomposed into stages. The set of stages, Stages , is a parameter of the semantics. Instruction i performing in stage st , (i.e. $i.\text{st}$) is an atomic event in an execution. The execution of \mathcal{P} is composed of the set of events: $\text{eventsOf}(\mathcal{P}) = \{i.\text{st} \mid i \in \text{instrsOf}(\mathcal{P}), \text{st} \in \text{Stages}\}$. The set of all possible events is $\mathbb{E} = \{i.\text{st} \mid i \in \mathbb{I}, \text{st} \in \text{Stages}\}$.

Definition 1 (Event). *An event e is of the form $i.\text{st}$. It represents the instruction $i \in \mathbb{I}$, (atomically) performing in stage $\text{st} \in \text{Stages}$.*

Example 1. *Following the example in Fig. 5 we consider an architecture with two opcodes: **add**, **lw** for add and load respectively. For each of these, we may have several actual operations (with different operands), thus giving us the set \mathbb{O} . The program \mathcal{P} in Fig. 5 has two cores: $\text{Cores} = \{c_0, c_1\}$ and four instructions: $\text{instrsOf}(\mathcal{P}) = \{i_0, i_1, i_2, i_3\}$. We have, for example, $c(i_1) = c_0, \lambda(i_1) = 1, \text{op}(i_1) = \text{add } r3, r2, r1$ while $c(i_2) = c_1, \lambda(i_2) = 0$. The instruction stream for core c_0 is $\mathcal{I}_0 = i_0 \cdot i_1$ and that of core c_1 is $\mathcal{I}_1 = i_2 \cdot i_3$.*

Let us suppose that this program is executed on a 4-stage microarchitecture with $\text{Stages} = \{\text{Fet}, \text{Dec}, \text{Exe}, \text{WB}, \text{Com}\}$. The events corresponding to the program are given by $\text{eventsOf}(\mathcal{P}) = \{i_0.\text{Fet}, i_0.\text{Dec}, \dots, i_3.\text{Com}\}$ with $|\text{eventsOf}(\mathcal{P})| = 4 \times 5 = 20$.

D. Formal μspec Semantics

We now define the formal semantics of μspec axioms.

Definition 2 (μhb graph). *For a program \mathcal{P} , a μhb graph is a directed acyclic graph, $G(V, E)$, with nodes $V = \text{eventsOf}(\mathcal{P})$ representing events and edges representing the happens-before relationships, i.e. $(e_1, e_2) \in E \equiv \text{hb}(e_1, e_2)$.*

Validity of μhb graph w.r.t. an axiomatic semantics: Consider an axiomatic semantics \mathcal{A} (i.e. a set of axioms). A μhb graph $G = (V, E)$ is said to represent a valid execution of program \mathcal{P} under \mathcal{A} if it satisfies all the axioms in \mathcal{A} . We denote the validity of a μhb graph G by $G \models_{\mathcal{P}} \mathcal{A}$.

Satisfaction w.r.t. an axiom: We first define satisfaction for the quantifier-free part, starting at the atoms. Let $s : \mathbb{I}(\text{AX}) \rightarrow \mathbb{I}$ be an assignment for the symbolic instruction variables $\mathbb{I}(\text{AX})$ in axiom AX .

$$G \models i_1[s] <_r i_2[s] \iff c(s(i_1)) = c(s(i_2)) \\ \wedge \lambda(s(i_1)) < \lambda(s(i_2)) \quad \dots(i)$$

$$G \models \text{P}(i_1, \dots, i_m)[s] \iff \\ (\text{op}(s(i_1)), \dots, \text{op}(s(i_m))) \in \mathbb{P}^{\mathcal{A}} \quad \dots(ii)$$

$$G \models \text{hb}(i_1.\text{st}_1, i_2.\text{st}_2)[s] \iff \\ (s(i_1).\text{st}_1, s(i_2).\text{st}_2) \in E^+ \quad \dots(iii)$$

In (i), the reference order $<_r$ relates instructions i_1, i_2 from the same instruction stream if i_1 is before i_2 . In (ii) we extend predicate interpretations, $\mathbb{P}^{\mathcal{A}}$, (defined over \mathbb{O}) to instructions by taking the $\text{op}(\cdot)$ component. Finally, **hb** atoms

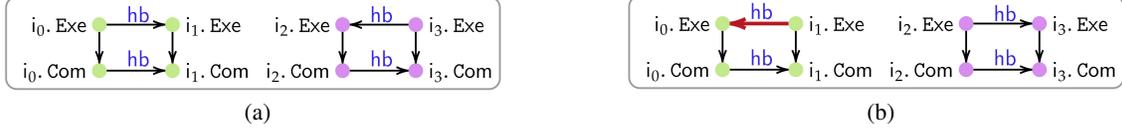


Fig. 4: Valid (a) and an invalid (b) execution graphs for the program in Fig. 5 and axioms in Fig. 3. All edges represent the **hb** relation. The red (bold) edge violates **ax1**.

```

i0: lw r1, 42(r0)   |   i2: lw r4, 42(r0)
i1: add r3, r2, r1 |   i3: add r3, r2, r1

```

Fig. 5: Example program snippet

are interpreted as E^+ , i.e. transitive closure of E , as stated in (iii). Operators \wedge, \vee, \neg have their usual semantics.

We now define the satisfaction of a (quantified) axiom AX by a graph G , denoted by $G \models_{\mathcal{P}} \text{AX}$ above.

$$\begin{aligned}
G \models_{\mathcal{P}} \phi[s] &\equiv G \models \phi[s] \\
&\quad \text{for quantifier-free } \phi \\
G \models_{\mathcal{P}} \forall i \phi[s] &\equiv G \models_{\mathcal{P}} \phi[s[i \leftarrow i]] \\
&\quad \text{for all } i \in \text{instrsOf}(\mathcal{P}) \setminus \text{range}(s) \\
G \models_{\mathcal{P}} \exists i \phi[s] &\equiv G \models_{\mathcal{P}} \phi[s[i \leftarrow i]] \\
&\quad \text{for some } i \in \text{instrsOf}(\mathcal{P}) \setminus \text{range}(s)
\end{aligned}$$

The base case is $G \models_{\mathcal{P}} \phi[s]$ (where ϕ is quantifier-free) and follows the earlier definitions. We extend $G \models_{\mathcal{P}} \phi$ with (almost) usual quantification semantics: $\forall (\exists)$ quantifies over all (some) instructions in $\text{instrsOf}(\mathcal{P})$. Execution G is a valid execution of \mathcal{P} under semantics \mathcal{A} , denoted as $G \models_{\mathcal{P}} \mathcal{A}$, if $G \models_{\mathcal{P}} \text{AX}$ for all axioms AX in \mathcal{A} .

III. OPERATIONAL MODEL OF COMPUTATION

To concretize our claims, we introduce a model of computation that characterizes the models of interest. We choose to focus on finite-state operational models that generate totally ordered traces, where transitions represent (i.st) events. While there are less restrictive models (e.g. event structures [22], [23]), such models require specialized, typically under-approximate, verification techniques (e.g. [24], [25], [26]). Our choice is motivated by the ability to (a) have finite-state implementations of generated models (e.g. in RTL) and (b) verify against these models with off-the-shelf tools (e.g. model checkers using BDD and SMT-based backends).

A. Model of computation

Intuitively, the model of computation resembles a 1-way transducer [27], [28] with multiple (read-only) input tapes (one tape for each instruction stream). This allows us to execute programs of unbounded length with a finite control state.³

1) *Model definition*: An operational model is parameterized by cores Cores , stages Stages , and a history parameter $h \in \mathbb{N} \cup \{\infty\}$ which bounds the length of tape to the left of the head. It is a tuple $(\mathcal{Q}, \Delta, q_{\text{init}}, q_{\text{final}})$:

- \mathcal{Q} is a finite set of control states

³A Kripke structure-based formalism is insufficient since we want to execute unbounded programs with distinguished instructions without explicitly modelling control logic.

- $\Delta \subseteq \mathcal{Q} \times (\mathbb{I} \cup \{\neg\})^{|\text{Cores}|} \times \mathcal{Q} \times \text{Act}$ is the transition relation where Act is the set of actions
- $q_{\text{init}} \in \mathcal{Q}$ is the initial state
- $q_{\text{final}} \in \mathcal{Q}$ is the final state which must be absorbing (i.e. it has a self-loop)

A model is finite-state if \mathcal{Q} is finite, and it has bounded-history if $h \in \mathbb{N}$. For the end goal of effective verification, we are interested in finite-state, bounded-history models since it is precisely such models that can be compiled to finite-state systems.

2) *Model semantics*: A configuration is a triple $\gamma = (U, q, V)$ where $U : \text{Cores} \rightarrow \mathbb{I}^*$, $V : \text{Cores} \rightarrow \mathbb{I}^*$ and $q \in \mathcal{Q}$. Intuitively U (V) represent, for each instruction stream, the contents of the input tape to the left (right) of the head respectively. For a bounded history machine, a configuration is *allowed* only if $|U(c)| \leq h$ for all $c \in \text{Cores}$. For unbounded history all configurations are allowed.

The set of actions is

$$\begin{aligned}
\text{Act} = & \{\text{right}(c) \mid c \in \text{Cores}\} \cup \\
& \{\text{stay}\} \cup \\
& \{\text{sched}(c, i, \text{st}) \mid c \in \text{Cores}, \text{st} \in \text{Stages}, i \in [h]\} \cup \\
& \{\text{drop}(c, i) \mid c \in \text{Cores}, i \in [h]\}
\end{aligned}$$

Intuitively, these represent in order: motion of the tape head for c to the right, silent (no-effect), generation of an event, and removing the i^{th} instruction from the left of the head. We provide full semantics in the supplementary material [17].

For word $w \in \mathbb{I}^*$, let $\text{fst}(w)$ denote its first element if $w \neq \epsilon$ and \neg otherwise. Transitions are enabled based on the control state and the instructions that the tape-heads point to: transition $(q_1, (i_1, \dots, i_{|\text{Cores}|}), q_2, _) \in \Delta$ is enabled in configuration $\gamma = (U, q, V)$ if $q_1 = q$ and $\text{fst}(V(c)) = i_c$ for each $c \in \text{Cores}$.

3) *Runs*: The initial configuration is given by $\gamma_{\text{init}}(\mathcal{P}) = (U_{\text{init}}, q_{\text{init}}, V_{\text{init}})$ where $U_{\text{init}} = \lambda c. \epsilon$ and $V_{\text{init}} = \lambda c. \mathcal{I}_c$, i.e. for each core, the left of the tape head is empty, and the right of the tape head consists of the instruction stream for that core. Starting from $\gamma_{\text{init}}(\mathcal{P})$, the machine transitions according to the transition rules. Such a sequence of configurations $\gamma_{\text{init}}(\mathcal{P}) = \gamma_0 \xrightarrow{e_1} \gamma_1 \cdots \xrightarrow{e_m} \gamma_m$, where all γ_i are allowed is called a run. A run is called accepting if it ends in the state q_{final} .

4) *Traces*: The sequence of event labels $\sigma = e_1 \cdots e_m$ annotating a run is the *trace* corresponding to the run. Each label is an event from \mathbb{E} and hence $\sigma \in \mathbb{E}^*$. We view σ as a (linear) μhb execution graph $e_1 \xrightarrow{\text{hb}} e_2 \cdots \xrightarrow{\text{hb}} e_m$, and hence define $\sigma \models \mathcal{A}$ in the usual way. Accordingly, we will sometimes refer to σ as an execution of a program \mathcal{P} . The set

of traces corresponding to accepting runs of an operational model \mathcal{M} on a program \mathcal{P} are denoted as $\text{traces}_{\mathcal{M}}(\mathcal{P}) \subseteq \mathbb{E}^*$.

IV. SOUNDNESS, COMPLETENESS, AND EQUIVALENCE

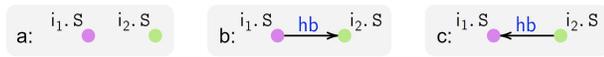
We proceed to formalize the notion of equivalence that relates axiomatic and operational models. In literature [29], [2], ISA-level behaviours of programs have been annotated by the read values of `load` operations. Hence, one notion of equivalence might be to require that identical read values be possible between the models. While this may be reasonable for ISA-level behaviours, it can hide microarchitectural features: different microarchitectural executions can have identical architectural results. Given that μspec models executions at the granularity of microarchitectural events, we adopt a stronger notion of equivalence. For soundness, we require that the operational semantics generates linearizations of μhb graphs that are valid under the axiomatic semantics. Formally:

Definition 3 (Soundness). *An operational model \mathcal{M} is sound w.r.t. \mathcal{A} if for any program \mathcal{P} , each trace in $\text{traces}_{\mathcal{M}}(\mathcal{P})$ is a linearization of some μhb graph that is valid under \mathcal{A} .*

Before defining completeness, we need to address a subtlety. Since operational executions are viewed as μhb graphs by interpreting trace-ordering as the `hb` ordering, the operational model always generates linearized μhb graphs. However, in general, linearizations of valid μhb graphs could end up being invalid w.r.t the axioms. Consider Example 2.

Example 2 (Non-refinable axiom). For the following axiom with $\text{Stages} = \{\mathbf{S}\}$, the graph (a) is a valid execution. However, both of its linearizations (b) and (c) are invalid. Thus, *all* of the (totally-ordered) traces generated by our operational models will be deemed invalid under the axiomatic semantics. This renders a direct comparison between operational and axiomatic executions infeasible.

$$\forall i_1, i_2. (\neg \text{hb}(i_1.S, i_2.S) \wedge \neg \text{hb}(i_2.S, i_1.S))$$



To address this issue, we develop the notion of refinability. For two μhb graphs $G = (V, E)$ and $G' = (V', E')$, we say that G' refines G , denoted $G \sqsubseteq G'$ if (1) $V = V'$ and (2) $(e_1, e_2) \in E^+ \implies (e_1, e_2) \in E'^+$.

Definition 4 (Refinable `hb`). *An axiomatic semantics \mathcal{A} is refinable if for any program \mathcal{P} , and μhb graph G s.t. $G \models_{\mathcal{P}} \mathcal{A}$, we have $G' \models_{\mathcal{P}} \mathcal{A}$ for all linear graphs G' satisfying $G \sqsubseteq G'$.*

Refinability says that all linearizations of a valid graph are valid. While executions under axiomatic semantics are given by (partially-ordered) μhb graphs, our class of operational models generate totally-ordered traces. Refinability bridges this gap by relating valid μhb graphs to valid traces. Interestingly, we can check whether a universal axiomatic semantics satisfies refinability, which at a high level, we show via a small model property (Lemma 1).

Lemma 1. *Given a universal axiomatic semantics we can decide whether the semantics is refinable.*

Refinability is especially important for completeness. For non-refinable semantics, validity of linearizations cannot be checked based on the axioms, as all linearizations may be invalid (Example 2).

We assume that the axiomatic semantics satisfies refinability.

We define completeness and our formal problem statement.

Definition 5 (Completeness). *An operational model \mathcal{M} is complete, if for any program \mathcal{P} and valid μhb graph $G \models_{\mathcal{P}} \mathcal{A}$, $\text{traces}_{\mathcal{M}}(\mathcal{P})$ contains all linearizations of G .*

Formal Problem Statement Given an axiomatic semantics \mathcal{A} , a set of cores Cores and stages Stages , generate a *finite state, bounded history* model, $\mathcal{M} = (\mathcal{Q}, \Delta, q_{\text{init}}, q_{\text{final}})$, which satisfies soundness and completeness (Defns. 3 and 5).

V. ENABLING SYNTHESIS BY BOUNDING REORDERINGS

In this section, we develop some theoretical results for the synthesis of operational models. First, we show that synthesis of sound and complete (viz. Defn. 3 and 5) finite-state operational models is not possible. Then we provide an underapproximation for the completeness requirement, called *t-completeness*, that enables the synthesis of finite-state models. This still does not allow for bounded-history models as future events can influence past orderings (Example 3). In §VI we add *extensibility* thus enabling our original goal of finite-state and bounded-history models.

A. An impossibility result

We show that it is in fact impossible to develop a finite-state transition system \mathcal{M} that satisfies the requirements prescribed in Defns. 3 and 5. Figure 6 gives an axiomatic semantics $\mathcal{A}^\#$ (with $\text{Stages} = \{\mathbf{S}, \mathbf{T}\}$) such that for all possible finite-state models, there is some program such that either soundness or completeness is violated. In words, the axioms in Fig. 6 state

$$\begin{aligned} \mathbf{ax0}: & \forall i_1. \quad \text{hb}(i_1.S, i_1.T) \\ \mathbf{ax1}: & \forall i_1, i_2. \quad \text{hb}(i_1.S, i_2.S) \implies \text{hb}(i_1.T, i_2.T) \end{aligned}$$

Fig. 6: Semantics $\mathcal{A}^\#$ that does not allow bounded synthesis

the following constraints: **ax0** says that for each instruction, the `S` stage event happens before the `T` stage, and **ax1** enforces that for any two instructions, the ordering between their `S` stage events implies an identical ordering between their `T` stage events. We have the following:

Theorem 1. *For a single-core program \mathcal{P} with an instruction stream of $|\mathcal{I}_{c_1}| = m$ instructions, there is no model $\mathcal{M} = (\mathcal{Q}, \Delta, q_{\text{init}}, q_{\text{final}})$ that is sound and complete w.r.t. $\mathcal{A}^\#$ and \mathcal{P} , and s.t. $|\mathcal{Q}| < \mathcal{O}(2^m/m)$, even with $h = \infty$.*

We provide an intuitive explanation, deferring details to the supplement [17]. In valid executions of $\mathcal{A}^\#$, `S` stage events

can be ordered arbitrarily, while **T** stage events must maintain the same ordering as that of corresponding **S** stages. Hence the machine must remember the **S** orderings in its finite control. However, the number of such orderings grows (exponentially) with the number of instructions m , implying that existence of a finite-state model that works for all programs is not possible.

Corollary 1. *There does not exist a finite state operational model (even with $h = \infty$) which is sound and complete with respect to the $\mathcal{A}^\#$ axioms.*

B. An underapproximation result

Given the results of the previous section, we must relax some constraint imposed on the operationalization: we choose to relax completeness. To do so, we define an underapproximation called *t-reordering bounded traces*. Intuitively, this imposes two constraints: (a) it bounds the depth of reorderings between instructions on each core, (b) it bounds the number of instructions executed on all other cores, while a core is executing a single instruction.

We observe that (a) is a reasonable assumption since most microarchitectures bound reordering depth, often due to finite reorder buffers. On the other hand, (b) can be thought of as a fairness/starvation-freedom property.

For two instructions i_1, i_2 on the same core, let $\text{diff}_r(i_1, i_2) = \lambda(i_2) - \lambda(i_1)$ (recall that $\lambda(i)$ is the instruction index of i). Consider a trace σ of program \mathcal{P} . For $i \in \text{instrsOf}(\mathcal{P})$, we define the starting index of i , denoted as $\text{start}(i)$, as the index of the first event of instruction i in σ . Similarly we define the ending index, $\text{end}(i)$ as the largest index for some event of i in σ . Let the *prefix-closed end index* of i be the max of end over instructions that are $\leq_r i$: $\text{pfxend}(i) = \max\{\text{end}(i') \mid i' \leq_r i\}$. Two instructions i_1 and i_2 are coupled in a trace (denoted as $\text{coup}(i_1, i_2)$) if the intervals $[\text{start}(i_1), \text{pfxend}(i_1)], [\text{start}(i_2), \text{pfxend}(i_2)]$ overlap.

Definition 6 (*t-reordering bounded traces*). *A trace is t-reordering bounded if, for any pair of instructions i_1, i_2 with $c(i_1) = c(i_2)$, (1) if $i_2.st_2 \xrightarrow{\text{hb}} i_1.st_1$ then $\text{diff}_r(i_1, i_2) < t$ and (2) if $\text{coup}(i_1, i), \text{coup}(i, i_2)$ for some i then $|\text{diff}_r(i_1, i_2)| < t$.*

Intuitively, (1) says that an instruction cannot be reordered with another that precedes it by $\geq t$ indices, while (2) says that instructions on a core cannot be *stalled* while more than t instructions are executed on another. Note that *t-reordering boundedness* is a property of traces, and not of axioms. We now relax completeness (and hence equivalence) to require that the operational model at least generate all *t-reordering bounded linearizations* (instead of all linearizations).

Definition 5* (*t-completeness*). *An operational model \mathcal{M} is t-complete w.r.t. an axiomatic model \mathcal{A} , if for each program \mathcal{P} and $G \models_{\mathcal{P}} \mathcal{A}$, $\text{traces}_{\mathcal{M}}(\mathcal{P})$ contains all t-reordering bounded linearizations of G .*

Replacing Defn. 5 with its *t*-bounded relaxation (Defn. 5*) addresses the issue of having to keep track of an unbounded number of orderings. However, to allow for finite implemen-

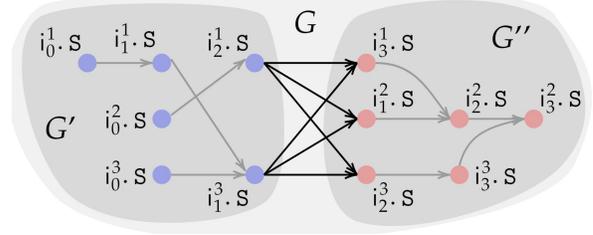


Fig. 7: \mathcal{P} has instruction streams $i_0^1 \cdot i_1^1 \cdot i_2^1 \cdot i_3^1, i_0^2 \cdot i_1^2 \cdot i_2^2 \cdot i_3^2$, and $i_0^3 \cdot i_1^3 \cdot i_2^3 \cdot i_3^3$. Blue instructions form the prefix \mathcal{P}' (i.e. $\mathcal{P}' \preceq \mathcal{P}$) and red its residual $\mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$. The figure shows executions G' of \mathcal{P}' , G'' of \mathcal{P}'' , and their composition $G = G' \triangleright G''$.

tations in practice, in addition to finite-state, we also require bounded-history ($h \in \mathbb{N}$). This is addressed in the next section.

VI. ADDING EXTENSIBILITY

As illustrated by the following example, the *t-reordering bounded underapproximation* is insufficient to achieve bounded-history operational model synthesis on its own.

Example 3 (Need for extensibility). Consider a single stage axiomatic semantics: $\text{Stages} = \{\mathbf{S}\}$, and predicate $\mathbb{P} = \{\mathbf{P}\}$.

$$\forall \mathbf{i0}, \mathbf{i1}, \mathbf{i2}. (\mathbf{P}(\mathbf{i0}, \mathbf{i1}, \mathbf{i2}) \wedge \mathbf{i0} <_r \mathbf{i1}) \implies \neg \text{hb}(\mathbf{i1.S}, \mathbf{i0.S})$$

There cannot be a sound, *t*-complete, and bounded-history (for bound h) model for this axiom (for some $t > 1$). To see this, consider a (single-core) program \mathcal{P} , with instructions $i_0 \cdot i_1 \cdots i_{h+1}$. Depending on the instructions in \mathcal{P} , the interpretation $\mathcal{P}^{\mathcal{A}}$ of \mathcal{P} can either be (a) $\mathcal{P}^{\mathcal{A}} = \{(i_0, i_1, i_{h+1})\}$ or (b) $\mathcal{P}^{\mathcal{A}} = \{\}$. In the former case, the ordering $i_1.S \xrightarrow{\text{hb}} i_0.S$ is invalid while in the latter it is valid. Since we only allow a h -sized history, $i_0.S$ must be scheduled before the tape-head reaches i_{h+1} , i.e. before the machine can determine which of (a)/(b) hold. Since the machine cannot determine whether events $i_0.S, i_1.S$ can be reordered, this leads either to a model which is unsound (always reorders) or incomplete (never reorders).

Thus, we need an additional restriction to enable generation of operational models with a finite history parameter h . We propose extensibility, which intuitively states that partial executions of program \mathcal{P} that have not violated any axioms can be composed with valid executions of the residual program to generate valid complete executions of \mathcal{P} . To do this, we extend the notion of validity to partial executions through *prefix programs*.

A program \mathcal{P} can be split into a prefix \mathcal{P}' (blue) and the residual suffix \mathcal{P}'' (red) (Fig. 7). Formally, \mathcal{P}' is a *prefix* of program \mathcal{P} , if \mathcal{P}' has instruction streams $\{\mathcal{I}'_i\}$, each of which is a prefix of the instr. streams $\{\mathcal{I}_i\}$ of \mathcal{P} . We denote that \mathcal{P}' is a prefix of \mathcal{P} by $\mathcal{P}' \preceq \mathcal{P}$. For programs $\mathcal{P}, \mathcal{P}'$ such that $\mathcal{P}' \preceq \mathcal{P}$ we denote the *residual* of \mathcal{P} w.r.t. \mathcal{P}' as $\mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$. \mathcal{P}' has instr. streams \mathcal{I}'_c : for each core c , $\mathcal{I}_c = \mathcal{I}'_c \cdot \mathcal{I}''_c$.

In Fig. 7, for example, the first instruction stream of \mathcal{P} is $i_0 \cdot i_1 \cdot i_2 \cdot i_3$. The prefix program \mathcal{P}' has (the prefix) $i_0 \cdot i_1 \cdot i_2$ as its first instr. stream. On the other hand, the residual program, $\mathcal{P}'' = \mathcal{P} \circ \mathcal{P}'$, has the suffix i_3 as its instruction stream.

For graphs $G' = (V', E')$ and $G'' = (V'', E'')$, with $V' \cap V'' = \emptyset$ we define $G' \triangleright G''$ as the graph $G = (V, E)$ where, (1) $V = V' \cup V''$, and (2) $E = E' \cup E'' \cup \{(e', e'') \mid e' \in \text{sink}(E'), e'' \in \text{source}(E'')\}$. The example in Fig. 7 illustrates such a composition: we have $G = G' \triangleright G''$.

Definition 7 (Extensibility). *An axiom AX satisfies extensibility if for any programs \mathcal{P} and \mathcal{P}' s.t. $\mathcal{P}' \preceq \mathcal{P}$, and $\mathcal{P}'' = \mathcal{P} \circ \mathcal{P}'$ if $G' \models_{\mathcal{P}'}$ AX and $G'' \models_{\mathcal{P}''}$ AX then $G' \triangleright G'' \models_{\mathcal{P}}$ AX. An axiomatic semantics \mathcal{A} satisfies extensibility if all axioms $AX \in \mathcal{A}$ satisfy extensibility.*

We require that the axiomatic model satisfies extensibility. We define μspecRE (RE stands for Refinable, Extensible) as the subset of μspec in which all axioms are refinable and extensible. Finite-state, bounded-history synthesis is feasible for universal axioms in μspecRE , as we discuss in the next section. Like refinability, we can check whether an axiom satisfies extensibility (Lemma 2).

Lemma 2. *Given a universal axiom we can decide whether it satisfies extensibility.*

VII. CONVERTING TO OPERATIONAL MODELS USING AXIOM AUTOMATA

In this section, we describe our approach that converts an axiomatic model into an equivalent operational model \mathcal{M} . In §VII-A we develop *axiom automata*, which are the building blocks of our operationalization: they are automata that check for axiom compliance as the operational model executes. In §VII-B we describe how these automata can be instantiated to ensure validity for bounded programs with arbitrary μspec axioms. §VII-C holds our main result: we describe how axiom automata can be instantiated to get a finite-state bounded-history model for universal axioms in μspecRE .

We focus on a single universal axiom $\forall i_1, \dots, i_k \phi$, but this can be easily extended to a set of axioms.

A. Axiom Automata

In what follows, we fix a (universal) axiom $AX = \forall i_1 \dots \forall i_k \phi(i_1, \dots, i_k)$, and let $I(AX) = \{i_1, \dots, i_k\}$, $E(AX) = \{i.\text{st} \mid i \in I(AX), \text{st} \in \text{Stages}\}$. This axiom enforces that $\phi(\cdot)$ holds for all k -tuples of instructions in the given program. An axiom automaton is a finite state automaton that monitors whether $\phi(\cdot)$ holds for a single k -tuple of instructions. Our operational model is composed of several such automata - thereby allowing us to check all k -tuples. We now define axiom automata, starting with some auxiliary definitions.

Let $\text{nonhb}(AX)$ denote the non-hb atoms in ϕ , i.e. instruction predicate applications and $<_r$ orderings. A *context* is an assignment (of true/false) to each atom in $\text{nonhb}(AX)$; $\text{cxt} : \text{nonhb}(AX) \rightarrow \mathbb{B}$. Each variable assignment $s : I(AX) \rightarrow \mathbb{I}$ fixes the valuation of all $\text{nonhb}(AX)$ atoms (following the

semantics in §II). Hence each assignment s leads to a unique context, which we denote as $\text{cxt}(s)$.

We extend assignments to events and words over events. For $e = i.\text{st}$, we define $s(e) = s(i).\text{st}$ and for $w \in E(AX)^*$,

$$s(w) = s(w[0]) \cdots s(w[|w| - 1]) \in \mathbb{E}^*$$

As mentioned in §III-A4, we interpret $s(w) \in \mathbb{E}^*$ as the μhb graph $w[0] \xrightarrow{\text{hb}} w[1] \cdots \xrightarrow{\text{hb}} w[|w| - 1]$.

Observe that once we fix the context, the validity of $\phi(\cdot)$ only depends on the value of the hb atoms in ϕ . Hence for two assignments s_1, s_2 with the same context: $\text{cxt}(s_1) = \text{cxt}(s_2)$, s_1 and s_2 share the same set of valid executions: $s_1(w)$ satisfies ϕ if and only if $s_2(w)$ does. This implies that across different assignments s , there are only finitely many valid sets of executions over events in $s(E(AX))$ - one for each context. Intuitively, contexts divide the set of all possible assignments into classes which admit similar orderings.

As a consequence of the above, for each AX and context cxt , we can construct a finite state automaton that recognizes acceptable orderings of $E(AX)$ (Lemma 3). The main observation behind Lemma 3 is that once the context (i.e. interpretation of the nonhb(AX) atoms) is fixed, the allowed orderings can be represented as a language over the symbolic events $E(AX)$.

Lemma 3 (Axiom-Automata). *Given an axiom AX and context cxt , there exists a finite-state automaton $\text{aa}(AX[\text{cxt}])$ over alphabet $E(AX)$ with language $\{w \mid w \in E(AX)^{\text{perm}}, s(w) \models \phi(i_1, \dots, i_k)[s] \text{ for all } s \text{ that agree with } \text{cxt}\}$.*

B. Deploying axiom automata

1) *Concretization of an axiom automaton:* The automaton $\text{aa}(AX[\text{cxt}])$ mentioned in Lemma 3 recognizes orderings over the symbolic alphabet $E(AX)$ that lead to ϕ being satisfied. Our end goal, however, is identifying acceptable orderings over the (non-symbolic) events \mathbb{E} . This requires us to generate concrete instances of axiom automata, one for each assignment $s : I(AX) \rightarrow \mathbb{I}$, which we now do.

Given an assignment $s : I(AX) \rightarrow \mathbb{I}$, we denote the (*concretized*) automaton for s w.r.t AX as $\text{aa}(AX, s)$. The automaton $\text{aa}(AX, s)$ is identical to $\text{aa}(AX[\text{cxt}(s)])$, except that the symbolic alphabet $E(AX)$ replaced by its image $s(E(AX))$ under s . Intuitively (by §VII-A), the set of valid orderings of events in $s(E(AX))$ is characterized by the context of s , $\text{cxt}(s)$. This means that the acceptable orderings of events in $s(E(AX))$ is identical to the set of words (orderings) accepted by $\text{aa}(AX[\text{cxt}(s)])$, except that the symbolic events $E(AX)$ should be replaced by their concrete counterparts, $s(E(AX))$. This justifies the definition of $\text{aa}(AX, s)$.

We extend the notation $\text{aa}(AX, s)$ from a single assignment to a set of assignments. For $I \subseteq \mathbb{I}$, we denote by $\text{aa}(AX, I)$ the set of axiom automata over I : $\{\text{aa}(AX, s) \mid s : I(AX) \rightarrow I\}$.

2) *A basic operationalization:* Lemma 3 and the concretization defined in §VII-B1 suggest an operationalization for AX. For a program \mathcal{P} , if a trace σ is accepted by *all* (concrete) automata $\text{aa}(AX, \text{instrsOf}(\mathcal{P}))$ then $\sigma \models \phi[s]$ holds for each assignment s , thus satisfying AX. The number of

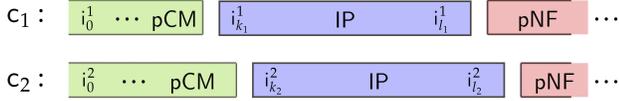


Fig. 8: Completed prefix (pCM), in-progress (IP) and not-fetched postfix (pNF) of instructions during execution.

these automata is $|\text{aa}(\text{AX}, \text{instrsOf}(\mathcal{P}))| \sim |\text{instrsOf}(\mathcal{P})|^k$ for an axiom with k universally quantified variables. Since this increases with \mathcal{P} , the model is not finite state. Even so, this enables us to construct operational models *for a given bound on* $|\text{instrsOf}(\mathcal{P})|$. We can do this even for non-universal axioms by converting existential quantifiers into finite disjunctions over $\text{instrsOf}(\mathcal{P})$. We demonstrate an application of this in §VIII, where we check that a processor satisfies an axiom ensuring correctness of read values.

C. Bounding the number of active instructions

As the discussion from §VII-B2 concludes, generating all concrete automata (statically) for arbitrary μspec specifications does not give us a finite state model. We need to bound the number of automata maintained at any point in the trace. In order to do this, for each index in the trace, we identify *active* instructions: an active instruction is one for which we need to maintain ordering information at that index. We observe that under the t -bounded reordering under-approximation, only a bounded number of instructions are active. This, in turn implies that we only need to maintain a bounded number of axiom automata. We now formalize these concepts.

For a t -reordering bounded trace σ of a program \mathcal{P} and a trace index $0 \leq j \leq |\sigma|$, let $\text{CM}(j)$ and $\text{NF}(j)$ be instructions which have executed all and none of their events at $\sigma[j]$ respectively. We define the following auxiliary terms:

$$\begin{aligned} \text{pCM}(j) &= \{i \mid \forall i'. i' \leq_r i \implies i' \in \text{CM}(j)\} \\ \text{pNF}(j) &= \{i \mid \forall i'. i \leq_r i' \implies i' \in \text{NF}(j)\} \\ \text{IP}(j) &= \text{instrsOf}(\mathcal{P}) \setminus (\text{pCM}(j) \cup \text{pNF}(j)) \end{aligned}$$

Intuitively $\text{pCM}(j)$ represents the prefix-closed set of *completed* instructions, $\text{pNF}(j)$ represents the postfix-closed set of *not-fetched* instructions, and $\text{IP}(j)$ are the rest - the *in-progress* instructions (see Fig. 8). By the first condition of t -reordering boundedness, in-progress (IP) instructions on each core are bounded by t for all j (Lemma 4):

Lemma 4. *For any t -reordering bounded trace σ , for all $0 \leq j \leq |\sigma|$, we have, $|\text{IP}(j)| \leq |\text{Cores}| \cdot t$.*

Active instructions Two instructions i, i' are k -coupled in a trace σ if they form a coupling chain of length k : i.e. there exist instructions i_1, \dots, i_{k-1} such that $\text{coup}(i, i_1), \text{coup}(i_1, i_2), \dots, \text{coup}(i_{k-1}, i')$. For trace σ , $0 \leq j \leq |\sigma|$ and $k \in \mathbb{N}$, we define k -active instructions at j , $\text{AC}_k(j)$, as instructions from $\text{pCM}(j) \cup \text{IP}(j)$ which are k -coupled with some instruction from $\text{IP}(j)$.

Intuitively, for a μspecRE axiom with k universally quantified variables, the execution of two instructions affect each

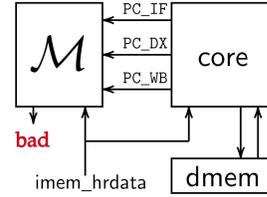


Fig. 9: Experimental setup.

other only if they are k -coupled. In particular, maintaining ordering information is important for instructions which are k -coupled with the in-progress instructions. As Lemma 5 shows, these *active* instructions - $\text{AC}_k(j)$ - are bounded at any given point in the trace.

Lemma 5. *For each k , there is a (program-independent) bound b_k , s.t. for any t -reordering bounded trace σ , for all $0 \leq j \leq |\sigma|$, we have $|\text{AC}_k(j)| \leq b_k$.*

The operational model Our operational model maintains the in-progress instructions (IP) on its tape. At each step it schedules an event from these instructions. The validity of event scheduling is ensured by maintaining orderings between events corresponding to the active instructions. Lemmas 4, 5 imply that at all points in the trace, (1) the set IP is bounded and (2) the active instructions - AC_k - are bounded (as a function of b_k). Consequently, this results in a model which has finite state (used to maintain orderings between events of AC_k) and bounded history (owing to (1)). This gives us the main result - a finite state, bounded history operational model.

Theorem 2. *For a (refinable) universal axiomatic semantics that satisfies extensibility, synthesis of finite-state, bounded-history operational models satisfying Def. 3 and 5* is feasible.*

VIII. CASE STUDIES

In this section, we demonstrate applications of operationalization. We discuss three case studies: (1) `multi_vscale` [30] is a multi-core extension of the 3-stage in-order `vscale` [31] processor, (2) `tomasulo` is an OoO processor based on [32], and (3) `sdram_ctrl` is an SDRAM-controller [33].

For each case, we instrument the hardware designs by exposing ports that signal the execution of events (e.g. the PC ports in Fig. 9). We convert axioms into an operational model \mathcal{M} based on the approach discussed in §VII. \mathcal{M} is compiled to RTL and is synchronously composed with the hardware design, where it transitions on the exposed event signals. Thus, any violating behaviour of the hardware will lead \mathcal{M} into a non-accepting (`bad`) state. Hence by specifying `!bad` as a safety property, we can perform verification of the RTL design w.r.t. the axioms. The operationalization approach enables us to perform both bounded and unbounded verification using off-the-shelf hardware model checkers. We highlight that this would not have been possible without operationalization.

We use the Yosys-based [34] SymbiYosys as the model-checker, with boolector [35] and abc [36] as backend solvers for BMC and PDR proof strategies respectively. Experiments

Instructions	PDR	BMC ($d = 20$)
ALU-R	1m46s	14m30s
ALU-I	2m11s	11m31s
Load+Store	2m18s	13m35s

Fig. 10: Proof runtimes for ($\mathbf{ax1} \wedge \mathbf{ax2}$).

are performed on an Intel Core i7 machine with 16GB of RAM. We use our algorithm to automatically generate axiom automata. The compilation of the generated automata to RTL and their instrumentation with the design is done manually. However, in the future this could be automated following the procedure developed in §VII. The experimental designs are available at <https://github.com/adwait/axiomatic-operational-examples>.

Highlights. We demonstrate how the operationalization framework enables us to leverage off-the-shelf model checking tools implementing bounded and (especially) unbounded proof techniques such as IC3/PDR. This would not have been possible directly with axiomatic models. Even when Thm. 2 does not apply (e.g. non-universal/non-extensible axioms), following §VII-B2 we can fall back on a BMC-based check over all possible programs under a bound on $|\text{instrsOf}(\mathcal{P})|$.

A. The *multi_vscale* processor

a) *Pipeline axioms on a single core:* We begin with the single-core variant of *multi_vscale*. We are interested in verifying the pipeline axioms for this core. The first axiom states that pipeline stages must be in **Fet-DX-WB** order and the second enforces in-order fetch.

$$\begin{aligned} \mathbf{ax1}: & \forall i1. (\mathbf{hb}(i1.\mathbf{Fet}, i1.\mathbf{DX}) \wedge \\ & \quad \mathbf{hb}(i1.\mathbf{DX}, i1.\mathbf{WB})) \\ \mathbf{ax2}: & \forall i1, i2. i1 <_r i2 \Rightarrow \mathbf{hb}(i1.\mathbf{Fet}, i2.\mathbf{Fet}) \end{aligned}$$

The setup schematic is given in Figure 9: \mathcal{M} is the operational model implemented in RTL (note that we could do this only because the model is finite state and requires a finite history h). Given that it is a 3-stage in-order processor, at any given point each core has at most 3 instructions in its pipeline and we can safely choose a history parameter of $h = 3$, and \mathcal{M} is complete for a reordering bound of $t = 3$. We replace the *imem_hrdata* (instruction data) connection to the core by an input signal that we can symbolically constrain. Using this input signal, we can control the program (instruction stream) executed by the core.

Verification is performed with a PDR based proof using the *abc pdr* backend. We experiment with various choices of instructions fed to the processor (by symbolically constraining *imem_hrdata*). In Fig. 10, we show the constraint and its PDR proof runtime, with BMC runtime (depth = 20) for comparison. These examples demonstrate our ability to prove unbounded correctness.

b) *Memory ordering on multi-core:* We now configure the design with 2 cores: c_0, c_1 , both initialized with symbolic load and store operations. We then perform verification w.r.t. the **ReadValues (RV)** axiom shown below. This axiom says

$ I $	$ AA $	BMC d	Time
4	16	12	3m10s
6	36	16	15m48s
8	64	20	1h58m

Fig. 11: Proof runtimes for the Read-Values axiom for different instruction counts ($|I|$).

that for any read instruction ($i1$), the value read should be the same as the most recent write instruction ($i2$) on the same address, or it should be the initial value.

$$\begin{aligned} \mathbf{RV}: & \forall i1, \exists i2, \forall i3. \mathbf{IsRead}(i1) \Rightarrow \\ & (\mathbf{DataInit}(i1) \vee (\mathbf{IsWrite}(i2) \wedge \\ & \quad \mathbf{SameAddr}(i1, i2) \wedge \mathbf{hb}(i2.\mathbf{DX}, i1.\mathbf{DX}) \\ & \quad \wedge \mathbf{ValEq}(i1, i2) \wedge ((\mathbf{IsWrite}(i3) \wedge \\ & \quad \mathbf{SameAddr}(i1, i3)) \Rightarrow \\ & \quad (\mathbf{hb}(i3.\mathbf{DX}, i2.\mathbf{DX}) \vee \mathbf{hb}(i1.\mathbf{DX}, i3.\mathbf{DX})))))) \end{aligned}$$

This not a universal axiom, and hence Thm. 2 does not apply. However, for bounded programs we can construct $|\text{instrsOf}(\mathcal{P})|^2$ concrete automata (since there are two universally quantified variables: $i1, i3$) as discussed in §VII-B2. We convert the existential quantifier over $i2$ into a finite disjunction over $\text{instrsOf}(\mathcal{P})$. We perform BMC queries for programs with $|I| = |\text{instrsOf}(\mathcal{P})| = 4, 6, 8$.

By keeping instructions symbolic, we effectively prove correctness for *all* programs within our bound $|I|$. The table alongside shows the instruction bound, $|I|$, the number of axiom automata $|AA|$, BMC depth d , and proof runtime. Though our theoretical results apply to universal axioms, this shows how an axiom automata-based operationalization can be applied to arbitrary axioms by bounding $|\text{instrsOf}(\mathcal{P})|$.

B. An OoO processor: *tomasulo*

Our second design is an out-of-order processor (based on [32]) that implements Tomasulo’s algorithm. The processor has stages: **F** (fetch), **D** (dispatch), **I** (issue), **E** (execute), **WB** (writeback), and **C** (commit). We verify in-order-commit, program-order fetch, and pipeline order axioms for this processor. A BMC proof (with $d = 20$) takes $\sim 2\text{m}$.

The axiom **axDep** given below is crucial for correct execution in an OoO processor. It enforces that execute (**E**) stages for consecutive instructions should be in program order if the destination of the first instruction is same as the source of the second, i.e. dependent instructions are executed in order.

$$\mathbf{axDep}: \forall i1, i2. (i1 <_r i2 \wedge \mathbf{Cons}(i1, i2) \wedge \mathbf{DepOn}(i1, i2)) \Rightarrow \mathbf{hb}(i1.\mathbf{E}, i2.\mathbf{E})$$

We add a program counter (*pc*) to instructions and define $\mathbf{Cons}(i1, i2) \equiv \text{pc}(i1) + 4 = \text{pc}(i2)$ and $\mathbf{DepOn}(i1, i2) \equiv \text{dest}(i1) = \text{src1}(i2) \vee \text{dest}(i1) = \text{src2}(i2)$.

As before, we compose the operational model \mathcal{M} corresponding to this axiom with the RTL design. We symbolically constrain the processor to execute a sequence of symbolic (add and sub) instructions and assert **!bad**. A BMC query

($d = 20$) results in an assertion violation. We manually identified the bug as being caused by the incorrect reset of entries in the Register Alias Table (RAT) in the `Com` stage. When committing instruction i_0 , the entry $\text{RAT}(\text{dest}(i_0))$ is reset, while some instruction i_1 with $\text{dest}(i_0) = \text{dest}(i_1)$ is issued at the same cycle. A third instruction i_2 with $\text{src1}(i_2) = \text{dest}(i_0)$ then reads the result of i_0 instead of i_1 , violating the axiom. We fix this bug and perform a BMC proof ($d = 20$), which takes $\sim 6\text{m}30\text{s}$. This demonstrates how our technique can be used to identify a bug, correct it and check the fixed design.

C. A memory controller: `sdram_ctrl`

To demonstrate the versatility of our approach, we experiment with an SDRAM controller [33], which interfaces a processor host with an SDRAM device, with a ready-valid interface for read/write requests. All intricacies related to interfacing with the SDRAM are handled by maintaining appropriate control state in the controller. In the following, we once again convert axioms into an operational model by our technique, and compose the generated model with the design.

First we verify pipeline-stage axioms for `sdram_ctrl` for write (4-stages) and read (5-stages) operations executed by the host. A PDR-based (unbounded) proof for the pipeline axioms requires $\sim 8\text{m}$. Next we verify properties related to SDRAMs refresh operation [37]. The controller ensures that the host-level behaviour is not affected by refreshes by creating an illusion of atomicity for writes and reads. This results in the axiom that once a write or read operation is underway, no refresh stage should execute before it is completed. We once again prove this property with PDR, which takes $\sim 1\text{m}30\text{s}$.

IX. RELATED WORK

There has been much work on developing axiomatic (declarative) models for memory consistency in parallel systems, at the ISA level [2], [38], [39], the microarchitectural level [12], [16], [11], and the programming language level [20], [40], [41], [42], [43]. There has also been work on constructing equivalent operationalizations for these models, e.g., for Power [2], ARMv8 [10], RA[8], C++ [7], and TSO [19], [9]. These constructions are accompanied by hand-written/theorem-prover based proofs, demonstrating equivalence with the axiomatic model. In principle, our work is related to these, however we enable *automatic* generation of equivalent operational models from axiomatic ones, eliminating most of the manual effort.

At an abstract level, we have been inspired by classic works that have developed connections between logics and automata [44], [45]. There is a large body of work on synthesis of operational implementations as well as monitors from temporal specifications (e.g. [46], [47], [48]), most commonly those written in Linear Temporal Logic (LTL) [49] and its variants (e.g. [50]). In this paper we perform a similar conversion but for a very different logic: μspec specifies constraints over partial orders while LTL does so over totally ordered traces. Additionally, the elements over which constraints are enforced

is also different: μspec constrains orderings of a known set of events, while LTL does so over traces with potentially differing sets of events (atoms). These differences make a direct comparison with the previously mentioned works ineffectual, and have required us to develop novel concepts in this work.

In terms of the application to proving properties, the work closest to ours is RTLCheck [13], which compiles constraints from μspec to SystemVerilog assertions. These assertions are checked on a per-program basis. On the other hand, we demonstrate the ability to prove unbounded correctness. Additionally, for axioms that are not generally operationalizable (for unbounded programs), we demonstrate the ability to generate an operational model for some apriori known bound on the program size. In this case, we can verify correctness for *all* programs of size upto that bound, as opposed to on a per-program basis as RTLCheck does. RTL2 μspec [51] aims to perform the reverse conversion: from RTL to μspec axioms.

X. CONCLUSION

In this paper we make strides towards enabling greater interoperability between operational and axiomatic models, both through theoretical results and case studies. We derive μspecRE , a restricted subset of the μspec domain-specific language for axiomatic modelling. We show that the generation of an equivalent finite-state operational model is impossible for general μspec axioms, though it is feasible for universal axioms in μspecRE . From a practical standpoint, we develop an approach based on axiom automata that enables us to automatically generate such equivalent operational models for universally quantified axioms in μspecRE (or for arbitrary μspec axioms if equivalence up to a bound is sufficient).

The challenges we surmount for our conversion (discussed in §I) find parallels in manual operationalization works [7], and we believe that the above concepts can be extended to formalisms such as Cat [2]. Our practical evaluation illustrates the key impact of this work—its ability to enable users of axiomatic models to take advantage of the vast number of techniques that have been developed for operational models in the fields of formal verification and synthesis.

XI. FUTURE WORK

An interesting direction for future work is to enrich μspec semantics (e.g., with quantitative operators) such that valid executions are guaranteed to satisfy t -reordering boundedness. In addition to allowing generation of finite-state operational models, we believe that such axioms would also capture processor executions more precisely.

While some aspects of executions are easier to specify operationally, others (e.g., non-deterministic scheduling) are better suited to axiomatic specifications. Another direction for future work is combining operational and axiomatic modelling, for example using tools such as UCLID5 [52], [53].

ACKNOWLEDGMENTS

This work was supported in part by Intel under the Scalable Assurance program and by DARPA contract FA8750-20-C-0156.

REFERENCES

- [1] Christel Baier and Joost-Pieter Katoen. Principles of model checking. 2008.
- [2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2), July 2014.
- [3] Yatin A. Manerkar. *Progressive Automated Formal Verification of Memory Consistency in Parallel Processors*. PhD thesis, Princeton University, Princeton, NJ, USA, 2020.
- [4] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, 1994.
- [5] Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.
- [6] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134, 2011.
- [7] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for C/C++11 concurrency. In *OOPSLA*, 2016.
- [8] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.
- [9] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLS*, 2009.
- [10] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proceedings of the ACM on Programming Languages*, 2:1 – 29, 2018.
- [11] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhat-tacharjee. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [12] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646, 2014.
- [13] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 463–476, 2017.
- [14] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 788–801, 2018.
- [15] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate : Automated exploit program generation for hardware security verification. 2018.
- [16] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μ hb graphs to verify the coherence-consistency interface. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 26–37, 2015.
- [17] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. Automated Conversion of Axiomatic to Operational Models: Theory and Practice. <https://arxiv.org/abs/2208.06733>, 2022.
- [18] Jeremy Manson. The Java memory model. In *POPL '05*, 2005.
- [19] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors. *Communications of the ACM*, 53:89 – 97, 2010.
- [20] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL '11*, 2011.
- [21] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 2005.
- [22] Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. Reconciling event structures with modern multiprocessors. *ArXiv*, abs/1911.06567, 2020.
- [23] Alan Jeffrey and James Riely. On thin air reads towards an event structures model of relaxed memory. *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–9, 2016.
- [24] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013.
- [25] Stavros Aronis. Effective techniques for Stateless Model Checking. 2018.
- [26] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: Model checking for hardware memory models. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.
- [27] Jean Berstel. Transductions and context-free languages. In *Teubner Studienbücher : Informatik*, 1979.
- [28] Jacques Sakarovitch. Elements of automata theory. 2009.
- [29] Luc Maranget, Jade Alglave, Susmit Sarkar, and Peter Sewell. Litmus: Running Tests against Hardware. In *TACAS'11, 17th International Conference on Tools And Algorithms for the Construction and Analysis of Systems*, Saarbrücken, Germany, March 2011.
- [30] Yatin A. Manerkar. multi-vsacle. https://github.com/ymanerka/multi_vsacle/tree/multicore.
- [31] LGTMCU. vscale. <https://github.com/LGTMCU/vscale>. [Online; accessed 11-05-2021].
- [32] Soham-Das-2021. Tomasulo. <https://github.com/Soham-Das-2021/Tomasulo-Machine>. [Online; accessed 11-05-2021].
- [33] Stafford Horne. SDRAM controller. <https://github.com/stffrdhrn/sdram-controller>. [Online; accessed 11-05-2021].
- [34] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-A Free Verilog Synthesis Suite. 2013.
- [35] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.
- [36] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, release 70930. <http://www.eecs.berkeley.edu/~alanmi/abc/>.
- [37] Bruce Jacob, Spencer W. Ng, and David T. Wang. Memory systems: Cache, DRAM, disk. 2007.
- [38] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10:282–312, 1988.
- [39] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2.
- [40] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.
- [41] Viktor Vafeiadis, Thibaut Balabonski, Soham Sundar Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.
- [42] Conrad Watt, Christopher Pulte, Anton Podkopaev, G. Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu yu Guo. Repairing and mechanising the JavaScript relaxed memory model. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.
- [43] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.
- [44] J. Richard Büchi. On a decision method in restricted second order arithmetic. 1990.
- [45] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 185–194, 1983.
- [46] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15:519–539, 2012.
- [47] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS*, 2002.
- [48] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6:158–173, 2003.
- [49] Zohar Manna and Amir Pnueli. The temporal logic of reactive and concurrent systems. In *Springer New York*, 1992.
- [50] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M. Murray, and Sanjit A. Seshia. Reactive synthesis from signal temporal logic specifications. *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, 2015.
- [51] Yao Hsiao, Dominic P. Mulligan, Nikos Nikolieris, Gustavo Petri, and Caroline Trippel. Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations. *MICRO-54*:

54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021.

- [52] Sanjit A. Seshia and Pramod Subramanyan. UCLID5: Integrating modeling, verification, synthesis and learning. *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–10, 2018.
- [53] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeuffer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. UCLID5: Multi-modal formal modeling, verification, and synthesis. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 538–551, Cham, 2022. Springer International Publishing.

Formally Verified Quite OK Image Format

Mario Bucev

School of Computer and Communication Sciences
EPFL

1015 Lausanne, Switzerland
mario.bucev@epfl.ch

Viktor Kunčák

School of Computer and Communication Sciences
EPFL

1015 Lausanne, Switzerland
viktor.kuncak@epfl.ch
<https://orcid.org/0000-0001-7044-9522>

Abstract—Lossless compression and decompression functions are ubiquitous operations that have a clear high-level specification and are thus suitable as verification benchmarks. Such functions are also important. On the one hand, they improve the performance of communication, storage, and computation. On the other hand, errors in them would result in a loss of data. These functions operate on sequences of unbounded length and contain unbounded loops or recursion that update large state space, which makes finite-state methods and symbolic execution difficult to apply.

We present deductive verification of an executable Stainless implementation of compression and decompression for the recently proposed Quite OK Image format (QOI). While fast and easy to implement, QOI is non-trivial and includes a number of widely used techniques such as run-length encoding and dictionary-based compression. We completed formal verification using the Stainless verifier, proving that encoding followed by decoding produces the original image. Stainless transpiler was also able to generate C code that compiles with GCC, is inter-operable with the reference implementation and runs with performance essentially matching the reference C implementation.

Index Terms—formal verification, compression, Stainless, SMT solver, mechanized induction

I. INTRODUCTION

Lossless conversions are ubiquitous. Examples include compression tools such as zip, as well as lossless image formats such as PNG. Unfortunately, common compression formats, especially ones for pictures, are more complex than one would expect a first. As a result of this complexity and the absence of precise specifications, it has proven difficult to reason about implementations of these algorithms. Consequently, the practice in the field is to use software testing, possibly backed by advanced testing algorithms [1], which do not guarantee correctness. As a reaction to the complexity of existing formats, Dominic Szablewski announced the “quite OK image format” [2] on 24 November 2021. The proposal was accompanied by a concise and efficient implementation. It attracted significant attention, with re-implementations quickly emerging in different programming languages (including Verilog) as well as variations such as streaming implementations.

Inspired by these developments, this paper presents an executable and formally verified implementation of the quite OK image encoding and decoding algorithms. We have presented

this formal development and shared the code on GitHub as part of the ASPLOS 2022 tutorial at EPFL in March 2022 [3], but no reviewed record of the work existed until now. The verified case study is now also available at:

<https://github.com/epfl-lara/bolts/tree/master/qoi/>

We are not aware of a formally verified implementation of functional correctness of QOI. Recently, a blog appeared referring to an implementation in Ada/SPARK¹. Our understanding is that this Ada/SPARK implementation only proves the absence of run-time errors and not full correctness.

In a broader line of work, formal verification was applied either to specific algorithms or domain-specific languages. The Deflate algorithm [4] specification has been formalized, implemented, and verified in [5] in Coq. Researchers also formalized common lemmas in information theory in Coq and apply these to Shannon-Fano codes [6].

Related approaches verify serialization tasks, which do not typically aim to compress data. Examples of such work include [7] formally verified Protocol buffer compiler implementation in Coq, for a commonly used subset of this serialization format. Correct by construction pretty printing in parsing libraries also ensures correctness subject to certain local invertibility conditions [8, Section 6.4], as do invertible lenses [9]. Our case study may thus also provide a starting point for exploring the expressive power of provably invertible domain-specific languages for data transformation.

II. BACKGROUND

A. Stainless Verifier and C Transpiler

Stainless [3], [10]–[12] accepts as input source code in a subset of the Scala programming language [13]. Typical Stainless programs can thus be compiled using the existing Scala compilers and run using the Java Virtual Machine.

Stainless supports formal verification of assertions, preconditions, postconditions, and invariants using the Inox solver. Inox in turn relies on unfolding of function definitions and uses SMT solvers, notably Z3, CVC4, and Princess.

Stainless also supports generation of C code (transpilation) for a subset of Scala. This subset targets programs without heap-allocated memory, in the spirit of our previous case study [14]. We wrote our QOI format case study to meet the

This project is supported in part by the EPFL School of Computer and Communication Sciences as well as the Swiss Science Foundation Project 200021_197288.

¹<https://blog.adacore.com/quite-proved-image-format>

expectations of the C code generator; it is the generated C code that we use for the performance comparison (Section IV-C).

B. QOI Format Overview

To encourage subsequent verification efforts and comparisons, we summarize here the QOI format definition. The format is structured with a header, followed by the actual data, and terminated by a marker (7 zero bytes followed by 01₁₆). Table I describes the header format. Images are encoded in a row-major order (left-to-right, top-to-bottom).

QOI encoder is single-pass. It manipulates the following data structures:

- The image to encode pixels. Each pixel is constituted of *chan* bytes.
- The current index *pxPos* within pixels (multiple of *chan*), the current pixel *px*, as well as the previous pixel *pxPrev* (initialized to $R = G = B = 0$ and $A = 255$).
- The encoded image bytes and the output position *outPos* within bytes.
- *index*, an array of 64 pixels denoting previously-seen pixels. It is zero-initialized.
- *run*, counting the number of equal consecutive pixels (initialized to 0).

In the following, we write *px.r*, *px.g*, *px.b*, *px.a* to refer to the red, green, blue, and alpha channels of a pixel *px*. When a pixel does not have an alpha channel, we default *px.a* to 255.

Each pixel is encoded in one of four different cases, two of which have two subcases. Encoded pixels are written in tagged chunks, uniquely identifying the applied (sub)case. The details of the chunk formats and computations can be found in [2].

Case A. If $px = pxPrev$, we increment the run counter. Whenever it reaches 62, we write a *run chunk*, reset *run* to 0 and continue with the next pixel.

Otherwise, if $px \neq pxPrev$ and $run > 0$, we write a run chunk as well, reset *run* to 0 and proceed to encode *px* using the remaining three methods.

Case B. We compute a hash of the current pixel *px*, denoted by *colorPos(px)*. The hash function is set by the QOI standard and yields a non-negative number smaller than 64. Then, if $index(colorPos(px)) = px$, we write an *index chunk* using the computed position and proceed with the next pixel. Otherwise, we update $index(colorPos(px))$ with *px* and encode *px* using the two remaining methods.

Cases C.i and C.ii. The idea is to encode a difference between the current and previous pixel, provided the difference

TABLE I
QOI FILE HEADER STRUCTURE. OFFSET AND SIZE ARE GIVEN IN BYTES.

Name	Offset	Size	Description
Magic	0	4	qoif to indicate a QOI image
w	4	4	Image width in pixels (in big-endian)
h	8	4	Image height in pixels (in big-endian)
chan	12	1	Channels: 3 for RGB; 4 for RGBA
Color space	13	1	0: sRGB with linear alpha, 1: all channels linear (informative)

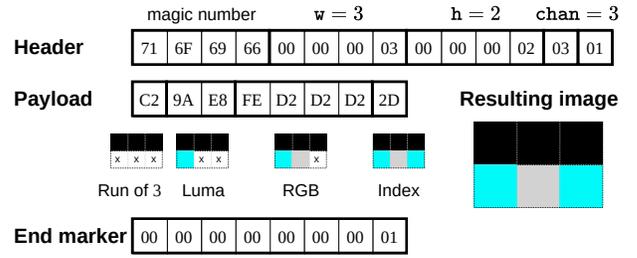


Fig. 1. Example of a Compressed Image in QOI format

is “small enough”. This case comes with two variants: the *diff* subcase (C.i) with a chunk size of 1 byte and the *luma* subcase (C.ii) for larger magnitudes with a chunk size of 2 bytes.

Cases D.i and D.ii. Whenever all above cases do not apply, we resort to encoding the full RGB value if $px.a = pxPrev.a$ (D.i) or the full RGBA value otherwise (D.ii).

Decompression is single-pass as well and maintains the same data structures as the compression counterpart. The decoder iterates over all chunks and applies the reverse transformation.

Example of decoding an image. Consider the encoded QOI image depicted in fig. 1. Squares denote bytes in hexadecimal while thick black boxes delimit the chunks. Though this figure actually transcribes the shown 3×2 image in the QOI format, knowing the exact details of the computations is unnecessary for this discussion.

The decoder starts with a black and opaque *pxPrev*. It reads the first data byte (C2₁₆) and uniquely identifies a run chunk indicating to repeat the previous pixel *pxPrev* 3 times (case A). The decoder then proceeds with the next chunk.

The following 9A₁₆ signals this byte and the following one, E8₁₆, constitutes a luma chunk (case C.ii). The decoder computes a cyan² pixel based on the previous pixel and the differences stored in this chunk. Before moving on, this pixel is stored in *index* at the position given by *colorPos(·)*.

Next, FE₁₆ identifies an RGB chunk (case D.i) with three following repeating bytes D2₁₆, producing a light gray pixel. The decoder computes a position for this pixel and stores it in *index* (which happens to not collide with the previous cyan pixel).

Finally, 2D₁₆ specifies an index chunk (case B) with the position of the cyan pixel decoded previously.

III. VERIFICATION APPROACH

We proved two classes of properties (memory safety is ensured by the programming language model):

- Runtime safety: for any input, the encoder and decoder do not access arrays out of bound or throw exceptions.
- Correctness: decoding is the inverse of encoding (invertibility).

It is much less work to show only the first property, so we focus our presentation on the second one.

²Dark gray in monochromatic.

To prove correctness, we proceed by “running” the encoder on an arbitrary but fixed input and decode the image at the same time as it is encoded. Once we are finished, the decoded image must be the same as the original one.

We establish not only separate invariants for the encoder and decoder’s respective states, but also an invariant that ties them. For example, if the encoder encounters a sequence of repeating pixels (case A), it delays writing down the chunk until the end of this sequence. In such a case, the decoder is expected to lag behind the encoder. On the other hand, for cases B, C and D, both the encoder and decoder are expected to advance at the same pace and are, in some sense, synchronized.

Then, given encoder and decoder states satisfying the invariants, we show that encoding a single pixel and decoding it should give the same pixel while maintaining these invariants. We then generalize this result to the entire image, leveraging induction.

To describe invertibility in Stainless, we write plain Scala code in terms of `encode` and `decode`, and provide the appropriate conditions. Before presenting the inversion theorem, we deem it helpful first to introduce some definitions.

The following snippet contains the declarations of three records (or *case classes* in Scala’s terminology). For conciseness, we abbreviate `a: T`, `b: T`, `c: T` to `a`, `b`, `c: T` below.

```
// Encoding context
case class EncCtx(pixels: Array[Byte], w, h, chan: Long) {
  // invariants on the fields (only one conjunct shown)
  require(pixels.length == w * h * chan)
}
case class EncodedResult(encoded: Array[Byte], length: Long)
case class DecodedResult(pixels: Array[Byte], w, h, chan: Long)
```

`EncCtx` contains the input of the encoder: the image (pixels, an array of RGBA bytes) as well as its dimensions and the number of channels. As these values may not be arbitrary (for instance, we must have `pixels.length == w * h * chan`), we add a **require** clause that specifies an invariant over these fields. Stainless then injects these assumptions into proofs when the values of the type appear in verification conditions.

`EncodedResult`, as its name suggests, holds the result of the encoding process. As `encoded` must be big enough to account for the worst case, the `length` field indicates the effective size of the compressed image.

We can now state the “invertibility theorem” with the `decodeEncodelsIdentityThm` function in the snippet below³.

```
def encode(ctx: EncCtx): EncodedResult = ...
def decode(bytes: Array[Byte], /* exclusive end index for decoding: */
  until: Long): Option[DecodedResult] = ...

def decodeEncodelsIdentityThm(ctx: EncCtx): Boolean = {
  val res = encode(ctx)
  decode(res.bytes, res.length) match
  case Some(DecodedResult(decoded, w, h, chan)) =>
    w == ctx.w && h == ctx.h && chan == ctx.chan &&
    // Predicate for comparing arrays within a range
    arraysEq(ctx.pixels, decoded, 0, pixels.length)
  case None() => false // i.e. should be unreachable
}.holds
```

³For brevity of presentation, code and specification snippets may slightly differ from the actual case study available on the URL shown in the introduction.

The **.holds** construct in `decodeEncodelsIdentityThm` asks Stainless to prove the following. Given a valid `EncCtx` – representing the encoder input – satisfying its stated invariant, if we feed the result `res` of the encoder to the decoder, it always succeeds (by having `case None()` returning **false**). Additionally, the decoded dimensions and number of channels correspond to the original input. Furthermore, the original and decoded images are equal.

To help Stainless prove this theorem, we must establish contracts for several functions, provide sufficient proof annotations to guide the solver, and write lemmas – which are just (possibly recursive) functions stating a property. However, `decodeEncodelsIdentityThm` does not contain any proof annotation, as everything needed to derive the conclusion is contained in the definitions of `encode` and `decode`.

In fact, `encode` and `decode` contain few annotations. They delegate the work (alongside the proofs) to `encodeLoop` and `decodeLoop`. In particular, `encodeLoop` iterates (through recursion) over the pixels and invokes `encodeSingleStep` for the actual work. By stating a sufficiently strong induction hypothesis (IH) on `encodeLoop` and combining the IH with the properties of `encodeSingleStep`, we obtain proof of invertibility.

As `encodeLoop` is “just” gluing the pieces together, we instead present `encodeSingleStep`:

```
// Pixel read from the pixels array, updated output
// position within the bytes array and updated run.
case class EncodingIteration(px: Int, outPos, run: Long)

// Contains the state of the decoder, that is mutated
// in encodeSingleStep ('var' marks a field as mutable).
case class GhostDecoded(var index: Array[Int],
  var pixels: Array[Byte], var inPos, var pxPos: Long)

def encodeSingleStep(index: Array[Int], bytes: Array[Byte],
  pxPrev: Int, run0, outPos0, pxPos: Long, ctx: EncCtx,
  @ghost decoded: GhostDecoded): EncodingIteration = // ...
```

`encodeSingleStep` returns `EncodingIteration` that gives the last read pixel (`px`) and one-past-the-end position of the last written byte (`outPos`). For a sequence of repeating pixels, the `run` field of the returned record is incremented. Otherwise, the encoded pixels are written (in-place) in bytes and `outPos` is updated accordingly.

Notably, `encodeSingleStep` takes a ghost parameter, `decoded`, which models the decoder state that would arise during possible future decoding runs. Ghost variables are subject to ghost elimination, which we discuss in more detail in IV-C. Intuitively, ghost variables allow tracking some extra information that may only be used for contracts and proof annotations: in particular, they cannot influence the execution of the algorithm [15].

The precondition of `encodeSingleStep` requires that the decoder state is consistent: for instance, the currently decoded pixels correspond to the original ones. At the end of the function, before returning, we “run” the decoder on `decoded` by calling `decodeLoop` with the updated index and bytes arrays.

Then, we can express local invertibility as follows. If we run the decoder from the *old* `decoded` state (i.e. before entering `encodeSingleStep`) on the bytes we wrote when executing

encodeSingleStep, then the decoded pixels must correspond to the pixels that have been encoded.

To prove this key property, we proceed in two phases, akin to how the encoder proceeds. The snippet below shows an excerpt of the encodeSingleStep, highlighting these two phases.

```
// Record returned by updateRun
case class RunUpdate(reset: Boolean, run, outPos: Long)

def encodeSingleStep(...) = {
  // ... Some preconditions
  // A copy of the "original" index, will be erased by ghost elimination:
  @ghost val oldIndex = freshCopy(index)
  // Phase 1: Run-length processing (case A)
  val runUpd = updateRun(bytes, run0, outPos0)
  val run1 = runUpd.run
  val outPos1 = runUpd.outPos
  // The premise holds when flushing (writing down the run chunk)
  assert(runUpd.reset ==>
    updateRunProp(pxPrev, px, bytes, run0, outPos0, outPos1))
  // ... other assertions
  // Phase 2: Encode pixel individually (cases B, C, D)
  val outPos2 = if px != pxPrev then
    val outPos2 = encodeNoRun(index, bytes, outPos1)
    // ... some assertions and lemmas to support this claim
    assert(encodeNoRunProp(pxPrev, px, oldIndex, index, bytes,
      outPos1, outPos2))
  else
    // ... assertions stating invariants are preserved
    outPos1
  // ... assertions to glue everything together
  EncodingIteration(px, outPos2, run1)
} ensuring /* postconditions stating distilled properties */
```

First, the encoder handles the run-length part of the algorithm, corresponding to case A as described in II-B. The work is delegated to updateRun and returns a record telling (through the reset field) whether a run chunk was written to bytes. If not, then invertibility is of course preserved as the encoded pixels are left untouched. Otherwise, updateRun guarantees that reading the written chunk gives us a run chunk whose value is the run counter we have just written – expressed with updateRunProp, presented afterward.

Second, in the case where the previous and current pixels are different, the encoder picks methods B, C or D to encode the current pixel. The task is handed over to encodeNoRun and states with encodeNoRunProp that reading the written chunk yields back the pixel.

updateRunProp and encodeNoRunProp both use doDecodeNext to decode the written chunk. The latter returns an ADT with two variants describing the decoded chunk. Run(r) indicates a run chunk with $r + 1$ repeating pixels. The $+1$ is a result of the run counter being shifted by one when encoded. DiffOrIndexOrColor(px) denotes a pixel encoded by method B, C or D. Due to the variable length nature of chunks, doDecodeNext also returns the position of the next chunk to be decoded (if any).

```
enum DecodedNext:
  case Run(run: Long)
  case DiffOrIndexOrColor(px: Int)

def doDecodeNext(bytes, index: Array[Int],
  pxPrev: Int, inPos0: Long): (DecodedNext, Long) = ...
```

Expressing the desired properties is then a matter of pattern-matching over the result of doDecodeNext and tying it with appropriate equalities.

```
def updateRunProp(pxPrev, px: Int, bytes: Array[Byte],
  run0, outPos0, outPos1: Long): Boolean =
  // ... preconditions including e.g. ordering on outPos0, outPos1
  // If px == pxPrev, the current run counter run0 is incremented
  // (reflected by the conditional +1).
  val run = run0 + bool2int(px == pxPrev)
  // The index does not matter for this case, we give an arbitrary array
  val dummyIndex = Array.fill(64)(0)
  doDecodeNext(bytes, dummyIndex, pxPrev, outPos0) match
    case (Run(r), inPos) => r + 1 == run && inPos == outPos1
    case _ => false

// oldIndex refers to the index at the beginning of encodeSingleStep
def encodeNoRunProp(pxPrev, px: Int, oldIndex, index: Array[Int],
  bytes: Array[Byte], outPos1, outPos2: Long): Boolean =
  // ... preconditions including e.g. ordering on outPos1, outPos2
  doDecodeNext(bytes, oldIndex, pxPrev, outPos1) match
    case (DiffOrIndexOrColor(decodedPx), inPosRes) =>
      decodedPx == px && inPosRes == outPos2 &&
      oldIndex.updated(colorPos(px), px) == index
    case _ => false
```

We rely on Inox (Stainless’ underlying solver) to unfold function definitions to prove that the calls to updateRunProp and encodeNoRunProp in encodeSingleStep hold. To help with the proof, we also provide assertions whose content is similar to the properties stated by updateRunProp and encodeNoRunProp.

Now that we have these two invertibility properties, we show that the composition of these two phases preserves invertibility by tying all facts together (see the end of the body of encodeSingleStep in the source code of encoder.scala).

IV. RESULTS

We first present some statistics and remarks about the verification before considering the performance of the generated C code with respect to the reference implementation.

For all experiments, we used a server with $2 \times$ Intel® Xeon® CPU E5-2680 v2 at 2.80GHz (release date Q3’13, for a total of 20 physical cores) running on Ubuntu 20.04.3 LTS.

A. Verification Statistics

Our QOI implementation in Scala without annotations consists of 313 lines of code (LOC)⁴. The annotated version has 2789 LOC, of which 1405 are for lemmas and helpers. This yields a ratio of 8.9 lines of specifications per executable line. The specification lines include 42 lemmas, 19 of which are general purpose and could become part of the standard library.

Table II shows for each category of verification condition (VC) their respective numbers and their cumulative times. It took roughly 1h30min to verify all VCs. The lower quartile, the median, and the upper quartile are 0.5s, 1.8s, and 5.7s respectively. Around 9.5% of VCs took more than 30s to verify, the highest being 3min.

For each function call, Stainless generates VCs corresponding to the function preconditions. Assertions annotations and postconditions of functions are translated into VCs as well.

⁴Counted with `cloc v1.82`

TABLE II
SUMMARY OF THE VERIFICATION CONDITIONS.

Verification Condition	#	Total time [min]
Preconditions	2387	370.9
Body assertions	787	203.3
Postconditions	145	31.2
Array index within bounds	126	4.9
Remainder by zero	87	10.6
Non-negative measure	23	2.1
Class invariant	21	1.5
Cast correctness	6	0.1
Match exhaustiveness	5	0.4
Measure decreases	4	4.4
Total	3591	629.4

Stainless furthermore generates other runtime safety verification conditions, such as array bounds checks and remainder by zero checks. It is sometimes necessary to provide sufficient annotations (e.g., assertions and invariants) to help Stainless prove these VCs.

B. Verification Effort

The case study was implemented and formally verified by the first author (who had a few months of experience with Stainless) over the period of approximately 4 to 5 weeks.

We have first implemented a version closely following the C reference version. Though we could prove runtime safety, describing deeper properties turned out to be difficult. For example, we could not refer to the result of decoding a range, but only the end-to-end decompression result of the entire image.

We have thus rewritten the implementation multiple times making both small and larger changes. Since the encoder and decoder are succinct, the rewrites took a relatively small amount of time compared to the remaining verification effort.

During repeated verification runs, the VC cache and the ability to selectively verify only provided functions greatly speed up the interactive experience. For example, making a few changes to a previously verified version requires less than two minutes to check all VCs, compared to the 1h30min for a clean-state re-run.

C. Generated C Code and Its Efficiency

We compare the encoding and decoding throughput of the transpiled C code with the reference implementation. Though the primary goal of the reference is simplicity, its decoding and encoding throughput are respectively 3.4x and 29x higher than `libpng` while achieving a similar compression ratio⁵.

As briefly mentioned in IV-B, we make use of ghost states for proving invertibility. Stainless first checks for correct usage of ghost variables before eliminating them in a phase of the C transpiler. Assertions and functions contracts are removed as well⁶. In summary, “proof infrastructure” is erased and incurs no cost at runtime.

⁵Derived from the section “Grand total for images (AVG)” at <https://qoiformat.org/benchmark/> (consulted the 11.08.2022).

⁶To ensure removal, developers should import the `StaticChecks` library.

The generated C code is 661 LOC long, against 311 for the reference implementation. For the purpose of evaluation, we also wrote unverified glue C code that performs I/O. We do not make any correctness claims about this code, only about the part that converts arrays of bytes between uncompressed and compressed form. We evaluated the throughput of the generated C code (`genc-qoi`) against the reference implementation (`qoi`) using a modified version of the benchmark utility shipped with `qoi`. We run the benchmark with 3 runs over 7 images ranging from 3 to 13.8 megapixels, and report the result in table III.

We compiled all involved C sources using GCC 11.1.0 with `-O3`. As our implementation uses tail recursion, so does the generated C code⁷. It is necessary to pass an optimization level of at least `-O2` or explicitly pass the `-foptimize-sibling-calls` to GCC in order to have the tail calls eliminated.

To our surprise, the transpiled version is on-par with the reference implementation: it is approximately 7% faster in decoding and 2% slower in encoding. Disassembling the decoding functions reveals that both were compiled similarly. Nevertheless, the `genc-qoi` version uses more instructions for all cases but index decoding (case B). These extra instructions are of an arithmetic and logical nature and do not involve memory operations. For case B, GCC produced one 4-bytes memory load operation for `genc-qoi`, while it emitted four 1-byte memory load operations for `qoi`. We conjecture that the reported difference may be explained by these three extra memory loads.

TABLE III
BENCHMARK RESULTS OF QOI AND GENC-QOI

	Decoding throughput [megapixels/s]	Encoding throughput [megapixels/s]
qoi (unverified)	90.92	86.24
genc-qoi (verified)	97.65	84.45

V. CONCLUSIONS

We have presented a QOI implementation in Scala and verified with Stainless that decoding is the inverse of encoding. We have also seen that the transpiled C version matches the performance of the reference implementation. Going forward, we expect that other verified implementations will emerge and that QOI will become a useful benchmark for testing verification approaches and tools.

ACKNOWLEDGMENT

We thank FMCAD 2022 reviewers for helpful comments. We thank Georg S. Schmid for useful discussions and Jad Hamza for developing the C code generator in Stainless. We thank the organizers of ASPLOS 2022 conference for the opportunity to present a summary of the case study as one part of the tutorial.

⁷We thank GCC! Our C code generator does not (yet) eliminate tail calls.

REFERENCES

- [1] A. Kanade, R. Alur, S. Rajamani, and G. Ramanlingam, “Representation dependence testing using program inversion,” in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 277–286. [Online]. Available: <https://doi.org/10.1145/1882291.1882332>
- [2] “The Quite OK Image format for fast, lossless compression.” [Online]. Available: <https://qoifformat.org/>
- [3] “Verifying programs with Stainless (ASPLOS 2022 tutorial on Stainless).” [Online]. Available: <https://epfl-lara.github.io/asplos2022tutorial/>
- [4] L. P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3,” Internet Engineering Task Force, Request for Comments RFC 1951, May 1996, num Pages: 17. [Online]. Available: <https://datatracker.ietf.org/doc/rfc1951>
- [5] C.-S. Senjak and M. Hofmann, “An implementation of deflate in coq,” 2016. [Online]. Available: <https://arxiv.org/abs/1609.01220>
- [6] R. Affeldt, J. Garrigue, and T. Saikawa, “Examples of Formal Proofs about Data Compression,” in *2018 International Symposium on Information Theory and Its Applications (ISITA)*. Singapore: IEEE, Oct. 2018, pp. 633–637. [Online]. Available: <https://ieeexplore.ieee.org/document/8664276/>
- [7] Q. Ye and B. Delaware, “A verified protocol buffer compiler,” in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 222–233. [Online]. Available: <https://doi.org/10.1145/3293880.3294105>
- [8] R. Edelmann, “Efficient parsing with derivatives and zippers,” Ph.D. dissertation, EPFL, Lausanne, 2021. [Online]. Available: <http://infoscience.epfl.ch/record/287059>
- [9] M. Hofmann, B. Pierce, and D. Wagner, “Symmetric lenses,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 371–384.
- [10] J. Hamza, N. Voirol, and V. Kunčák, “System FR: Formalized foundations for the Stainless verifier,” *Proc. ACM Program. Lang*, no. OOPSLA, November 2019.
- [11] V. Kuncak and J. Hamza, “Stainless verification system tutorial,” in *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 2–7.
- [12] “Stainless,” 2022. [Online]. Available: <https://github.com/epfl-lara/stainless/>
- [13] M. Odersky, L. Spoon, B. Venners, and F. Sommers, *Programming in Scala (Fifth Edition, Updated for Scala 3.0)*. Artima Press, 2021.
- [14] J. Hamza, S. Felix, V. Kunčák, I. Nussbaumer, and F. Schramka, “From verified Scala to STIX file system embedded code using Stainless,” in *NASA Formal Methods (NFM)*, 2022, p. 18. [Online]. Available: <http://infoscience.epfl.ch/record/292424>
- [15] M. Abadi and L. Lamport, “The existence of refinement mappings,” in *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, July 1988, pp. 165–175, IICS 1988 Test of Time Award. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/the-existence-of-refinement-mappings/>

Split Transition Power Abstraction for Unbounded Safety

Martin Blicha*[‡] , Grigory Fedyukovich[†] , Antti E. J. Hyvärinen*  and Natasha Sharygina* 

*Università della Svizzera Italiana, Lugano, Switzerland

{blichm, hyvaeria, sharygin}@usi.ch

[†]Florida State University, Tallahassee, FL, USA

grigory@cs.fsu.edu

[‡]Charles University, Prague, Czech Republic

Abstract—Transition Power Abstraction (TPA) is a recent symbolic model checking approach that leverages Craig interpolation to create a sequence of symbolic abstractions for transition paths that double in length with each new element. This doubling abstraction allows the approach to find bugs that require long executions much faster than traditional approaches that unfold transitions one at a time, but its ability to prove system safety is limited. This paper proposes a novel instantiation of the TPA approach capable of proving unbounded safety efficiently while preserving the unique capability to detect deep counterexamples. The idea is to split the transition over-approximations in two complementary parts. One part focuses only on reachability in fixed number of steps, the second part complements it by summarizing all shorter paths. The resulting split abstractions are suitable for discovering safe transition invariants, making the SPLIT-TPA approach much more efficient in proving safety and even improving the counterexample detection. The approach is implemented in the constrained Horn clause solver GOLEM and our experimental comparison against state-of-the-art solvers shows it to be both competitive and complementary.

I. INTRODUCTION

Automated formal verification by means of model checking is popular because of the ability to both 1) find error paths for unsafe systems, and 2) *prove* the absence of error paths for safe systems. Recent techniques based on Satisfiability Modulo Theories (SMT) as well as the continuing improvements of SMT solvers [1, 12, 16, 27, 35] enable scalable applications of model checking to software verification [3]. Specifically, the idea of building a safe inductive invariant incrementally—pioneered by the hardware model checking algorithm IC3/PDR [8, 17]—has been successfully applied in several IC3-inspired approaches [10, 11, 18, 24, 29, 30], thus improving the capabilities of verification tools significantly.

Although this progress is undeniably encouraging, model checking still suffers from scalability issues associated with an exhaustive exploration of a system’s states. For many systems, a large set of states need to be observed to eventually detect a counterexample or synthesize an invariant. Multi-phase loops [39] often exhibit such behaviour, in particular. A recently introduced approach based on Transition Power Abstraction (TPA) [5] successfully attacks the first part of the problem. It uses abstraction to summarize the reachability of an exponentially increasing number of steps. Thus TPA can quickly focus on the essential part of the search space and not waste time examining short paths that cannot lead to a counterexample. Interestingly, the abstractions that enable TPA

to detect long counterexample paths quickly can also be used to prove safety by discovering safe transition invariants [5]. However, the required condition that the over-approximating relation must be closed under composition with transition relation is rarely satisfied, and the algorithm performs rather poorly on safe systems.

In this paper we leverage the ideas from TPA that enable a fast exploration of large parts of the state space to detect invariants in the system that hold only after a specific (often very large) number of transitions. Our new approach, called SPLIT-TPA, also uses the idea of the transition power abstraction sequence but computes the abstractions in a different way that generates significantly more suitable candidates for transition invariants. In the original TPA sequence n^{th} element over-approximates reachability up to 2^n steps of the transition relation. The TPA sequence is used to check reachability by doubling the number of explored states at every iteration of the verification run. At the same time the sequence is expanded and its elements are refined as a direct consequence of information learned in these bounded reachability checks.

The novelty of SPLIT-TPA lies in splitting the over-approximating sequence into two complementary parts: $\text{TPA}^=$ and $\text{TPA}^<$. Elements of $\text{TPA}^=$ summarize paths of a fixed number of steps: n^{th} element covers *exactly* 2^n steps of the transition relation. The elements of $\text{TPA}^<$ complement the first sequence: n^{th} element summarizes all paths of length less than 2^n . The abstractions of $\text{TPA}^=$ sequence allow SPLIT-TPA to discover a special type of safe transition invariants, which are not possible to obtain in the original TPA algorithm. These invariants are composed of two orthogonal parts: one part summarizes safe transitions up to a specific bound; the second part summarizes unbounded safety, but only from that specific bound onwards. The final invariant is a disjunction of these two orthogonal parts which together cover any number of transitions. This specific structure makes these invariants suitable for proving safety of a large class of problems including some challenging instances that cannot be tackled by other state-of-the-art approaches.

We have implemented SPLIT-TPA in our publicly available CHC solver GOLEM and compared it against the original TPA approach and other state-of-the-art solvers ELDARICA and SPACER. On a set of challenging public benchmarks representing multi-phase loops [39], SPLIT-TPA significantly outperforms TPA on the safe version of these benchmarks and

This procedure returns a subset of reachable states of Tgt if there exists a path from Src to Tgt of length at most 2^{n+1} . If no such path exists, it returns an empty set. First, it checks existence of an *abstract* path consisting of *two* steps of $ATr^{\leq n}$, the n^{th} element of the TPA sequence (lines 2-3). If no such abstract path exists (line 4), then no real path of length $\leq 2^{n+1}$ exists (line 7). Additionally, $n + 1^{\text{st}}$ element of the TPA sequence is constructed or refined using *Craig interpolation* [15] (lines 5-6).

If an abstract path does exist, the procedure attempts to refine it to a real path. The refinement begins by applying quantifier elimination (QE) to determine a set of candidate *intermediate* states (line 10). These are states that can be reached from Src by one step of $ATr^{\leq n}$ and also can reach Tgt by another step of $ATr^{\leq n}$. Given a set of intermediate states, the procedure *recursively* determines the existence of a *real* path from Src to the intermediate states (line 11) and then the existence of a real path from the truly reachable intermediate states to Tgt (line 13). The bound for these recursive calls is decremented, and $n = 0$ represents the base case where no recursive calls are needed as $ATr^{\leq 0}$ represents true reachability in the system (line 9). If any of the two abstract steps cannot be refined, the procedure tries to find a new abstract path and repeats the whole process. The strengthening of $ATr^{\leq n}$ in the recursive call to ISREACHABLE with $n - 1$ guarantees that refuted abstract paths cannot repeat, and the procedure makes progress.

Note that instead of full quantifier elimination, any under-approximation can be used in ISREACHABLE. In particular, experiments in [5] showed that TPA works much better with *model-based projection* [4, 30].

One way to understand the procedure ISREACHABLE in TPA is that it mimics bounded reachability checks using a sequence of (precise) relations $R^{\leq n}$ defined inductively as

$$\begin{aligned} R^{\leq 0} &= Id \cup Tr, \\ R^{\leq n+1} &= R^{\leq n} \circ R^{\leq n}. \end{aligned} \quad (1)$$

However, this precise sequence is over-approximated by the TPA sequence. The over-approximation keeps the satisfiability queries manageable: Each $ATr^{\leq n}$ is a formula only over *two* copies of the state variables, no matter how large n is. This is guaranteed by using *Craig interpolation* to compute the abstractions. Compared to that, representing relation $R^{\leq n}$ *precisely* requires $2^n + 1$ copies of the state variables.

The TPA algorithm has been designed to detect long counterexample paths quickly and in this has achieved significant improvements over the state-of-the-art. Interestingly, the TPA sequence can also provide candidates for safe transition invariant, which could be used to prove safety. However, the capabilities of TPA in this respect are very limited, as also exhibited by the experimentation in [5].

Fig. 1 illustrates the limitations of TPA in generating safe transition invariants. The loop on the left has been studied extensively in the context of loop invariants, e.g., in [39]. We scaled the constants to better demonstrate the behaviour of TPA. TPA proves safety up to $8192 = 2^{13}$ iterations

```

x=0; y=5000;
while (x<10000) {
  if (x>=5000)
    y=y+1
  x=x+1;
}
assert (y==10000);

v=0; w=0;
assume (x>z);
while (v<1000) {
  if (x<z)
    v=v+1;
  else
    w=w+1;
  x=x+1;
  z=z+2;
}
assert (w>0);

```

Fig. 1. Examples of multi-phase loops

of the loop very quickly. Each of the first 13 top-level calls to ISREACHABLE determines bounded safety with a single satisfiability query. In the process, TPA learns that $ATr^{\leq n} \equiv x' \leq x + 2^n$ for $n = 1 \dots 13$. It utilizes the fact that x must be incremented more than 2^{13} times to exit the loop and reach the `assert`. However, in the next iteration of Algorithm 1 an abstract path of two steps of $ATr^{\leq 13}$ is discovered and the refinement process in ISREACHABLE begins. To make progress, the algorithm must refine the over-approximating relation $ATr^{\leq 13}$ in order to show that the error is not reachable in two steps of $ATr^{\leq 13}$. This requires learning a suitable relation between variables x and y . However, since $ATr^{\leq 13}$ must capture *all* paths of length $\leq 2^{13}$, it is not easy to learn such a relation. At least in our implementation, TPA is continuously discovering and refuting new abstract paths, making very little progress in refining the elements of the TPA sequence with each refutation. Due to this slow progress, the algorithm fails to prove safety in a reasonable amount of time.

The second loop depicted on the right of Fig. 1 is benchmark 17 from the suite of multi-phase benchmarks used in our experiments (Section VI). The behaviour of TPA is similar to the previous case, but this time it can find a safe invariant, though at a considerable cost, as illustrated below. It uses variable v and the fact that at least 1000 increments are required and quickly proves bounded safety up to 2^9 iterations of the loop. In the next iteration of its main procedure TPA spends a considerable amount of time in ISREACHABLE refining the abstraction and capturing the behaviour of the other variables and the relations between them. Finally, after proving safety up to 2^{11} iterations of the loop, it manages to discover a safe transition invariant.

We will see in the next section that SPLIT-TPA is able to prove the first loop safe and it can find a safe transition invariant for the second loop much faster.

IV. SPLIT TRANSITION POWER ABSTRACTION

In this section we present SPLIT-TPA, a new instantiation of the TPA approach suitable for proving unbounded safety. We start by revisiting R^{\leq} from Eq. (1) and show that the idea of *splitting* the TPA sequence arises naturally from a redundancy present in the inductive definition of R^{\leq} . Then we show how SPLIT-TPA performs bounded reachability checks with the split sequences and how it discovers safe transition invariants.

A. Overview

As mentioned previously, the TPA algorithm has been designed to be a simple and efficient procedure for detecting deep counterexample paths. It can also prove safety by discovering a safe transition invariant for the system. However, the only source of candidates for the required safe transition invariants are the elements $ATr^{\leq n}$ of the TPA sequence. $ATr^{\leq n}$ can be proved to be a transition invariant if it is closed under composition with one step of Tr . The problem is that this condition is rarely fulfilled. The abstractions $ATr^{\leq n}$ are primarily constructed as proofs of bounded safety in the system: they must summarize all paths of lengths from 0 to 2^n and they must be safe. While it is possible that such bounded proof is in fact an unbounded proof, in many cases these abstractions are not closed under composition with Tr , and the bounded proofs do not generalize to unbounded proofs.

Our solution to TPA's lack of ability to prove unbounded safety in practice is to introduce *new* source of candidates for transition invariants. We split the over-approximating TPA sequence into two complementary parts: $TPA^=$ and $TPA^<$. Elements of $TPA^=$ summarize paths of fixed length and the corresponding elements of $TPA^<$ summarize all shorter paths. While $TPA^<$ leads to similar transition invariants as TPA , $TPA^=$ leads to invariants with different structure and different properties, which allows SPLIT-TPA to prove safety of some challenging problems.

The idea of splitting is motivated not only by the need for another source of candidates for invariants, but also by a possible redundancy in the TPA algorithm, which could lead to unnecessary work. TPA sequence is based on the sequence R^{\leq} from Eq. (1). The intuition behind this inductive definition is that every path of length $\leq 2^{n+1}$ can be obtained as a concatenation of two paths of length $\leq 2^n$. However, there can be multiple ways to decompose such a path into two smaller paths (see Fig. 2) and proving one such decomposition infeasible does not entail that others are infeasible as well.

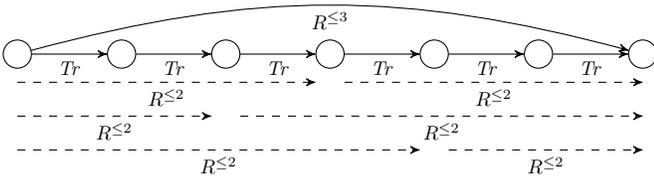


Fig. 2. Three different ways of decomposing path of length 6 into two paths of length at most 4

Splitting arises naturally from an attempt to fix this redundancy. The reasoning is as follows: Instead of concatenating two steps of $R^{\leq n}$ to obtain $R^{\leq n+1}$, we replace one of these steps with a step of $R^=n = Tr^{2^n}$, which represents reachability in *exactly* 2^n steps. However, $R^{\leq n} \circ R^=n$ covers only paths of length from 2^n to 2^{n+1} . To keep the smaller lengths covered as well, we can add $R^{\leq n}$. The result, $R^{\leq n+1} = R^{\leq n} \cup R^{\leq n} \circ R^=n$, *almost* gives us the unique deconstruction we are seeking. The exceptions are paths of length *exactly* 2^n

which are covered by both $R^{\leq n}$ and $R^{\leq n} \circ R^=n$. The final step is a realization that this last redundancy is removed by replacing the relation $R^{\leq n}$ by $R^{<n}$. The sequence $R^{<}$ has the following inductive definition:¹

$$\begin{aligned} R^{<0} &= Id, \\ R^{<n+1} &= R^{<n} \cup R^{<n} \circ R^=n, \end{aligned} \quad (2)$$

with the sequence $R^=$ also defined inductively:

$$\begin{aligned} R^=0 &= Tr, \\ R^=n+1 &= R^=n \circ R^=n. \end{aligned} \quad (3)$$

Notice that we have effectively *split* the R^{\leq} sequence into two sequences $R^{<}$ and $R^=$, because $R^{\leq n} = R^{<n} \cup R^=n$. Now, decomposing a path according to the inductive definitions from Eq. (2) and (3) is *unique*. For example, there is only one way to decompose the path of length six from Fig. 2, now viewed as one step of $R^{<3}$, according to Eq. (2): first two steps are covered by $R^{<2}$ and the last four steps are covered by $R^=2$.

Following the TPA template, we do not use the sequences $R^{<}$ and $R^=$ directly. We build over-approximating sequences $TPA^{<}$ and $TPA^=$ whose representation in terms of copies of state variables does not blow up with increasing n . The elements of the over-approximating sequences $TPA^{<}$ and $TPA^=$ are denoted as $ATr^{<n}$ and $ATr^=n$, respectively, and we require that

$$ATr^{<n} \supseteq R^{<n} = Id \cup Tr \cup Tr^2 \cup \dots \cup Tr^{2^n-1}, \quad (4)$$

$$ATr^=n \supseteq R^=n = Tr^{2^n}. \quad (5)$$

SPLIT-TPA uses these over-approximating sequences $TPA^{<}$ and $TPA^=$ both for bounded reachability checks and for detecting safe transition invariants. We will see later that $TPA^=$ sequence allows SPLIT-TPA to find interesting invariants and prove safety of challenging problems. The main procedure of SPLIT-TPA is similar to Algorithm 1 and is given in Algorithm 3.

Algorithm 3: ISSAFESPLITTPA($\langle Init, Tr, Bad \rangle$):
SPLIT-TPA's main procedure

input : transition system $\mathcal{S} = \langle Init, Tr, Bad \rangle$
global : $TPA^{<}$ sequence $ATr^{<0}, \dots, ATr^{<n}, \dots$
 $TPA^=$ sequence $ATr^=0, \dots, ATr^=n, \dots$ (lazily
initialized to *true*)

```

1  $ATr^{<0} \leftarrow Id; \quad ATr^=0 \leftarrow Tr; \quad n \leftarrow 0$ 
2 while true do
3   if ISREACHABLELT( $n, Init, Bad$ )  $\neq \emptyset$  or
     ISREACHABLEEQ( $n, Init, Bad$ )  $\neq \emptyset$  then return
     UNSAFE
4   if HASTRANSITIONINVARIANT( $\mathcal{S}, n$ ) then return SAFE
5    $n \leftarrow n + 1$ 

```

In the rest of this section we present the implementation of the methods ISREACHABLELT and ISREACHABLEEQ for bounded reachability checks and the implementation of the method HASTRANSITIONINVARIANT for discovering safe transition invariant.

¹An alternative inductive definition $R^{<n+1} = R^{<n} \cup R^=n \circ R^{<n}$ leads to a different variant of our algorithm.

B. Bounded reachability checks with $TPA^=$ and $TPA^<$

SPLIT-TPA performs the bounded reachability check at level n in two phases. First, all paths of length *strictly smaller* than 2^{n+1} are checked in ISREACHABLELT. Then all paths of length *exactly* 2^{n+1} are checked in ISREACHABLEEQ.

To implement ISREACHABLEEQ, we can reuse Algorithm 2, with the modification that all references to TPA sequence and its elements $ATr^{\leq n}$ are replaced by $TPA^=$ sequence and its elements $ATr^{=n}$ (we do not repeat the pseudocode for the sake of space). To understand why this works, compare the inductive definitions of the underlying sequences R^{\leq} and $R^=$ from Eq. (1) and (3). The induction step is *the same* in both cases. The only difference is the base case: $TPA^=$ sequence starts with $ATr^{=0} = R^{=0} = Tr$, as opposed to $ATr^{\leq 0} = R^{\leq 0} = Id \cup Tr$. The output of ISREACHABLEEQ is either a non-empty subset of Tgt that is truly reachable from Src in exactly 2^{n+1} steps of Tr , or an empty set if no path from Src to Tgt of length 2^{n+1} exists.

The procedure ISREACHABLELT is designed to complement ISREACHABLEEQ by covering all paths with $<2^{n+1}$ steps. The implementation is given in Algorithm 4. It follows the inductive definition of $R^<$ from Eq. (2) in the same manner as ISREACHABLEEQ follows the inductive definition of $R^=$.

Algorithm 4: Reachability query using $TPA^<$ sequence

```

input : level  $n$ , source states  $Src$ , target states  $Tgt$ 
output: subset of target states truly reachable in  $<2^{n+1}$  steps
global:  $TPA^<$  sequence  $ATr^{<0}, \dots, ATr^{<n}, \dots$ ,
          $TPA^=$  sequence  $ATr^{=0}, \dots, ATr^{=n}, \dots$ 
1 while true do
2    $opt1 \leftarrow ATr^{<n}[x' \mapsto x'']$ 
3    $opt2 \leftarrow ATr^{<n}(x, x') \wedge ATr^{=n}(x', x'')$ 
4    $q \leftarrow Src(x) \wedge (opt1 \vee opt2) \wedge Tgt(x'')$ 
5    $sat\_res, model \leftarrow CHECKSAT(q)$ 
6   if  $sat\_res = UNSAT$  then
7      $Itp(x, x'') \leftarrow GETItp(opt1 \vee opt2, Src(x) \wedge Tgt(x''))$ 
8      $ATr^{<n+1} \leftarrow ATr^{<n+1} \wedge Itp[x'' \mapsto x']$ 
9     return  $\emptyset$ 
10  else
11    if  $n = 0$  then return  $QE(\exists x, x' : q)[x'' \mapsto x]$ 
12    if  $model \models opt1$  then
13       $TgtReach \leftarrow ISREACHABLELT(n-1, Src, Tgt)$ 
14      if  $TgtReach = \emptyset$  then continue
15      return  $TgtReach$ 
16    else
17       $Inter \leftarrow QE(\exists x, x'' :$ 
18         $Src(x) \wedge opt2 \wedge Tgt(x''), x')[x' \mapsto x]$ 
19       $InterReach \leftarrow ISREACHABLELT(n-1, Src, Inter)$ 
20      if  $InterReach = \emptyset$  then continue
21       $TgtReach \leftarrow$ 
22         $ISREACHABLEEQ(n-1, InterReach, Tgt)$ 
23      if  $TgtReach = \emptyset$  then continue
24      return  $TgtReach$ 

```

ISREACHABLELT first assembles the query for an abstract path (lines 2–4) and sends it to the satisfiability solver (line 5). Following the inductive definition of Eq. (2), the abstract path consists of either one step of $ATr^{<n}$ or a step of $ATr^{<n}$

followed by a step of $ATr^{=n}$. If no such abstract path exists (line 6), the procedure reports that no real path of length $<2^{n+1}$ exists (line 9). Before reporting the result, it uses Craig interpolation [15] to refine the abstraction at the next level (line 8).

If an abstract path exists (line 10), the procedure checks whether there is a corresponding real path. On level 0 (line 11), the discovered abstract path is real, and the procedure returns a reachable subset of target states. On other levels, the procedure first needs to determine which abstract path has been found and then try to refine it.

The first possibility is that the abstract path is a single step of $ATr^{<n}$ (line 12). The refinement of this single abstract step is checked with a single recursive call. If the refinement is not successful, the procedure attempts to find a new abstract path (line 14). Otherwise, the reached target states from the recursive call are returned (line 15).

The second possibility is that abstract path consists of one step of $ATr^{<n}$ followed by one step of $ATr^{=n}$ (line 16). One after another, the procedure attempts to refine these abstract steps into a real path by calling the corresponding procedures ISREACHABLELT and ISREACHABLEEQ with decreased bound. If any of the two steps cannot be refined, that abstract path has been refuted and the procedure attempts to find a new abstract path (lines 19, 21). If both abstract steps have been successfully refined, a reachable subset of target states is reported (line 22).

Similarly to Algorithm 2, quantifier elimination can be replaced by its under-approximation, such as model-based projection, and we do so in our implementation.

The correctness of the reachability procedures guarantees the correctness of UNSAFE answer of SPLIT-TPA.

Lemma 1: If ISREACHABLEEQ(n, Src, Tgt) or ISREACHABLELT(n, Src, Tgt) returns a non-empty set Res , then $Res \subseteq Tgt$ and every state in Res can be reached from some state in Src in exactly 2^{n+1} steps (for ISREACHABLEEQ) or in $<2^{n+1}$ steps (for ISREACHABLELT).

Proof: By induction on n , relying on the properties of quantifier elimination (QE) and the fact that $ATr^{<0} = Id$ and $ATr^{=0} = Tr$ represent true reachability. ■

Theorem 1: If SPLIT-TPA (Algorithm 3) returns UNSAFE, then there exists a counterexample path in the system, i.e., some bad state is reachable from some initial state.

Proof: Follows directly from Lemma 1. ■

C. Proving safety by discovering safe transition invariants

If a bounded safety has been proved on level n in Algorithm 3, i.e., there is no counterexample path of length $\leq 2^{n+1}$ in the system, then the algorithm attempts to extend the bounded proofs to unbounded ones. The procedure HASTRANSITIONINVARIANT tries to construct a (grounded) transition invariant based on the elements of $TPA^=$ and $TPA^<$ sequences. If a safe transition invariant is found, SPLIT-TPA has proven *unbounded safety*.

We have identified sufficient conditions for the elements $ATr^{<n}$ and $ATr^{=n}$ that guarantee the existence of a transition

invariant. These conditions are formalized in Lemma 2 and Lemma 3, respectively.

Lemma 2: Assume that for some n , $Init \triangleleft ATr^{<n} \circ Tr \subseteq Init \triangleleft ATr^{<n}$. Then $ATr^{<n}$ is a left-grounded transition invariant.

If $Tr \circ ATr^{<n} \triangleright Bad \subseteq ATr^{<n} \triangleright Bad$, then $ATr^{<n}$ is a right-grounded transition invariant.

Proof: Suppose that $s \in Init$ and $(s, t) \in Tr^*$, i.e., t is reachable from s . We show that $(s, t) \in ATr^{<n}$ by induction on d , the length of minimal path from s to t .

Base case $d < 2^n$: $(s, t) \in ATr^{<n}$ holds by Eq. (4).

Induction step: Suppose that the claim holds for all paths of length d . We show that then it also holds for all paths of length $d+1$. Consider a path between s and t of length $d+1$. Then t has a predecessor m on this path, i.e., m lies d steps from s and reaches t in 1 step. Then $(s, m) \in ATr^{<n}$ by the induction hypothesis. Since $(m, t) \in Tr$ it follows that $(s, t) \in ATr^{<n} \circ Tr$. Since $s \in Init$, it follows by the assumption of the lemma that $(s, t) \in ATr^{<n}$.

We have shown that if $s \in Init$ and $(s, t) \in Tr^*$ then $(s, t) \in ATr^{<n}$. Thus $ATr^{<n}$ is a left-grounded transition invariant. The case of the right-grounded transition invariant is analogous. In the inductive case, we consider m the successor of s on the path from s to t . ■

Note that with a slightly stronger assumption we can use the same proof idea to discover full transition invariants:

Observation 1: If $ATr^{<n} \circ Tr \subseteq ATr^{<n}$ or $Tr \circ ATr^{<n} \subseteq ATr^{<n}$ then $ATr^{<n}$ is a transition invariant.

Discovering transition invariants based on $TPA^{<}$ sequence is similar to how the invariants were detected in TPA sequence in [5]. This is not surprising, as the elements $ATr^{<n}$ and $ATr^{\leq n}$ have similar properties. The key advantage of SPLIT-TPA is the additional ability to discover transition invariants by detecting fixed points in the $TPA^=$ sequence.

Lemma 3: Assume that for some n , $Init \triangleleft ATr^{<n} \circ ATr^{=n} \circ ATr^{=n} \subseteq Init \triangleleft ATr^{<n} \circ ATr^{=n}$ then $Init \triangleleft Tr^* \subseteq Init \triangleleft ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$.

If $ATr^{=n} \circ ATr^{=n} \circ ATr^{<n} \triangleright Bad \subseteq ATr^{=n} \circ ATr^{<n} \triangleright Bad$ then $Tr^* \triangleright Bad \subseteq ATr^{<n} \cup ATr^{=n} \circ ATr^{<n} \triangleright Bad$.

Proof: The proof uses the same ideas as the proof of Lemma 2. Suppose that $s \in Init$ and $(s, t) \in Tr^*$, i.e., t is reachable from s . We proceed by induction on d , the length of minimal path from s to t .

Base case $d < 2^{n+1}$: It follows by Eq. (4) and (5) that $(s, t) \in ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$.

Induction step: Assuming the claim holds for all paths of length d , we show that it also holds for all paths of length $d+2^n$. Consider a path between s and t of length $d+2^n$. There exists m on this path that lies d steps from s and reaches t in exactly 2^n steps. Then $(m, t) \in ATr^{=n}$ by Eq. (5) and $(s, m) \in ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$ by induction hypothesis. Consider the two cases:

- $(s, m) \in ATr^{<n}$: It follows that $(s, t) \in ATr^{<n} \circ ATr^{=n}$.
- $(s, m) \in ATr^{<n} \circ ATr^{=n}$: It follows that $(s, t) \in ATr^{<n} \circ ATr^{=n} \circ ATr^{=n}$. Then $(s, t) \in ATr^{<n} \circ ATr^{=n}$ by the assumption of the lemma.

We have shown that if $s \in Init$ and $(s, t) \in Tr^*$ then $(s, t) \in ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$. Thus $ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$ is a left-grounded transition invariant. For the right-grounded transition invariant, in the induction step pick m that lies exactly 2^n steps from s (and reaches Bad in d steps). ■

Similarly to Lemma 2, full transition invariants can be discovered by checking a stronger condition:

Observation 2: If $ATr^{=n} \circ ATr^{=n} \subseteq ATr^{=n}$ then both $ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$ and $ATr^{<n} \cup ATr^{=n} \circ ATr^{<n}$ are full transition invariants.

Note that transition invariants obtained using Lemma 3 are *disjunctive by definition*. The disjunctive structure reflects the inductive nature of the proof of Lemma 3. $ATr^{<n}$ corresponds to the base case and represents the bounded part of the proof; $ATr^{=n}$ corresponds to the induction step and represents the unbounded part of the proof. Since the induction step makes 2^n steps of Tr instead of 1, the unbounded proof corresponds to k -induction rather than induction.

The procedure `HASTRANSITIONINVARIANT` checks the conditions of Lemma 2 and Lemma 3 using an SMT solver. For example, $ATr^{=n} \circ ATr^{=n} \circ ATr^{<n} \triangleright Bad \subseteq ATr^{=n} \circ ATr^{<n} \triangleright Bad$ iff $ATr^{=n}(x, x') \wedge ATr^{=n}(x', x'') \wedge ATr^{<n}(x'', x''') \wedge Bad(x''') \wedge \neg ATr^{=n}(x, x'')$ is *unsatisfiable*.

When the procedure discovers a grounded transition invariant it must also verify that the invariant is *safe*, i.e., it does not relate any initial with any bad state. This can also be checked with a single satisfiability query. In the case of transition invariant detected using conditions of Lemma 2, the check is not even necessary. The invariant, which is $ATr^{<n}$ for some n , is guaranteed to be safe after `ISREACHABLELT` proved bounded safety on level $n-1$.

The detection of safe (grounded) transition invariants as described above allows SPLIT-TPA to prove safety and the correctness is guaranteed by Lemma 2 and Lemma 3.

Theorem 2: If SPLIT-TPA returns `SAFE`, there is no counterexample path from $Init$ to Bad in S .

To demonstrate the behaviour of SPLIT-TPA, recall the loops from Fig. 1. For the first loop, similarly to TPA, SPLIT-TPA quickly proves bounded safety up to $8192 = 2^{13}$ iterations of the loop, and in the process learns that $ATr^{<n} \equiv x' < x + 2^n$ and that $ATr^{=n} \equiv x' \leq x + 2^n$ for $n = 1 \dots 13$. In the next iteration of its main loop, SPLIT-TPA discovers an abstract path consisting of a step of $ATr^{<13}$ followed by a step of $ATr^{=13}$. After some time spent in the refinement, the algorithm manages to refute all abstract paths and proves bounded safety for $< 2^{14}$ iterations. As part of the refinement, it strengthens $ATr^{=13}$ to include the facts $x' = x + 8192$ and $x \leq 1808$. With this strengthened information, it can easily prove that no path of length exactly $2^{14} = 16384$ exists because it is not possible to make two steps of the abstract relation $ATr^{=13}$ from the initial state. In addition, it learns that $ATr^{=14} \equiv x \leq -6384$. This satisfies the condition of Observation 2, namely $ATr^{=14} \circ ATr^{=14} \subseteq ATr^{=14}$. Thus SPLIT-TPA concludes at this point that the system is safe.

When analyzing the second loop, SPLIT-TPA behaves differently than TPA. After proving bounded safety in the first

iteration of Algorithm 3, SPLIT-TPA learns that $ATr^{=1} \equiv x > z \implies w' \geq w + 2$. In the next iteration, $ATr^{=1}$ is strengthened with facts $x \geq z \implies w' \geq w + 1$ and $x < z \implies w' \geq w$. These three facts together concisely over-approximate the change to w after precisely two iterations of the loop. Moreover, $ATr^{=1}$ with these three components is closed under composition, i.e., $ATr^{=1} \circ ATr^{=1} \subseteq ATr^{=1}$. Thus, SPLIT-TPA concludes already at this point that the system is safe (based on Observation 2). The transition invariant, using $\vec{a} = (x, z, v, w)$, is then $ATr^{<1}(\vec{a}, \vec{a}') \vee (ATr^{<1}(\vec{a}, \vec{a}') \wedge ATr^{=1}(\vec{a}', \vec{a}'))$, where

$$ATr^{<1}(\vec{a}, \vec{a}') \equiv w' \geq w \wedge v' \leq v \wedge ((x' \geq x \wedge z' \leq z) \vee (x' \geq x + 1 \wedge z' \leq z + 2)),$$

$$ATr^{=1}(\vec{a}, \vec{a}') \equiv x > z \rightarrow w' \geq w + 2 \wedge x \geq z \rightarrow w' \geq w + 1 \wedge x < z \rightarrow w' \geq w.$$

Note that the exact value of $ATr^{<1}$ is not important in this case, as long as it over-approximates all paths of length < 2 .

V. FROM TRANSITION INVARIANTS TO STATE INVARIANTS

In Section IV-C, we have shown how SPLIT-TPA can prove a transition system safe by finding a safe transition invariant. However, many applications require a proof of safety in the form of a safe inductive (state) invariant. Here we show that (k -)inductive invariants can be obtained from the discovered transition invariants by quantifying over the source or target states. This follows Lemma 2 and Lemma 3 and their proofs.

Lemma 4: Assume that for some n , the following holds:

$$Init \triangleleft ATr^{\leq n} \circ Tr \subseteq Init \triangleleft ATr^{\leq n}.$$

Then the following is an inductive invariant:

$$Inv(x') \equiv \exists x : Init(x) \wedge ATr^{\leq n}(x, x').$$

Proof: Analogous to the proof of Lemma 2. Intuitively, Inv represents all states reachable by one step of $ATr^{<n}$ from $Init$. Since $ATr^{<n}$ is a left-grounded transition invariant by Lemma 2, making one additional step of Tr cannot end up outside this set. Also, $Init \subseteq Inv$, because $Id \subseteq ATr^{\leq n}$, i.e., Inv holds in the initial states. ■

Lemma 5: Assume that for some n , the following holds:

$$Tr \circ ATr^{\leq n} \triangleright Bad \subseteq ATr^{\leq n} \triangleright Bad.$$

If $ATr^{<n}$ is safe, then the following is an inductive invariant:

$$Inv(x) \equiv \neg(\exists x' : ATr^{\leq n}(x, x') \wedge Bad(x')).$$

Proof: Analogous to the proof of Lemma 4. ■

Compared to Lemma 2, the proof of Lemma 3 uses an inductive step of size 2^n . Following that proof we can turn the transition invariant from $TPA^=$ into 2^n -inductive invariant.

Lemma 6: Assume that for some n , the following holds:

$$Init \triangleleft ATr^{<n} \circ ATr^{=n} \circ ATr^{=n} \subseteq Init \triangleleft ATr^{<n} \circ ATr^{=n}.$$

Then the following is 2^n -inductive invariant:

$$Inv(x'') \equiv \exists x, x' : Init(x) \wedge (ATr^{<n}(x, x'') \vee (ATr^{<n}(x, x') \wedge ATr^{=n}(x', x''))).$$

Proof: We follow the proof of Lemma 3. Inv represents the set of states reachable from $Init$ either by one step of $ATr^{<n}$ or by a combined step of $ATr^{<n}$ and $ATr^{=n}$. It follows that Inv over-approximates the set of states reachable from $Init$ in less than 2^{n+1} steps of Tr . Thus, Inv satisfies the base step of k -induction (for $k = 2^n$).

For the inductive step, we need to prove that making 2^n steps of Tr from an Inv -state leads again to an Inv -state. We rely on Eq. (5), i.e., $ATr^{=n} \supseteq Tr^{2^n}$. If s is an Inv -state, then it is reachable from some initial state i either in one step of $ATr^{<n}$ or in one step of $ATr^{<n} \circ ATr^{=n}$. Moreover, all states reachable from s in 2^n steps of Tr are reachable from s by one step of $ATr^{=n}$. Thus, in the first case, they are reachable from i in one step of $ATr^{<n} \circ ATr^{=n}$. In the second case, they are reachable from i in one step of $ATr^{<n} \circ ATr^{=n} \circ ATr^{=n}$. Based on the assumption of the lemma, they are reachable from i also in one step of $ATr^{<n} \circ ATr^{=n}$. ■

Lemma 7: Assume that for some n , the following holds:

$$ATr^{=n} \circ ATr^{=n} \circ ATr^{<n} \triangleright Bad \subseteq ATr^{=n} \circ ATr^{<n} \triangleright Bad.$$

If $ATr^{<n}(x, x'') \vee (ATr^{=n}(x, x') \wedge ATr^{<n}(x', x''))$ is safe then the following is 2^n -inductive invariant:

$$Inv(x) \equiv \neg(\exists x', x'' : Bad(x'') \wedge (ATr^{<n}(x, x'') \vee (ATr^{=n}(x, x') \wedge ATr^{<n}(x', x'')))).$$

Proof: Analogous to the proof of Lemma 6. ■

Note that in each given case, the (k -)inductive invariants are quantified and quantifier elimination must be applied if quantifier-free inductive invariants are required. Inductive invariants can be obtained from k -inductive invariants by quantifying over the intermediate states [29].

VI. EXPERIMENTS

We have implemented SPLIT-TPA in our Horn solver GOLEM². In our experiments we used GOLEM 0.1.0, which uses OPENSMT 2.3.2 for SMT solving and interpolation.

The goal of the experiments was to compare SPLIT-TPA to TPA [5], which is also available in GOLEM, and to state-of-the-art tools ELGARICA 2.0.8 [26], Z3-SPACER [30] implemented in Z3 4.8.17 [35], and GSPACER [22] a more recent version enriched with global guidance. All experiments were conducted on a machine with AMD EPYC 7452 32-core processor and 8x32 GiB of memory. We used a timeout of 5 minutes for every task.³

For the evaluation we used the set of benchmarks representing multi-phase loops [39], which are known to be challenging for automated analysis techniques. We used both the safe

²<https://github.com/usi-verification-and-security/golem.git>

³Full results at <http://verify.inf.usi.ch/content/split-tpa-experiments>, artifact at <https://doi.org/10.5281/zenodo.6988735>

TABLE I
SUMMARY OF THE EXPERIMENTS ON MULTI-PHASE BENCHMARKS.

Benchmark suite	SPLIT-TPA	TPA	Z3SPACER	GSPACER	ELДАРICA
multi-phase safe	19 (7)	12 (0)	6 (0)	24 (3)	26 (4)
multi-phase unsafe	37 (3)	35 (2)	20 (0)	17 (0)	17 (0)

Solved (unique) instances out of 54 benchmarks.

TABLE II
FULL RESULTS ON SAFE (LEFT) AND UNSAFE BENCHMARKS (RIGHT)

Ben.	SPLIT-TPA	TPA	Z3SPACER	GSPACER	ELДАРICA	Ben.	SPLIT-TPA	TPA	Z3SPACER	GSPACER	ELДАРICA
01	26.28	TO	TO	TO	TO	01	14.53	10.12	TO	TO	TO
02	TO	TO	133.28	<1	TO	02	<1	<1	1.25	TO	TO
03	TO	TO	TO	TO	1.33	03	<1	<1	<1	<1	1.16
04	TO	TO	TO	<1	3.70	04	TO	TO	TO	TO	TO
05	<1	<1	<1	<1	1.19	05	<1	<1	<1	<1	1.18
06	TO	TO	TO	TO	3.95	06	TO	TO	TO	TO	TO
07	TO	TO	TO	TO	1.32	07	TO	TO	TO	TO	TO
08	TO	TO	TO	TO	TO	08	TO	TO	TO	TO	TO
09	TO	TO	TO	TO	TO	09	TO	TO	TO	TO	TO
10	TO	TO	TO	TO	TO	10	20.40	233.78	TO	TO	TO
11	TO	TO	TO	5.68	TO	11	152.28	TO	TO	TO	TO
12	TO	TO	TO	TO	1.62	12	TO	TO	TO	TO	TO
13	<1	<1	ERR	<1	1.16	13	<1	<1	<1	<1	1.13
14	53.94	TO	TO	TO	118.78	14	<1	<1	<1	8.91	89.78
15	TO	TO	TO	TO	TO	15	TO	TO	TO	TO	TO
16	TO	TO	TO	TO	TO	16	TO	TO	TO	TO	TO
17	<1	37.50	TO	<1	7.53	17	14.84	15.81	181.59	TO	TO
18	<1	<1	TO	<1	3.66	18	<1	<1	<1	<1	1.57
19	TO	TO	<1	<1	1.22	19	<1	<1	<1	<1	20.74
20	TO	TO	TO	TO	TO	20	TO	TO	TO	TO	TO
21	<1	10.39	TO	<1	15.45	21	<1	<1	<1	<1	10.63
22	TO	TO	TO	TO	TO	22	TO	TO	TO	TO	TO
23	<1	<1	ERR	<1	1.79	23	<1	<1	<1	<1	1.17
24	TO	TO	TO	TO	TO	24	<1	TO	96.64	TO	TO
25	TO	45.93	TO	TO	9.33	25	<1	<1	<1	<1	1.19
26	2.60	1.55	TO	<1	TO	26	2.01	1.46	TO	TO	TO
27	TO	TO	TO	TO	TO	27	<1	<1	TO	TO	TO
28	<1	TO	TO	TO	1.61	28	<1	<1	TO	TO	162.43
29	3.94	TO	TO	118.98	34.22	29	<1	<1	2.76	32.56	45.75
30	TO	TO	TO	TO	20.48	30	<1	<1	<1	<1	10.22
31	TO	TO	TO	<1	1.60	31	TO	TO	TO	TO	TO
32	TO	TO	TO	11.49	TO	32	<1	<1	<1	<1	7.17
33	TO	TO	TO	TO	TO	33	<1	<1	<1	<1	1.21
34	TO	TO	TO	<1	5.86	34	<1	<1	<1	<1	1.15
35	TO	TO	TO	<1	1.80	35	<1	<1	<1	<1	1.20
36	<1	<1	TO	<1	1.92	36	16.68	14.45	TO	TO	TO
37	<1	<1	<1	<1	14.33	37	<1	<1	<1	<1	13.37
38	TO	<1	TO	<1	1.36	38	262.18	TO	TO	TO	TO
39	TO	TO	67.41	58.73	2.48	39	TO	TO	ERR	ERR	TO
40	109.05	TO	TO	TO	ERR	40	<1	<1	<1	133.07	ERR
41	TO	TO	TO	TO	TO	41	TO	4.60	TO	TO	TO
42	TO	TO	TO	<1	4.37	42	18.31	40.39	TO	TO	TO
43	TO	TO	TO	5.20	TO	43	TO	TO	TO	TO	TO
44	TO	TO	TO	TO	TO	44	34.18	TO	TO	TO	TO
45	TO	TO	TO	TO	TO	45	TO	TO	TO	TO	TO
46	TO	288.20	13.07	<1	1.28	46	TO	239.05	TO	TO	TO
47	TO	TO	TO	TO	TO	47	5.71	6.79	TO	TO	TO
48	47.00	TO	TO	TO	TO	48	17.52	12.10	TO	TO	TO
49	122.96	TO	TO	TO	TO	49	32.59	12.49	TO	TO	TO
50	TO	TO	TO	TO	TO	50	TO	TO	TO	TO	TO
51	TO	TO	TO	TO	TO	51	6.71	11.57	TO	TO	TO
52	235.24	TO	TO	TO	TO	52	70.83	82.43	TO	TO	TO
53	147.28	TO	TO	TO	TO	53	57.42	33.00	TO	TO	TO
54	133.63	TO	TO	TO	TO	54	40.74	15.15	TO	TO	TO

TO: timeout; ERR: memory out or other inconclusive answer.

versions of the benchmarks from CHC-COMP repository⁴ and the unsafe versions of the benchmarks from [5]. The results are summarized in Table I and times for each tool/benchmark pair are given in Table II.

Regarding safety, Table I shows that SPLIT-TPA overall solved 7 more benchmarks than TPA, but still less than GSPACER or ELДАРICA. However, it solved seven benchmarks *uniquely* (the other competitors did not solve them). This indicates that SPLIT-TPA is quite orthogonal to the existing techniques for proving safety.

The results on unsafe benchmarks show that SPLIT-TPA not only preserves the capability of TPA to detect deep counterexample, but it was even able to outperform it by solving two more benchmarks overall.

Besides the multi-phase benchmarks, we also evaluated the tools on a general benchmark set from the LRA-TS category of CHC-COMP 2021, the latest edition with a publicly available selected benchmark set.⁵ Out of 498 benchmarks, SPLIT-TPA proved 128 benchmarks safe and 72 unsafe. TPA proved 62 benchmarks safe and 71 unsafe. Even though the performance of SPLIT-TPA still lacks behind Z3-SPACER and GSPACER (ELДАРICA does not support arithmetic over reals) on CHC-COMP benchmarks, it still achieved a significant improvement over TPA, especially on safe benchmarks.

To better understand the advantage of SPLIT-TPA over TPA, we collected statistics from the runs of SPLIT-TPA on safe instances to see which transition invariants it used to prove safety. In our implementation $TPA^<$ is checked before $TPA^=$. Moreover, each sequence element is first checked for a full transition invariant. This is followed by checks for left-grounded and finally right-grounded transition invariant.

On CHC-COMP2021 LRA-TS benchmarks, out of 128 benchmarks proven safe, 63 invariants were discovered from $TPA^<$ and 65 invariants were discovered with $TPA^=$. Regardless of the sequence, 81 were full transition invariants and 47 were left-grounded transition invariants. Surprisingly, no (purely) right-grounded transition invariants were discovered. For safe multi-phase benchmarks the results were similar. Out of 19 invariants, 15 invariants were found with $TPA^=$ and 4 invariants were found with $TPA^<$. Fifteen of these invariants were full transition invariants and 4 were left-grounded. Again, no purely right-grounded transition invariant was found. These statistics confirm the essential role of the $TPA^=$ sequence in SPLIT-TPA as a source of transition invariants.

VII. RELATED WORK

Many model-checking algorithms search for a safe inductive invariant to prove safety. Candidates for inductive invariants are typically obtained from proofs of bounded safety. The algorithms try to construct the safe inductive invariant either in monolithic [32, 34, 38] or incremental way [8, 10, 17, 24, 30]. Our work follows a similar strategy, but it primarily computes *transition* invariants, not state invariants.

Transition invariants have been introduced in [36] as a proof rule for program verification, especially termination and other liveness properties. Transition predicate abstraction [37] has been introduced as a way to compute transition invariants. In contrast, we use transition invariants to prove safety, with candidates automatically obtained from proofs of bounded safety using Craig interpolation.

Craig interpolation [15] is a popular abstraction technique widely used in model checking. We use standard algorithms to compute interpolants from proofs of unsatisfiability [6, 13, 33]. The integration of domain-specific knowledge [31] is future work.

While in most model checking algorithms interpolants are used as over-approximations of *states*, we use them to over-approximate *transitions*. The idea of abstracting transition

⁴<https://github.com/chc-comp/aeval-benchmarks>

⁵<https://github.com/chc-comp/chc-comp21-benchmarks/tree/main/LRA-TS>

relation with interpolants originates from [28]. However, they maintained an abstraction of only a single step of the transition relation. We build two sequences of relations over-approximating doubling number of steps of the transition relation, which are useful both for detecting deep counterexamples and as a source of candidates for safe transition invariant.

Loop acceleration [2, 7, 19] is a loop analysis technique that can prove safety and detect deep counterexamples. However, on its own, it is applicable only to limited types of integer loops. Acceleration have also been successfully integrated into interpolation-based model checking [9, 25] where interpolants computed from accelerated paths lead to much better abstraction refinement in the traditional CEGAR algorithm [14]. In contrast, SPLIT-TPA computes transition interpolants, not state interpolants. It also does not try to capture all possible behaviour of a loop (by accelerating it). Instead, it builds over-approximations of (exponentially increasing) bounded number of iterations. By relying purely on Craig interpolation it can handle transition relations where acceleration is not possible.

The k -induction principle [20] has been successfully used as a replacement for basic inductive reasoning in IC3-style algorithms [21, 23, 29]. k -inductive invariants can be more compact than inductive invariants and for some theories k -induction is a strictly stronger proof rule [29]. SPLIT-TPA uses both inductive reasoning (applied to $\text{TPA}^<$) and k -inductive reasoning (applied to $\text{TPA}^=$) to discover transition invariants. We believe that SPLIT-TPA's success on challenging systems can be in large part attributed to the inclusion of k -inductive reasoning, which was missing in TPA [5].

VIII. CONCLUSION

In this work we have presented SPLIT-TPA, a novel instantiation of a recently introduced TPA approach. Splitting the transition power abstraction into two complementary parts makes the algorithm more efficient in proving safety by detecting safe transition invariants while still retaining and even improving the capability of detecting long counterexamples. The advantage of our instantiation has been confirmed experimentally on a set of challenging multi-phases benchmarks and on an extensive general benchmark set from CHC-COMP. The experiments also show that SPLIT-TPA is both competitive and complementary compared to state-of-the-art in safety verification. As the next step, we plan to study extensions of SPLIT-TPA from transition systems to general CHC systems.

ACKNOWLEDGMENTS

This work was partially supported by Swiss National Science Foundation grant 200021_185031, Czech Science Foundation grant 20-07487S, and the National Science Foundation (of the United States) grant 2106949. The authors thank the anonymous reviewers for their valuable feedback and suggestions.

REFERENCES

- [1] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 415–442. Springer International Publishing, Cham (2022)
- [2] Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer* **10**(5), 401–424 (2008)
- [3] Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. *Journal of Automated Reasoning* **60**(3), 299–335 (Mar 2018)
- [4] Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations*. EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
- [5] Blicha, M., Fedyukovich, G., Hyvärinen, A., Sharygina, N.: Transition power abstraction for deep counterexample detection. In: *Tools and Algorithms for Construction and Analysis of Systems* (2022)
- [6] Blicha, M., Hyvärinen, A.E.J., Kofroň, J., Sharygina, N.: Decomposing Farkas interpolants. In: Vojnar, T., Zhang, L. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 3–20. Springer International Publishing, Cham (2019)
- [7] Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) *Computer Aided Verification*. pp. 227–242. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
- [8] Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) *Verification, Model Checking, and Abstract Interpretation*. pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
- [9] Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 428–442. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
- [10] Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) *Computer Aided Verification*. pp. 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
- [11] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Abraham, E., Havelund, K. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 46–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
- [12] Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 93–107. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
- [13] Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Logic* **12**(1), 7:1–7:54 (Nov 2010)
- [14] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) *Computer Aided Verification*. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
- [15] Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic* **22**(3), 269–285 (1957)
- [16] Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) *Computer-Aided Verification (CAV'2014)*. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (July 2014)
- [17] Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. pp. 125–134. FMCAD '11, FMCAD Inc, Austin, TX (2011)
- [18] Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving Constrained Horn Clauses Using Syntax and Data. In: *FMCAD*. pp. 170–178. IEEE (2018)
- [19] Frohn, F.: A calculus for modular loop acceleration. In: Biere, A., Parker, D. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 58–76. Springer International Publishing, Cham (2020)

- [20] Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) *Computer Aided Verification*. pp. 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
- [21] Gurfinkel, A., Ivrii, A.: K-induction without unrolling. In: *2017 Formal Methods in Computer Aided Design (FMCAD)*. pp. 148–155 (Oct 2017)
- [22] Hari Govind, V.K., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: Lahiri, S.K., Wang, C. (eds.) *Computer Aided Verification*. pp. 101–125. Springer International Publishing, Cham (2020)
- [23] Hari Govind, V.K., Vizek, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Dillig, I., Tasiran, S. (eds.) *Computer Aided Verification*. pp. 367–385. Springer International Publishing, Cham (2019)
- [24] Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2012*. pp. 157–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
- [25] Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) *Automated Technology for Verification and Analysis*. pp. 187–202. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
- [26] Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: *FMCAD*. pp. 158–164. IEEE (2018)
- [27] Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2016*. pp. 547–553. Springer International Publishing, Cham (2016)
- [28] Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) *Computer Aided Verification*. pp. 39–51. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
- [29] Jovanović, D., Dutertre, B.: Property-directed k-induction. In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. pp. 85–92 (Oct 2016)
- [30] Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. *Formal Methods in System Design* **48**(3), 175–205 (Jun 2016)
- [31] Leroux, J., Rümmer, P., Subotić, P.: Guiding Craig interpolation with domain-specific abstractions. *Acta Informatica* **53**(4), 387–424 (2016)
- [32] McMillan, K.L.: Interpolation and SAT-based model checking. In: *Computer Aided Verification*. pp. 1–13. Springer, Berlin Heidelberg (2003)
- [33] McMillan, K.L.: An interpolating theorem prover. *Theoretical Computer Science* **345**(1), 101 – 121 (2005)
- [34] McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) *Computer Aided Verification*. pp. 123–136. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
- [35] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
- [36] Podolski, A., Rybalchenko, A.: Transition invariants. In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004*. pp. 32–41 (2004)
- [37] Podolski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 132–144. POPL '05, Association for Computing Machinery, New York, NY, USA (2005)
- [38] Rümmer, P., Hojjat, H., Kuncak, V.: On recursion-free Horn clauses and Craig interpolation. *Formal Methods In System Design* **47**(1), 1–25 (2015)
- [39] Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Computer Aided Verification*. pp. 703–719. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

Automating Geometric Proofs of Collision Avoidance with Active Corners

Nishant Kheterpal* 
 Robotics Institute
 University of Michigan
 Ann Arbor, MI, USA
 nskh@umich.edu

Elanor Tang 
 Computer Science and Engineering
 University of Michigan
 Ann Arbor, MI, USA
 elanor@umich.edu

Jean-Baptiste Jeannin 
 Aerospace Engineering
 University of Michigan
 Ann Arbor, MI, USA
 jeannin@umich.edu

Abstract—Avoiding collisions between obstacles and vehicles such as cars, robots, or aircraft is essential to the development of autonomy. To simplify the problem, many collision avoidance algorithms and proofs consider vehicles to be a point mass, though the actual vehicles are not points. In this paper, we consider a convex polygonal vehicle with nonzero area traveling along a 2-dimensional trajectory. We derive an easily-checkable, quantifier-free formula to check whether a given obstacle will collide with the vehicle moving on the planned trajectory. We apply our active corner method to two case studies of aircraft collision avoidance and benchmark its performance.

I. INTRODUCTION

Preventing collisions with obstacles or foreign objects is crucial when developing autonomous capabilities for robots, cars, aircraft, and many other vehicles. As such, collision avoidance remains a major research theme of the autonomy, robotics, and formal methods communities. In particular, for safety-critical tasks such as vehicles interacting with humans or animals, it is imperative to provide *formal* proofs that the vehicle will not collide with agents in its environment.

In many papers studying trajectory planning or collision avoidance, e.g. [34], [3], [20], the vehicle is modeled as a point, and the volume — or surface area — occupied by the vehicle is ignored. In reality, land and air vehicles are not points but have a certain volume, and contact of any external object with any part of the vehicle would constitute a collision. In this paper, we present a novel, automated, and general technique to *transform* a planned trajectory of a vehicle with volume into explicit boundaries of the region in which an obstacle will not be at risk of a collision. This transformation provides an efficient, runtime-checkable test to determine whether a given obstacle will collide with a vehicle on the planned trajectory, even when the vehicle has volume.

Given a part of a trajectory \mathcal{T} , a vehicle occupying the volume $v(x_{\mathcal{T}}, y_{\mathcal{T}})$ when centered on position $(x_{\mathcal{T}}, y_{\mathcal{T}})$ along the trajectory, and a point-obstacle (x_O, y_O) , the vehicle will not collide with the obstacle if and only if:

$$\forall (x_{\mathcal{T}}, y_{\mathcal{T}}) \in \mathcal{T}, (x_O, y_O) \notin v(x_{\mathcal{T}}, y_{\mathcal{T}}) \quad (1)$$

In the rest of the paper, we will call this formulation the *implicit* formulation of collision avoidance. This implicit formulation is a correct definition, but it has one major drawback:

because of the universal quantifier on $(x_{\mathcal{T}}, y_{\mathcal{T}})$, it is not easy to check systematically or at runtime whether an obstacle is indeed at risk of a collision. Ideally, we would want to obtain a quantifier-free, easily checkable formula that is equivalent to (1); in the rest of this paper we will call that formula, which represents a region in the plane, the *explicit* formulation. In theory, one could use quantifier elimination, but for trajectories containing more than a few symbolic parameters, the algorithm does not finish in a reasonable time due to its doubly-exponential time complexity in the number of variables [14].

This issue arose before, notably in the verification of the Next-Generation Aircraft Collision Avoidance System ACAS X [22], [23]. In that work, the formal proof of correctness was divided into: (i) establishing the trajectory of the aircraft from its equations of motion, leading to a formula of the form of (1); and (ii) establishing an equivalent quantifier-free formula that can be checked efficiently at runtime. Both tasks required a proof in the KeYmaera X theorem prover, with significant manual effort [35]. A similar approach was used in the verification of collision avoidance for ACAS Xu, the unmanned version of ACAS X, with horizontal maneuvers [1]. The object of this paper is to automate and generalize task (ii) of this process.

In order to automate task (ii), we propose a different approach based on geometric intuition. Let us examine an example of a rectangular vehicle performing a simple maneuver (Figure 1). The central idea of the method presented in this paper is that the boundaries of the explicit formulation are either *trajectories of a corner* of the vehicle or *sides of the vehicle* at a few particular points.

The corners to consider at every point depend on the slope of the trajectory: for a rectangular vehicle, the boundaries follow the top-right and bottom-left corners when the vehicle’s velocity is directed “northwest” (towards the top left) or southeast; and, symmetrically, the boundaries follow the top-left and bottom-right corners when the vehicle’s velocity is directed towards the northeast or southwest of the plane. We call these corners *active corners*. But this is not enough: at points where the trajectory switches from following one set of corners to another, the boundary may follow a side of the vehicle at that point, e.g., its bottom boundary at the lowest point of the trajectory on Figure 1. We call these points

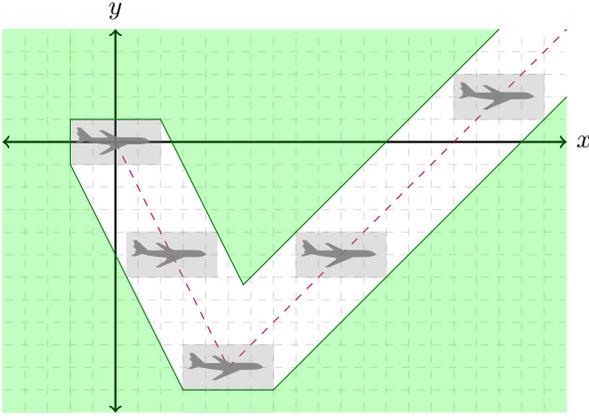


Fig. 1: Safe region for a rectangle with $w = 2, h = 1$ and its center following Equation (2). The trajectory is dashed in purple, safe region shaded in green, and unsafe region is unshaded.

transition points. By capturing the motion of these boundaries — both corners depending on the slope or the sides of the vehicle at certain points — we can construct a quantifier-free formula equivalent to (1), corresponding to its equivalent explicit formulation. Our approach is fully symbolic with no approximation.

In this paper, we formalize and generalize our approach to different trajectories and polygons and show how to find active corners and transitions symbolically and how to form a quantifier free explicit formulation equivalent to the input implicit formulation. We carefully prove that our transformation is both sound (any obstacle shown safe using the explicit formulation is safe using the implicit formulation) and complete (no obstacles that are actually safe appear unsafe using the explicit formulation). Finally, we detail a fully symbolic Python implementation of our work and present an evaluation of its performance on two applications from previous papers (where we fully automate results that had required significant manual proof effort) and a third non-polynomial example.

II. OVERVIEW

This section provides an overview of our approach, by walking through a simple example constructing a geometric safe region used to verify obstacle avoidance for aircraft. At present, our method applies only to two-dimensional planar motion due to the increased complexity of three-dimensional motion when analyzing trajectories and polyhedra. The example uses linear motion and a rectangle, though the method generalizes to other planar motion and convex polygons, as detailed in Section IV.

A. Trajectory

Consider the planar side-view of an airplane flying, initially descending at constant velocity and then ascending with constant velocity. For simplicity, assume the aircraft has infinite acceleration. In this example, we represent the bounds of the

aircraft as an axis-aligned rectangle with width $2w$ and height $2h$ that moves in the (x, y) plane. The airplane begins at the origin and moves in the plane with piecewise trajectory \mathcal{T} :

$$\mathcal{T} = \begin{cases} y = -2x & x \in [0, 5] \\ y = x - 15 & x \in [5, \infty) \end{cases} \quad (2)$$

B. Implicit Formulation

Suppose the rectangle translates with its center moving along this piecewise trajectory. Additionally, assume there is a point obstacle at (x_O, y_O) to be avoided; that is, the rectangle never intersects the obstacle. Then we can state an quantified (or *implicit*) formulation of obstacle avoidance:

$$\forall (x_{\mathcal{T}}, y_{\mathcal{T}}) \in \mathcal{T}, (|x_O - x_{\mathcal{T}}| > w \vee |y_O - y_{\mathcal{T}}| > h) \quad (3)$$

The implicit formulation of the safe region (3) straightforwardly represents a safety property of obstacle avoidance — if an object moving with its center fixed along the nominal trajectory \mathcal{T} is far enough away (either width w in the x -axis or height h in the y -axis) from a point obstacle at (x_O, y_O) , then the object is safe. Here we use *safe region* to mean the set of all obstacle locations for which collision is avoided.

C. Explicit Formulation

The need for a quantifier-free equivalent of Equation (3) motivates an *explicit formulation* of the safe region for the obstacle. The goal of this work is to automate the generation of such a formulation. We compute the reachable set of the object as it moves along a trajectory in order to compute the complement of the safe region: the *unsafe region*. We can express the *unsafe region* (the set of all locations for which an obstacle will collide with the object as it moves along a given trajectory) directly as a union of regions in the plane, each defined (in this case) by an intersection of linear inequalities (Equation (4)) bounding the region, plotted in Figure 1. The *safe region* is simply the negation of (4).

$$\begin{aligned} & \left((x_O \geq -w) \wedge (y_O \leq h) \wedge (y_O \geq -2x_O - 2w - h) \right. \\ & \quad \wedge (x_O \leq 5 + w) \wedge (y_O \leq -2x_O + 2w + h) \\ & \quad \left. \wedge (y_O \geq -10 - h) \right) \vee \left((x_O \geq 5 - w) \right. \\ & \quad \wedge (y_O \geq -10 - h) \wedge (y_O \leq x_O + w + h - 15) \\ & \quad \left. \wedge (y_O \geq x_O - w - h - 15) \right) \end{aligned} \quad (4)$$

Note the first disjunction in Equation (4) corresponds to the motion of the aircraft on the left side of Figure 1 as it descends; the second corresponds to the right side as the aircraft ascends.

III. ALGORITHM

A. Preliminaries

In this work, we define the *safe region* as the set of obstacle locations where, given a polygon's trajectory, a collision will not occur. Correspondingly, the *unsafe region* will be unsafe if an obstacle invades its area. As such, the *unsafe region* corresponds to the reachable set of the polygon as it moves

along a trajectory. We can define a quantified representation of the safe region: the *implicit formulation* from (1). We also use the term *explicit formulation* in this work; we use that to mean an equivalent to the *implicit formulation*, but without quantifiers like \forall and \exists . Our method primarily applies to convex polygons. In this paper, we discuss polygons with central symmetry for ease of exposition, though the method straightforwardly extends to irregular and asymmetric convex polygons, and can be extended to concave polygons (Section III-F).

We consider two-dimensional planar trajectories defined piecewise, with each piece a function $y = f(x)$ or $x = f(y)$ and f a C^1 function (differentiable and having a continuous derivative). Trajectories must have a *finite* number of these C^1 pieces. The pieces themselves need not be continuous, though the applications we study do include continuous piecewise trajectories. The subdomains for the piecewise trajectory must be non-overlapping and exhaustive, meaning their union should cover the entire domain of the trajectory. Polygons move along the trajectory without rotating. Since the polygons translate along the trajectory, there is a constant vector offset $\begin{bmatrix} \Delta x_i \\ \Delta y_i \end{bmatrix}$ from the center to the i -th vertex, and there are n vertices for an n -sided polygon. Thus, the trajectory for vertex i is $y - \Delta y_i = f(x - \Delta x_i)$ or $x - \Delta x_i = f(y - \Delta y_i)$. We consider the trajectories of all vertices of the polygon in an attempt to bound its motion and compute the reachable set of the object as it moves along the trajectory.

B. Active Corners

Throughout this section we consider a trajectory of the form $y = f(x)$; the case for $x = f(y)$ is symmetric. The boundaries of the safe region are (for centrally symmetric polygons) formed by the trajectories of a pair of opposite corners of the vehicle (Figure 2) — we call this pair of corners *active corners*. For asymmetric polygons, the corners may not directly oppose each other.

We choose the active corners to represent the outermost extent of the object along the trajectory; as such, their motion bounds the safe region. Which corners are active depends on the slope of the trajectory (which can be computed from the derivative of f) and the shape of the convex object. A corner v_i is active when the slope θ of the trajectory is between the slopes of the sides adjacent to v_i ; when a corner is active, its opposite corner is also active based on the symmetry of the polygon. More precisely, if we number the corners v_1 through v_n counter-clockwise (with $v_{n+1} = v_0$ and $v_{-1} = v_n$), corner v_i is active if and only if the slope θ of the trajectory is in the angle interval $[\angle \overrightarrow{v_{i-1}v_i}, \angle \overrightarrow{v_i v_{i+1}}]$, or symmetrically in the angle interval $[\angle \overrightarrow{v_{i+1}v_i}, \angle \overrightarrow{v_i v_{i-1}}]$. Because the direction of the trajectory is inconsequential for our purpose, θ is modulo 180° .

For example, on the hexagon in Figure 2, v_1 and v_4 are active when $\theta \in [0^\circ, 60^\circ] \cup [180^\circ, 240^\circ]$; v_2 and v_5 are active when $\theta \in [60^\circ, 120^\circ] \cup [240^\circ, 300^\circ]$; and v_3 and v_6 are active when $\theta \in [120^\circ, 180^\circ] \cup [300^\circ, 360^\circ]$.

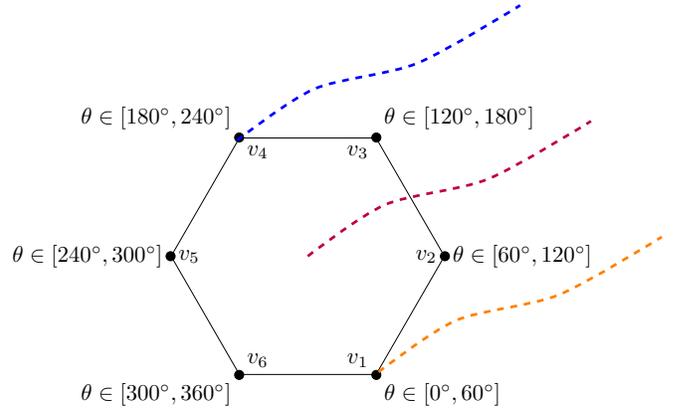


Fig. 2: A hexagon, the angles of its sides, and shifted active corner-trajectories

At transition points (where the active corners change), the boundary of the safe region may not follow an active corner. We detail what happens then in Section III-C. Note that when a linear trajectory is parallel to a side of the polygon (e.g. $\theta = 60^\circ$ for the hexagon in Figure 2), two adjacent corners may both be active and either can be classified as such. For such trajectories, the active corners technically do not change for the length of the linear path, so there would be no transition points as long as the trajectory parallels a polygon edge.

Given an obstacle at point (x_O, y_O) , we can check if it is inside the unsafe region (or reachable set) in a computationally efficient fashion. If an obstacle lies outside the unsafe region, it would be either above both corner-trajectories or below both corner-trajectories for whichever corners are active. We can express the location of the obstacle with respect to a corner-trajectory in a single equation by considering the value of $y_O - f(x_O - \Delta x_i) - \Delta y_i$ for active corner (vertex) v_i . This term will be positive for both vertices v_i, v_j if the obstacle is above both corner-trajectories, and similarly negative if the obstacle is below both. Therefore, any point (x_O, y_O) in the safe region has a positive value for the product of the two expressions above, and any point in the unsafe region has a negative or zero value for this product. This yields the following test to check if an object lies in (part of) the *unsafe region*:

$$(y_O - f(x_O - \Delta x_i) - \Delta y_i) \cdot (y_O - f(x_O - \Delta x_j) - \Delta y_j) \leq 0 \quad (5)$$

where $\Delta x_i, \Delta y_i, \Delta x_j, \Delta y_j$ are the (constant) offsets from the center of the polygon to the active corners (vertices) v_i, v_j .

When implementing this algorithm, the trajectories of all other vertices lie within the trajectories of the active corners, so to check whether an obstacle lies in the portion of the unsafe region defined by the active corners, it suffices to check over all pairs of vertices (v_i, v_j) with $i, j \in \{1, 2, \dots, n\}$. This check can be made more efficient by considering only all possible pairs of active corners based on the polygon shape and discarding, say, pairs of adjacent vertices. If the test in (5) indicates that an object lies within the unsafe region, trajectory $y = f(x)$ or $x = f(y)$ is clearly unsafe.

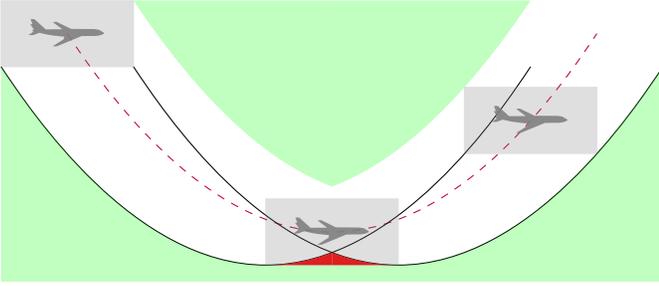


Fig. 3: A rectangular airplane moving along a planar trajectory. At the transition point at the parabola’s vertex, the “notch” is visible and shaded in red; part of the object lies outside the corner-trajectories at this point.

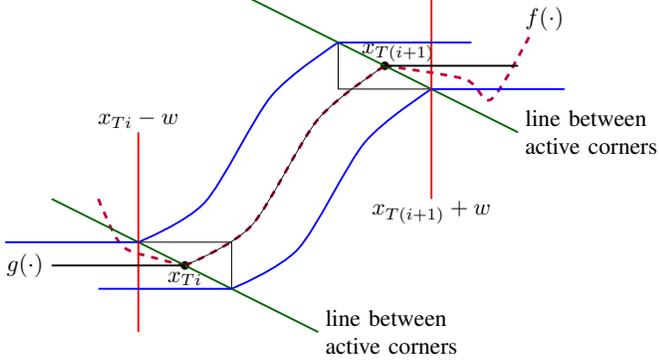


Fig. 4: For piecewise functions, between transition points and/or piecewise boundaries, this figure shows the difference between $f(\cdot)$ and $g(\cdot)$ and the two additional checks on (x_O, y_O) described in Section III-D.

C. Notches at Transition Points Between Active Corners

It turns out that using only active corners would yield an underestimate of the reachable set, which would be unsound for verifying safety. Figure 3 illustrates why: the white area bounded by the trajectories of the corners does not contain the red “notch,” even though a collision would occur with an obstacle in this notch. Therefore, if the test in (5) yields a value > 0 , the trajectory is not necessarily safe; we **additionally** check safety at all *transition points* (x_T, y_T) to see whether the obstacle at (x_O, y_O) lies within the polygon centered at (x_T, y_T) . Recall that transition points are defined as points on the trajectory where the active corners switch. In the full test for safety (Equation 7), this check is represented as `in_polygon()` and can leverage one of many point-in-polygon implementations, which generally run in linear time on the order of number of vertices. As the slope of the function may change at the boundary between piecewise subfunctions, we also add a notch check at each subdomain boundary.

D. Handling Piecewise Functions

In order to account for piecewise functions, we modify our method in two ways to avoid using a subfunction outside the subdomain over which it holds. The first is a modification to hold subfunctions constant outside of the subdomain over which they’re defined; and the second is an additional

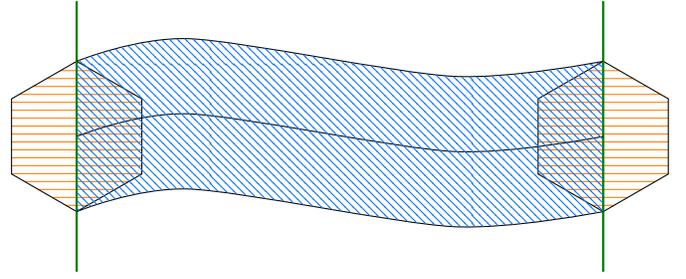


Fig. 5: Terms in Equation (7), illustrated. **Green** terms restrict test to relevant piecewise subdomains, **blue, diagonally-hatched** terms check if obstacles are between active corner pairs, and **orange, horizontally-hatched** terms check if obstacles are in the notch at transition points and subdomain bounds.

boolean clause to the safety test in (5) so it only applies over a valid subdomain. In this case, the subdomain is an interval $[x_{T_k}, x_{T(k+1)}]$, where $x_{T_k}, x_{T(k+1)}$ may be piecewise boundaries *or* transition points. Because of this, there may be many subdomains for a single piecewise case in which there happen to be many transition points.

First, we define a function $g(x)$ (or $g(y)$ symmetrically) that holds constant the value of each subfunction outside of its subdomain $[x_{T_k}, x_{T(k+1)}]$. The function $g(\cdot)$ is used in place of $f(\cdot)$ in (5) above. Let $y_{T_k} = f(x_{T_k})$.

$$g(x) = \begin{cases} y_{T_k} & \text{if } x \leq x_{T_k} \\ f(x) & \text{if } x_{T_k} < x < x_{T(k+1)} \\ y_{T(k+1)} & \text{if } x \geq x_{T(k+1)} \end{cases} \quad (6)$$

Additionally, we add a clause to ensure the modified subfunction $g(\cdot)$ is only used over the correct subdomain. First, we check $x_{T_k} - w < x_O < x_{T(k+1)} + w$, where w is the half-width of the object. We also construct a line between the two active corners of the object in each of the two piecewise boundary locations (x_{T_k}, y_{T_k}) and $(x_{T(k+1)}, y_{T(k+1)})$ and check (x_O, y_O) is between the two lines. This way, we ensure the test for being unsafe holds only for the region on which each subfunction applies. Figure 4 illustrates the function $g(\cdot)$ and the additional subdomain-related clauses.

E. Generic Explicit Formulation

This leads to a generic quantifier-free explicit formulation to test whether an obstacle is in the safe region, where $\{(x_T, y_T)_k\}$ represents the set of all transition *and* boundary points on the trajectory between piecewise subdomains. $g(\cdot)$ is used as defined previously in Section III-D. As defined previously, $\Delta x_i, \Delta y_i, \Delta x_j, \Delta y_j$ are the (constant) offsets from the center of the polygon to active corners v_i and v_j . Our algorithm generates a test for whether an obstacle is unsafe (if a collision will occur); negating the boolean formula or its result allows testing whether an obstacle lies in the safe region.

Equation (7) is color-coded in correspondence with Figure 5. The first, third, and fourth lines ensure the test applies only over the correct piecewise domain and are in **green**; the

$$\begin{aligned}
\text{unsafe?} = & \bigvee_{\{(g_k, x_{T_k}, x_{T(k+1)})\}} \left(x_{T_k} - w < x_O < x_{T(k+1)} + w \wedge (x_O, y_O) \text{ between} \right. \\
& \left(\text{Line}((x_{T_k} + \Delta x_i, y_{T_k} + \Delta y_i), (x_{T_k} + \Delta x_j, y_{T_k} + \Delta y_j)), \text{Line}((x_{T(k+1)} + \Delta x_i, y_{T(k+1)} + \Delta y_i), (x_{T(k+1)} + \Delta x_j, y_{T(k+1)} + \Delta y_j)) \right) \wedge \\
& \left. \bigvee_{\{(v_i, v_j)\}} (y_O - g(x_O + \Delta x_i) - \Delta y_i)(y_O - g(x_O + \Delta x_j) - \Delta y_j) \leq 0 \right) \vee \left(\bigvee_{\{(x_T, y_T)_i\}} \text{in_polygon}(x_{T_i}, y_{T_i}, x_O, y_O) \right)
\end{aligned} \tag{7}$$

second line checks the obstacle is between the active corners and is in blue; the fifth line is in orange and checks for the notch at transition points and piecewise subdomain boundaries.

F. Extensions

Thus far, we have considered point-mass obstacles, but the reasoning extends to obstacles that have the same properties as the object (convex and centrally symmetric). This is achieved through a reduction where the shape of the obstacle is incorporated into the shape of the object. For example, in the simple case where both are horizontal rectangles, with the object of height $2h$ and width $2w$, and the obstacle of height $2h_O$ and width $2w_O$, the object and obstacle intersect if and only if the center of the obstacle is contained in a virtual object with the same center as the initial object, but of height $2(h + h_O)$ and width $2(w + w_O)$. We have thus reduced the problem of collision avoidance with a convex object to a problem of avoidance with a point-mass object. A similar reasoning — albeit a little more complicated — can be applied to any convex, centrally symmetric obstacle.

For ease of presentation, and because they appear in most practical applications, we have focused on objects that are convex and centrally symmetric. We can extend the reasoning to non-centrally symmetric objects: the only difference in that case is that pairs of active corners do not change together, but rather one active corner may change on one side, and another active corner may change on the other side later. Pairs of active corners are thus not opposite corners of the object anymore. The convexity of the object (and obstacles) is essential for active corners; however, we can extend our reasoning to non-convex, polygonal objects by seeing them as unions of convex sub-objects and ensuring collision avoidance with each sub-object. Finally, due to its reliance on corners, our method cannot handle circles or ellipses, but they can be approximated by polygons.

IV. PROOF OF EQUIVALENCE

We prove the equivalence of the safe regions represented by 1) the implicit formulation and 2) the explicit output of our active-corner method for trajectories of form $y = f(x)$. The proof of soundness follows; the proof of completeness is in our full paper on arXiv. The proof structure considers segments of the trajectory in which no active corner switch occurs; that is, where the angle of the tangent to the trajectory is bounded. In these segments, the bounds on the trajectory tangent angle

allow us to bound the location of points in the interior of the polygon and show they lie between the two active corners. The two endpoints of a segment represent locations at which 1) the notch exists or 2) the trajectory switches to a new piecewise subfunction. In our method, these cases are handled by testing if obstacle (x_O, y_O) is inside the polygon at various transition points $\{(x_T, y_T)\}_i$.

A. Proof Preliminaries

Consider a segment of the motion along trajectory $y = f(x)$ or $x = f(y)$ in which no active corner switches or piecewise trajectory segment switches occur. We can arbitrarily rotate this segment of motion and the proof will hold, since the object translates along the trajectory without rotation. Assume, then, that a rotation is made by an angle θ such that the active corners are oriented along a vertical line. This rotation is an invertible transformation, so the logic of this proof holds through the entire trajectory. Because of this coordinate rotation, we consider only trajectories $y = f(x)$ for the proof; any trajectory $x = f(y)$ can be rotated into the form $y = f(x)$ invertibly, so our results hold for these forms as well. Let v_i, v_j denote the active corners for this segment, with corresponding offsets $\Delta x_i, \Delta y_i, \Delta x_j, \Delta y_j$ in the rotated coordinate system.

Since no active corner switch occurs, then we know the slope of function $y = f(x)$ is limited by the shape of the polygon itself — let these bounds be $\pm m$, with m representing the slope of the relevant sides of the polygon. Slopes of f beyond this range cannot occur over the trajectory segment in consideration, due to our assumption that the no active corner switches occur. Because the polygon is symmetric, the lower bound on slope is the negative of the upper bound (illustrated further in Figure 7). This proof is presented considering regular, symmetric polygons for simplicity, but extends to asymmetric polygons as discussed in our full paper on arXiv. To prove the soundness of our method, we must prove that all obstacles shown safe using our method ($\text{safe}_{\text{expl}}$) are also safe using the input implicit formulation ($\text{safe}_{\text{impl}}$). To prove $\text{safe}_{\text{expl}} \implies \text{safe}_{\text{impl}}$, we prove the contrapositive $\text{unsafe}_{\text{impl}} \implies \text{unsafe}_{\text{expl}}$.

Specifically, $\text{unsafe}_{\text{impl}}$ means that an obstacle at (x_O, y_O) is inside a polygon centered at some coordinates (x, y) ; $\text{unsafe}_{\text{expl}}$ means that the below holds from (5):

$$(y_O - f(x_O - \Delta x_i) - \Delta y_i) \cdot (y_O - f(x_O - \Delta x_j) - \Delta y_j) \leq 0$$

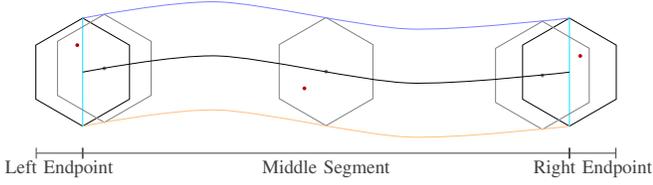


Fig. 6: Sections of proof

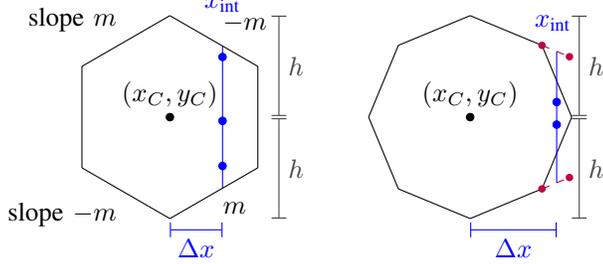


Fig. 7: Figure of the slope of the sides of a regular hexagon and octagon.

This proof has three sections: one holds for the majority of the trajectory segment, one for the beginning of the segment, and one of the end of the segment. The beginning- and end-of-segment proofs follow the form of the main proof but consider polygons fixed at the trajectory segment endpoints. They are included in our full paper available on arXiv.

B. Middle Segment Proof

Consider a (symmetric) polygon P , with half-height h and half-width w , centered at (x_C, y_C) , where $y_C = f(x_C)$. Let $(x_{\text{int}}, y_{\text{int}})$ be a point inside or on the edges of P . We prove that interior point $(x_{\text{int}}, y_{\text{int}})$ lies between the active corners of an identical polygon \bar{P} located at $(x_{\text{int}}, f(x_{\text{int}}))$. We do this by bounding three terms: 1) $f(x_{\text{int}})$, 2) y_{int} , and 3) the active corners of \bar{P} to prove that y_{int} lies between them.

First, we bound $f(x_{\text{int}})$ (the center of \bar{P}). Let $x_{\text{int}} = x_C + \Delta x$, for $\Delta x \in [-w, w]$. Let $f(x_{\text{int}}) = f(x_C + \Delta x) = y_C + \Delta y$, for some Δy which we will bound. The slope of the trajectory $\frac{dy}{dx}$ is bounded by $(-m, m)$, because this proof considers a segment of motion with no changes in active corner. Hence Δy is bounded proportionally to Δx , with $\Delta y \in (-|m\Delta x|, |m\Delta x|)$. Therefore, $f(x_{\text{int}}) \in (y_C - |m\Delta x|, y_C + |m\Delta x|)$. Our proof proceeds assuming $\Delta x \neq 0$, since if $\Delta x = 0$, x_{int} will lie on the vertical centerline of P . In that case, it is trivial to show x_{int} lies between the active corners.

Recall $x_{\text{int}} = x_C + \Delta x$, for $\Delta x \in [-w, w]$. Let $y_{\text{int}} = y_C + \Delta y_{\text{int}}$, for some Δy_{int} which we will bound. Given that the slopes of the sides on the top and bottom of P are $\pm m$, we assert that any $(x_{\text{int}}, y_{\text{int}})$ with $x_{\text{int}} = x_C + \Delta x$ has a corresponding $\Delta y_{\text{int}} \in [-h + |m\Delta x|, h - |m\Delta x|]$. This is illustrated in Figure 7 with a hexagon, but it generalizes to any symmetric convex polygon. Given this, we can bound the x and y interior coordinates as below:

$$(x_{\text{int}}, y_{\text{int}}) = \left[\begin{array}{c} x_C + \Delta x \\ [y_C - h + |m\Delta x|, y_C + h - |m\Delta x|] \end{array} \right] \quad (8)$$

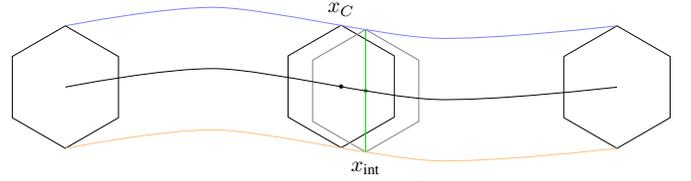


Fig. 8: Shifted polygon illustration

Finally, we show interior point y_{int} lies within the active corners of \bar{P} . Because we consider a rotated coordinate frame such that the active corners are oriented along the vertical axis, the top and bottom active corners are located at $(\bar{x}_{\text{top}}, \bar{y}_{\text{top}}) = (x_{\text{int}}, f(x_{\text{int}}) + h)$ and $(\bar{x}_{\text{bot}}, \bar{y}_{\text{bot}}) = (x_{\text{int}}, f(x_{\text{int}}) - h)$, respectively. The bounds on \bar{y}_{top} and \bar{y}_{bot} are given by the following:

$$\begin{aligned} y_C - |m\Delta x| + h &< \bar{y}_{\text{top}} < y_C + |m\Delta x| + h \\ y_C - |m\Delta x| - h &< \bar{y}_{\text{bot}} < y_C + |m\Delta x| - h \end{aligned} \quad (9)$$

Then $y_{\text{int}} \leq y_C - |m\Delta x| + h < \bar{y}_{\text{top}}$ and $y_{\text{int}} \geq y_C + |m\Delta x| - h > \bar{y}_{\text{bot}}$.

The top active corner trajectory is given by $f_{\text{top}}(x) = f(x) + h$ and the bottom active corner trajectory is given by $f_{\text{bot}}(x) = f(x) - h$. By definition, $f_{\text{top}}(x_{\text{int}}) = \bar{y}_{\text{top}}$ and $f_{\text{bot}}(x_{\text{int}}) = \bar{y}_{\text{bot}}$, or equivalently, $\bar{y}_{\text{top}} - f_{\text{top}}(x_{\text{int}}) = 0$ and $\bar{y}_{\text{bot}} - f_{\text{bot}}(x_{\text{int}}) = 0$. Since $y_{\text{int}} < \bar{y}_{\text{top}}$ and $y_{\text{int}} > \bar{y}_{\text{bot}}$,

$$y_{\text{int}} - f_{\text{top}}(x_{\text{int}}) < 0 \quad y_{\text{int}} - f_{\text{bot}}(x_{\text{int}}) > 0 \quad (10)$$

By multiplying the equations in (10), we get

$$(y_{\text{int}} - f_{\text{top}}(x_{\text{int}})) \cdot (y_{\text{int}} - f_{\text{bot}}(x_{\text{int}})) < 0 \quad (11)$$

This is an equivalent test for whether an object lies in the unsafe region from (5). Therefore we have shown that for all (x_C, y_C) points satisfying $y_C = f(x_C)$, all points $(x_{\text{int}}, y_{\text{int}})$ inside and on the boundary of a polygon centered at (x_C, y_C) also have $(f_{\text{top}}(x_{\text{int}}) - y_{\text{int}}) \cdot (f_{\text{bot}}(x_{\text{int}}) - y_{\text{int}}) \leq 0$. These are exactly the definitions of $\text{unsafe}_{\text{impl}}$ and $\text{unsafe}_{\text{expl}}$ from IV-A earlier. Therefore, we have shown $\text{unsafe}_{\text{impl}} \implies \text{unsafe}_{\text{expl}}$ and the contrapositive $\text{safe}_{\text{expl}} \implies \text{safe}_{\text{impl}}$ holds as well.

V. IMPLEMENTATION

We have implemented our automated method in Python using SymPy, a symbolic math library [30]. The code implementing our algorithm and the applications in Section VI is available on GitHub at <https://github.com/nskh/automatic-safety-proofs>.

Given a fully symbolic trajectory and object, we first identify the angles corresponding to sides of the object. Then, following Section III-B, we identify points on the trajectory corresponding to the angles θ_i of each side of the object. To avoid discontinuities in the arctan function, the implementation solves a reformulation: $\frac{\partial f}{\partial x} \sin(\theta_i) = \frac{\partial f}{\partial y} \cos(\theta_i)$. Solving this equation may yield either y in terms of x or the reverse. In this case, we substitute the implicit solution for x or y into the trajectory equation, eliminate the remaining variable, and identify transition points. Given transition points,

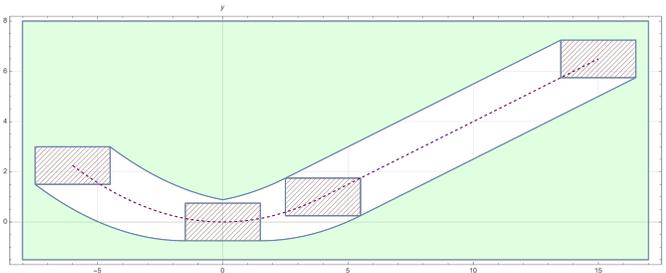


Fig. 9: Safe region for an instance of [22]. The notches are the red-hatched rectangles and the trajectory is dashed in purple.

we can implement the test from (7). SymPy includes a “point-in-polygon” method, which we use to identify if an obstacle (x_O, y_O) lies in the “notch” at any transition point. The output explicit formulation can be expressed either in \LaTeX or in Mathematica format; output in Mathematica also supports generating code to copy-paste directly and plot the safe region using Mathematica’s `RegionPlot[]` functionality. Examples can be found in Figures 9 and 10.

In order to implement our method in a fully symbolic fashion, we must account for the potential values of symbols when instantiated. We can leverage SymPy’s built-in “assumptions” to specify that certain symbols representing, say, trajectory parameters or object dimensions are real, positive, and/or nonzero, but these assumptions may not suffice to construct a fully symbolic safe region. In that case, our fully symbolic implementation computes a number of potential valid safe regions. As detailed in Section III-D, we construct the explicit formulation using many clauses defined on intervals between transition points and/or piecewise boundaries. In the symbolic case, the order of these terms may differ, depending on, say, the sign of a variable in the trajectory. Additionally, symbolic piecewise cases for, say, $x < b$ may mean that certain transition points do not occur at all if b lies in some range. Correspondingly, our fully symbolic implementation computes all valid orderings of piecewise boundaries and transition points; it additionally considers all valid combinations of transition points to account for “notches” that may not exist when piecewise bounds and/or trajectory parameters are instantiated. In order to check if orderings are valid, we attempt to sort using the SymPy assumptions: if we know b is positive, no returned ordering will place b before a transition point at 0, for example. Additionally, we enforce that adjacent points in the ordering “come from” the same functions: we will not return an ordering where a transition point from piecewise subfunction f_1 lies between the piecewise boundaries for f_2 . Doing so ensures that we often generate relatively few (~ 10) potential orderings despite considering many combinations, though examples with intractably many orderings do exist.

VI. APPLICATIONS AND EVALUATION

A. Verification of vertical maneuvers in ACAS X

A collision avoidance system intended to prevent near mid-air collisions, ACAS X, was verified in [22]. The KeYmaera X proof presented in [22] required a significant amount of human

interaction (on the order of hundreds of hours), while the method presented in this paper generates an explicit formulation from the trajectory fully automatically. ACAS X prevents collisions between aircraft by issuing advisories (control commands) to one aircraft, the *ownship*. The bounds of aircraft in this work are shaped like hockey pucks (cylinders wider than they are tall) of a radius r_p and half-height h_p . From a side perspective of an encounter between aircraft, the bounds are rectangular. In [22], verification was performed in a side-view perspective, assuming two aircraft approach each other in a vertically-oriented planar slice of three dimensions. A careful choice of reference frame can reduce a three-dimensional encounter between aircraft into a two-dimensional system, by modeling the encounter as a 1-dimensional vertical encounter and the distance of a horizontal encounter [22, Section 6].

To simplify calculations, [22] used the relative horizontal speed r_v of the two aircraft and assumed it constant; the vertical velocity of the oncoming aircraft \dot{h} is also assumed constant. Advisories consist of climb and descent speed advisories, yielding ownship trajectories that are piecewise combinations of parabolas and straight lines. One example trajectory is below in (12), which assumes the advisory issued is for the ownship to climb at a rate \dot{h}_f greater than its current vertical velocity \dot{h}_0 . (r_t, h_t) are the (x, y) coordinates for trajectory \mathcal{T} in this example, and a_r is the acceleration.

$$(r_t, h_t) = \begin{cases} \left(r_v t, \frac{a_r}{2} t^2 + \dot{h}_0 t \right) & \text{for } 0 \leq t < \frac{\dot{h}_f - \dot{h}_0}{a_r} \\ \left(r_v t, \dot{h}_f t - \frac{(\dot{h}_f - \dot{h}_0)^2}{2a_r} \right) & \text{for } \frac{\dot{h}_f - \dot{h}_0}{a_r} \leq t \end{cases} \quad (12)$$

The implicit formulation of the safe region is below, for an oncoming aircraft at relative coordinates r_O, h_O .

$$\forall t. \forall r_t. \forall h_t. ((r_t, h_t) \in \mathcal{T} \implies |r_O - r_t| > r_p \vee |h_O - h_t| > h_p) \quad (13)$$

In [22], the authors eliminate the parametrization over t , which yields an initial parabolic section and then straight-line motion after. We use this t -free trajectory to compute the unsafe region, which is displayed in Figure 9. A boolean formulation of the unsafe region is available in our full paper on arXiv.

B. Verified Turning Maneuvers for Unmanned Aerial Vehicles

Turning maneuvers for unmanned aerial vehicles (UAVs) have been verified as safe in [1], where the UAV was represented as a circular safety buffer around a point object fixed along the trajectory. The KeYmaera X proof presented in [1] required a significant amount of human interaction (on the order of hundreds of hours); in contrast, the method presented in this paper generates an explicit formulation from the trajectory fully automatically. This work represents motion in a two-dimension plane viewed top-down, with the buffer “puck” taking the form of a circle. The turning maneuver trajectory moves along a circular arc then in a straight line:

$$(x_{\mathcal{T}}, y_{\mathcal{T}}) = \begin{cases} x_{\mathcal{T}}^2 + y_{\mathcal{T}}^2 = R^2 & y_{\mathcal{T}} < x_{\mathcal{T}} \tan \theta \\ y_{\mathcal{T}} = \frac{R \cos \theta - x_{\mathcal{T}}}{\tan \theta} + R \sin \theta & y_{\mathcal{T}} \geq x_{\mathcal{T}} \tan \theta \end{cases} \quad (14)$$

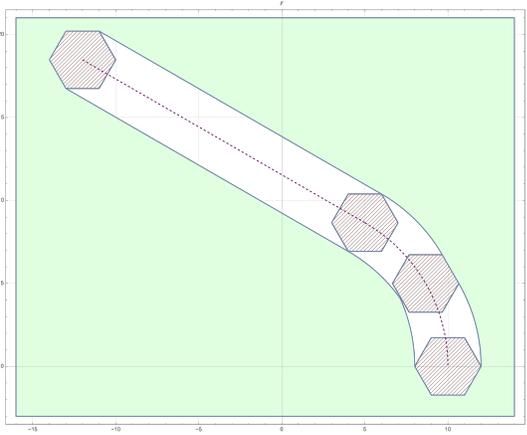


Fig. 10: Approximated safe region for an instance of [1]. The notches are the red-hatched hexagons, the trajectory is dashed in purple.

With a circular safety buffer of radius r_p , the implicit formulation ensures for all points along the trajectory, the obstacle (x_O, y_O) is at least r_p away.

$$\forall x_{\mathcal{T}}, \forall y_{\mathcal{T}}. (\text{traj}(x_{\mathcal{T}}, y_{\mathcal{T}}) \implies (x_O - x_{\mathcal{T}})^2 + (y_O - y_{\mathcal{T}})^2 \geq r_p^2) \quad (15)$$

Note that our method does not support circular objects, only polygons, so we overapproximate the circular safety buffer as a regular hexagon inscribing a circle. This approximation allows a valid overapproximation of the unsafe region, since the hexagon contains the original circle in [1]. Note that the approximation of a circle can be made arbitrarily precise by increasing the number of sides of a polygon used. A plot of the unsafe region is in Figure 10 and a boolean formulation of the unsafe region is in our full paper on arXiv.

C. Runtime Evaluation

This section presents a comparison of our method to quantifier elimination via cylindrical algebraic decomposition (CAD) [12]. We consider a variety of cases, from fully numeric to fully symbolic. Fully symbolic cases use trajectories without real constants, like $ax^2 + bx + c = d$, and polygons with variable dimensions like rectangles of width w and height h . Fully numeric cases instantiate all parameters with reals to yield, say, trajectory $4x^2 + 2x + 1$ and a rectangle of width 2 and height 1. In Table I, our “Numeric Trajectory” examples instantiate only the trajectory with reals but leave the polygon symbolic, and the “Numeric Hexagon/Rectangle” examples leave the trajectory symbolic but use reals for the polygon dimensions.

Results were generated using a 2017 iMac Pro workstation with 128 GB of RAM, with CAD results using Mathematica’s `Resolve` implementation. A table of results is shown in Table I. We use the examples from VI-A (ACAS X) and VI-B (UAV). The Dubins path example is inspired by common path planners and takes the form of two circular arcs connected by a straight line and ending with a line; its symbolic trajectory equation is included in our full paper on arXiv.

Our findings in Table I demonstrate the advantages and disadvantages of our method relative to quantifier elimination using CAD. For non-polynomial examples like a rectangle moving along the Dubins path described above or the UAV example from [1], the active corner method is able to compute fully symbolic formulations of the safe region when CAD fails to return an answer when run overnight (8+ hours). We do note that due to the complexity of a symbolic *hexagon* moving along the Dubins path, the number of transition points means our method cannot compute an answer, though neither can CAD. For a fully numeric example from [1], CAD took 2381 seconds to run but returned `False` incorrectly in place of a region. Additionally, memory is often a constraint for symbolic computation given the CAD algorithm’s doubly-exponential runtime [14]; many examples consumed 100+ GB of RAM and one case grew to consume 350 GB of RAM without returning an answer. In the worst case, however, our method consumes under 100MB of RAM. On the other hand, for strictly polynomial examples like that in [22], CAD runs quickly and efficiently, though our method remains competitive.

VII. RELATED WORK

Reachability computation is a vital question in safety-critical cases where users seek to guarantee properties or behavior. One method of constructing reachable sets for safety is zonotope reachability [3]. Reachability computation using zonotopes offers efficient algorithmic methods and supports analysis of dynamical systems with uncertainty. Zonotopes have been used in verification of automated vehicles [2], the design of safe trajectories for quadrotor aircraft [24], and the analysis of power systems [15], among other applications. Zonotope reachability methods discretize a dynamical system and iteratively propagate an estimate of the reachable set forward in time. Their input is a differential equation, while our method requires an explicit closed-form trajectory. For the purpose of checking safety, the estimate of the reachable set must be either exact or an overestimate; in order to deal with discretization error, zonotope methods repeatedly overestimate the reachable interval. Zonotope methods for nonlinear systems rely on linearization and again account for error that may occur by expanding the reachable set [4]. Our method yields exact reachable sets. While it is possible to model convex object reachability with zonotopes, the reachable set expands with the time horizon because the dimensions of the object are treated not as constant dimensions but as uncertainty in initial conditions that is propagated forward through time [20].

Interval-based reachability methods share similarities to zonotope methods but do not aim for the tightest approximations possible; instead simpler axis-aligned sets (hyper-rectangles in high dimensions) are used for computation [32], [33]. These representations simplify storage in memory and operations like intersections but do not compute exact estimates in the way our method does. However, they do support both continuous [31], [25] and discrete [13] dynamic systems with uncertainty. Set-valued constraint solving may be used but similarly relies on inexact discretization [21]. Other reach-

Example	Instance	Active Corners Time	Active Corners RAM	CAD Time	CAD RAM
UAV	Fully Numeric	0.48 sec	7.1 MB	2381* <i>sec</i>	30.89 MB
UAV	Numeric Trajectory	0.82 sec	8.4 MB	DNF	50+ GB
UAV	Numeric Hexagon	38 sec	22 MB	DNF	100+ GB
UAV	Fully Symbolic	45 sec	24 MB	DNF	100+ GB
Dubins	Fully Numeric	1.2 sec	9.0 MB	DNF	11+ GB
Dubins	Fully Symbolic: Rectangle	4505 sec	91 MB	DNF	4+ GB
Dubins	Fully Symbolic: Hexagon	DNF	N/A	DNF	8+ GB
ACAS X	Fully Numeric	0.13 sec	5.9 MB	0.04 sec	160 KB
ACAS X	Numeric Trajectory	0.48 sec	6.6 MB	0.04 sec	188 KB
ACAS X	Numeric Rectangle	0.51 sec	6.6 MB	0.2 sec	325 KB
ACAS X	Fully Symbolic	0.57 sec	6.6 MB	1.1 sec	1.8 MB

TABLE I: Evaluation results, with better results **bolded**. DNF: example did not finish in 8+ hours. *: *incorrect answer*.

ability methods for differential equations include Hamilton-Jacobi reachability for systems with complex, nonlinear, high-dimensional dynamics [7], and control barrier functions, which enable the construction of safe optimization-based controllers [5].

A counterpart to reachability is automatic invariant generation for hybrid systems, in which a formal statement showing a system never evolves into an unsafe state is proved. In [17], the authors proved a polynomial and its Lie derivatives can represent algebraic sets of polynomial vector fields. A procedure to check invariance of polynomial equalities was proposed in [18]. Semi-algebraic invariants for polynomial ODEs were studied in [19], [40], [28]. Invariants for hybrid systems were studied in [37] and [29]. Relational abstractions bridge the gap between continuous and discrete modes by over-approximating continuous system evolution to summarize the system as a purely discrete one using invariant generation [38]. Barrier certificates have also been used as invariants for safety verification in hybrid systems [36].

Our work has similar aims to swept-volume collision checking, from path planning and graphics, in which approximate, efficient collision-checking is performed as a volume is moved along a path. A convex over-approximation swept-volume approach was presented in [16]. Swept-volume checking in four dimensions was performed using an intersection test in space-time in [10]. An efficient algorithm computing distances between convex polytopes, the Lin-Canny algorithm, was proposed for this task in [26], [27]. Methods are typically discrete and approximate for performance in online applications. That said, there are some exact methods such as collision checking for straight-line segments like those on robotic arms [39] and an algorithm for large-scale environments [11]. However, these methods operate on individual collision checking instances, such as graphics simulations or video game environments, and their results cannot be used repeatedly. Our method yields provably correct, fully symbolic, and exact safe regions for continuous trajectories and supports, for example, quantifier-free and efficient testing in runtime or in large-scale settings once a desired safe region formulation has been generated.

Another alternative to this work is quantifier elimination, a general algorithm for converting formulas with quantified

variables into equivalent statements that are quantifier-free [41], [12]. Quantifier elimination can be performed using Cylindrical Algebraic Decomposition (CAD), an algorithm that operates on semialgebraic sets [12], [6]. QEPCAD is one notable software tool implementing CAD that could be used in this work [8]. The runtime of the CAD algorithm is doubly exponential in the number of total variables (not the number of quantified variables) [14], [9]; we offer a detailed comparison to CAD in Mathematica in Section VI-C.

VIII. CONCLUSION AND FUTURE WORK

We have presented an automated approach to construct explicit safe regions for convex polygons moving in the plane with piecewise equations of motion of the form $y = f(x)$ and $x = f(y)$. We have also proved the equivalence of the implicit and explicit formulations of the safe region, discussed an automated implementation of our method, and benchmarked the performance of our method compared to quantifier elimination using cylindrical algebraic decomposition.

We would like to study how our method extends to objects translating in 3 or n dimensions; we conjecture that active corners will become active edges in three dimensions (and $(n - 2)$ -polyhedra in n dimensions). Additionally, we would like to expand to handle trajectories in the form of inequalities; rotating objects; and invariants of differential equations of the form $f(x, y) = 0$ rather than explicit trajectories. On the implementation side, we are currently exploring how to automatically output a machine-checkable proof of equivalence between the implicit and explicit formulations, using the PVS theorem prover.

ACKNOWLEDGEMENTS

The authors would like to thank Nikos Aréchiga and Gabor Orosz for their feedback during the development of this research, as well as Jiawei Chen, Necmiye Ozay, and Mohit Tekriwal for comments on earlier versions of this paper. This work was partially funded by a NASA Fellowship. Toyota Research Institute (“TRI”) provided funds to assist the authors with their research, but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity. Thanks also to the SISL lab at Stanford for their `aircraftshapes` TikZ library.

REFERENCES

- [1] Eytan Adler and Jean-Baptiste Jeannin. Formal verification of collision avoidance for turning maneuvers in uavs. In *AIAA Aviation 2019 Forum*, page 2845, 2019.
- [2] Matthias Althoff and John M Dolan. Online verification of automated road vehicles using reachability analysis. *IEEE Transactions on Robotics*, 30(4):903–918, 2014.
- [3] Matthias Althoff, Goran Frehse, and Antoine Girard. Set propagation techniques for reachability analysis. *Annual Review of Control, Robotics, and Autonomous Systems*, 4:369–395, 2021.
- [4] Matthias Althoff, Olaf Stursberg, and Martin Buss. Reachability analysis of nonlinear systems with uncertain parameters using conservative linearization. In *2008 47th IEEE Conference on Decision and Control*, pages 4042–4048, 2008.
- [5] Aaron D Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath, and Paulo Tabuada. Control barrier functions: Theory and applications. In *2019 18th European Control Conference (ECC)*, pages 3420–3431. IEEE, 2019.
- [6] Dennis S Arnon, George E Collins, and Scott McCallum. Cylindrical algebraic decomposition i: The basic algorithm. *SIAM Journal on Computing*, 13(4):865–877, 1984.
- [7] Somil Bansal, Mo Chen, Sylvia Herbert, and Claire J Tomlin. Hamilton-jacobi reachability: A brief overview and recent advances. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2242–2253. IEEE, 2017.
- [8] Christopher W. Brown. Qepcad b: A program for computing with semi-algebraic sets using cads. *SIGSAM Bull.*, 37(4):97–108, December 2003.
- [9] Christopher W Brown and James H Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 54–60, 2007.
- [10] Stephen Cameron. Collision detection by four-dimensional intersection testing. 1990.
- [11] Jonathan D Cohen, Ming C Lin, Dinesh Manocha, and Madhav Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff, 1995.
- [12] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, pages 134–183, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.
- [13] Samuel Coogan and Murat Arcak. Efficient finite abstraction of mixed monotone systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 58–67, 2015.
- [14] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1):29–35, 1988.
- [15] Ahmed El-Guindy, Yu Christine Chen, and Matthias Althoff. Compositional transient stability analysis of power systems via the computation of reachable sets. In *2017 American Control Conference (ACC)*, pages 2536–2543. IEEE, 2017.
- [16] A Foisy and V Hayward. A safe swept volume method for collision detection. In *The Sixth International Symposium of Robotics Research*, pages 61–68, 1993.
- [17] Khalil Ghorbal and André Platzer. Characterizing algebraic invariants by differential radical invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 279–294. Springer, 2014.
- [18] Khalil Ghorbal, Andrew Sogokon, and André Platzer. Invariance of conjunctions of polynomial equalities for algebraic differential equations. In *International Static Analysis Symposium*, pages 151–167. Springer, 2014.
- [19] Khalil Ghorbal, Andrew Sogokon, and André Platzer. A hierarchy of proof rules for checking positive invariance of algebraic and semi-algebraic sets. *Computer Languages, Systems & Structures*, 47:19–43, 2017.
- [20] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *International Workshop on Hybrid Systems: Computation and Control*, pages 291–305. Springer, 2005.
- [21] Luc Jaulin. Solving set-valued constraint satisfaction problems. *Computing*, 94(2):297–311, 2012.
- [22] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and André Platzer. A formally verified hybrid system for the next-generation airborne collision avoidance system. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–36. Springer, 2015.
- [23] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Aurora Schmidt, Ryan W. Gardner, Stefan Mitsch, and André Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *Int. J. Softw. Tools Technol. Transf.*, 19(6):717–741, 2017.
- [24] Shreyas Kousik, Patrick Holmes, and Ram Vasudevan. Safe, aggressive quadrotor flight via reachability-based trajectory design. In *Dynamic Systems and Control Conference*, volume 59162, page V003T19A010. American Society of Mechanical Engineers, 2019.
- [25] Thomas Le Mézo, Luc Jaulin, and Benoit Zerr. An interval approach to compute invariant sets. *IEEE Transactions on Automatic Control*, 62(8):4236–4242, 2017.
- [26] Ming C Lin. Efficient collision detection for animation. In *Proceedings of the Third Euro-graphics Workshop on Animation Cambridge*, 1992.
- [27] Ming C Lin. *Efficient collision detection for animation and robotics*. PhD thesis, PhD thesis, Department of Electrical Engineering and Computer Science, UC Berkeley, 1993.
- [28] Jiang Liu, Naijun Zhan, and Hengjun Zhao. Computing semi-algebraic invariants for polynomial dynamical systems. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 97–106, 2011.
- [29] Nadir Matringe, Arnaldo Vieira Moura, and Rachid Rebiha. Generating invariants for non-linear hybrid systems by linear algebraic methods. In *International Static Analysis Symposium*, pages 373–389. Springer, 2010.
- [30] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMIT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Tereš, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.
- [31] Pierre-Jean Meyer, Samuel Coogan, and Murat Arcak. Sampled-data reachability analysis using sensitivity and mixed-monotonicity. *IEEE control systems letters*, 2(4):761–766, 2018.
- [32] Pierre-Jean Meyer, Alex Devonport, and Murat Arcak. Tira: Toolbox for interval reachability analysis. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 224–229, 2019.
- [33] Pierre-Jean Meyer, Alex Devonport, and Murat Arcak. *Interval Reachability Analysis: Bounding Trajectories of Uncertain Systems with Boxes for Control and Verification*. Springer Nature, 2021.
- [34] Stefan Mitsch, Khalil Ghorbal, and André Platzer. On provably safe obstacle avoidance for autonomous robotic ground vehicles. In *Robotics: Science and Systems IX, Technische Universität Berlin, Berlin, Germany, June 24-June 28, 2013*, 2013.
- [35] André Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.*, 59(2):219–265, 2017.
- [36] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *International Workshop on Hybrid Systems: Computation and Control*, pages 477–492. Springer, 2004.
- [37] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. Constructing invariants for hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 539–554. Springer, 2004.
- [38] Sriram Sankaranarayanan and Ashish Tiwari. Relational abstractions for continuous and hybrid systems. In *International Conference on Computer Aided Verification*, pages 686–702. Springer, 2011.
- [39] Fabian Schwarzzer, Mitul Saha, and Jean-Claude Latombe. Exact collision checking of robot paths. In *Algorithmic foundations of robotics V*, pages 25–41. Springer, 2004.
- [40] Andrew Sogokon, Khalil Ghorbal, Paul B Jackson, and André Platzer. A method for invariant generation for polynomial continuous systems. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 268–288. Springer, 2016.
- [41] Alfred Tarski. A decision method for elementary algebra and geometry. University of California Press, Berkeley, 1951.

Differential Testing of Pushdown Reachability with a Formally Verified Oracle

Anders Schlichtkrull  Morten Konggaard Schou  Jiří Srba 
Department of Computer Science
Aalborg University
 Aalborg, Denmark
 {andsch,mksc,srba}@cs.aau.dk

Dmitriy Traytel 
Department of Computer Science
University of Copenhagen
 Copenhagen, Denmark
 traytel@di.ku.dk

Abstract—Pushdown automata are an essential model of recursive computation. In model checking and static analysis, numerous problems can be reduced to reachability questions about pushdown automata and several efficient libraries implement automata-theoretic algorithms for answering these questions. These libraries are often used as core components in other tools, and therefore it is instrumental that the used algorithms and their implementations are correct. We present a method that significantly increases the trust in the answers provided by the libraries for pushdown reachability by (i) formally verifying the correctness of the used algorithms using the Isabelle/HOL proof assistant, (ii) extracting executable programs from the formalization, (iii) implementing a framework for the differential testing of library implementations with the verified extracted algorithms as oracles, and (iv) automatically minimizing counter-examples from the differential testing based on the delta-debugging methodology. We instantiate our method to the concrete case of PDAAAL, a state-of-the-art library for pushdown reachability. Thereby, we discover and resolve several nontrivial errors in PDAAAL.

I. INTRODUCTION

In 1964, Büchi [7] proved that the possibly infinite set of all reachable pushdown configurations (from a given initial configuration) can be effectively described by a regular language. In fact, even for a given regular set of pushdown configurations, its $post^*$ and pre^* closures (representing all forward and backward reachable configurations from a given set of configurations) are also regular. Büchi’s *automata-theoretic approach* gave rise to a rich theory of pushdown reachability with numerous algorithms and applications to, e.g., interprocedural control-flow analysis of recursive programs [9], [11], model checking [4], [13], [45], [46], communication network analysis [10], [21], [22] and others. A number of tools have been developed to support the theory, including Moped [45], [46], WALi [25], and PDAAAL [23] with applications ranging from the static analysis of Java [46] and C/C++ code [26], [43] to the analysis of MPLS communication protocols [22].

Even though the automata-theoretic approach for pushdown reachability is based on relatively simple saturation procedures, the proofs of correctness are nontrivial and the implementation of the algorithms in the different tools often includes numerous performance optimizations as well as additional improvements to the theory itself [23]. To be able to rely on

the output of model checking tools and other applications of pushdown reachability, it is important that the theory is not only sound but also correctly implemented. A positive reachability answer is typically accompanied by a finite evidence (trace) that can function as an efficiently checkable certificate. A negative answer is, on the other hand, much harder to check, and designing a finite evidence for non-reachability is difficult, primarily because the number of reachable pushdown configurations can be infinite. One approach is to establish an invariant that (i) includes the initial configuration(s) of the system, (ii) is maintained by the transition relation and (iii) has an empty intersection with the set of undesirable configurations. Such approaches have been studied [16], [17] but are usually incomplete and require another complex tool (that can be error-prone, too) to verify such invariants.

We instead use a proof assistant, Isabelle/HOL [37] (§II), to formally verify the correctness of the pushdown reachability algorithms $post^*$ (forward search), pre^* (backward search), and $dual^*$ (bi-directional search) (§III) that lie at the heart of the automata-theoretic analysis of pushdown systems [4], [23], [44]. From the formalization of pre^* , we extract an executable program with strong correctness guarantees (§IV). For a given input, the extracted program’s output can be compared with the output of other, unverified but optimized tools solving the same problem (§V). This approach is known as differential testing [14], [18], [34] with a twist that the testing oracle has been formally verified and thus is extremely trustworthy. When testing reveals a disagreement between a verified and an unverified algorithm, we know who is to blame. To help localize errors in unverified algorithms, we minimize the tests causing disagreement using the delta-debugging technique [51]. Our main contributions are as follows.

- The formalization of $post^*$, pre^* and $dual^*$ algorithms in Isabelle/HOL and verification of their correctness based on the proofs provided by Schwoon [44] for $post^*$ and pre^* , and following Jensen et al. [23] for $dual^*$.
- The refinement to and the extraction of an executable program of the formalized pre^* algorithm that serves as the verified oracle for differential testing.
- The automatic minimization of the input automata in cases where an unverified tool disagrees with the oracle.

This research was supported by the Independent Research Fund Denmark (DFF project QASNET) and by Novo Nordisk Fonden (NNF20OC0063462).

- The application of our method to a modern state-of-the-art library for pushdown reachability, PDAAAL [23], and the identification, localization (using the minimized counter-examples), and correction of three, previously unknown, implementation errors (§VI). The corrected implementation passes all differential tests successfully.

Our Isabelle formalization as well as the case study are publicly available [40].

a) **Related work.** Differential testing with a verified oracle has been used in the context of runtime verification and automatic theorem proving. The runtime monitor VeriMon [2], [41] served as the verified oracle used to detect errors in unverified monitors. Compared to our approach, VeriMon’s differential testing case study is from a different application domain, does not include exhaustive test generation for small input sizes (which is difficult in runtime monitoring) and does not minimize the tests automatically. To assess its performance but also to evaluate the benchmark’s correctness, the verified first-order prover RPx [39] was evaluated on a standard benchmark for first-order logic problems. RPx’s answers have in all cases coincided with the expected ones recorded in the benchmark.

The verified C compiler CompCert [32] and several verified distributed systems [20], [33], [48] have been themselves put onto the testbed [15], [50]. A few errors in these tools’ unverified parts or in scenarios violating the verification assumptions were found, but none in the verified components themselves.

Many works extract efficient executable code from formalizations, but do not use it as an oracle in testing. Examples include verified model checkers for LTL [12] and timed automata [49] and verified algorithms for finite automata [3], [6], [24], [31] and context-free grammars [35], [38].

The only formalization of pushdown automata we are aware of is part of Lammich et al.’s work on dynamic pushdown networks (DPN) [30]. Lammich describes the Isabelle formalization of an executable pre^* algorithm for DPNs stemming from this work in an unpublished technical report [29]. DPNs generalize pushdown automata, but their $post^*$ is not regular [5] and so we cannot extend this work for our purposes. Moreover, Lammich’s formalization does not support ε -transitions in the underlying automata, an essential component needed for our formalization of $post^*$ and $dual^*$.

b) **Background definitions.** Let P be a finite set of control locations and Γ a finite stack alphabet. A *pushdown system* (PDS) is a tuple (P, Γ, Δ) , where $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of *rules*, written $(p, \gamma) \hookrightarrow (q, w)$ whenever $((p, \gamma), (q, w)) \in \Delta$. Without loss of generality, we assume $|w| \leq 2$, so that $w = \varepsilon$ represents a *pop* operation that removes the topmost stack symbol, $|w| = 1$ is a *swap* that replaces the topmost symbol with another one, and $|w| = 2$ is a *push* that incorporates a *swap* followed by adding a new symbol on top.

A *configuration* of a pushdown system is a pair (p, w) of the current control location $p \in P$ and the current stack content $w \in \Gamma^*$ where we assume that the top of the stack is on the left. The set of all configurations is denoted by \mathcal{C} . A PDS can take a *computation step* $(p, \gamma w') \Rightarrow (q, ww')$ between configurations whenever $(p, \gamma) \hookrightarrow (q, w)$ and $w' \in \Gamma^*$. For a given $C \subseteq \mathcal{C}$,

we define $post^*(C) = \{c' \in \mathcal{C} \mid c \Rightarrow^* c' \text{ for some } c \in C\}$ and $pre^*(C) = \{c \in \mathcal{C} \mid c \Rightarrow^* c' \text{ for some } c' \in C\}$.

The *reachability problem* for PDSs is to decide whether $c \Rightarrow^* c'$ for configurations c and c' , and it is equivalent to asking whether $c' \in post^*(\{c\})$ or equivalently whether $c \in pre^*(\{c'\})$. Büchi [7] showed that for any regular set $C \subseteq \mathcal{C}$, the sets $post^*(C)$ and $pre^*(C)$ are also regular.

To represent regular sets of pushdown configurations, we use *P-automata* [44], which are nondeterministic finite automata with multiple initial states for each of the control locations from the set P . Formally, let N be a finite set of noninitial states and $F \subseteq P \cup N$ a finite set of final states. A *P-automaton* is a tuple $\mathcal{A} = (P \cup N, \rightarrow, P, F)$ with the transition relation $\rightarrow \subseteq (P \cup N) \times \Gamma \times (P \cup N)$ so that $P \cup N$ is the set of its states and the pushdown alphabet Γ is the input alphabet of the automaton. The language $L(\mathcal{A})$ of *P-automaton* \mathcal{A} contains the pushdown configurations accepted by \mathcal{A} : a configuration $(p, w) \in P \times \Gamma^*$ is accepted if and only if there is a path from p to q for some $q \in F$ in the *P-automaton* (defined via the transition relation \rightarrow) labelled with w . The *reachability problem for P-automata* is as follows: given a PDS (P, Γ, Δ) and *P-automata* \mathcal{A}_1 and \mathcal{A}_2 , does there exist $c \in L(\mathcal{A}_1)$ and $c' \in L(\mathcal{A}_2)$ such that $c \Rightarrow^* c'$ using the rules Δ ?

II. ISABELLE/HOL

Isabelle/HOL [37] is a proof assistant based on classical higher-order logic (HOL), a simply typed lambda calculus with Hilbert choice, axiom of infinity, and rank-1 polymorphism. We present our formalization using HOL’s syntax, which mixes functional programming and mathematical notation.

Types are built from type variables $'a, 'b, \dots$ and type constructors like pairs $_ \times _$ and functions $_ \Rightarrow _$ (both written infix) and sets $_ set$ (written postfix). Type constructors can also be nullary, e.g., the Boolean type *bool*. Type variables can be restricted by type classes: $'a :: finite$ is a type variable $'a$ that can only be instantiated with finite types (i.e., types with finitely many inhabitants). New type constructors are introduced as abbreviations for complex type expressions and as inductive datatypes using commands **type synonym** and **datatype** respectively, e.g., the types of transitions **type synonym** $(state, label) transition = state \times label \times state$ and finite lists **datatype** $'a list = [] \mid 'a \# ('a list)$.

Terms are built from variables x, y, \dots , constants c, d, \dots , lambda abstractions $\lambda x. t$ and applications written as juxtaposition $f x$. Isabelle includes many constants and syntax for them, e.g., infix operators $\wedge, \vee, \longrightarrow, \longleftarrow, \in, \cup, \cap$, unbounded and bounded quantifiers $\exists x. P x$ and $\forall y \in A. Q y$, and set comprehensions $\{x. P x\}$. Non-recursive functions are defined and given readable syntax using the **definition** command:

definition image (infix $'$) where

$$f ' A = \{y. \exists x \in A. y = f x\}$$

Type annotations like $image :: ('a \Rightarrow 'b) \Rightarrow 'a set \Rightarrow 'b set$ can be omitted as they are inferred. Recursive definitions are supported using the **fun** command:

fun append (infix $@$) where

$$[] @ ys = ys \mid (x \# xs) @ ys = x \# (xs @ ys)$$

```

locale LTS = fixes trans_rel :: ('state, 'label) transition set
begin
  definition step_relp (infix  $\Rightarrow$ ) where
     $c \Rightarrow c' \iff (\exists l. (c, l, c') \in \text{trans\_rel})$ 
  definition step_starp (infix  $\Rightarrow^*$ ) where
     $c \Rightarrow^* c' \iff \text{step\_relp}^{**} c c'$ 
  definition pre_star  $C = \{c'. \exists c \in C. c' \Rightarrow^* c\}$ 
  definition post_star  $C = \{c'. \exists c \in C. c \Rightarrow^* c'\}$ 
  definition srcs =  $\{p. \nexists q \gamma. (q, \gamma, p) \in \text{trans\_rel}\}$ 
  definition sinks =  $\{p. \nexists q \gamma. (p, \gamma, q) \in \text{trans\_rel}\}$ 
  inductive_set trans_star where
     $(p, [], p) \in \text{trans\_star}$ 
    |  $(p, \gamma, q') \in \text{trans\_rel} \longrightarrow (q', w, q) \in \text{trans\_star} \longrightarrow$ 
     $(p, \gamma \# w, q) \in \text{trans\_star}$ 
end

```

Fig. 1: The locale for labeled transition systems

Internally, **fun** performs an automatic termination proof. More complex recursion schemes may require a manual proof.

Another way to define a function is as Prolog-style monotone rules. The **inductive** command allows such definitions as least fixed points. Take, e.g., the reflexive transitive closure:

```

inductive rtranclp (_**) where
   $R^{**} x x \mid R x y \longrightarrow R^{**} y z \longrightarrow R^{**} x z$ 

```

Theorems and lemmas are terms of type *bool* that have been proved to be equivalent to *True*. All proofs pass through Isabelle’s kernel, which relies only on a few well-understood reasoning rules such as *modus ponens*. We refer to a textbook [36] for a practical introduction to proving in Isabelle.

Structures and assumptions common to many theorems can be organized via locales [1]—Isabelle’s module mechanism for fixing parameters and stating and assuming their properties. In the context of a locale, the parameters are available as constants and the assumptions as facts. Locales can be interpreted, which involves instantiating the parameters and proving the assumptions. As the result, one obtains the (instantiated) theorems proved in the context of the locale.

Consider our locale for labeled transition systems (LTSs) in Fig. 1. It fixes the parameter *trans_rel*, and its context consists of the definitions between the **begin** and **end** keywords. All definitions should be self-explanatory except perhaps *trans_star*: the set of triples (p, w, q) for which the LTS can move from p to q by consuming word w . This relation is defined inductively, first for the empty sequence and then extending it by one more symbol—here we use in conjunction two assumptions on the symbol γ and sequence w . (Following an Isabelle convention, we formalize it equivalently as two implications.) In the formalization, the locale has more definitions than shown here and a number of lemmas. Outside LTS’s context, we can access its definitions, e.g., *pre_star* is available under the name *LTS.pre_star* and can be applied to any transition relation A and a set of states C as follows: *LTS.pre_star A C*.

```

datatype 'label op = pop | swap 'label | push 'label 'label
type_synonym ('ctr_loc, 'label) rule =
  ('ctr_loc  $\times$  'label)  $\times$  ('ctr_loc  $\times$  'label op)
type_synonym ('ctr_loc, 'label) conf = 'ctr_loc  $\times$  'label list
locale PDS = fixes  $\Delta :: ('ctr\_loc, 'label:: \text{finite})$  rule set
begin
  fun lbl where
     $\text{lbl pop} = [] \mid \text{lbl (swap } \gamma) = [\gamma] \mid \text{lbl (push } \gamma \gamma') = [\gamma, \gamma']$ 
  definition is_rule (infix  $\hookrightarrow$ ) where
     $(p, \gamma) \hookrightarrow (p', w) \iff ((p, \gamma), (p', w)) \in \Delta$ 
  inductive_set step where
     $(p, \gamma) \hookrightarrow (p', w) \longrightarrow$ 
     $((p, \gamma \# w'), (), (p', \text{lbl } w @ w')) \in \text{step}$ 
  interpretation LTS step .
end

```

end

```

datatype ('ctr_loc, 'noninit) state =
  Init 'ctr_loc | Noninit 'noninit
locale PDS_with_finals = PDS  $\Delta$ 
  for  $\Delta :: ('ctr\_loc :: \text{enum}, 'label :: \text{finite})$  rule set +
  fixes F_inits :: 'ctr_loc set and F_noninits :: 'noninit set
begin
  definition finals = Init  $\setminus$  F_inits  $\cup$  Noninit  $\setminus$  F_noninits
  definition inits =  $\{q. \exists p. q = \text{Init } p\}$ 
  definition accepts  $A (p, w) =$ 
     $(\exists q \in \text{finals}. (\text{Init } p, w, q) \in \text{LTS.trans\_star } A)$ 
  definition lang  $A = \{c. \text{accepts } A c\}$ 
end

```

Fig. 2: The types and locales for pushdown systems

III. PUSHDOWN REACHABILITY

We formalize pushdown systems (PDSs) and saturation algorithms for calculating *pre** and *post** following Schwoon [44] and *dual** following Jensen et al. [23].

Fig. 2 shows our modeling of PDSs. We use type variables to represent control locations (*'ctr_loc*) and stack labels (*'label*). We introduce types for operations (*'label op*), rules (*('ctr_loc, 'label) rule*) and configurations (*('ctr_loc, 'label) conf*). A PDS is given by the locale *PDS*, which fixes a set of rules Δ . Each PDS gives rise to an unlabeled transition relation, which we model by an LTS step with label $()$ —the only element of type *unit*. The definition is a non-recursive **inductive** definition. We use the **interpretation** command to interpret LTS with *step*. This means that *pre_star* refers to *LTS.pre_star* step in *PDS*. Likewise, *trans_star* refers to *LTS.trans_star* step and similarly for other LTS definitions. The type *('ctr_loc, 'noninit) state* represents *P*-automata states, where *'noninit* is the type variable for noninitial states. The locale *PDS_with_finals* extends *PDS* with a set of final initial states *F_inits* and final noninitial states *F_noninits*. For the rest of this section, we work within the *PDS_with_finals* locale. In this locale, a *P*-automaton is a set of transitions.

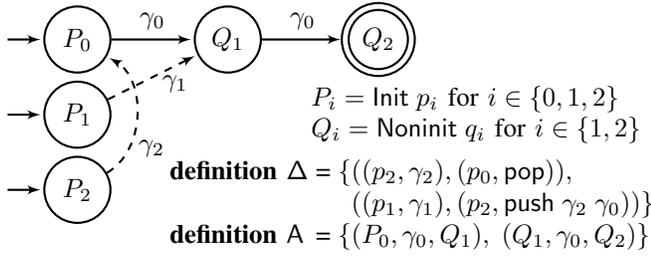


Fig. 3: Adding two transitions (dashed arrows) to a P -automaton. Initially (solid arrows) the P -automata encodes only configuration $(p_0, [\gamma_0, \gamma_0])$. After saturation, the configurations $(p_1, [\gamma_1, \gamma_0])$ and $(p_2, [\gamma_2, \gamma_0, \gamma_0])$ are also encoded.

A. Nondeterministic pre^* Saturation

Schwoon [44] presents the pre^* saturation which is a nondeterministic algorithm that given a P -automaton A returns a P -automaton whose language is $pre_star(\text{lang } A)$. The algorithm proceeds by iteratively adding transitions to A . In each step, the algorithm nondeterministically chooses a transition to add that satisfies a number of criteria. The P -automaton is saturated when no more transitions can be added. We formalize a step of the algorithm by the relation:

inductive pre_star_rule where

$$\begin{aligned}
 &(\text{Init } p, \gamma, q) \notin A \longrightarrow (p, \gamma) \hookrightarrow (p', w) \longrightarrow \\
 &(\text{Init } p', \text{lbl } w, q) \in \text{LTS.trans_star } A \longrightarrow \\
 &\text{pre_star_rule } A \ (A \cup \{(\text{Init } p, \gamma, q)\})
 \end{aligned}$$

The pre_star_rule relation relates two P -automata if the latter can be obtained from the former via one step of the algorithm. The criteria of the algorithm are expressed as the premises of the implication shown in pre_star_rule 's definition. The last two premises are taken directly from Schwoon's definition of the algorithm and the first one ensures that the transition we add into the new P -automaton is a new one. A single P -automaton can be related to different P -automata via pre_star_rule , which captures nondeterministic choice.

Consider the PDS defined by Δ in Fig. 3, and let the P -automaton A consist of the two solid transitions in the figure. Let A' be $A \cup \{(P_2, \gamma_2, P_0)\}$. Notice that $(P_2, \gamma_2, P_0) \notin A$ and $(p_2, \gamma_2) \hookrightarrow (p_0, \text{pop})$ and $(P_0, \text{lbl } \text{pop}, P_0) \in \text{LTS.trans_star } A$. From pre_star_rule 's definition then follows that $pre_star_rule A A'$. Let A'' be $A' \cup \{(P_1, \gamma_1, Q_1)\}$. From pre_star_rule 's definition it follows that $pre_star_rule A' A''$.

We formalize what it means for a P -automaton A to be saturated w.r.t a rule r , and for A' to be a saturation of A :

definition saturated $r A = (\nexists A'. r A A')$

definition saturation $r A A' = (r^{**} A A' \wedge \text{saturated } r A')$

In our example, A'' is saturated and thus formally we have saturated $pre_star_rule A''$ and saturation $pre_star_rule A A''$.

We next prove the pre^* saturation algorithm correct. Here, we focus on the proof's most interesting aspects, especially those where we had to deviate from Schwoon's pen-and-paper proof, and refer to our formalization for full details [40].

The correctness theorem states that if a transition system A' is a saturation of a transition system A then the language

of A' is indeed the pre^* closure of the language of A . Like Schwoon, we assume that the initial states are sources:

theorem pre_star_rules_correct:

assumes $\text{inits} \subseteq \text{LTS.srcs } A$

and saturation $pre_star_rule A A'$

shows $\text{lang } A' = pre_star(\text{lang } A)$

Schwoon's Lemma 3.1 is used to prove the \supseteq direction of the theorem's conclusion. He proves it by considering an arbitrary predecessor configuration (p', w) of a configuration (p, v) in A 's language. The proof proceeds by induction on the number of \Rightarrow transitions from (p', w) to (p, v) . We do not keep track of this number, but we instead prove the lemma by induction on the transitive and reflexive closure of \Rightarrow . The formalization of the proof is written in Isabelle's structured proof language Isar (not shown) and follows Schwoon's arguments. Schwoon's Lemma 3.2 is used to prove the \subseteq direction of $pre_star_rules_correct$'s conclusion. We showcase Lemma 3.2 in Schwoon's formulation, but adapted to our notation:

Lemma 3.2 If saturation $pre_star_rule A A'$ and

$(p, w, q) \in \text{LTS.trans_star } A'$ then:

- (a) $(p, w) \Rightarrow^* (p', w')$ for a configuration (p', w') such that $(p', w', q) \in A$;
- (b) moreover, if q is an initial state, then $w' = []$.

In his proof, Schwoon claims to prove (a) by an induction and then that (b) will follow immediately from a simple argument. However, reading his proof we notice that he uses (b) in the proof of (a). We resolve this by noticing that we can strengthen (b) to hold for any stack w and not just the one w' claimed to exist in (a). Our formulation of (b) looks as follows:

lemma word_into_init_empty:

assumes $(p, w, \text{Init } q) \in \text{LTS.trans_star } A$

and $\text{inits} \subseteq \text{LTS.srcs } A$

shows $w = [] \wedge p = \text{Init } q$

We prove (a) using the strengthened version of (b). Like Schwoon, we prove (a) by a nested induction. His outer induction is on the number of times the algorithm added transitions to the P -automaton. We instead prove the lemma by induction on the transitive reflexive closure of pre_star_rule . The inner induction is more challenging to formalize. Here, Schwoon considers a specific transition t which he defines as the i th transition added to P -automaton A . In the same context he considers a word w and two states, $\text{Init } p$ and q , such that $(\text{Init } p, w, q) \in \text{LTS.trans_star } A'$. He then defines j as the number of times t is used in $(\text{Init } p, w, q) \in \text{LTS.trans_star } A'$. We may argue that this number is not well-defined, because there can be several paths from $\text{Init } p$ to q consuming w , and on these paths t may not occur the same number of times. It turns out we can choose among these paths completely freely—any one of them will work, and so we just choose one arbitrarily. Formalizing this required us to define a variant of $trans_star$ that keeps track of the intermediate states.

B. Nondeterministic $post^*$ Saturation

We call states with no incoming or outgoing transitions isolated. The $post^*$ saturation algorithm requires the addition

of new noninitial states that are isolated in the automaton on which the algorithm is run. Under certain conditions the algorithm adds transitions into and out of these. Each such new state corresponds to a control location and a label. We extend the datatype of states with a new constructor `Isolated` for these:

```
datatype ('ctr_loc, 'noninit, 'label) state =
  Init 'ctr_loc | Noninit 'noninit | Isolated 'ctr_loc 'label
```

Moreover, we define `isols = {q. ∃p. q = Isolated p}`.

Steps in the $post^*$ saturation are formalized as follows:

inductive `post_star_rules` **where**

```
(p, γ) ↦ (p', pop) ⟶ (Init p', ε, q) ∉ A ⟶
(Init p, [γ], q) ∈ LTS_ε.trans_star_ε A ⟶
post_star_rules A (A ∪ {(Init p', ε, q)})
| (p, γ) ↦ (p', swap γ') ⟶ (Init p', Some γ', q) ∉ A ⟶
(Init p, [γ], q) ∈ LTS_ε.trans_star_ε A ⟶
post_star_rules A (A ∪ {(Init p', Some γ', q)})
| (p, γ) ↦ (p', push γ' γ'') ⟶
(Init p', Some γ', Isolated p' γ') ∉ A ⟶
(Init p, [γ], q) ∈ LTS_ε.trans_star_ε A ⟶
post_star_rules A (A ∪ {(Init p', Some γ', Isolated p' γ')})
| (p, γ) ↦ (p', push γ' γ'') ⟶
(Isolated p' γ', Some γ'', q) ∉ A ⟶
(Init p', Some γ', Isolated p' γ') ∈ A ⟶
(Init p, [γ], q) ∈ LTS_ε.trans_star_ε A ⟶
post_star_rules A (A ∪ {(Isolated p' γ', Some γ'', q)})
```

The relation has one rule for `pop`, one for `swap`, and two for `push`. It uses `LTS_ε.trans_star_ε`, which is similar to `LTS.trans_star` but allows ε -transitions that do not consume stack symbols. The transition $(\text{Init } p', \varepsilon, q)$ is an ε -transition and $(\text{Init } p', \text{Some } \gamma', q)$ is a γ' -labeled non- ε -transition. The function `lang_ε` returns the language of a P -automaton with ε -transitions. We prove $post^*$ saturation correct:

theorem `post_star_rules_correct`:

```
assumes saturation post_star_rules A A'
and inits ⊆ LTS.srcs A and isols ⊆ LTS.isolated A
shows lang_ε A' = post_star (lang_ε A)
```

Schwoon's definition of the $post^*$ rule has only one rule for `push` (in contrast to our two rules). In his rule, Schwoon *first* adds a transition $(\text{Init } p', \text{Some } \gamma', \text{Isolated } p' \gamma')$ and *then* adds a transition $(\text{Isolated } p' \gamma', \text{Some } \gamma'', q)$. Consider his rule here presented in his formulation but our notation:

```
If (p, γ) ↦ (p', push γ' γ'') and
(Init p, γ, q) ∈ LTS_ε.trans_star_ε A,
first add (Init p', Some γ', Isolated p' γ');
then add (Isolated p' γ', Some γ'', q).
```

We were at first surprised that he specified this *first/then* order, but his correctness proof actually relies on it. Specifically, the order is used in his proof of Lemma 3.4, which is the key to prove the \supseteq direction of `post_star_rules_correct`. We present Lemma 3.4 in Schwoon's formulation but our notation:

Lemma 3.4 If saturation `post_star_rules A A'` and $(\text{Init } p, w, q) \in \text{LTS}_\varepsilon.\text{trans_star}_\varepsilon A'$ then:

- (a) if $q \notin \text{isols}$, then $(p', w') \Rightarrow^* (p, w)$ for a configuration (p', w') such that $(\text{Init } p', w', q) \in \text{LTS}_\varepsilon.\text{trans_star}_\varepsilon A$;

- (b) if $q = \text{Isolated } p' \gamma'$, then $(p', \gamma') \Rightarrow^* (p, w)$.

Schwoon's proof is a nested induction. The outer induction is on the number of transitions $post^*$ has added. The induction step proceeds by an inner induction on the number of times the most recently added transition t was used in $(\text{Init } p, w, q') \in \text{LTS}_\varepsilon.\text{trans_star}_\varepsilon A'$. (We resolve the ambiguity of that number's meaning in a similar way as for pre^* .) The proof then proceeds by a case distinction on which of the $post^*$ saturation rules added t . Consider the case where t was added by the "first" part of the rule for `push`. In this case, t has the form $(\text{Init } p', \text{Some } \gamma', \text{Isolated } p' \gamma')$. Schwoon states that "Then since `Isolated` $p' \gamma'$ has no transitions leading into it initially, it cannot have played part in an application rule before this step, and t is the first transition leading to it. Also, there are no transitions leading away from t so far." Had Schwoon not forced the algorithm to *first* add the transition into `Isolated` $p' \gamma'$ and *then* add the one out of it, then he could not have claimed that there are no transition leading away from t . We capture this idea in the following two lemmas, stating that if t is not present, then `Isolated` $p' \gamma'$ must be a source and a sink:

lemma `post_star_rules_Isolated_source_invariant`:

```
assumes post_star_rules** A A'
and isols ⊆ LTS.isolated A
and (Init p', Some γ', Isolated p' γ') ∉ A'
shows Isolated p' γ' ∈ LTS.srcs A'
```

lemma `post_star_rules_Isolated_sink_invariant`:

```
assumes post_star_rules** A A'
and isols ⊆ LTS.isolated A
and (Init p', Some γ', Isolated p' γ') ∉ A'
shows Isolated p' γ' ∈ LTS.sinks A'
```

Formalizing Schwoon's `push` rule as a single rule in `post_star_rules` does not capture the order in which the two transition are added to the set. This is why we split the rule in two—one adding the transition into the new noninitial state and another adding the transition out of the new noninitial state. This does not yet impose the needed *first/then* order. However, we can impose the order by letting the latter rule be only applicable if the transition added by the former is indeed already in the automaton. This is possible because the transition added into state `Isolated` $p' \gamma'$ is $(\text{Init } p', \text{Some } \gamma', \text{Isolated } p' \gamma')$, and thus we can refer to the states comprising this transition in any context where `Isolated` $p' \gamma'$ is available, in particular, the second `push` rule. Note that our $post^*$ saturation algorithm is slightly more general than Schwoon's as we do not require the transition out of the new noninitial state to be added immediately after the transition into it, rather we allow this to happen at any time after.

C. Combined $dual^*$ Saturation

We now consider the recent bi-directional search approach, called $dual^*$ [23]. With $dual^*$ we can check if the configurations of one P -automaton A_2 are reachable from another P -automaton A_1 by alternating between saturating A_2 towards its pre^* closure and A_1 towards its $post^*$ closure, while simultaneously (on-the-fly) keeping track of their intersection

fun (in LTS) reach where
 reach $p \ [] = \{p\}$
 | reach $p (\gamma \# w) = (\bigcup q' \in (\bigcup (p', \gamma', q') \in \text{step}.$
 if $p' = p \wedge \gamma' = \gamma$ **then** $\{q'\}$ **else** $\{\}$). reach $q' w$)

definition (in PDS) pre_star1 $A = (\bigcup ((p, \gamma), (p', w)) \in \Delta.$
 $\bigcup q \in \text{LTS.reach } A (\text{Init } p') (\text{lbl } w). \{\text{Init } p, \gamma, q\}\}$)

definition (in PDS) pre_star_exec = the \circ while_option
 $(\lambda s. s \cup \text{pre_star1 } s \neq s) (\lambda s. s \cup \text{pre_star1 } s)$

Fig. 4: Executable pre^*

automaton. As soon as the intersection automaton becomes nonempty, we know that there is a state in A_2 that is reachable from A_1 . This is the case even if the pre^* and $post^*$ automata are not saturated. Our correctness theorem is formalized here:

theorem dual_star_correct_early_termination:

assumes $\text{inits} \subseteq \text{LTS.srcs } A_1$ **and** $\text{inits} \subseteq \text{LTS.srcs } A_2$
and $\text{isols} \subseteq \text{LTS.isolated } A_1 \cap \text{LTS.isolated } A_2$
and $\text{post_star_rules}^{**} A_1 A_1'$ **and** $\text{pre_star_rule}^{**} A_2 A_2'$
and $\text{lang_}\varepsilon\text{-inters} (\text{inters_}\varepsilon A_1' (\text{LTS_}\varepsilon\text{-of } A_2')) \neq \{\}$
shows $\exists c_1 \in \text{lang_}\varepsilon A_1. \exists c_2 \in \text{lang } A_2. c_1 \Rightarrow^* c_2$

The function $\text{LTS_}\varepsilon\text{-of}$ trivially converts a P -automaton to a P -automaton with ε -transitions. The function $\text{inters_}\varepsilon$ calculates the intersection P -automaton with ε -transitions of two P -automata with ε -transitions using a product construction. The function $\text{lang_}\varepsilon\text{-inters}$ gives the language of an intersection automaton. Since the \subseteq directions of $\text{pre_star_rule_correct}$ and $\text{post_star_rules_correct}$ do not rely on A' being saturated we prove them assuming only respectively $\text{pre_star_rule}^{**} A_2 A_2'$ and $\text{post_star_rules}^{**} A_1 A_1'$ instead of saturation $\text{pre_star_rule } A_2 A_2'$ and saturation $\text{post_star_rules } A_1 A_1'$. We use these more general lemmas to prove *dual_star_correct_early_termination*.

IV. EXECUTABLE PUSHDOWN REACHABILITY

To get an executable algorithm for pre^* , we resolve the non-determinism by defining a functional program pre_star_exec , presented in Fig. 4 (where we indicate the corresponding locale for each definition), with this characteristic property:

theorem pre_star_exec_language_correct:

assumes $\text{inits} \subseteq \text{LTS.srcs } A$
shows $\text{lang } (\text{pre_star_exec } A) = \text{pre_star } (\text{lang } A)$

The function reach is trans_star 's executable counterpart: for a state p and a word w , $\text{reach } p w$ computes the set of states reachable from p via w using step (fixed in the LTS locale). In other words, we have $q \in \text{reach } p w$ iff $(p, w, q) \in \text{trans_star}$.

The definition of pre_star_exec uses while_option , the functional while loop counterpart. Given a test predicate b , a loop body c and a loop state s , the expression $\text{while_option } b c s$ computes the optional state $\text{Some } (c (\dots (c (c s))))$ not satisfying b with the minimal number of applications of c , or None if no such state exists. Our specific loop keeps adding the results of a single step pre_star1 to the P -automaton comprising the loop state. We prove that our loop never returns None ,

definition nonempty $A P Q =$
 $(\exists p \in P. \exists q \in Q. \exists w. (p, w, q) \in \text{trans_star } A)$

definition inters $A B =$
 $\{((p_1, p_2), w, (q_1, q_2)). (p_1, w, q_1) \in A \wedge (p_2, w, q_2) \in B\}$

definition nonempty_inter $\Delta A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni} =$
 $\text{nonempty } (\text{inters } A_1 (\text{pre_star_exec } \Delta A_2))$
 $((\lambda x. (x, x)) \text{ 'inits}) (\text{finals } F_1 F_1^{ni} \times \text{finals } F_2 F_2^{ni})$

definition check $\Delta A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni} =$
 $(\text{if } \neg \text{inits} \subseteq \text{LTS.srcs } A_2 \text{ then None}$
 $\text{else Some } (\text{nonempty_inter } \Delta A_1 F_1 F_1^{ni} A_2 F_2 F_2^{ni}))$

Fig. 5: Reachability check for P -automata

i.e., it always terminates. We thus use the, defined partially as the $(\text{Some } x) = x$, in pre_star_exec to extract the resulting P -automaton. The step pre_star1 computes the set of all transitions that can be added by a single application of pre_star_rule .

Fig. 4's definitions are executable: Isabelle can interpret them as functional programs and extract Standard ML, Haskell, OCaml, or Scala code [19], but it is usually not possible to extract code for inductive predicates (such as trans_star or the transitive closure in saturation) or definitions involving quantifiers ranging over an infinite domain (as in saturated). The definition of pre_star_exec has an obvious inefficiency. In every iteration, pre_star1 is evaluated twice: once as a part of the loop body and once as a part of the test. Instead we use the following improved equation, which replaces while_option with explicit recursion, for code extraction.

lemma pre_star_exec_code[code]:

$\text{pre_star_exec } s = (\text{let } s' = \text{pre_star1 } s \text{ in}$
 $\text{if } s' \subseteq s \text{ then } s \text{ else } \text{pre_star_exec } (s \cup s'))$

With the executable algorithm for pre^* , we decide the reachability problem for P -automata using the check function shown in Fig. 5. It inputs a PDS Δ along with two P -automata represented by their transition relations (A_1 and A_2), their final initial states (F_1 and F_2) and their final noninitial states (F_1^{ni} and F_2^{ni}). The computation proceeds by intersecting (inters) the initial P -automaton with the pre^* saturation of the final P -automaton and checking the result's nonemptiness (nonempty). Fig. 5 refers to functions pre_star_exec , inits , finals , and trans_star which we introduced earlier in the context of different locales, outside of the respective locale. Therefore, these functions take additional parameters that correspond to the fixed parameters of the respective locale if they are used by the function (e.g., we write $\text{pre_star_exec } \Delta$ instead of pre_star_exec for an implicitly fixed Δ).

The definition of nonempty is not executable because of the quantification over words w . We implement, but omit here, the straightforward executable algorithm that starts with the set of initial states P and iteratively adds transitions from A until it reaches Q or saturates without reaching Q , in which case the language is empty since no state in Q is reachable from P .

Overall, check returns an optional Boolean value, where None signifies a well-formedness violation on the final

P -automaton: a non-source initial state in A_2 . If check returns Some b , then b is the answer to the reachability problem for P -automata. We formalize this characterization of check by the following two theorems (phrased outside of locales).

theorem *check_None*:

$$\text{check } \Delta \ A_1 \ F_1 \ F_1^{\text{ni}} \ A_2 \ F_2 \ F_2^{\text{ni}} = \text{None} \longleftrightarrow \\ \neg \text{inits} \subseteq \text{LTS.srscs } A_2$$

theorem *check_Some*:

$$\text{check } \Delta \ A_1 \ F_1 \ F_1^{\text{ni}} \ A_2 \ F_2 \ F_2^{\text{ni}} = \text{Some } b \longleftrightarrow \\ (\text{inits} \subseteq \text{LTS.srscs } A_2 \wedge (b \longleftrightarrow \\ (\exists p \ w \ p' \ w'. \text{step_starp } \Delta \ (p, w) \ (p', w') \wedge \\ (p, w) \in \text{lang } A_1 \ F_1 \ F_1^{\text{ni}} \wedge (p', w') \in \text{lang } A_2 \ F_2 \ F_2^{\text{ni}})))$$

V. DIFFERENTIAL TESTING

Differential testing [14], [18], [34] is a technique for finding implementation errors by executing different algorithms solving the same problem on a set of test cases and comparing the outputs. Differential testing has been effective for finding errors in a wide range of domains, from network certificate validation [47] to JVM implementations [8]. Yet, even different algorithms do not necessarily fail independently, e.g., when built from the same specification [27] or when sharing potentially faulty components, e.g., input parsers or preprocessing. To reduce the danger of missing such errors, we suggest to incorporate a formally verified implementation in differential testing. Moreover, in case of a discrepancy the verified oracle reliably tells us which of the unverified implementations is wrong.

A. Differential Testing of Pushdown Reachability

Our executable formalization of pushdown reachability allows us to perform differential testing on unverified tools for the same problem. A test case for pushdown reachability consists of a PDS with rules Δ and two P -automata \mathcal{A}_1 and \mathcal{A}_2 representing the initial and final configurations of interest. The answer to the test case is whether there exist $c \in L(\mathcal{A}_1)$ and $c' \in L(\mathcal{A}_2)$ such that $c \Rightarrow^* c'$ using the rules Δ .

To execute the formalization on a given test case, we generate an Isabelle theory file, which first defines the control locations, labels, and automata states as finite subsets of the natural numbers (their sizes depending on the specific test case), and then includes for the pushdown rules Δ and the two P -automata, each represented by its transitions A_i along with the accepting (initial and noninitial) states F_i and F_i^{ni} for $i \in \{1, 2\}$. Fig. 3 shows a specific example of Δ and A definitions.

We generate a lemma that uses our check function, where the expected result Some True or Some False is inserted depending on the answer produced by an unverified tool under test (invoked before generating the theory on the same inputs):

lemma $\text{check } \Delta \ A_1 \ F_1 \ F_1^{\text{ni}} \ A_2 \ F_2 \ F_2^{\text{ni}} = \text{Some True}$ **by** eval

The eval proof method extracts Standard ML code for check and other constants in the lemma and executes the lemma statement as an expression. It succeeds iff the lemma evaluates to True. We call a test case a counter-example, if the proof method fails. One could also run the extracted code outside

Input: Reachability tools *tool* and *oracle*, PDS (P, Γ, Δ) , P -automata $\mathcal{A}_i = (P \cup N_i, \rightarrow_i, P, F_i)$ for $i \in \{1, 2\}$.

Output: Minimal counter-example (failing testcase)

- 1: $c \leftarrow \Delta \cup (\{1\} \times (\rightarrow_1 \cup F_1)) \cup (\{2\} \times (\rightarrow_2 \cup F_2))$
▷ Convert to a set of features
- 2: **return** DD(c , 2) ▷ returned set of features can be converted to PDS and P -automata as on lines 10-11
- 3: **function** DD(c , n) ▷ c is a test case, n is granularity
- 4: let $c_1 \uplus \dots \uplus c_n = c$, all c_i as evenly sized as possible
- 5: **if** $\exists i. \text{BAD}(c_i)$ **return** DD(c_i , 2)
- 6: **else if** $\exists i. \text{BAD}(c \setminus c_i)$ **return** DD($c \setminus c_i$, $\max(n-1, 2)$)
- 7: **else if** $n < |c|$ **return** DD(c , $\min(2n, |c|)$)
- 8: **else return** c
- 9: **function** BAD(c) ▷ c is a test case
- 10: let $\Delta' = c \cap \Delta$ ▷ extract PDS rules and P -automata
- 11: for $i \in \{1, 2\}$ let $\mathcal{A}'_i = (P \cup N_i, \rightarrow'_i, P, F'_i)$ where
 $\rightarrow'_i = \{t \in \rightarrow_i \mid (i, t) \in c\}$ and $F'_i = \{q \in F_i \mid (i, q) \in c\}$
- 12: with both tools check if \mathcal{A}'_1 reaches \mathcal{A}'_2 via (P, Γ, Δ')
- 13: **return false** if *tool* and *oracle* agree, else *true*

Algorithm 1: Specialization of delta-debugging [51] to PDS.

Isabelle, but our setup allows us to generate the inputs to check on the formalization level instead of that of the extracted code.

To efficiently check a large number of test cases, we batch multiple definitions and lemmas into one theory file, thus reducing the overhead of starting Isabelle. We run Isabelle from the command line and check the output log for any failing eval proofs, which correspond to failing test cases.

B. Automatic Counter-Example Minimization

If differential testing finds a failing test case, we use delta-debugging [51] to automatically reduce it to a minimal failing test case to help the subsequent debugging process. We use the minimizing delta debugging algorithm [51] that sees a test case as a set of features, and works by systematically testing different subsets until a minimal failing test case is found.

We use delta debugging on any discovered counter-example and fix the set of features to contain: (i) each pushdown rule, (ii) each transition in either of the P -automata, and (iii) each final state in a P -automaton (as opposed to it not being final).

States and labels are identified by unique names, and the initial P -automata states are exactly the states mentioned in any pushdown rule in the feature set. We specialize the general delta debugging algorithm to pushdown systems as shown in Algorithm 1. The algorithm first creates the set of features and calls the recursive function DD with this set of features and the granularity 2. The function then splits the set of features into a number of equally sized subsets (according to the granularity) and checks if any of these subsets or their complements still fail. If yes, then the function tries to recursively reduce the set of features further, otherwise it will increase the granularity and try again. The function BAD converts the set

$\Delta = \{$	$(p_0, B) \leftrightarrow (p_2, \text{push } D B),$	$(p_1, B) \leftrightarrow (p_1, \text{swap } C),$	$(p_2, C) \leftrightarrow (p_0, \text{push } C C),$	$(p_2, B) \leftrightarrow (p_2, \text{swap } E),$	$(p_3, E) \leftrightarrow (p_2, \text{swap } C),$
$(p_0, D) \leftrightarrow (p_0, \text{swap } A),$	$(p_0, C) \leftrightarrow (p_2, \text{swap } D),$	$(p_1, E) \leftrightarrow (p_1, \text{swap } C),$	$(p_2, D) \leftrightarrow$	$(p_2, E) \leftrightarrow (p_3, \text{push } A E),$	$(p_3, B) \leftrightarrow (p_2, \text{push } D B),$
$(p_0, E) \leftrightarrow (p_0, \text{push } B E),$	$(p_0, E) \leftrightarrow (p_2, \text{swap } E),$	$(p_1, A) \leftrightarrow (p_2, \text{swap } A),$	$(p_0, \text{push } B D),$	$(p_2, B) \leftrightarrow (p_3, \text{push } C B),$	$(p_3, E) \leftrightarrow (p_3, \text{swap } A),$
$(p_0, D) \leftrightarrow$	$(p_0, E) \leftrightarrow (p_3, \text{push } B E),$	$(p_1, D) \leftrightarrow (p_2, \text{swap } D),$	$(p_2, C) \leftrightarrow (p_1, \text{push } C C),$	$(p_3, D) \leftrightarrow$	$(p_3, A) \leftrightarrow (p_3, \text{push } C A),$
$(p_0, \text{push } D D),$	$(p_0, C) \leftrightarrow (p_3, \text{swap } E),$	$(p_1, C) \leftrightarrow (p_2, \text{swap } E),$	$(p_2, A) \leftrightarrow (p_1, \text{push } B A),$	$(p_0, \text{push } B D),$	$(p_3, E) \leftrightarrow (p_3, \text{swap } D),$
$(p_0, D) \leftrightarrow (p_0, \text{pop}),$	$(p_1, B) \leftrightarrow (p_0, \text{swap } C),$	$(p_1, C) \leftrightarrow (p_3, \text{swap } D),$	$(p_2, A) \leftrightarrow (p_2, \text{push } A A),$	$(p_3, C) \leftrightarrow (p_0, \text{push } E C),$	$(p_3, C) \leftrightarrow (p_3, \text{pop})\}$
$(p_0, D) \leftrightarrow (p_1, \text{swap } A),$	$(p_1, D) \leftrightarrow (p_0, \text{swap } C),$	$(p_1, D) \leftrightarrow (p_3, \text{pop}),$	$(p_2, C) \leftrightarrow (p_2, \text{swap } A),$	$(p_3, C) \leftrightarrow (p_0, \text{swap } E),$	
$(p_0, A) \leftrightarrow (p_1, \text{push } C A),$	$(p_1, C) \leftrightarrow (p_0, \text{swap } B),$	$(p_2, B) \leftrightarrow (p_0, \text{push } A B),$	$(p_2, E) \leftrightarrow (p_2, \text{swap } A),$	$(p_3, C) \leftrightarrow (p_1, \text{push } A C),$	
$(p_0, E) \leftrightarrow (p_2, \text{push } A E),$	$(p_1, C) \leftrightarrow (p_0, \text{swap } E),$	$(p_2, A) \leftrightarrow (p_0, \text{push } C A),$	$(p_2, A) \leftrightarrow (p_2, \text{push } B A),$	$(p_3, B) \leftrightarrow (p_1, \text{pop}),$	

$A_1 =$	$\{(\text{Init } p_0, B, \text{Noninit } q_1), (\text{Init } p_0, D, \text{Noninit } q_0),$	$\Delta = \{(p_0, D) \leftrightarrow (p_0, \text{pop})\}$
	$(\text{Init } p_2, B, \text{Noninit } q_0), (\text{Init } p_3, A, \text{Noninit } q_2),$	
	$(\text{Noninit } q_0, D, \text{Noninit } q_1), (\text{Noninit } q_2, C, \text{Noninit } q_0)\}$	$A_1 = \{(\text{Init } p_0, D, \text{Noninit } q_0), (\text{Noninit } q_0, D, \text{Noninit } q_1)\}$
$F_1 = \{ \}$	$F_1^{\text{ni}} = \{q_1\}$	$F_1 = \{ \}$ $F_1^{\text{ni}} = \{q_1\}$
$A_2 =$	$\{(\text{Init } p_2, A, \text{Noninit } q_0), (\text{Init } p_2, B, \text{Noninit } q_0)\}$	$A_2 = \{ \}$
$F_2 = \{p_0, p_2\}$	$F_2^{\text{ni}} = \{ \}$	$F_2 = \{p_0\}$ $F_2^{\text{ni}} = \{ \}$

Fig. 6: Original and minimized (bottom right) counter-example

of features into a reduced pushdown system and two reduced P -automata and checks if the given tool implementation is still inconsistent with the oracle. We note that minimal failing counter-examples are only locally minimal and not necessarily unique. Yet, minimization is effective and necessary. Fig. 6 shows a real bug example we discovered by random differential testing in the PDAAAL library for pushdown reachability [23] and its minimization by Algorithm 1.

VI. CASE STUDY: ANALYSIS OF PDAAAL

We apply differential testing with automatic counter-example minimization to PDAAAL [42], a recent C++ implementation of pushdown reachability checking, which appears to be the currently most efficient library for pushdown reachability [23]. PDAAAL implements $post^*$, pre^* and $dual^*$ [23].

These three different algorithms can be used in classical differential testing without a verified oracle, but given the large amount of shared code this is bound to miss some errors. And without a verified oracle, manual effort is needed to determine which implementation is faulty in case of discrepancies. This motivates using our verified reachability check via pre^* , and we compare the output of each unverified algorithm to the output of our trustworthy oracle on a large number of test cases.

A. Methodology of Test Case Generation

We structure our test case generation in three phases.

In phase one, we use real-world tests generated from the domain of network verification, which PDAAAL was originally built for as a backend [22]. We generate pushdown reachability problems from realistic network verification use-cases on (up to) 100 random reachability queries on each of the 260 different networks derived from the Internet Topology Zoo [28] giving a total of 25 512 test cases.

In phase two, we randomly generate valid pushdown systems and P -automata. We generate 15 000 cases of varying sizes with 4 control locations, 5 labels, up to 200 pushdown rules, and up to 13 automata transitions. Our generator writes all ingredients (pushdown system and P -automata) to a JSON

file, which is then translated to the Isabelle definitions and correctness lemmas that incorporate the unverified answers.

Finally, in phase three, we exhaustively enumerate the set of all test cases up to a certain (small) size. For the pushdown systems $|P| = |\Gamma| = 2$ and $|\Delta| \leq 2$, and for P -automata $|N_1| = 2$, $|N_2| = 1$ and $|\rightarrow| \leq 2$. We remove symmetric cases, where swapping state names or labels gives an identical case. In total, this yields close to 27 million combinations of pushdown systems and P -automata. For the exhaustive tests, we output both JSON files and Isabelle definitions directly from the test case generator. A bash script stitches together the Isabelle definitions into a single theory file with a batch of test cases to benefit from Isabelle’s parallel processing of proofs.

B. Results

The real-world test cases showed no discrepancies between the verified oracle and PDAAAL. This indicates that PDAAAL has already been thoroughly tested on this type of problem instances. Isabelle ran out of memory in 30 of the 25 512 test cases. The average CPU time (on AMD EPYC 7642 processors at 1.5 GHz) per test case was 35 seconds for Isabelle, while PDAAAL used less than 0.02 seconds on most cases.

Phase two, however, resulted in 1 334 discrepancies. By applying our counter-example minimization, we noted that all these cases had a common trait: the P -automaton \mathcal{A}_2 accepted the empty word. This helped us find the first implementation error in the implementation of the on-the-fly automata intersection when using $post^*$. The $post^*$ algorithm can introduce ε -transitions, which were not handled correctly by the intersection implementation. In most cases, this does not matter, as for any ε -transition followed by a normal transition the $post^*$ algorithm adds a direct transition at some later point. However, in the case of an empty stack being accepted by \mathcal{A}_2 , this does not happen, which causes the unverified algorithm to return the wrong answer False. We resolved the error and re-ran the generated tests. After that only one discrepancy remained.

This second error was found in the implementation of pre^* . The minimized counter-example helped us find the source

10: **function** ADDTRANSITION($q_i \xrightarrow{\gamma} q'_i$) ▷ with $i \in \{1, 2\}$
11: **add** $q_i \xrightarrow{\gamma} q'_i$ to \mathcal{A}_i
12: **for all** $q_{3-i}, q'_{3-i} \in Q_{3-i}$ s.t. $(q_1, q_2) \in R$ and $q_{3-i} \xrightarrow{\gamma} q'_{3-i}$ **do**
13: **add** $(q_1, q_2) \xrightarrow{\gamma} (q'_1, q'_2)$ to \mathcal{A}_\cap
14: ADDSTATE(q'_1, q'_2)

(a) Snippet of (correct) intersection pseudocode by Jensen et al. [23]

```

@@ -119,10 +119,10 @@ namespace pdaaal {
119 119         if (res.second) { // New edge is not already in edges (rel U workset).
120 120             _workset.emplace(from, label, to);
121 121             if (trace != nullptr) { // Don't add existing edges
122 122 +             _automaton.add_edge(from, to, label, trace_ptr_from<W>(trace));
123 123                 if constexpr (ET) {
124 124                     _found = _found || _early_termination(from, label, to, trace_ptr_from<W>(trace));
125 125 -             _automaton.add_edge(from, to, label, trace_ptr_from<W>(trace));
126 126         }
127 127     }
128 128 };

```

(b) PDAAAL's C++ code showing the resolution of the second error

Fig. 7: Discovered second implementation error and its correct pseudocode

of the implementation error: the set of automata transitions was updated only after calling the function that performs the nonemptiness check of the intersection automaton, but it should have been updated before that call. We argue that this error is subtle, as it only causes a single failure out of 15 000 randomly generated test cases. Fig. 7a shows the correct pseudocode by Jensen et al. [23]. Fig. 7b shows PDAAAL's corresponding C++ code and the change resolving the error, where the line that needed to be moved corresponds to the pseudocode's Line 11.

For both errors, the affected test cases resulted in a correct answer for at least one of the other search strategies in PDAAAL. This is not the case for the last error, which is found in code shared by all three methods, and where PDAAAL's algorithms disagree only with Isabelle. This error is caused by a mismatch between the assumptions of the parser that builds the pushdown system and the data structure that stores the pushdown rules. The parser assumes that it can incrementally add rules to the data structure without knowing all labels in advance, but the data structure assumes to know all labels from the start to implement a memory optimization that replaces a rule that applies to all labels by a wildcard.

For the first two test phases, the program that generated Isabelle definitions also depended on this parser, so the bug was not discovered until the third phase, which has a different setup. After the three bugs were fixed, all test cases pass.

VII. CONCLUSION

We presented a methodology that increases the reliability of tools and libraries for pushdown reachability analysis. To this

end, we formalized and proved in Isabelle/HOL the correctness of the essential saturation algorithms used in such tools. We extracted an executable program from our formalization and used it as a trustworthy oracle for differential testing. Putting the modern pushdown analysis library PDAAAL on the testbed, we discovered a number of implementation errors in its code, even though the library performed flawlessly in its application domain. Using our automatic counter-example minimization based on delta-debugging, we were able to identify the sources of these errors and suggested fixes to PDAAAL's implementation that now passes all the differential tests.

This process significantly increased PDAAAL's reliability and shows that with a moderate effort, the combination of proof assistants with code generation, differential testing, and delta-debugging is highly fruitful. The execution of all tests in the three phases took 303 CPU days. We executed the tests on a compute cluster with 1536 CPU cores. The formalization work took about two person-months for experienced formalizers, creating about 4 400 nonempty lines of Isabelle definition and proofs. An additional half person-month of work was needed to implement the differential testing and counter-example minimization, set up the tests, and localize and resolve the discovered errors. This one-time effort will also benefit the future development of PDAAAL.

Too often, the race for better performance can lead to subtle implementation errors. Our methodology shows how formally verified algorithms that were not tuned for performance can be used to improve the quality of tuned but unverified algorithms.

REFERENCES

- [1] Ballarin, C.: Locales: A module system for mathematical theories. *J. Autom. Reason.* **52**(2), 123–153 (2014). <https://doi.org/10.1007/s10817-013-9284-7>
- [2] Basin, D.A., Dardinier, T., Heimes, L., Krstic, S., Raszyc, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) *IJCAR 2020*. LNCS, vol. 12166, pp. 432–453. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_25
- [3] Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLS 2009*. LNCS, vol. 5674, pp. 147–163. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_12
- [4] Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) *CONCUR 1997*. LNCS, vol. 1243, pp. 135–150. Springer (1997). https://doi.org/10.1007/3-540-63141-0_10
- [5] Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) *CONCUR 2005*. LNCS, vol. 3653, pp. 473–487. Springer (2005). https://doi.org/10.1007/11539452_36
- [6] Braibant, T., Pous, D.: Deciding Kleene algebras in Coq. *Log. Methods Comput. Sci.* **8**(1) (2012). [https://doi.org/10.2168/LMCS-8\(1:16\)2012](https://doi.org/10.2168/LMCS-8(1:16)2012)
- [7] Büchi, J.R.: Regular canonical systems. *Archiv für mathematische Logik und Grundlagenforschung* **6**(3–4), 91–111 (1964). <https://doi.org/10.1007/BF01969548>
- [8] Chen, Y., Su, T., Su, Z.: Deep differential testing of JVM implementations. In: Atlee, J.M., Bultan, T., Whittle, J. (eds.) *ICSE 2019*, pp. 1257–1268. IEEE / ACM (2019). <https://doi.org/10.1109/ICSE.2019.00127>
- [9] Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 449–461. Springer (2005). https://doi.org/10.1007/11513988_45
- [10] van Duijn, I., Jensen, P., Jensen, J., Krøgh, T., Madsen, J., Schmid, S., Srba, J., Thorgersen, M.: Automata-theoretic approach to verification of MPLS networks under link failures. *IEEE/ACM Transactions on Networking* pp. 1–16 (2021). <https://doi.org/10.1109/TNET.2021.3126572>
- [11] Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural data-flow analysis. In: Thomas, W. (ed.) *FoSSaCS 1999*. LNCS, vol. 1578, pp. 14–30. Springer (1999). https://doi.org/10.1007/3-540-49019-1_2
- [12] Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) *CAV 2013*. LNCS, vol. 8044, pp. 463–478. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_31
- [13] Esparza, J., Schwoon, S.: A bdd-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) *CAV 2001*. LNCS, vol. 2102, pp. 324–336. Springer (2001). https://doi.org/10.1007/3-540-44585-4_30
- [14] Evans, R.B., Savoia, A.: Differential testing: a new approach to change detection. In: Crnkovic, I., Bertolino, A. (eds.) *ESEC-FSE 2007*, pp. 549–552. ACM (2007). <https://doi.org/10.1145/1287624.1287707>
- [15] Fonseca, P., Zhang, K., Wang, X., Krishnamurthy, A.: An empirical study on the correctness of formally verified distributed systems. In: Alonso, G., Bianchini, R., Vukolic, M. (eds.) *EuroSys 2017*, pp. 328–343. ACM (2017). <https://doi.org/10.1145/3064176.3064183>
- [16] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) *CAV 2014*. LNCS, vol. 8559, pp. 69–87. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_5
- [17] Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Bodik, R., Majumdar, R. (eds.) *POPL 2016*, pp. 499–512. ACM (2016). <https://doi.org/10.1145/2837614.2837664>
- [18] Groce, A., Holzmann, G.J., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: *ICSE 2007*, pp. 621–631. IEEE Computer Society (2007). <https://doi.org/10.1109/ICSE.2007.68>
- [19] Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) *FLOPS 2010*. LNCS, vol. 6009, pp. 103–117. Springer (2010). https://doi.org/10.1007/978-3-642-12251-4_9
- [20] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM* **60**(7), 83–92 (2017). <https://doi.org/10.1145/3068608>
- [21] Jensen, J.S., Krøgh, T.B., Madsen, J.S., Schmid, S., Srba, J., Thorgersen, M.T.: P-Rex: fast verification of MPLS networks with multiple link failures. In: Dimitropoulos, X.A., Dainotti, A., Vanbever, L., Benson, T. (eds.) *CoNEXT 2018*, pp. 217–227. ACM (2018). <https://doi.org/10.1145/3281411.3281432>
- [22] Jensen, P.G., Kristiansen, D., Schmid, S., Schou, M.K., Schrenk, B.C., Srba, J.: AalWiNes: a fast and quantitative what-if analysis tool for MPLS networks. In: Han, D., Feldmann, A. (eds.) *CoNEXT 2020*, pp. 474–481. ACM (2020). <https://doi.org/10.1145/3386367.3431308>
- [23] Jensen, P.G., Schmid, S., Schou, M.K., Srba, J., Vanerio, J., van Duijn, I.: Faster pushdown reachability analysis with applications in network verification. In: Hou, Z., Ganesh, V. (eds.) *ATVA 2021*. LNCS, vol. 12971, pp. 170–186. Springer (2021). https://doi.org/10.1007/978-3-030-88885-5_12
- [24] Jiang, D., Li, W.: The verification of conversion algorithms between finite automata. *Sci. China Inf. Sci.* **61**(2), 028101:1–028101:3 (2018). <https://doi.org/10.1007/s11432-017-9155-x>
- [25] Kidd, N., Lal, A., Repts, T.: Wali: The weighted automaton library (2007). <https://research.cs.wisc.edu/wpis/wpds/wali/>
- [26] Kincaid, Z., Breck, J., Boroujeni, A.F., Repts, T.W.: Compositional recurrence analysis revisited. In: Cohen, A., Vechev, M.T. (eds.) *PLDI 2017*, pp. 248–262. ACM (2017). <https://doi.org/10.1145/3062341.3062373>
- [27] Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.* **12**(1), 96–109 (1986). <https://doi.org/10.1109/TSE.1986.6312924>
- [28] Knight, S., Nguyen, H., Falkner, N., Bowden, R., Roughan, M.: The internet topology Zoo. *IEEE Journal on Selected Areas in Comm.* **29**(9), 1765–1775 (2011)
- [29] Lammich, P.: Formalization of dynamic pushdown networks in Isabelle/HOL (2009). <https://www21.in.tum.de/~lammich/isabelle/dpn-document.pdf>
- [30] Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Bouajjani, A., Maler, O. (eds.) *CAV 2009*. LNCS, vol. 5643, pp. 525–539. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_39
- [31] Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft’s algorithm. In: Beringer, L., Felty, A.P. (eds.) *ITP 2012*. LNCS, vol. 7406, pp. 166–182. Springer (2012). https://doi.org/10.1007/978-3-642-32347-8_12
- [32] Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* **52**(7), 107–115 (2009). <https://doi.org/10.1145/1538788.1538814>
- [33] Lesani, M., Bell, C.J., Chipala, A.: Chapar: certified causally consistent distributed key-value stores. In: Bodik, R., Majumdar, R. (eds.) *POPL 2016*, pp. 357–370. ACM (2016). <https://doi.org/10.1145/2837614.2837622>
- [34] McKeeman, W.M.: Differential testing for software. *Digit. Tech. J.* **10**(1), 100–107 (1998). <http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf>
- [35] Minamide, Y.: Verified decision procedures on context-free grammars. In: Schneider, K., Brandt, J. (eds.) *TPHOLS 2007*. LNCS, vol. 4732, pp. 173–188. Springer (2007). https://doi.org/10.1007/978-3-540-74591-4_14
- [36] Nipkow, T., Klein, G.: *Concrete Semantics - With Isabelle/HOL*. Springer (2014). <https://doi.org/10.1007/978-3-319-10542-0>
- [37] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). <https://doi.org/10.1007/3-540-45949-9>
- [38] Ramos, M.V.M., Almeida, J.C.B., Moreira, N., de Queiroz, R.J.G.B.: Formalization of the pumping lemma for context-free languages. *J. Formaliz. Reason.* **9**(2), 53–68 (2016). <https://doi.org/10.6092/issn.1972-5787/5595>
- [39] Schlichtkrull, A., Blanchette, J.C., Traytel, D.: A verified prover based on ordered resolution. In: Mahboubi, A., Myreen, M.O. (eds.) *CPP 2019*, pp. 152–165. ACM (2019). <https://doi.org/10.1145/3293880.3294100>
- [40] Schlichtkrull, A., Schou, M.K., Srba, J., Traytel, D.: Repeatability package for "Differential testing of pushdown reachability with a formally verified oracle" (2022). <https://doi.org/10.5281/zenodo.6952978>
- [41] Schneider, J., Basin, D.A., Krstic, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) *RV 2019*. LNCS, vol. 11757, pp. 310–328. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_18

- [42] Schou, M.K., Jensen, P.G., Kristiansen, D., Schrenk, B.C.: PDAAAL. GitHub (2021), <https://github.com/DEIS-Tools/PDAAAL>
- [43] Schubert, P.D., Hermann, B., Bodden, E.: PhASAR: An inter-procedural static analysis framework for C/C++. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 393–410. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_22
- [44] Schwoon, S.: Model checking pushdown systems. Ph.D. thesis, Technical University Munich, Germany (2002), <https://d-nb.info/96638976X/34>
- [45] Schwoon, S.: Moped. In: <http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/> (2002)
- [46] Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A Java bytecode checker based on Moped. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 541–545. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_35
- [47] Tian, C., Chen, C., Duan, Z., Zhao, L.: Differential testing of certificate validation in SSL/TLS implementations: An RFC-guided approach. *ACM Trans. Softw. Eng. Methodol.* **28**(4), 24:1–24:37 (2019). <https://doi.org/10.1145/3355048>
- [48] Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Grove, D., Blackburn, S.M. (eds.) PLDI 2015. pp. 357–368. ACM (2015). <https://doi.org/10.1145/2737924.2737958>
- [49] Wimmer, S.: Munta: A verified model checker for timed automata. In: André, É., Stoelinga, M. (eds.) FORMATS 2019. LNCS, vol. 11750, pp. 236–243. Springer (2019). https://doi.org/10.1007/978-3-030-29662-9_14
- [50] Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Hall, M.W., Padua, D.A. (eds.) PLDI 2011. pp. 283–294. ACM (2011). <https://doi.org/10.1145/1993498.1993532>
- [51] Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.* **28**(2), 183–200 (2002). <https://doi.org/10.1109/32.988498>

TRICERA: Verifying C Programs Using the Theory of Heaps

Zafer Esen 

Uppsala University, Uppsala, Sweden
zafer.esen@it.uu.se

Philipp Rümmer 

University of Regensburg, Regensburg, Germany
Uppsala University, Uppsala, Sweden
philipp.ruemmer@it.uu.se

Abstract—TRICERA is an automated, open-source verification tool for C programs based on the concept of Constrained Horn Clauses (CHCs). In order to handle programs operating on heap, TRICERA applies a novel theory of heaps, which enables the tool to hand off most of the required heap reasoning directly to the underlying CHC solver. This leads to a cleaner interface between the language-specific verification front-end and the language-independent CHC back-end, and enables verification tools for different programming languages to share a common heap back-end. The paper introduces TRICERA, gives an overview of the theory of heaps, and presents preliminary experimental results using SV-COMP benchmarks.

I. INTRODUCTION

This paper presents TRICERA, an automated open-source verification tool for C programs. TRICERA accepts programs in a subset of the C11 standard [1] with the purpose of checking whether explicit and implicit safety assertions in a program are valid. The tool has been developed mainly with applications in the embedded systems area in mind: restrictions in the supported language features are aligned with the recommendations made in the MISRA C coding guidelines [2]. TRICERA works by translating C programs to sets of Constrained Horn Clauses (CHCs), which are then processed and solved by the CHC solvers ELДАРICA [33] or SPACER [37], thus either proving that assertions can never fail, or computing counterexample traces leading to an assertion violation.

TRICERA is a model checker for C programs, but includes a plethora of additional features that go beyond C11, such as processing specifications in the ACSL language [6], modelling concurrent and parameterised systems, and augmenting programs with timing constraints. A distinguishing feature of TRICERA is the handling of heap data-structures, which are among the most challenging aspects in the verification of imperative programs. Existing verification tools based on CHCs tend to handle heap either using the theory of arrays (e.g., as done by SEAHORN [30]), or apply bespoke encodings of heap data using refinement types [28], invariants (JAYHORN [35]) or prophecies (RUSTHORN [43]). As the heap encoder is often one of the most complex components of a CHC-based verification tool, this implies repeated implementation effort when designing verification tools for different programming languages, and migrating a tool to a different style of heap encoding is an extremely complex task.

We propose a departure from this conventional architecture of CHC-based verification tools, instead using a language-independent *theory of heaps* [24] augmenting the interface between verification tools and CHC solvers. The theory of heaps is designed to cover the features of many existing programming languages; it is deliberately kept simple, so that it can be integrated easily in verifiers; and it is kept high-level, so that CHC solvers are able to implement a wide range of methods for solving problems involving heap, including the aforementioned encodings through arrays and invariants. The resulting architecture is shown in Figure 1.

TRICERA is the first verification tool that produces CHCs modulo the theory of heaps. At the point of writing this paper, in addition a project is underway to convert the Java verification tool JAYHORN [35] to use the theory. The development of effective solvers for CHCs modulo heaps is an ongoing effort as well; currently the CHC solver ELДАРICA provides direct support for CHCs modulo heaps by integrating a native decision and interpolation procedure for the theory of heaps [23]. In addition a tool is available for translating CHCs with heaps to CHCs with algebraic data-types (ADTs) and arrays. A more detailed description of the theory of heaps is available as a technical report [24].

TRICERA is developed at Uppsala University and the University of Regensburg. It is open source¹ and distributed under a 3-Clause BSD license. A web-interface to try it online is available².

The contributions of this paper are (i) a presentation of the verification tool TRICERA, including an overview of its features, the verification approach, and architecture; (ii) a definition of a high-level encoding of heap data using the theory of heaps; (iii) an experimental evaluation of TRICERA, on C benchmarks taken from SV-COMP, with and without heap.

II. TRICERA FEATURES

A. Input Language

We start with an overview of the features and languages supported by TRICERA. As its main input language TRICERA can handle a large subset of C11 [1], extended with additional features that are useful for verification purposes. An overview

¹<https://github.com/uuverifiers/tricera>

²<http://logicrunch.it.uu.se:4096/~zafer/tricera/>

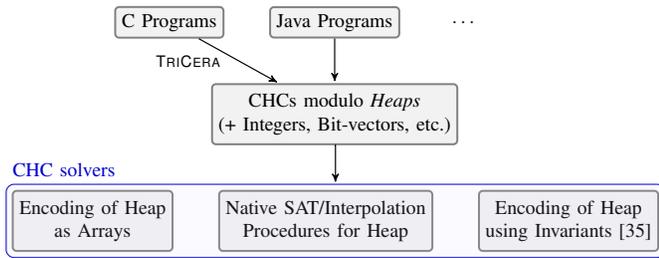


Fig. 1: Program verification using the theory of heaps.

of the currently supported and unsupported types, operations and constructs is given in Table I.

The initially supported subset of C11 is selected as to provide a strong foundation that can be easily extended. Our choice of language features to support is mainly influenced by safety-critical programs from the embedded systems area, and largely aligned with the recommendations made by the MISRA C [2].

TRICERA supports most of the standard C types with the exception of floating-point numbers, function pointers and strings. Integer types can be treated either as mathematical integers or as bit-vectors, in the latter case modelling the standard wrap-around semantics. TRICERA has full support for the C operators and statements, including several extensions discussed later in this section. TRICERA can also handle a (small) part of the C library, in particular functions for memory allocation. Heap data, pointers, and arrays are encoded through the theory of heaps, which we discuss in more detail in Section V.

The partial support in Table I for arrays refers to the following restrictions: (i) the type of array cells must be specified during allocation using `malloc`; (ii) pointers used to index arrays (either through brackets or pointer arithmetic) must be declared as arrays when they are first declared. For instance, `int *a` cannot be used to index an array, but `int a[]` can. Pointers to array cells are allowed: for instance `int *b = &a[i]` is allowed where `a` is an `int` array, but `b` cannot later be indexed as an array. Since arrays are a recent addition to TRICERA, these are restrictions of the current TriCera C front-end and not a theoretical limitation of the

TABLE I: Supported subset of the C language (not exhaustive). ✓ represents fully or almost-fully supported, † represents partially supported, and ✗ represents unsupported features.

Types	✓integers (mathematical, machine arithmetic), ✓structs, ✓enums, ✓heap pointers, †arrays, †stack pointers, ✗floating point, ✗strings, ✗function pointers,
Expressions	✓(postfix, unary, logical, bitwise, arithmetic, cast operators)
Statements and Blocks	✓(compound, expression, selection, iteration statements), ✓(atomic, within and thread blocks (non-standard C))
Other	✓(assert and assume statements), ✓(malloc, calloc, and free) ✓threads, ✓communicating timed systems, ✓function contract and loop invariant inference, †ACSL parser (only for function contracts)

theory of heaps.

TRICERA has limited support for pointers to stack variables. Such pointers are statically associated with the variables they point to on the stack. This imposes some restrictions on such pointers: they cannot be mixed and matched with pointers to the heap, and they cannot be reassigned. The restrictions result in easier to solve encodings.

The following paragraphs survey some of the additional features beyond C11.

B. Supported Code Annotations

In line with other model checkers, TRICERA uses `assert` and `assume` statements for explicitly specifying properties, which have their usual semantics as given by Flanagan and Saxe [27]. TRICERA in addition automatically adds several implicit properties:

- all pointer de-references are checked for type safety,
- array accesses are checked to be within array bounds,
- (optionally) memory leaks are detected by ensuring all allocated memory on the heap is freed at program exit.

Checking pointer de-references for type safety also implies memory safety, because TRICERA encodes unallocated locations using a special type. More information is provided in Section V.

Given a program with an entry function (default `main`), TRICERA will attempt to prove that none of the explicit and implicit properties can be violated. When TRICERA reports that an assertion is reachable, a counterexample trace is provided for debugging purposes.

TRICERA supports the declaration of non-deterministically initialised (local or global) variables (with program type \mathbb{T}) using the notation $\mathbb{T} \times = _$.

Function calls in a program are handled, by default, through inlining; by annotating a function with the comment `/*@ contract @*/`, TRICERA can be instructed to instead compute a contract consisting of a pre- and post-condition for the function (also see Section II-C). Functions that do not have a body are assumed to produce some non-deterministic result, but not change global variables or heap data.

Function contracts can optionally be specified using the ACSL specification language [6]. At the moment, TRICERA can parse and encode `requires`, `ensures` and `assigns` clauses. Listing 1 shows an example program that TRICERA can check. Programs annotated with contracts are verified modularly: for each function f with a contract, TRICERA will try to prove that f will never violate its contract that will then be used for encoding f at its call sites. More details about the supported ACSL features are given in [21].

C. Annotation Inference

TRICERA can be used to automatically infer function contracts and loop invariants for *safe* programs (with respect to implicit and explicit assertions) [4]. An example program is given in Listing 2, encoding the `tak` function [44]. Based on the properties assumed and asserted at lines 12 and 14,

Listing 1: Example of ACSL function contracts in TRICERA. The program is *unsafe* because `q` is accessed but only `p` is specified in the `assigns` clause.

```

1 /*@
2   requires \valid(p, q);
3   assigns *p;
4 */
5 void foo(int* p, int* q) {
6   *q = 42;
7 }

```

Listing 2: An example contract inference problem in TRICERA

```

1 /*@ contract @*/ ↗
2 int tak(int x, int y, int z) {
3   if (y < x)
4     return tak(tak(x-1, y, z),
5               tak(y-1, z, x),
6               tak(z-1, x, y));
7   else return y;
8 }
9
10 void main() {
11   int x, y, z;
12   assume(x > y && y <= z);
13   int r = tak(x, y, z);
14   assert(r == z);
15 }

```

respectively, TRICERA is able to compute a contract for `tak` that is sufficient to show the safety of the program:

$$f_{pre} : true$$

$$f_{post} : (r \neq z \vee y \geq z \vee x > y) \wedge (r \neq y \vee y \geq z \vee y \geq x) \wedge (r = z \vee r = y \vee y > z) \wedge (r = y \vee z \geq y \vee x > y)$$

where f_{pre} and f_{post} are the pre- and post-conditions of `tak`.

The inferred contracts and invariants can be printed in the ACSL language [6], as well as in SMT-LIB2 and in Prolog. As of writing this paper, ACSL printing is limited to programs without heap.

D. Uninterpreted Predicates

TRICERA allows declaration of *uninterpreted predicates* as annotations, which can then be used in `assert` and `assume` statements. Uninterpreted predicates provide a way to directly affect the generated set of CHCs, as assumptions about the shape of invariants can be manually specified. A program annotated with uninterpreted predicates is considered safe if and only if an interpretation of the predicates (in the sense of first-order logic) exists such that all assertions hold.

An example application is given in Figure 2. In the left column, an array `a` is updated in a loop, and the loop at

line 8 encodes the property $\forall j : 0 \leq j < n \rightarrow a[j] = 2j$. Although the program is simple, it turns out to be challenging for software model checkers, since a universally quantified property about the array elements is needed.

The right column shows a version of the program rewritten for verification purposes; the uninterpreted predicate `p_a` is now used to specify a data invariant for the array `a`. The two arguments are selected to correspond to the index, and the value residing at that index, respectively. Writes to `a` are replaced with assertions to `p_a` as in line 6, which asserts that the array `a` contains the value `2*i` at index `i`. Reads from `a` are replaced with assumptions with an additional fresh variable in lines 9–10. The program in the right column can be verified by TRICERA almost instantaneously.

The encoding in Figure 2 closely corresponds to the encoding of universally quantified properties in [13], and is also similar to the invariant encoding of [35], where heap data and operations are encoded through data invariants. Uninterpreted predicates in TRICERA make it possible to easily experiment with encoding tricks of this kind.

E. Concurrency

TRICERA has basic support for handling concurrency in programs. Static threads, executing concurrently with the main program, can be declared using the keyword `thread`. TRICERA currently applies a relatively simple, sequentially consistent thread model that is defined in [34]. This support for concurrency is mainly intended for modelling purposes, but is also useful, e.g., for defining monitors that check temporal properties during execution. For instance, the following thread asserts that the global variable `x` will never decrease during program execution.

```

1 thread Monitor {
2   int t = x;
3   assert(x >= t);
4 }

```

Thread interleaving can be controlled using `atomic` blocks, which mandate that all statements in the block are executed in one atomic step. Threads can moreover be controlled using synchronous rendezvous, which are introduced through UPPAAL-style binary communication channels [7]. In the following program, the two statements `chan_send` and `chan_receive` can only be executed together, thus ensuring that the assertion will be checked after the assignment:

```

1 chan s; int x;
2 thread A {x = 42; chan_send(s);}
3 thread B {chan_receive(s); assert(x > 0);}

```

Finally, TRICERA also supports the declaration of infinitely replicated threads, which are useful to model dynamic thread creation and parameterised systems. An example of a parameterised model is given in the next section.

<pre> 1 2 void main () { 3 int i, n = _; 4 int a[n]; 5 for (i = 0; i < n; ++i) { 6 a[i] = 2*i; 7 } 8 for (i = 0; i < n; ++i) { 9 10 11 assert(a[i] == 2*i); 12 } 13 } </pre>	<pre> 1 /*\$ p_a(int, int) \$*/ ⌘ 2 void main () { 3 int i, n = _; 4 5 for (i = 0; i < n; ++i) { 6 assert(p_a(i, 2*i)); 7 } 8 for (i = 0; i < n; ++i) { 9 int v = _; 10 assume(p_a(i, v)); 11 assert(2*i == v); 12 } 13 } </pre>
--	---

Fig. 2: Encoding an array program (left column) using uninterpreted predicates (right column).

Listing 3: The parameterised Fischer protocol [3]

```

1 int lock = 0; ⌘
2 thread[tid] Proc {
3   clock C;
4   assume(tid > 0);
5
6   while (1) {
7     atomic { assume(lock == 0); C = 0; }
8     within (C <= 1) { lock = tid; }
9     C = 0; assume(C > 1);
10
11    if (lock == tid) { // critical sect.
12      assert(lock == tid);
13      lock = 0;
14    }
15  }
16 }

```

F. Timing Constraints

For modelling purposes, TRICERA supports timing constraints in C programs. C programs with time have semantics similar to UPPAAL timed automata [7], which means that computations (program instructions) consume zero time, but are interleaved with explicit time-elapse transitions. The passing of time can be observed using clocks, which are declared as variables of type `clock`, can be reset to 0, and can be compared with constants in `assert` and `assume` statements.

As an example, Listing 3 shows a parameterised version of the well-known Fischer mutual exclusion protocol [3]. An arbitrary number of processes can participate in the protocol by communicating through a shared variable `lock`. In line 2, for this purpose an infinitely replicated thread `Proc` is declared. Each instance of `Proc` has a unique thread id `tid` of type `int` and a clock `C`. Each process executes a simple loop: it waits until it observes that `lock == 0`, and then writes its

thread id to `lock`. The `within` block in line 8 has similar semantics as an UPPAAL time invariant: it enforces execution of the assignment before the condition $C \leq 1$ has become false, i.e., at most one time unit after executing the block in line 7. The process then waits for more than one time unit in line 9, and then checks that no other process has meanwhile overwritten the value in `lock`. Line 12 asserts that at most one process is able to enter the critical section at a time.

TRICERA is able to verify the safety of this model for an unbounded number of participating threads, using an encoding of the program as CHCs over k -indexed invariants [34].

III. THE TRICERA VERIFICATION APPROACH

A. Constrained Horn Clauses

TRICERA analyses programs by translating them to sets of Constrained Horn Clauses (CHCs, or just *clauses* in this paper), in such a way that the CHCs are satisfiable iff the program is safe. A Constrained Horn Clause is a sentence $\forall \bar{x}. (C \wedge B_1 \wedge \dots \wedge B_n \rightarrow H)$ where H is either an *atom* (application of a predicate to first-order terms) or *false*, B_i (for $1 \leq i \leq n$) is an atom, and C is a constraint over some background theories (including heaps). A CHC with at least one positive *literal* (an atom or its negation) is called a *definite clause*, and a CHC with no positive literals is called a *goal clause* (or an *assertion clause*). In the rest of the paper we leave the universal quantification of variables implicit, and write the clauses from right to left in the spirit of logic programming.

B. The Architecture of TRICERA

An overview of the TRICERA architecture is given in Figure 3. The preprocessor and the CHC solver are external tools; we call the whole toolchain “TRICERA”.

a) Preprocessor: Input programs are preprocessed in order to simplify parsing and encoding. To simplify parsing, all `typedefs` are removed and some language constructs are normalised into a standard form. Unused type and function declarations are removed; removing unused data-types makes

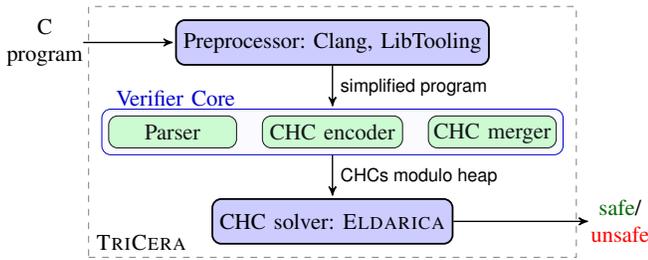


Fig. 3: An overview of TRICERA.

modelling heap simpler as the number of possible types for a heap object is reduced. The preprocessor is written as a stand-alone tool using the LibTooling³ library in C++.

b) Verifier Core: The TRICERA core component is a translator from C programs into CHCs, written in Scala. The verifier core works by first creating a parse tree of the input program, and then translating this tree into a set of CHCs. The translator supports the language features given in Table I.

The CHC encoder also includes a CHC simplifier that post-processes the generated CHCs before being sent to a CHC solver. This simplifier attempts to merge the CHCs in order to produce a smaller but equisatisfiable set of CHCs.

c) CHC solver: The resulting set of CHCs are finally sent to a CHC solver to check if their conjunction is satisfiable. TRICERA primarily uses ELDARICA [33] for this purpose as it has native support for the theory of heaps, and can easily be integrated as a Scala library; however, the final set of CHCs can also be post-processed by eliminating heap operations, instead encoding using the theory of arrays, and then be checked by other solvers such as Z3/SPACER [37].

C. Programs as Constrained Horn Clauses (CHCs)

The overall translation from sequential programs to CHCs applied by TRICERA follows the strategy defined, e.g., in [12], [29]. In this setting, linear CHCs are used to model the control-flow graph of a program: each node of the graph is interpreted to represent the set of possible states at a program location and each transition corresponds to a program control instruction. Asserted properties add additional sink nodes to the graph, whose edges are the negations of those properties. The goal of the process is to discover program *invariants* that are sufficient to show that none of the sink nodes is reachable.

A program is thus encoded in CHCs as follows:

- An uninterpreted predicate is declared for each program location to represent program invariants: the interpretations of these predicates (provided by the CHC solver when the set of CHCs is satisfiable) correspond to sets of program states that hold at each location. The arguments to a predicate are all program variables currently in scope, as well as additional terms required in the encoding, for instance terms representing the heap.

- A definite clause consisting of only a single positive literal is added as program entry, e.g., $P(\dots) \leftarrow true$. The CHC (1) in Figure 4 is an example.
- A definite clause is introduced for each program control instruction. These CHCs encode Hoare triples between locations [32]. The set of CHCs can be cyclic (e.g., $\{P_1(\dots) \leftarrow P_0(\dots), P_0(\dots) \leftarrow P_1(\dots)\}$), representing program loops. Guarded control instructions are encoded by adding the guards as constraints. The CHCs (2) – (5) in Figure 4 provide an example.
- Two clauses are added for each asserted property: a goal clause whose constraint is the negation of the asserted property, and a definite clause whose constraint is the asserted property. The CHCs (6) and (7) in Figure 4 provide an example.
- Functions are encoded either through predicates representing their pre-/post-conditions, or by inlining them.

An example encoding is provided in Figure 4.

The translation of concurrent and timed programs follows the calculus defined in [34]. To handle concurrency, TRICERA uses a variant of the Owicki-Gries proof rules [47], [34], to which explicit variables to represent time and clocks are added. The representation of replicated threads uses the k -indexed invariants approach [52], [34].

IV. THE THEORY OF HEAPS

One of the most challenging aspects of encoding computer programs as CHCs is the encoding of heap-allocated data-structures and heap-related operations. One approach to represent such data-structures is using the theory of arrays (e.g., [36], [17]). This is a natural encoding since a heap can be seen as an array of memory locations; however, as the encoding is byte-precise, in the context of CHCs it tends to be low-level and often yields clauses that are hard to solve.

An alternative approach is to transform away such data-structures with the help of invariants or refinement types (e.g., [49], [13], [45], [35], and the example in Section II-D). The resulting CHCs tend to be over-approximate (i.e., can lead to false positives), even with smart refinement strategies that aim at increasing precision. This is because every operation that reads, writes, or allocates a heap object is replaced with assertions and assumptions about local object invariants, so that global program invariants might not be expressible. In cases where local invariants are sufficient, however, they can enable efficient and modular verification even of challenging programs.

Both approaches leave little design choice with respect to handling of heaps to CHC solvers. Dealing with heaps at the encoding level also implies repeated effort when designing verifiers for different programming languages.

The vision of the presented line of research is to extend CHCs to a standardised interchange format for programs with heaps. We apply a high-level theory of heaps [24] that does not restrict the way in which CHC solvers approach heap reasoning, while covering the main functionality needed for program verification: (i) representation of the type system

³<https://clang.lvm.org/docs/LibTooling.html>

```

1  /* P0 */
2  if (x > 0)
3    x+=1; /* P1 */
4  else
5    x-=1; /* P2 */
6  /* P3 */
7  assert (x > 0);
8  /* P4 */

```

$$\begin{aligned}
P_0(x) &\leftarrow true & (1) \\
P_1(x) &\leftarrow P_0(x) \wedge x > 0 & (2) \\
P_2(x) &\leftarrow P_0(x) \wedge x \leq 0 & (3) \\
P_3(x') &\leftarrow P_1(x) \wedge x' = x + 1 & (4) \\
P_3(x') &\leftarrow P_2(x) \wedge x' = x - 1 & (5) \\
P_4(x') &\leftarrow P_3(x) \wedge x > 0 & (6) \\
false &\leftarrow P_3(x) \wedge x \leq 0 & (7)
\end{aligned}$$

Fig. 4: The CHC encoding of a branching statement

Listing 4: SMT-LIB-style declaration of a heap. In lines 4–8 the constructors and the selectors of the data-types are declared. The constructors and selectors in lines 5–6 serve as the wrappers and the getters for the program types `Node` and `int`. `Node` is encoded as an ADT (line 4) and the C type `int` is encoded using mathematical integers (`Int`).

```

1 (declare-heap
2  Heap Addr Object O_Empty
3  ((Node 0) (Object 0))
4  (((Node (data Int) (next Addr))))
5  ((O_Node (getNode Node))
6   (O_Int (getInt Int))
7   (O_Uninit_Node) (O_Uninit_Int)
8   (O_Empty))))

```

associated with heap data; (ii) reading and updating of data on the heap; (iii) object allocation.

The theory of heaps employs algebraic data-types (ADTs), as already standardised by SMT-LIB [5], as a flexible way to handle (i). The theory offers operations akin to the theory of arrays to handle (ii) and (iii). Arithmetic operations on pointers are excluded in the theory, as are low-level tricks like extracting individual bytes from bigger pieces of data through pointer manipulation. Being language-agnostic, the theory of heaps allows for common encodings across different applications.

a) Sorts: To encode a program using the theory of heaps, first a heap data-type has to be declared that covers the required program types; a declaration in SMT-LIB notation is shown in Listing 4. Each declared heap introduces the three sorts, *Heap*, *Addr* and *AddrRange*, and in addition can declare any number of ADTs later used to represent the data stored on the heap (lines 5–8, see Section V). A *Heap* address has the sort *Addr*. Although an address itself does not carry type information, the type of a heap *Object* can be checked using ADT discriminator functions. A range of addresses can be defined with the *AddrRange* sort, which is needed when encoding contiguous data-structures such as arrays.

The objects on the heap are represented with a single *Object* sort, which can either be selected from one of the pre-declared sorts, or declared as an ADT in a heap theory declaration. The latter makes referring to heap theory sorts possible, such as *Addr*, as done in Line 4 of Listing 4. In the sequel we call a

constructor function that produces an *Object* a *wrapper*, and a selector that returns the underlying term from an *Object* a *getter*.

b) Operations: The operations of the theory of heaps are given in Table II. The function `allocate` is used for allocating new objects on the heap, and each allocation returns a new $\langle \text{Heap}, \text{Addr} \rangle$ pair that is valid and contains the passed object. The allocatedness of an *Addr* in a *Heap* can be tested using the predicate `valid`. The function `emptyHeap` returns a heap that is invalid at all addresses, and `nullAddr` returns an address that is invalid in all heaps.

The functions `read` and `write` are used for reading from and writing to heap addresses. If a read address is invalid, the *default object* is returned (`O_Empty` in line 2 in Listing 4). An invalid write returns the heap that was passed to the function without any modifications. The default *Object* to be returned on invalid reads is specified in the heap declaration, and this is needed to make the read function total.

Operations (14)–(17) are used for *batch* heap operations, which are needed when encoding array-like data on the heap. These operations operate over address ranges rather than single addresses (*Addr*). The functions `batchAllocate` and `batchWrite` allow batch allocation and batch update of address ranges. Given an address range, `nthInAddrRange` allows the extraction of an individual address, and the predicate `withinAddrRange` allows testing if an address is within a range.

c) Implementation: The theory of heaps is currently implemented in the SMT solver PRINCESS [50] and in the CHC solver ELGARICA [33]. The decision procedure for solving

TABLE II: Operations defined by the theory of heaps

<code>emptyHeap</code>	$: () \rightarrow \text{Heap}$	(8)
<code>nullAddr</code>	$: () \rightarrow \text{Addr}$	(9)
<code>allocate</code>	$: \text{Heap} \times \text{Object} \rightarrow \text{Heap} \times \text{Addr}$	(10)
<code>valid</code>	$: \text{Heap} \times \text{Addr} \rightarrow \text{Bool}$	(11)
<code>read</code>	$: \text{Heap} \times \text{Addr} \rightarrow \text{Object}$	(12)
<code>write</code>	$: \text{Heap} \times \text{Addr} \times \text{Object} \rightarrow \text{Heap}$	(13)
<code>batchAllocate</code>	$: \text{Heap} \times \text{Object} \times \mathbb{N} \rightarrow \text{Heap} \times \text{AddrRange}$	(14)
<code>batchWrite</code>	$: \text{Heap} \times \text{AddrRange} \times \text{Object} \rightarrow \text{Heap}$	(15)
<code>nthInAddrRange</code>	$: \text{AddrRange} \times \mathbb{N} \rightarrow \text{Addr}$	(16)
<code>withinAddrRange</code>	$: \text{AddrRange} \times \text{Addr} \rightarrow \text{Bool}$	(17)

TABLE III: O_T is the object wrapper for sort T , which is the encoding of the program type T . $O_T(T_0)$ constructs a zero-valued term of sort T . h represents the heap. Non-primed and primed terms encode the same program variable (and the heap) before and after the execution of a statement. x is a variable, p is a (non-array) pointer. a and b are pointers to arrays of type T . i , j and n are integers.

C statement	Mathematical encoding with heaps
$*p = x;$	$h' = \text{write}(h, p, O_T(x)) \wedge$ $(\text{is-}O_T(\text{read}(h, p)) \vee \text{is-}O_T(\text{read}(h, p)))$
$x = *p;$	$x = \text{get}T(\text{read}(h, p)) \wedge \text{is-}O_T(\text{read}(h, p))$
$x = a[i];$	$x = \text{get}T(\text{read}(h, \text{nthInAddrRange}(a, i))) \wedge$ $\text{is-}O_T(\text{read}(h, \text{nthInAddrRange}(a, i))) \wedge$ $\text{withinAddrRange}(a, i)$
$a[i] = x;$	$h' = \text{write}(h, \text{nthInAddrRange}(a, i), O_T(x)) \wedge$ $(\text{is-}O_T(\text{read}(h, \text{nthInAddrRange}(a, i))) \vee$ $\text{is-}O_T(\text{read}(h, \text{nthInAddrRange}(a, i)))) \wedge$ $\text{withinAddrRange}(a, i)$
$p = \text{malloc}(\text{sizeof}(T));$	$\langle h', p \rangle = \text{allocate}(h, O_T(\text{Uninit}_T))$
$p = \text{calloc}(\text{sizeof}(T));$	$\langle h', p \rangle = \text{allocate}(h, O_T(T_0))$
$a = \text{malloc}(\text{sizeof}(T) * n);$	$\langle h', p \rangle = \text{batchAllocate}($ $h, O_T(\text{Uninit}_T), n)$
$\text{free}(p);$	$h' = \text{write}(h, p, O_T(\text{Empty})) \wedge$ $\neg \text{is-}O_T(\text{read}(h, p))$
$\text{free}(a);$	$h' = \text{batchWrite}(h, a, O_T(\text{Empty})) \wedge$ $\forall q : \text{Addr.}(\text{withinAddrRange}(a, q) \rightarrow$ $\neg \text{is-}O_T(\text{read}(h, q)))$

formulas over the theory in PRINCESS is introduced in [23]. ELDARICA mostly defers the solving of heap theory formulas to PRINCESS; there is ongoing work to implement additional static analysis of heap properties directly in ELDARICA.

V. ENCODING OF C PROGRAMS WITH HEAP

When translating programs with heaps, TRICERA augments all introduced relation symbols (state invariants and pre-conditions) with explicit *Heap* arguments; post-conditions receive both the pre- and the post-heap.

A heap *Addr* can be seen as a direct counterpart of an (untyped) C pointer. Any program type that makes use of an *Addr*, such as a list node, needs to be declared as part of the heap theory declaration. Lastly, *Object wrappers* and *getters* need to be declared for all program types that can be on the heap. For instance, Listing 4 shows a heap declaration for a program over (mathematical) integers and a node struct:

```

struct Node {
  int data;
  struct Node* next;
};

```

Since *Node* has a pointer field, it is declared as an ADT as part of the heap declaration as shown in line 4 of Listing 4. The object wrappers and getters for all program types are declared in lines 5–6. Additional *empty* object wrappers are defined in lines 7–8 to serve as the *uninitialised* and *default* objects respectively. TRICERA uses the default object to mark de-allocated locations as shown in Table III. The

uninitialised objects are used as initial values for allocated memory locations with uninitialised values, as is the case with `malloc`. An uninitialised object constructor is declared for each programming type on the heap.

After the heap is declared, every statement that accesses the heap is encoded as shown in Table III. A new heap term h' is produced for statements modifying the starting heap term h .

Each de-reference of a pointer is also coupled with a type-safety assertion. In the table, those assertions are conjoined with the actual transition relations of the CHCs; as a result, the stated formulas describe all correct executions of a statement. TRICERA in addition introduces assertions that will detect cases in which these conditions are violated. For instance, for the expression $*p$, assuming that p is encoded using the sort T , TRICERA asserts the predicate $\text{is-}O_T(\text{read}(h, p))$. $\text{is-}O_T$ is the discriminator predicate for the ADT sort T . Since invalid reads would return the default object, this type-safety assertion doubles as a memory safety assertion. C also allows the allocation of uninitialised memory; TRICERA models this by placing the object $O_T(\text{Uninit}_T)$ in these addresses, which represents an uninitialised value for the sort T .

Functions that require byte-level access to data-structures such as `memset` are currently not supported by TRICERA; however, these can be handled without introducing a full byte-level memory representation. It is sufficient to infer which values a heap object can assume when setting all its bytes to a certain value, taking into account the compiler and architecture when necessary. To prevent aliasing when using such functions, safety assertions that ensure the accessed memory region belongs to a single object can be automatically added to each access.

a) *Arrays*: TRICERA uses the theory of heaps also to model C arrays. Arrays are allocated and freed using the *batch* operations of the theory. The address of an array cell is obtained with the `nthInAddrRange` function, which can then be used as any other address. Whenever an array cell is accessed (`a[i]`), TRICERA automatically asserts that the accessed index is within bounds (`withinAddrRange(a, i)`).

Arithmetic operations on array pointers can be supported by augmenting *AddrRange* terms with offsets (not shown in Table III). This yields a model in which arithmetic on array pointers is possible, but modified pointers have to remain in the same array, which is again in line with the MISRA C coding guidelines [2].

The theory of heaps does not provide a direct operation for de-allocation, i.e., an allocated address always remains valid. TRICERA overcomes this limitation by writing the default object ($O_T(\text{Empty})$) to de-allocated addresses, and provides an option to add a memory safety assertion such that all addresses must contain the default object at program exit. Double-freeing of memory is caught by an additional assertion that the freed addresses do not contain the default object.

Stack-allocated arrays are also modeled using the theory of heaps, and the functions to free their memory are automatically added by TRICERA when they go out of scope. Non-array



Fig. 5: The three sets of heap benchmarks: (i) the source C benchmarks from SV-COMP are encoded into (ii) CHCs modulo the theory of heaps using TRICERA, then heap2array is applied to these benchmarks to produce (iii) CHCs modulo the theory of arrays.

stack pointers do not make use of the theory, and are supported with some limitations as described in [22].

Table III does not show the encoding of field updates for record types, for instance $p \rightarrow f = x$ where p is a pointer to a record type with f as one of its fields. This is encoded by first reading the record from p , creating a new ADT term with only the field f updated, and then writing back the new ADT to the address pointed to by p .

VI. EXPERIMENTAL RESULTS

A. Benchmarks

As TRICERA does not have a model of pthreads yet, we focus in our evaluation on sequential C programs. We collected C benchmarks from SV-COMP 2022’s *ReachSafety* and *MemSafety* categories [10], and generated CHCs in the SMT-LIB [5] format for all benchmarks that the current version of TRICERA could parse and encode (see Section II-A). This resulted in 396 heap (i.e., where the heap is modelled using the theory of heaps) benchmarks (349 in the *ReachSafety* and 128 in the *MemSafety* categories, with some benchmarks occurring in both categories), and 1453 non-heap benchmarks in the *ReachSafety* category. Many of the benchmarks that TRICERA could not parse were under the Juliet test and the Linux device driver suites, which failed mainly due to currently unsupported operations and constructs such as `memcpy` and function pointers. Mathematical integer semantics was used in the benchmarks encoded by TRICERA.

For the heap benchmarks, an additional set of benchmarks was created through a translation of the theory of heaps into the theory of arrays, using an extended version of the encoding given in [24] implemented in the tool *heap2array*⁴. This serves the purpose of making additional back-ends available to solve the generated CHCs. Similarly generated benchmarks were also submitted to CHC-Comp 2022 and were part of the *LIA-nonlin-Arrays-nonrecADT* track⁵. The benchmark creation process is depicted in Figure 5.

We then applied two of the top CHC solvers currently available [26] to the CHCs: ELDARICA, which is the default solver in TRICERA and natively supports the theory of heaps, and Z3/SPACER [37]. We have used the default settings in both ELDARICA and Z3/SPACER. ELDARICA was used in two different configurations for the heap benchmarks: TRICERA (ELDARICA-heap), using the native solver for the theory of

heaps on the CHCs with heaps, and TRICERA (ELDARICA-array), applying ELDARICA’s array solver to the array version of the CHCs. Z3/SPACER was only applied to the array benchmarks (TRICERA (Z3/SPACER)). The *portfolio* rows in the result tables show the results achieved by running both back-ends of TRICERA in parallel and taking the first result (TRICERA (portfolio)).

B. Experimental Setup

The experiments were ran on an AMD Opteron 2220 SE (2.8 GHZ with 4 CPUs) machine running 64-bit Linux with 6 GB of RAM and a wall-clock timeout of 900 seconds. To compare TRICERA⁶ against the state of the art, we gathered the results published by SV-COMP 2022 [9] for the *ReachSafety* and *MemSafety* tracks.

C. Results

The results are given in Table IV for the non-heap benchmarks in the *ReachSafety* category, in Table V for the heap benchmarks in the *ReachSafety* category and in Table VI for the heap benchmarks in the *MemSafety* category. All benchmarks can be found in [25].

For non-heap, TRICERA showed performance competitive with the best tools evaluated at SV-COMP, in particular on safe problems. The TRICERA results are not completely comparable to the results of SV-COMP tools due to the use of mathematical integer semantics in TRICERA, however. For 19 benchmarks, the statuses reported by TRICERA were inconsistent with the expected SV-COMP statuses for this reason. The two TRICERA back-ends, ELDARICA and Z3/SPACER, always produced the same answer.

For heap problems, TRICERA performed worse than some of the tools based on bounded model checking or symbolic execution, but was comparable with CEGAR-based tools like CPACHECKER. Comparing the TRICERA back-ends, ELDARICA applied to the array encoding performs best by some margin (TRICERA (ELDARICA-array)).

TRICERA currently cannot check for reachability and memory-safety properties separately, it always adds the implicit memory-safety assertions. This, coupled with the use of mathematical integers, led to results that did not match their expected SV-COMP statuses in 25 heap benchmarks (13 reported incorrectly unsafe, 12 reported incorrectly safe) using the portfolio method; again there were no inconsistencies between the different TRICERA back-ends.

VII. RELATED WORK

There are several other verification tools that make use of CHCs, and many others for verifying C programs. As discussed in Section I, these tools either transform away the heap, or use the theory of arrays for encoding heap.

JAYHORN, a model checker for Java programs, encodes heap by using invariants that summarise the possible states of a reference at a program location [35], which is inspired by methods like liquid types [49]. Although this method is

⁴<https://github.com/zafer-esen/heap2array>

⁵<https://github.com/zafer-esen/tricera-adt-arr>

⁶<https://github.com/uuverifiers/tricera/commit/5ffd2b6>

TABLE IV: Results for the non-heap benchmarks in the *ReachSafety* category. The column “solved” gives the total number of “safe” or “unsafe” results.

	safe	unsafe	unknown	solved
GOBLINT [51]	180	0	1273	180
THETA [53]	250	140	1063	390
UKOJAK [46]	278	221	954	499
VERIFUZZ [15]	0	515	938	515
2LS [42]	428	265	760	693
CBMC [38]	313	394	746	707
TRICERA (Z3/SPACER)	442	271	740	713
CRUX [19]	293	427	733	720
LART [40]	346	392	715	738
ESBMC-KIND [41]	484	380	589	864
SYMBIOTIC [14]	423	458	572	881
UTAIPAN [18]	598	298	557	896
UAUTOMIZER [31]	612	302	539	914
PESCO [48]	584	458	411	1042
TRICERA (ELDARICA)	698	360	395	1058
GRAVES-CPA [41]	636	442	375	1078
TRICERA (portfolio)	730	379	344	1109
CPACHECKER [11]	666	470	317	1136
VERIABS [16]	739	507	207	1246

TABLE V: Results for the heap benchmarks in the *ReachSafety* category.

	safe	unsafe	unknown	solved
THETA [53]	10	7	332	17
GOBLINT [51]	27	0	322	27
GRAVES-CPA [41]	22	26	301	48
TRICERA (ELDARICA-heap)	12	36	301	48
2LS [42]	35	21	293	56
TRICERA (Z3/SPACER)	20	40	289	60
UTAIPAN [18]	32	33	284	65
UAUTOMIZER [31]	32	35	282	67
UKOJAK [46]	25	42	282	67
VERIFUZZ [15]	0	71	278	71
TRICERA (ELDARICA-array)	36	49	264	85
TRICERA (portfolio)	39	58	252	97
CRUX [19]	55	48	246	103
CPACHECKER [11]	58	46	245	104
PESCO [48]	65	47	237	112
CBMC [38]	65	51	233	116
LART [40]	90	30	229	120
SYMBIOTIC [14]	102	62	185	164
ESBMC-KIND [41]	122	49	178	171
VERIABS [16]	223	81	45	304

TABLE VI: Results for the heap benchmarks in the *MemSafety* category.

	safe	unsafe	unknown	solved
VERIFUZZ [15]	0	5	123	5
SESL	0	11	117	11
UAUTOMIZER [31]	6	11	111	17
UTAIPAN [18]	7	10	111	17
UKOJAK [46]	9	10	109	19
2LS [42]	14	11	103	25
TRICERA (ELDARICA-heap)	12	19	97	31
TRICERA (Z3/SPACER)	23	16	89	39
CPACHECKER [11]	53	10	65	63
TRICERA (ELDARICA-array)	39	24	65	63
TRICERA (portfolio)	40	26	62	66
ESBMC-KIND [41]	54	18	56	72
CPA-BAM-SMG	53	30	45	83
CBMC [38]	54	36	38	90
SYMBIOTIC [14]	68	36	24	104

incomplete (i.e., can lead to false positives), with various optimisations the authors have managed to significantly improve its effectiveness. Using the theory of heaps, much of the work in JayHorn could be shifted to a CHC solver. TRICERA and JAYHORN both use ELDARICA for solving the generated CHCs, but otherwise do not share any infrastructure.

RUSTHORN is a verifier for Rust programs, and also transforms away the heap [43] by exploiting the ownership system of Rust. Since the method is not directly applicable in case of unsafe code blocks, a theory of heaps could be used to extend the tool in this direction.

SEAHORN is a verifier for LLVM-based languages [30]. SEAHORN employs Z3/SPACER as one of its back-ends for CHC-based model-checking. It also employs various static analyses that can be used on their own as a verification engine, or to provide invariants to its CHC back-ends. SEAHORN encodes the heap as a set of non-overlapping arrays that are created by a *data structure analysis* (DSA) [39]. Since SEAHORN works with the LLVM intermediate representation, it can be used to target other LLVM-based languages than C. In contrast, TRICERA comes with its own parser that currently cannot handle all the peculiarities of C; however, its custom parser can handle several non-standard C constructs as shown in Table I and can easily be extended.

KORN is a verifier for C programs that uses CHCs; however its main focus is showing the feasibility of using *loop contracts* as opposed to loop invariants and currently supports a small fragment of C [20]. KORN uses ELDARICA as one of its back-ends.

Information about the other verification tools evaluated in Section VI can be found in the SV-COMP report [8].

VIII. CONCLUSIONS AND OUTLOOK

This paper has introduced the verification tool TRICERA, given an overview of the encoding of C programs using the theory of heaps, and provided first experimental results using SV-COMP benchmarks. Both TRICERA and the theory of heaps are still under development, and planned future work includes support for further features of C (see Table I), improved decision and interpolation procedures for the theory of heaps, and the development of additional heap back-ends (in particular along the lines of [35]). Once multiple CHC solvers with support for the theory of heaps are available, we will also propose a heap track at the Horn solver competition CHC-COMP.

ACKNOWLEDGEMENTS

This work was supported by the Swedish Research Council (VR) under grant 2018-04727, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and by the Knut and Alice Wallenberg Foundation under the project UPDATE.

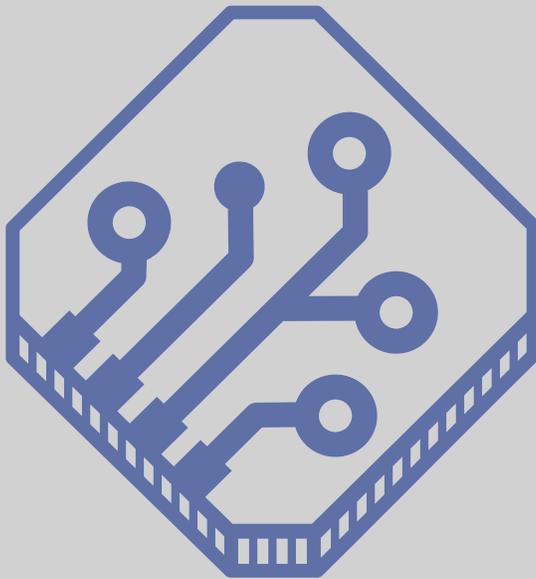
REFERENCES

- [1] Information technology — programming languages — C. ISO/IEC 9899:2011, International Organization for Standardization, Geneva, Switzerland (2011)
- [2] Guidelines for the use of the C language in critical systems. MISRA C:2012, The MISRA Consortium Limited, Norfolk, England (2012)
- [3] Abadi, M., Lamport, L.: An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst.* **16**(5), 1543–1571 (sep 1994). <https://doi.org/10.1145/186025.186058>, <https://doi.org/10.1145/186025.186058>
- [4] Amilon, J., Esen, Z., Gurov, D., Lidström, C., Rümmer, P.: An exercise in mind reading: Automatic contract inference for Frama-C. In: *Guide to Software Verification with Frama-C. Core Components, Usages, and Applications* (2022), (To appear)
- [5] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
- [6] Baudin, P., Cuoq, P., Filiâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language Version 1.17. (2021)
- [7] Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UP-PAAL - a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) *Hybrid Systems III: Verification and Control*, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22–25, 1995, Rutgers University, New Brunswick, NJ, USA. *Lecture Notes in Computer Science*, vol. 1066, pp. 232–243. Springer (1995). <https://doi.org/10.1007/BFb0020949>, <https://doi.org/10.1007/BFb0020949>
- [8] Beyer, D.: Progress on software verification: SV-COMP 2022. In: Fisman, D., Rosu, G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II. *Lecture Notes in Computer Science*, vol. 13244, pp. 375–402. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20, https://doi.org/10.1007/978-3-030-99527-0_20
- [9] Beyer, D.: Results of the 11th Intl. Competition on Software Verification (SV-COMP 2022) (Jan 2022). <https://doi.org/10.5281/zenodo.5831008>, <https://doi.org/10.5281/zenodo.5831008>
- [10] Beyer, D.: SV-Benchmarks: Benchmark Set for Software Verification and Testing (SV-COMP 2022 and Test-Comp 2022) (Jan 2022). <https://doi.org/10.5281/zenodo.5831003>, <https://doi.org/10.5281/zenodo.5831003>
- [11] Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Computer Aided Verification - 23rd International Conference, CAV 2011*, Snowbird, UT, USA, July 14–20, 2011. Proceedings. *Lecture Notes in Computer Science*, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16, https://doi.org/10.1007/978-3-642-22110-1_16
- [12] Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. *Lecture Notes in Computer Science*, vol. 9300, pp. 24–51. Springer (2015). https://doi.org/10.1007/978-3-319-23534-9_2, https://doi.org/10.1007/978-3-319-23534-9_2
- [13] Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) *Static Analysis - 20th International Symposium, SAS 2013*, Seattle, WA, USA, June 20–22, 2013. Proceedings. *Lecture Notes in Computer Science*, vol. 7935, pp. 105–125. Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_8, https://doi.org/10.1007/978-3-642-38856-9_8
- [14] Chalupa, M., Mihalkovic, V., Rečtáková, A., Zaoral, L., Strejcek, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding - (competition contribution). In: Fisman, D., Rosu, G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II. *Lecture Notes in Computer Science*, vol. 13244, pp. 462–467. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32, https://doi.org/10.1007/978-3-030-99527-0_32
- [15] Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing - (competition contribution). In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics*, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III. *Lecture Notes in Computer Science*, vol. 11429, pp. 244–249. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22, https://doi.org/10.1007/978-3-030-17502-3_22
- [16] Darke, P., Agrawal, S., Venkatesh, R.: Veriabs: A tool for scalable verification by abstraction (competition contribution). In: Groote, J.F., Larsen, K.G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. *Lecture Notes in Computer Science*, vol. 12652, pp. 458–462. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32, https://doi.org/10.1007/978-3-030-72013-1_32
- [17] De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Program verification using constraint handling rules and array constraint generalizations. *Fundam. Inform.* **150**(1), 73–117 (2017). <https://doi.org/10.3233/FI-2017-1461>, <https://doi.org/10.3233/FI-2017-1461>
- [18] Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate taipan with symbolic interpretation and fluid abstractions - (competition contribution). In: Biere, A., Parker, D. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II. *Lecture Notes in Computer Science*, vol. 12079, pp. 418–422. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_32, https://doi.org/10.1007/978-3-030-45237-7_32
- [19] Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Blazy, S., Chechik, M. (eds.) *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016*, Toronto, ON, Canada, July 17–18, 2016. Revised Selected Papers. *Lecture Notes in Computer Science*, vol. 9971, pp. 56–72 (2016). https://doi.org/10.1007/978-3-319-48869-1_5, https://doi.org/10.1007/978-3-319-48869-1_5
- [20] Ernst, G.: A complete approach to loop verification with invariants and summaries. *CoRR* **abs/2010.05812** (2020), <https://arxiv.org/abs/2010.05812>
- [21] Ernst, G.: Contract-Based Verification in TriCera. Master’s thesis, Uppsala University, Department of Information Technology (2022)
- [22] Esen, Z.: Extension of the ELGARICA C model checker with heap memory. Master’s thesis, Uppsala University, Department of Information Technology (2019)
- [23] Esen, Z., Rümmer, P.: Reasoning in the theory of heap: Satisfiability and interpolation. In: Fernández, M. (ed.) *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020*, Bologna, Italy, September 7–9, 2020, Proceedings. *Lecture Notes in Computer Science*, vol. 12561, pp. 173–191. Springer (2020). https://doi.org/10.1007/978-3-030-68446-4_9, https://doi.org/10.1007/978-3-030-68446-4_9
- [24] Esen, Z., Rümmer, P.: A theory of heap for constrained horn clauses (extended technical report). *CoRR* **abs/2104.04224** (2021), <https://arxiv.org/abs/2104.04224>
- [25] Esen, Z., Rümmer, P.: TriCera Benchmarks: SMT-LIB Encodings of SV-COMP 2022 Benchmarks by TriCera (Aug 2022). <https://doi.org/10.5281/zenodo.6950363>, <https://doi.org/10.5281/zenodo.6950363>
- [26] Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021*, Virtual, 28th March 2021. *EPTCS*, vol. 344, pp. 91–108 (2021). <https://doi.org/10.4204/EPTCS.344.7>, <https://doi.org/10.4204/EPTCS.344.7>
- [27] Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Hankin, C., Schmidt, D. (eds.) *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, London, UK, January 17–19, 2001. pp. 193–205. ACM (2001). <https://doi.org/10.1145/360204.360220>, <https://doi.org/10.1145/360204.360220>

- [28] Freeman, T., Pfenning, F.: Refinement types for ML. In: PLDI. pp. 268–277. ACM, New York, NY, USA (1991). <https://doi.org/10.1145/113445.113468>, <http://doi.acm.org/10.1145/113445.113468>
- [29] Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416. ACM (2012). <https://doi.org/10.1145/2254064.2254112>, <https://doi.org/10.1145/2254064.2254112>
- [30] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20, https://doi.org/10.1007/978-3-319-21690-4_20
- [31] Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate automizer and the search for perfect interpolants - (competition contribution). In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30, https://doi.org/10.1007/978-3-319-89963-3_30
- [32] Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969). <https://doi.org/10.1145/363235.363259>, <https://doi.org/10.1145/363235.363259>
- [33] Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: FMCAD 2018. pp. 1–7 (2018). <https://doi.org/10.23919/FMCAD.2018.8603013>
- [34] Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. In: Bjørner, N., Fioravanti, F., Rybalchenko, A., Senni, V. (eds.) Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014. EPTCS, vol. 169, pp. 39–52 (2014). <https://doi.org/10.4204/EPTCS.169.6>, <https://doi.org/10.4204/EPTCS.169.6>
- [35] Kahsai, T., Kersten, R., Rümmer, P., Schäfer, M.: Quantified heap invariants for object-oriented programs. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7–12, 2017. EPIC Series in Computing, vol. 46, pp. 368–384. EasyChair (2017), <https://easychair.org/publications/paper/Pmh>
- [36] Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using Horn clauses over integers and arrays. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27–30, 2015. pp. 89–96. IEEE (2015)
- [37] Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 846–862. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_59, https://doi.org/10.1007/978-3-642-39799-8_59
- [38] Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 389–391. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26, https://doi.org/10.1007/978-3-642-54862-8_26
- [39] Lattner, C., Adve, V.S.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12–15, 2005. pp. 129–142. ACM (2005). <https://doi.org/10.1145/1065010.1065027>, <https://doi.org/10.1145/1065010.1065027>
- [40] Lauko, H., Rockai, P.: LART: compiled abstract execution - (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 457–461. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_31, https://doi.org/10.1007/978-3-030-99527-0_31
- [41] Leeson, W., Dwyer, M.B.: Graves-cpa: A graph-attention verifier selector (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 440–445. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28, https://doi.org/10.1007/978-3-030-99527-0_28
- [42] Malík, V., Schrammel, P., Vojnar, T.: 2ls: Heap analysis and memory safety - (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 368–372. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_22, https://doi.org/10.1007/978-3-030-45237-7_22
- [43] Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* **43**(4), 15:1–15:54 (2021). <https://doi.org/10.1145/3462205>, <https://doi.org/10.1145/3462205>
- [44] McCarthy, J.: An interesting lisp function. *ACM Lisp Bulletin* (3), 6–8 (1979)
- [45] Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free Horn clauses. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8–10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 361–382. Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_18, https://doi.org/10.1007/978-3-662-53413-7_18
- [46] Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks - (competition contribution). In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 458–460. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44, https://doi.org/10.1007/978-3-662-46681-0_44
- [47] Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs i. *Acta Inf.* **6**, 319–340 (1976). <https://doi.org/10.1007/BF00268134>
- [48] Richter, C., Wehrheim, H.: Pesco: Predicting sequential combinations of verifiers - (competition contribution). In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 229–233. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19, https://doi.org/10.1007/978-3-030-17502-3_19
- [49] Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008. pp. 159–169. ACM (2008). <https://doi.org/10.1145/1375581.1375602>, <https://doi.org/10.1145/1375581.1375602>
- [50] Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNCS, vol. 5330, pp. 274–289. Springer (2008)
- [51] Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints - (competition contribution). In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of

- Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 438–442. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28, https://doi.org/10.1007/978-3-030-72013-1_28
- [52] Sánchez, A., Sankaranarayanan, S., Sánchez, C., Chang, B.Y.E.: Invariant generation for parametrized systems using self-reflection - (extended version). In: Miné, A., Schmidt, D. (eds.) SAS. Lecture Notes in Computer Science, vol. 7460, pp. 146–163. Springer (2012). https://doi.org/10.1007/978-3-642-33125-1_12, <https://doi.org/10.1007/978-3-642-33125-1>
- [53] Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., Majzik, I.: Theta: A framework for abstraction refinement-based model checking. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 176–179. IEEE (2017). <https://doi.org/10.23919/FMCAD.2017.8102257>, <https://doi.org/10.23919/FMCAD.2017.8102257>

The Conference on Formal Methods in Computer-Aided Design (FMCAD) is an annual conference on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing.



ISBN 978-3-85448-053-2



9 783854 480532

www.tuwien.at/academicpress