Constantin Enea
Akash Lal (Eds.)

LNCS 13964

Computer Aided Verification

35th International Conference, CAV 2023
Paris, France, July 17–22, 2023
Proceedings, Part I

OPEN ACCESS

Lecture Notes in Computer Science 13964

Founding Editors

Gerhard Goos
Juris Hartmanis

Editorial Board Members

Elisa Bertino, *Purdue University, West Lafayette, IN, USA*
Wen Gao, *Peking University, Beijing, China*
Bernhard Steffen, *TU Dortmund University, Dortmund, Germany*
Moti Yung, *Columbia University, New York, NY, USA*

The series Lecture Notes in Computer Science (LNCS), including its subseries Lecture Notes in Artificial Intelligence (LNAI) and Lecture Notes in Bioinformatics (LNBI), has established itself as a medium for the publication of new developments in computer science and information technology research, teaching, and education.

LNCS enjoys close cooperation with the computer science R & D community, the series counts many renowned academics among its volume editors and paper authors, and collaborates with prestigious societies. Its mission is to serve this international community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings. LNCS commenced publication in 1973.

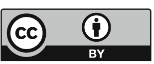
Constantin Enea · Akash Lal
Editors

Computer Aided Verification

35th International Conference, CAV 2023
Paris, France, July 17–22, 2023
Proceedings, Part I

Editors
Constantin Enea
LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris
Palaiseau, France

Akash Lal
Microsoft Research
Bangalore, India



ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-37705-1 ISBN 978-3-031-37706-8 (eBook)
https://doi.org/10.1007/978-3-031-37706-8

© The Editor(s) (if applicable) and The Author(s) 2023. This book is an open access publication.

Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.
The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

It was our privilege to serve as the program chairs for CAV 2023, the 35th International Conference on Computer-Aided Verification. CAV 2023 was held during July 19–22, 2023 and the pre-conference workshops were held during July 17–18, 2023. CAV 2023 was an in-person event, in Paris, France.

CAV is an annual conference dedicated to the advancement of the theory and practice of computer-aided formal analysis methods for hardware and software systems. The primary focus of CAV is to extend the frontiers of verification techniques by expanding to new domains such as security, quantum computing, and machine learning. This puts CAV at the cutting edge of formal methods research, and this year's program is a reflection of this commitment.

CAV 2023 received a large number of submissions (261). We accepted 15 tool papers, 3 case-study papers, and 49 regular papers, which amounts to an acceptance rate of roughly 26%. The accepted papers cover a wide spectrum of topics, from theoretical results to applications of formal methods. These papers apply or extend formal methods to a wide range of domains such as concurrency, machine learning and neural networks, quantum systems, as well as hybrid and stochastic systems. The program featured keynote talks by Ruzica Piskac (Yale University), Sumit Gulwani (Microsoft), and Caroline Trippel (Stanford University). In addition to the contributed talks, CAV also hosted the CAV Award ceremony, and a report from the Synthesis Competition (SYNTCOMP) chairs.

In addition to the main conference, CAV 2023 hosted the following workshops: Meeting on String Constraints and Applications (MOSCA), Verification Witnesses and Their Validation (VeWit), Verification of Probabilistic Programs (VeriProP), Open Problems in Learning and Verification of Neural Networks (WOLVERINE), Deep Learning-aided Verification (DAV), Hyperproperties: Advances in Theory and Practice (HYPER), Synthesis (SYNT), Formal Methods for ML-Enabled Autonomous Systems (FoMLAS), and Verification Mentoring Workshop (VMW). CAV 2023 also hosted a workshop dedicated to Thomas A. Henzinger for this 60th birthday.

Organizing a flagship conference like CAV requires a great deal of effort from the community. The Program Committee for CAV 2023 consisted of 76 members—a committee of this size ensures that each member has to review only a reasonable number of papers in the allotted time. In all, the committee members wrote over 730 reviews while investing significant effort to maintain and ensure the high quality of the conference program. We are grateful to the CAV 2023 Program Committee for their outstanding efforts in evaluating the submissions and making sure that each paper got a fair chance. Like recent years in CAV, we made artifact evaluation mandatory for tool paper submissions, but optional for the rest of the accepted papers. This year we received 48 artifact submissions, out of which 47 submissions received at least one badge. The Artifact Evaluation Committee consisted of 119 members who put in significant effort to evaluate each artifact. The goal of this process was to provide constructive feedback to tool developers and

help make the research published in CAV more reproducible. We are also very grateful to the Artifact Evaluation Committee for their hard work and dedication in evaluating the submitted artifacts.

CAV 2023 would not have been possible without the tremendous help we received from several individuals, and we would like to thank everyone who helped make CAV 2023 a success. We would like to thank Alessandro Cimatti, Isil Dillig, Javier Esparza, Azadeh Farzan, Joost-Pieter Katoen and Corina Pasareanu for serving as area chairs. We also thank Bernhard Kragl and Daniel Dietsch for chairing the Artifact Evaluation Committee. We also thank Mohamed Faouzi Atig for chairing the workshop organization as well as leading publicity efforts, Eric Koskinen as the fellowship chair, Sebastian Bardin and Ruzica Piskac as sponsorship chairs, and Srinidhi Nagendra as the website chair. Srinidhi, along with Enrique Román Calvo, helped prepare the proceedings. We also thank Ankush Desai, Eric Koskinen, Burcu Kulahcioglu Ozkan, Marijana Lazic, and Matteo Sammartino for chairing the mentoring workshop. Last but not least, we would like to thank the members of the CAV Steering Committee (Kenneth McMillan, Aarti Gupta, Orna Grumberg, and Daniel Kroening) for helping us with several important aspects of organizing CAV 2023.

We hope that you will find the proceedings of CAV 2023 scientifically interesting and thought-provoking!

June 2023

Constantin Enea
Akash Lal

Organization

Conference Co-chairs

Constantin Enea — LIX, École Polytechnique, France
Akash Lal — Microsoft Research, India

Artifact Co-chairs

Bernhard Kragl — Amazon Web Services, USA
Daniel Dietsch — Qt Group/University of Freiburg, Germany

Workshop Chair

Mohamed Faouzi Atig — Uppsala University, Sweden

Verification Mentoring Workshop Organizing Committee

Ankush Densai — AWS CA, USA
Eric Koskinen — Stevens Institute of Technology, USA
Burcu Kulahcioglu Ozkan — TU Delft, The Netherlands
Marijana Lazic — TU Munich, Germany
Matteo Sammartino — Royal Holloway, University of London, UK

Fellowship Chair

Eric Koskinen — Stevens Institute of Technology, USA

Website Chair

Srinidhi Nagendra — Université Paris Cité, CNRS, IRIF, France and Chennai Mathematical Institute, India

Sponsorship Co-chairs

Sebastian Bardin	CEA LIST, France
Ruzica Piskac	Yale University, USA

Proceedings Chairs

Srinidhi Nagendra	Université Paris Cité, CNRS, IRIF, France and Chennai Mathematical Institute, India
Enrique Román Calvo	Université Paris Cité, CNRS, IRIF, France

Program Committee

Aarti Gupta	Princeton University, USA
Abhishek Bichhawat	IIT Gandhinagar, India
Aditya V. Thakur	University of California, USA
Ahmed Bouajjani	University of Paris, France
Aina Niemetz	Stanford University, USA
Akash Lal	Microsoft Research, India
Alan J. Hu	University of British Columbia, Canada
Alessandro Cimatti	Fondazione Bruno Kessler, Italy
Alexander Nadel	Intel, Israel
Anastasia Mavridou	KBR, NASA Ames Research Center, USA
Andreas Podelski	University of Freiburg, Germany
Ankush Desai	Amazon Web Services
Anna Slobodova	Intel, USA
Anthony Widjaja Lin	TU Kaiserslautern and Max-Planck Institute for Software Systems, Germany
Arie Gurfinkel	University of Waterloo, Canada
Arjun Radhakrishna	Microsoft, India
Aws Albarghouthi	University of Wisconsin-Madison, USA
Azadeh Farzan	University of Toronto, Canada
Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Bettina Koenighofer	Graz University of Technology, Austria
Bor-Yuh Evan Chang	University of Colorado Boulder and Amazon, USA
Burcu Kulahcioglu Ozkan	Delft University of Technology, The Netherlands
Caterina Urban	Inria and École Normale Supérieure, France
Cezara Dragoi	Amazon Web Services, USA

Christoph Matheja	Technical University of Denmark, Denmark
Claudia Cauli	Amazon Web Services, UK
Constantin Enea	LIX, CNRS, Ecole Polytechnique, France
Corina Pasareanu	CMU, USA
Cristina David	University of Bristol, UK
Dirk Beyer	LMU Munich, Germany
Elizabeth Polgreen	University of Edinburgh, UK
Elvira Albert	Complutense University, Spain
Eunsuk Kang	Carnegie Mellon University, USA
Gennaro Parlato	University of Molise, Italy
Hossein Hojjat	Tehran University and Tehran Institute of Advanced Studies, Iran
Ichiro Hasuo	National Institute of Informatics, Japan
Isil Dillig	University of Texas, Austin, USA
Javier Esparza	Technische Universität München, Germany
Joost-Pieter Katoen	RWTH-Aachen University, Germany
Juneyoung Lee	AWS, USA
Jyotirmoy Deshmukh	University of Southern California, USA
Kenneth L. McMillan	University of Texas at Austin, USA
Kristin Yvonne Rozier	Iowa State University, USA
Kshitij Bansal	Google, USA
Kuldeep Meel	National University of Singapore, Singapore
Kyungmin Bae	POSTECH, South Korea
Marcell Vazquez-Chanlatte	Alliance Innovation Lab (Nissan-Renault-Mitsubishi), USA
Marieke Huisman	University of Twente, The Netherlands
Markus Rabe	Google, USA
Marta Kwiatkowska	University of Oxford, UK
Matthias Heizmann	University of Freiburg, Germany
Michael Emmi	AWS, USA
Mihaela Sighireanu	University Paris Saclay, ENS Paris-Saclay and CNRS, France
Mohamed Faouzi Atig	Uppsala University, Sweden
Naijun Zhan	Institute of Software, Chinese Academy of Sciences, China
Nikolaj Bjorner	Microsoft Research, USA
Nina Narodytska	VMware Research, USA
Pavithra Prabhakar	Kansas State University, USA
Pierre Ganty	IMDEA Software Institute, Spain
Rupak Majumdar	Max Planck Institute for Software Systems, Germany
Ruzica Piskac	Yale University, USA

Sebastian Junges	Radboud University, The Netherlands
Sébastien Bardin	CEA, LIST, Université Paris Saclay, France
Serdar Tasiran	Amazon, USA
Sharon Shoham	Tel Aviv University, Israel
Shaz Qadeer	Meta, USA
Shuvendu Lahiri	Microsoft Research, USA
Subhajit Roy	Indian Institute of Technology, Kanpur, India
Suguman Bansal	Georgia Institute of Technology, USA
Swarat Chaudhuri	UT Austin, USA
Sylvie Putot	École Polytechnique, France
Thomas Wahl	GrammaTech, USA
Tomáš Vojnar	Brno University of Technology, FIT, Czech Republic
Yakir Vizel	Technion - Israel Institute of Technology, Israel
Yu-Fang Chen	Academia Sinica, Taiwan
Zhilin Wu	State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China

Artifact Evaluation Committee

Alejandro Hernández-Cerezo	Complutense University of Madrid, Spain
Alvin George	IISc Bangalore, India
Aman Goel	Amazon Web Services, USA
Amit Samanta	University of Utah, USA
Anan Kabaha	Technion, Israel
Andres Noetzli	Cubist, Inc., USA
Anna Becchi	Fondazione Bruno Kessler, Italy
Arnab Sharma	University of Oldenburg, Germany
Avraham Raviv	Bar Ilan University, Israel
Ayrat Khalimov	TU Clausthal, Germany
Baoluo Meng	General Electric Research, USA
Benjamin Jones	Amazon Web Services, USA
Bohua Zhan	Institute of Software, Chinese Academy of Sciences, China
Cayden Codel	Carnegie Mellon University, USA
Charles Babu M.	CEA LIST, France
Chungha Sung	Amazon Web Services, USA
Clara Rodriguez-Núñez	Universidad Complutense de Madrid, Spain
Cyrus Liu	Stevens Institute of Technology, USA
Daniel Hausmann	University of Gothenburg, Sweden

Daniela Kaufmann	TU Wien, Austria
Debasmita Lohar	MPI SWS, Germany
Deivid Vale	Radboud University Nijmegen, Netherlands
Denis Mazzucato	Inria, France
Dorde Žikelić	Institute of Science and Technology Austria, Austria
Ekanshdeep Gupta	New York University, USA
Enrico Magnago	Amazon Web Services, USA
Ferhat Erata	Yale University, USA
Filip Cordoba	Graz University of Technology, Austria
Filipe Arruda	UFPE, Brazil
Florian Dorfhuber	Technical University of Munich, Germany
Florian Sextl	TU Wien, Austria
Francesco Parolini	Sorbonne University, France
Frédéric Recoules	CEA LIST, France
Goktug Saatcioglu	Cornell, USA
Goran Piskachev	Amazon Web Services, USA
Grégoire Menguy	CEA LIST, France
Guy Amir	Hebrew University of Jerusalem, Israel
Habeeb P.	Indian Institute of Science, Bangalore, India
Hadrien Renaud	UCL, UK
Haoze Wu	Stanford University, USA
Hari Krishnan	University of Waterloo, Canada
Hünkar Tunç	Aarhus University, Denmark
Idan Refaeli	Hebrew University of Jerusalem, Israel
Ignacio D. Lopez-Miguel	TU Wien, Austria
Ilina Stoilkovska	Amazon Web Services, USA
Ira Fesefeldt	RWTH Aachen University, Germany
Jahid Choton	Kansas State University, USA
Jie An	National Institute of Informatics, Japan
John Kolesar	Yale University, USA
Joseph Scott	University of Waterloo, Canada
Kevin Lotz	Kiel University, Germany
Kirby Linvill	CU Boulder, USA
Kush Grover	Technical University of Munich, Germany
Levente Bajczi	Budapest University of Technology and Economics, Hungary
Liangcheng Yu	University of Pennsylvania, USA
Luke Geeson	UCL, UK
Lutz Klinkenberg	RWTH Aachen University, Germany
Marek Chalupa	Institute of Science and Technology Austria, Austria

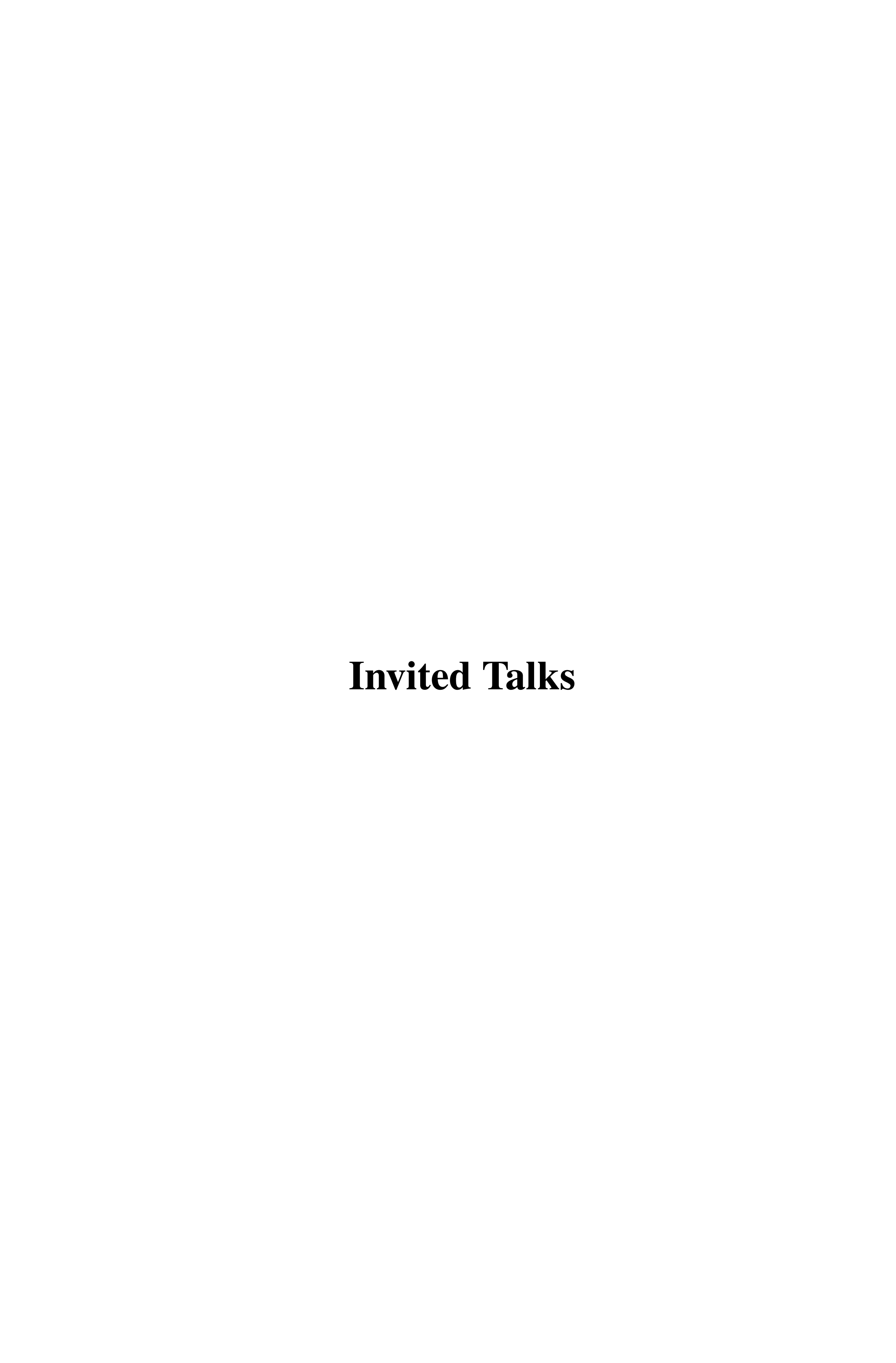
Mario Bucev	EPFL, Switzerland
Mário Pereira	NOVA LINCS—Nova School of Science and Technology, Portugal
Marius Mikucionis	Aalborg University, Denmark
Martin Jonáš	Masaryk University, Czech Republic
Mathias Fleury	University of Freiburg, Germany
Matthias Hetzenberger	TU Wien, Austria
Maximilian Heisinger	Johannes Kepler University Linz, Austria
Mertcan Temel	Intel Corporation, USA
Michele Chiari	TU Wien, Austria
Miguel Isabel	Universidad Complutense de Madrid, Spain
Mihai Nicola	Stevens Institute of Technology, USA
Mihály Dobos-Kovács	Budapest University of Technology and Economics, Hungary
Mikael Mayer	Amazon Web Services, USA
Mitja Kulczynski	Kiel University, Germany
Muhammad Mansur	Amazon Web Services, USA
Muqsit Azeem	Technical University of Munich, Germany
Neelanjana Pal	Vanderbilt University, USA
Nicolas Koh	Princeton University, USA
Niklas Metzger	CISPA Helmholtz Center for Information Security, Germany
Omkar Tuppe	IIT Bombay, India
Pablo Gordillo	Complutense University of Madrid, Spain
Pankaj Kalita	Indian Institute of Technology, Kanpur, India
Parisa Fathololumi	Stevens Institute of Technology, USA
Pavel Hudec	HKUST, Hong Kong, China
Peixin Wang	University of Oxford, UK
Philippe Heim	CISPA Helmholtz Center for Information Security, Germany
Pritam Gharat	Microsoft Research, India
Priyanka Darke	TCS Research, India
Ranadeep Biswas	Informal Systems, Canada
Robert Rubbens	University of Twente, Netherlands
Rubén Rubio	Universidad Complutense de Madrid, Spain
Samuel Judson	Yale University, USA
Samuel Pastva	Institute of Science and Technology Austria, Austria
Sankalp Gambhir	EPFL, Switzerland
Sarbojit Das	Uppsala University, Sweden
Sascha Klüppelholz	Technische Universität Dresden, Germany
Sean Kauffman	Aalborg University, Denmark

Shaowei Zhu	Princeton University, USA
Shengjian Guo	Amazon Web Services, USA
Simmo Saan	University of Tartu, Estonia
Smruti Padhy	University of Texas at Austin, USA
Stanly Samuel	Indian Institute of Science, Bangalore, India
Stefan Pranger	Graz University of Technology, Austria
Stefan Zetzsche	Amazon Web Services, USA
Sumanth Prabhu	TCS Research, India
Sumit Lahiri	Indian Institute of Technology, Kanpur, India
Sunbeom So	Korea University, South Korea
Syed M. Iqbal	Amazon Web Services, USA
Tobias Meggendorfer	Institute of Science and Technology Austria, Austria
Tzu-Han Hsu	Michigan State University, USA
Verya Monjezi	University of Texas at El Paso, USA
Wei-Lun Tsai	Academia Sinica, Taiwan
William Schultz	Northeastern University, USA
Xiao Liang Yu	National University of Singapore, Singapore
Yahui Song	National University of Singapore, Singapore
Yasharth Bajpai	Microsoft Research, USA
Ying Sheng	Stanford University, USA
Yuriy Biktairov	University of Southern California, USA
Zafer Esen	Uppsala University, Sweden

Additional Reviewers

Azzopardi, Shaun
Baier, Daniel
Belardinelli, Francesco
Bergstraesser, Pascal
Boker, Udi
Ceska, Milan
Chien, Po-Chun
Coglio, Alessandro
Correas, Jesús
Doveri, Kyveli
Drachsler Cohen, Dana
Durand, Serge
Fried, Dror
Genaim, Samir
Ghosh, Bishwamittra
Gordillo, Pablo
Guillermo, Roman Diez
Gómez-Zamalloa, Miguel
Hernández-Cerezo, Alejandro
Holík, Lukáš
Isabel, Miguel
Ivrii, Alexander
Izza, Yacine
Jothimurugan, Kishor
Kaivola, Roope
Kaminski, Benjamin Lucien
Kettl, Matthias
Kretinsky, Jan
Lengal, Ondrej
Losa, Giuliano
Luo, Ning
Malik, Viktor

Markgraf, Oliver
Martin-Martin, Enrique
Meller, Yael
Perez, Mateo
Petri, Gustavo
Pote, Yash
Preiner, Mathias
Rakamaric, Zvonimir
Rastogi, Aseem
Razavi, Niloofar
Rogalewicz, Adam
Sangnier, Arnaud
Sarkar, Uddalok
Schoepe, Daniel
Sergey, Ilya
Stoilkovska, Ilina
Stucki, Sandro
Tsai, Wei-Lun
Turrini, Andrea
Vafeiadis, Viktor
Valiron, Benoît
Wachowitz, Henrik
Wang, Chao
Wang, Yuepeng
Wies, Thomas
Yang, Jiong
Yen, Di-De
Zhu, Shufang
Žikelić, Đorđe
Zohar, Yoni



Invited Talks

Privacy-Preserving Automated Reasoning

Ruzica Piskac

Yale University, USA

Formal methods offer a vast collection of techniques to analyze and ensure the correctness of software and hardware systems against a given specification. In fact, modern formal methods tools scale to industrial applications. Despite this significant success, privacy requirements are not considered in the design of these tools. For example, when using automated reasoning tools, the implicit requirement is that the formula to be proved is public. This raises an issue if the formula itself reveals information that is supposed to remain private to one party. To overcome this issue, we propose the concept of privacy-preserving automated reasoning.

We first consider the problem of privacy-preserving Boolean satisfiability [1]. In this problem, two mutually distrustful parties each provides a Boolean formula. The goal is to decide whether their conjunction is satisfiable without revealing either formula to the other party. We present an algorithm to solve this problem. Our algorithm is an oblivious variant of the classic DPLL algorithm and can be integrated with existing secure two-party computation techniques.

We next turn to the problem where one party wants to prove to another party that their program satisfies a given specification without revealing the program. We split this problem into two subproblems: (1) proving that the program can be translated into a propositional formula without revealing either the program or the formula; (2) proving that the obtained formula entails the specification. To solve the latter subproblem, we developed a zero-knowledge protocol for proving the unsatisfiability of formulas in propositional logic [2] (ZKUNSAT). Our protocol is based on a resolution proof of unsatisfiability. We encode verification of the resolution proof using polynomial equivalence checking, which enables us to use fast zero-knowledge protocols for polynomial satisfiability.

Finally, we will outline future directions towards extending ZKUNSAT to first-order logic modulto theories (SMT) and translating programs to formulas in zero-knowledge to realize fully automated privacy-preserving program verification.

References

1. Luo, N., Judson, S., Antonopoulos, T., Piskac, R., Wang, X.: ppSAT: towards two-party private SAT solving. In: Butler, K.R.B., Thomas, K., (eds.) 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, 10–12 August 2022, pp. 2983–3000. USENIX Association (2022)
2. Luo, N., Antonopoulos, T., Harris, W.R., Piskac, R., Tromer, E., Wang, X.: Proving UNSAT in zero knowledge. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the

2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, 7–11 November 2022, pp. 2203–2217. ACM (2022)

Enhancing Programming Experiences Using AI: Leveraging LLMs as Analogical Reasoning Engines and Beyond

Sumit Gulwani

Microsoft, USA

AI can significantly improve programming experiences for a diverse range of users: from professional developers and data scientists (proficient programmers) who need help in software engineering and data wrangling, to spreadsheet users (low-code programmers) needing help in authoring formulas, and students (novice programmers) seeking hints when tackling programming homework. To effectively communicate their needs to AI, users can express their intent explicitly through input-output examples or natural language specifications, or implicitly by presenting bugs or recent code edits for AI to analyze and suggest improvements.

Analogical reasoning is at the heart of problem solving as it allows to make sense of new information and transfer knowledge from one domain to another. In this talk, I will demonstrate that analogical reasoning is a fundamental emergent capability of Large Language Models (LLMs) and can be utilized to enhance various types of programming experiences.

However, there is significant room for innovation in building robust experiences tailored to specific task domains. I will discuss how various methods from symbolic AI (particularly programming-by-examples-or-analogies) such as search-and-rank, failure-guided refinement, and neuro-symbolic cooperation, can help fill this gap. This comes in three forms: (a) Prompt engineering that involves synthesizing specification-rich, context-aware prompts from various sources, sometimes using the LLM itself, to elicit optimal output. (b) Post-processing techniques that guide, rank, and validate the LLM's output, occasionally employing the LLM for these purposes. (c) Multi-turn workflows that involve multiple LLM invocations, allowing the model more time and iterations to optimize results. I will illustrate these concepts using various capabilities in Excel, PowerQuery, and Visual Studio.

Verified Software Security Down to Gates

Caroline Trippel

Stanford University, USA

Hardware-software (HW-SW) contracts are critical for high-assurance computer systems design and an enabler for software design/analysis tools that find and repair hardware-related bugs in programs. E.g., memory consistency models define what values shared memory loads can return in a parallel program. Emerging security contracts define what program data is susceptible to leakage via hardware side-channels and what speculative control- and data-flow is possible at runtime. However, these contracts and the analyses they support are useless if we cannot guarantee microarchitectural compliance, which is a "grand challenge." Notably, some contracts are still evolving (e.g., security contracts), making hardware compliance a moving target. Even for mature contracts, comprehensively verifying that a complex microarchitecture implements some abstract contract is a time-consuming endeavor involving teams of engineers, which typically requires resorting to incomplete proofs.

Our work takes a radically different approach to the challenge above by synthesizing HW-SW contracts from advanced (i.e., industry-scale/complexity) processor implementations. In this talk, I will present our work on: synthesizing security contracts from processor specifications written in Verilog; designing compiler approaches parameterized by these contracts that can find and repair hardware-related vulnerabilities in programs; and updating hardware microarchitectures to support scalable verification and efficient security-hardened programs. I will conclude by outlining remaining challenges in attaining the vision of verified software security down to gates.

Contents – Part I

Automata and Logic

Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization 3
Masaki Waga

Automated Analyses of IOT Event Monitoring Systems 27
Andrew Apicelli, Sam Bayless, Ankush Das, Andrew Gacek, Dhiva Jaganathan, Saswat Padhi, Vaibhav Sharma, Michael W. Whalen, and Raveesh Yadav

Learning Assumptions for Compositional Verification of Timed Automata 40
Hanyue Chen, Yu Su, Miaomiao Zhang, Zhiming Liu, and Junri Mi

Online Causation Monitoring of Signal Temporal Logic 62
Zhenya Zhang, Jie An, Paolo Arcaini, and Ichiro Hasuo

Process Equivalence Problems as Energy Games 85
Benjamin Bisping

Concurrency

Commutativity for Concurrent Program Termination Proofs 109
Danya Lette and Azadeh Farzan

Fast Termination and Workflow Nets 132
Piotr Hofman, Filip Mazowiecki, and Philip Offtermatt

Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM 156
Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh

`nekton`: A Linearizability Proof Checker 170
Roland Meyer, Anton Opaterny, Thomas Wies, and Sebastian Wolff

Overcoming Memory Weakness with Unified Fairness: Systematic Verification of Liveness in Weak Memory Models 184
Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, Shankaranarayanan Krishna, and Mihir Vahanwala

Rely-Guarantee Reasoning for Causally Consistent Shared Memory 206
Ori Lahav, Brijesh Dongol, and Heike Wehrheim

Unblocking Dynamic Partial Order Reduction 230
Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis

Cyber-Physical and Hybrid Systems

3D Environment Modeling for Falsification and Beyond with Scenic 3.0 253
Eric Vin, Shun Kashiwa, Matthew Rhea, Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia

A Unified Model for Real-Time Systems: Symbolic Techniques and Implementation 266
S. Akshay, Paul Gastin, R. Govind, Aniruddha R. Joshi, and B. Srivathsan

Closed-Loop Analysis of Vision-Based Autonomous Systems: A Case Study 289
Corina S. Păsăreanu, Ravi Mangal, Divya Gopinath, Sinem Getir Yaman, Calum Imrie, Radu Calinescu, and Huafeng Yu

Hybrid Controller Synthesis for Nonlinear Systems Subject to Reach-Avoid Constraints 304
Zhengfeng Yang, Li Zhang, Xia Zeng, Xiaochao Tang, Chao Peng, and Zhenbing Zeng

Safe Environmental Envelopes of Discrete Systems 326
Rômulo Meira-Góes, Ian Dardik, Eunsuk Kang, Stéphane Lafortune, and Stavros Tripakis

Verse: A Python Library for Reasoning About Multi-agent Hybrid System Scenarios 351
Yangge Li, Haoqing Zhu, Katherine Braught, Keyi Shen, and Sayan Mitra

Synthesis

Counterexample Guided Knowledge Compilation for Boolean Functional Synthesis 367
S. Akshay, Supratik Chakraborty, and Sahil Jain

Guessing Winning Policies in LTL Synthesis by Semantic Learning 390
Jan Křetínský, Tobias Meggendorfer, Maximilian Prokop, and Sabine Rieder

Policy Synthesis and Reinforcement Learning for Discounted LTL 415
Rajeev Alur, Osbert Bastani, Kishor Jothimurugan, Mateo Perez, Fabio Somenzi, and Ashutosh Trivedi

Synthesizing Permissive Winning Strategy Templates for Parity Games 436
Ashwani Anand, Satya Prakash Nayak, and Anne-Kathrin Schmuck

Synthesizing Trajectory Queries from Examples 459
Stephen Mell, Favyen Bastani, Steve Zdancewic, and Osbert Bastani

Author Index 485

Contents – Part II

Decision Procedures

Bitwuzla 3
Aina Niemetz and Mathias Preiner

Decision Procedures for Sequence Theories 18
Artur Jeż, Anthony W. Lin, Oliver Markgraf, and Philipp Rümmer

Exploiting Adjoints in Property Directed Reachability Analysis 41
Mayuko Kori, Flavio Ascari, Filippo Bonchi, Roberto Bruni, Roberta Gori, and Ichiro Hasuo

Fast Approximations of Quantifier Elimination 64
Isabel Garcia-Contreras, V. K. Hari Govind, Sharon Shoham, and Arie Gurfinkel

Local Search for Solving Satisfiability of Polynomial Formulas 87
Haokun Li, Bican Xia, and Tianqi Zhao

Partial Quantifier Elimination and Property Generation 110
Eugene Goldberg

Rounding Meets Approximate Model Counting 132
Jiong Yang and Kuldeep S. Meel

Satisfiability Modulo Finite Fields 163
Alex Ozdemir, Gereon Kremer, Cesare Tinelli, and Clark Barrett

Solving String Constraints Using SAT 187
Kevin Lotz, Amit Goel, Bruno Dutertre, Benjamin Kiesl-Reiter, Soonho Kong, Rupak Majumdar, and Dirk Nowotka

The GOLEM Horn Solver 209
Martin Blicha, Konstantin Britikov, and Natasha Sharygina

Model Checking

COQCRYPTOLINE: A Verified Model Checker with Certified Results 227
Ming-Hsien Tsai, Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang

Incremental Dead State Detection in Logarithmic Time 241
Caleb Stanford and Margus Veanes

Model Checking Race-Freedom When "Sequential Consistency for Data-Race-Free Programs" is Guaranteed 265
Wenhao Wu, Jan Hückelheim, Paul D. Hovland, Ziqing Luo, and Stephen F. Siegel

Searching for i-Good Lemmas to Accelerate Safety Model Checking 288
Yechuan Xia, Anna Becchi, Alessandro Cimatti, Alberto Griggio, Jianwen Li, and Geguang Pu

Second-Order Hyperproperties 309
Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger

Neural Networks and Machine Learning

Certifying the Fairness of KNN in the Presence of Dataset Bias 335
Yannan Li, Jingbo Wang, and Chao Wang

Monitoring Algorithmic Fairness 358
Thomas A. Henzinger, Mahyar Karimi, Konstantin Kueffner, and Kaushik Mallik

`nl2spec`: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models 383
Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel

NNV 2.0: The Neural Network Verification Tool 397
Diego Manzanas Lopez, Sung Woo Choi, Hoang-Dung Tran, and Taylor T. Johnson

QEBVerif: Quantization Error Bound Verification of Neural Networks 413
Yedi Zhang, Fu Song, and Jun Sun

Verifying Generalization in Deep Learning 438
Guy Amir, Osher Maayan, Tom Zelazny, Guy Katz, and Michael Schapira

Author Index 457

Contents – Part III

Probabilistic Systems

A Flexible Toolchain for Symbolic Rabin Games under Fair and Stochastic Uncertainties 3
Rupak Majumdar, Kaushik Mallik, Mateusz Rychlicki, Anne-Kathrin Schmuck, and Sadegh Soudjani

Automated Tail Bound Analysis for Probabilistic Recurrence Relations 16
Yican Sun, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady

Compositional Probabilistic Model Checking with String Diagrams of MDPs 40
Kazuki Watanabe, Clovis Eberhart, Kazuyuki Asada, and Ichiro Hasuo

Efficient Sensitivity Analysis for Parametric Robust Markov Chains 62
Thom Badings, Sebastian Junges, Ahmadreza Marandi, Ufuk Topcu, and Nils Jansen

MDPs as Distribution Transformers: Affine Invariant Synthesis for Safety Objectives 86
S. Akshay, Krishnendu Chatterjee, Tobias Meggendorfer, and Đorđe Žikelić

Search and Explore: Symbiotic Policy Synthesis in POMDPs 113
Roman Andriushchenko, Alexander Bork, Milan Češka, Sebastian Junges, Joost-Pieter Katoen, and Filip Macák

Security and Quantum Systems

AutoQ: An Automata-Based Quantum Circuit Verifier 139
Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai

Bounded Verification for Finite-Field-Blasting: In a Compiler for Zero Knowledge Proofs 154
Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark Barrett

Formally Verified EVM Block-Optimizations ... 176
Elvira Albert, Samir Genaim, Daniel Kirchner, and Enrique Martin-Martin

SR-SFLL: Structurally Robust Stripped Functionality Logic Locking ... 190
Gourav Takhar and Subhajit Roy

Symbolic Quantum Simulation with Quasimodo ... 213
Meghana Sistla, Swarat Chaudhuri, and Thomas Reps

Verifying the Verifier: eBPF Range Analysis Verification ... 226
Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte

Software Verification

Automated Verification of Correctness for Masked Arithmetic Programs ... 255
Mingyang Liu, Fu Song, and Taolue Chen

Automatic Program Instrumentation for Automatic Verification ... 281
Jesper Amilon, Zafer Esen, Dilian Gurov, Christian Lidström, and Philipp Rümmer

Boolean Abstractions for Realizability Modulo Theories ... 305
Andoni Rodríguez and César Sánchez

Certified Verification for Algebraic Abstraction ... 329
Ming-Hsien Tsai, Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang

Complete Multiparty Session Type Projection with Automata ... 350
Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey

Early Verification of Legal Compliance via Bounded Satisfiability Checking ... 374
Nick Feng, Lina Marsso, Mehrdad Sabetzadeh, and Marsha Chechik

Formula Normalizations in Verification ... 398
Simon Guilloud, Mario Bucev, Dragana Milovančević, and Viktor Kunčak

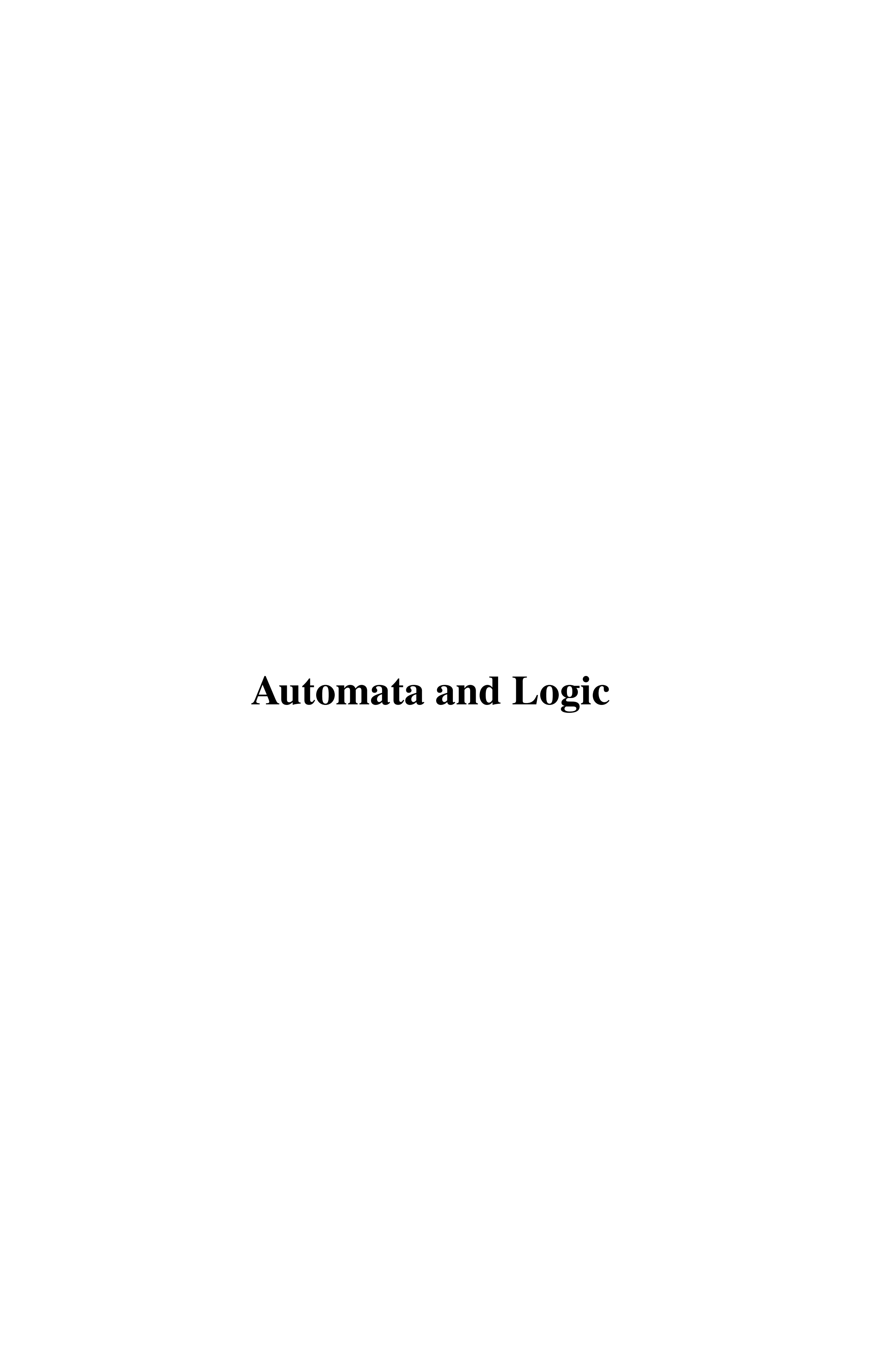
Kratos2: An SMT-Based Model Checker for Imperative Programs ... 423
Alberto Griggio and Martin Jonáš

Making IP = PSPACE Practical: Efficient Interactive Protocols for BDD Algorithms ... 437
Eszter Couillard, Philipp Czerner, Javier Esparza, and Rupak Majumdar

Ownership Guided C to Rust Translation 459
Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang

R2U2 Version 3.0: Re-Imagining a Toolchain for Specification, Resource Estimation, and Optimized Observer Generation for Runtime Verification in Hardware and Software 483
Chris Johannsen, Phillip Jones, Brian Kempa, Kristin Yvonne Rozier, and Pei Zhang

Author Index 499


Automata and Logic

Active Learning of Deterministic Timed Automata with Myhill-Nerode Style Characterization

Masaki Waga(✉)

Graduate School of Informatics, Kyoto University, Kyoto, Japan
mwaga@fos.kuis.kyoto-u.ac.jp

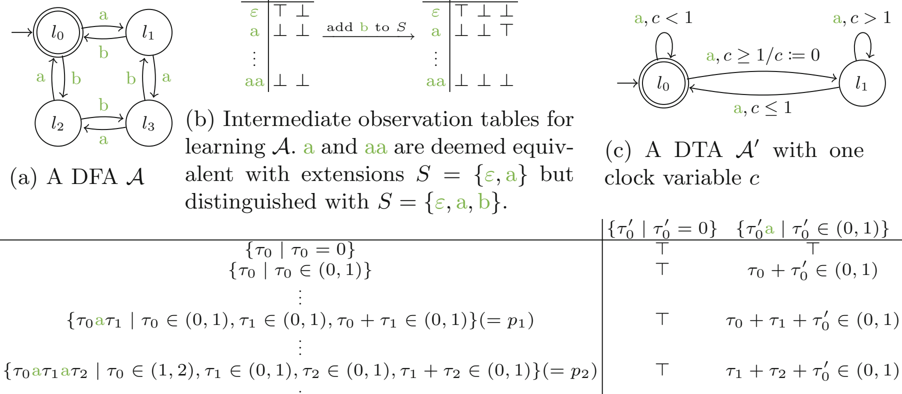
Abstract. We present an algorithm to learn a deterministic timed automaton (DTA) via membership and equivalence queries. Our algorithm is an extension of the L* algorithm with a Myhill-Nerode style characterization of recognizable timed languages, which is the class of timed languages recognizable by DTAs. We first characterize the recognizable timed languages with a Nerode-style congruence. Using it, we give an algorithm with a smart teacher answering *symbolic* membership queries in addition to membership and equivalence queries. With a symbolic membership query, one can ask the membership of a certain set of timed words at one time. We prove that for any recognizable timed language, our learning algorithm returns a DTA recognizing it. We show how to answer a symbolic membership query with finitely many membership queries. We also show that our learning algorithm requires a polynomial number of queries with a smart teacher and an exponential number of queries with a normal teacher. We applied our algorithm to various benchmarks and confirmed its effectiveness with a normal teacher.

Keywords: timed automata · active automata learning · recognizable timed languages · L* algorithm · observation table

1 Introduction

Active automata learning is a class of methods to infer an automaton recognizing an unknown target language $\mathcal{L}_{\mathrm{tgt}} \subseteq \Sigma^*$ through finitely many queries to a teacher. The L* algorithm [8], the best-known active DFA learning algorithm, infers the minimum DFA recognizing $\mathcal{L}_{\mathrm{tgt}}$ using *membership* and *equivalence* queries. In a membership query, the learner asks if a word $w \in \Sigma^*$ is in the target language $\mathcal{L}_{\mathrm{tgt}}$, which is used to obtain enough information to construct a hypothesis DFA $\mathcal{A}_{\mathrm{hyp}}$. Using an equivalence query, the learner checks if the hypothesis $\mathcal{A}_{\mathrm{hyp}}$ recognizes the target language $\mathcal{L}_{\mathrm{tgt}}$. If $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \neq \mathcal{L}_{\mathrm{tgt}}$, the teacher returns a counterexample $cex \in \mathcal{L}_{\mathrm{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$ differentiating the target language and the current hypothesis. The learner uses cex to update $\mathcal{A}_{\mathrm{hyp}}$ to classify cex correctly. Such a learning algorithm has been combined with formal verification, e. g., for testing [22,24,26,28] and controller synthesis [31].

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 3–26, 2023.
https://doi.org/10.1007/978-3-031-37706-8_1



(a) A DFA $\mathcal{A}$

	ε	a
ε	$\top$	$\bot$
a	$\bot$	$\bot$
$\vdots$		
aa	$\bot$	$\bot$

	ε	a	b
ε	$\top$	$\bot$	$\bot$
a	$\bot$	$\bot$	$\top$
$\vdots$			
aa	$\bot$	$\bot$	$\bot$

(b) Intermediate observation tables for learning $\mathcal{A}$. a and aa are deemed equivalent with extensions $S = \{\varepsilon, \mathrm{a}\}$ but distinguished with $S = \{\varepsilon, \mathrm{a}, \mathrm{b}\}$.

(c) A DTA $\mathcal{A}'$ with one clock variable c

	$\{\tau'_0 \mid \tau'_0 = 0\}$	$\{\tau'_0\mathrm{a} \mid \tau'_0 \in (0,1)\}$
$\{\tau_0 \mid \tau_0 = 0\}$	$\top$	$\top$
$\{\tau_0 \mid \tau_0 \in (0,1)\}$	$\top$	$\tau_0 + \tau'_0 \in (0,1)$
$\vdots$		
$\{\tau_0\mathrm{a}\tau_1 \mid \tau_0 \in (0,1), \tau_1 \in (0,1), \tau_0 + \tau_1 \in (0,1)\} (= p_1)$	$\top$	$\tau_0 + \tau_1 + \tau'_0 \in (0,1)$
$\vdots$		
$\{\tau_0\mathrm{a}\tau_1\mathrm{a}\tau_2 \mid \tau_0 \in (1,2), \tau_1 \in (0,1), \tau_2 \in (0,1), \tau_1 + \tau_2 \in (0,1)\} (= p_2)$	$\top$	$\tau_1 + \tau_2 + \tau'_0 \in (0,1)$
$\vdots$		

(d) Timed observation table for learning $\mathcal{A}'$. Each cell is indexed by a pair $(p, s) \in P \times S$ of elementary languages. The cell indexed by (p, s) shows a constraint Λ such that $w \in p \cdot s$ satisfies $w \in \mathcal{L}_{\mathrm{tgt}}$ if and only if Λ holds. Elementary languages p_1 and p_2 are deemed equivalent with the equation $\tau_0^1 + \tau_1^1 = \tau_1^2 + \tau_2^2$, where τ_i^j represents τ_i in p_j.

Fig. 1. Illustration of observation tables in the L* algorithm for DFA learning (Fig. 1b) and our algorithm for DTA learning (Fig. 1d)

Most of the DFA learning algorithms rely on the characterization of regular languages by *Nerode's congruence*. For a language $\mathcal{L}$, words p and p' are equivalent if for any extension s, $p \cdot s \in \mathcal{L}$ if and only if $p' \cdot s \in \mathcal{L}$. It is well known that if $\mathcal{L}$ is regular, such an equivalence relation has finite classes, corresponding to the locations of the minimum DFA recognizing $\mathcal{L}$ (known as *Myhill-Nerode theorem*; see, e. g., [18]). Moreover, for any regular language $\mathcal{L}$, there are finite extensions S such that p and p' are equivalent if and only if for any $s \in S$, $p \cdot s \in \mathcal{L}$ if and only if $p' \cdot s \in \mathcal{L}$. Therefore, one can learn the minimum DFA by learning such finite extensions S and the finite classes induced by Nerode's congruence.

The L* algorithm learns the minimum DFA recognizing the target language $\mathcal{L}_{\mathrm{tgt}}$ using a 2-dimensional array called an *observation table*. Figure 1b illustrates observation tables. The rows and columns of an observation table are indexed with finite sets of words P and S, respectively. Each cell indexed by $(p, s) \in P \times S$ shows if $p \cdot s \in \mathcal{L}_{\mathrm{tgt}}$. The column indices S are the current extensions approximating Nerode's congruence. The L* algorithm increases P and S until: 1) the equivalence relation defined by S converges to Nerode's congruence and 2) P covers all the classes induced by the congruence. The equivalence between $p, p' \in P$ under S can be checked by comparing the rows in the observation table indexed with p and p'. For example, Fig. 1b shows that a and aa are deemed equivalent with extensions $S = \{\varepsilon, \mathrm{a}\}$ but distinguished by adding b to S. The refinement of P and S is driven by certain conditions to validate the DFA construction and by addressing the counterexample obtained by an equivalence query.

Timed words are extensions of conventional words with real-valued dwell time between events. *Timed* languages, sets of timed words, are widely used to formalize real-time systems and their properties, e.g., for formal verification. Among various formalisms representing timed languages, *timed automata (TAs)* [4] is one of the widely used formalisms. A TA is an extension of an NFA with finitely many clock variables to represent timing constraints. Figure 1c shows an example.

Despite its practical relevance, learning algorithms for TAs are only available for limited subclasses of TAs, e.g., real-time automata [6,7], event-recording automata [15,16], event-recording automata with unobservable reset [17], and one-clock deterministic TAs [5,30]. Timing constraints representable by these classes are limited, e.g., by restricting the number of clock variables or by restricting the edges where a clock variable can be reset. Such restriction simplifies the inference of timing constraints in learning algorithms.

Contributions. In this paper, we propose an active learning algorithm for *deterministic* TAs (DTAs). The languages recognizable by DTAs are called *recognizable timed languages* [21]. Our strategy is as follows: first, we develop a Myhill-Nerode style characterization of recognizable timed languages; then, we extend the L* algorithm for recognizable timed languages using the similarity of the Myhill-Nerode style characterization.

Due to the continuity of dwell time in timed words, it is hard to characterize recognizable timed languages by a Nerode-style congruence between timed words. For example, for the DTA in Fig. 1c, for any $\tau, \tau' \in [0, 1)$ satisfying $\tau < \tau'$, $(1 - \tau')\mathrm{a}$ distinguishes τ and τ' because $\tau(1 - \tau')\mathrm{a}$ leads to l_0 while $\tau(1 - \tau)\mathrm{a}$ leads to l_1. Therefore, such a congruence can make *infinitely* many classes.

Instead, we define a Nerode-style congruence between sets of timed words called *elementary languages* [21]. An elementary language is a timed language defined by a word with a conjunction of inequalities constraining the time difference between events. We also use an equality constraint, which we call, a *renaming equation* to define the congruence. Intuitively, a renaming equation bridges the time differences in an elementary language and the clock variables in a TA. We note that there can be multiple renaming equations showing the equivalence of two elementary languages.

Example 1. Let p_1 and p_2 be elementary languages $p_1 = \{\tau_0^1 \mathrm{a} \tau_1^1 \mid \tau_0^1 \in (0, 1), \tau_1^1 \in (0, 1), \tau_0^1 + \tau_1^1 \in (0, 1)\}$ and $p_2 = \{\tau_0^2 \mathrm{a} \tau_1^2 \mathrm{a} \tau_2^2 \mid \tau_0^2 \in (1, 2), \tau_1^2 \in (0, 1), \tau_2^2 \in (0, 1), \tau_1^2 + \tau_2^2 \in (0, 1)\}$. For the DTA in Fig. 1c, p_1 and p_2 are equivalent with the renaming equation $\tau_0^1 + \tau_1^1 = \tau_1^2 + \tau_2^2$ because for any $w_1 = \tau_0^1 \mathrm{a} \tau_1^1 \in p_1$ and $w_2 = \tau_0^2 \mathrm{a} \tau_1^2 \mathrm{a} \tau_2^2 \in p_2$: 1) we reach l_0 after reading either of w_1 and w_2 and 2) the values of c after reading w_1 and w_2 are $\tau_0^1 + \tau_1^1$ and $\tau_1^2 + \tau_2^2$, respectively.

We characterize recognizable timed languages by the finiteness of the equivalence classes defined by the above congruence. We also show that for any recognizable timed language, there is a finite set S of elementary languages such that the equivalence of any prefixes can be checked by the extensions S.

By using the above congruence, we extend the L* algorithm for DTAs. The high-level idea is the same as the original L* algorithm: 1) the learner makes membership queries to obtain enough information to construct a hypothesis DTA $\mathcal{A}_{\mathrm{hyp}}$ and 2) the learner makes an equivalence query to check if $\mathcal{A}_{\mathrm{hyp}}$ recognizes the target language. The largest difference is in the cells of an observation table. Since the concatenation $p \cdot s$ of an index pair $(p, s) \in P \times S$ is not a timed word but a set of timed words, its membership is not defined as a Boolean value. Instead, we introduce the notion of *symbolic* membership and use it as the value of each cell of the *timed* observation table. Intuitively, the symbolic membership is the constraint representing the subset of $p \cdot s$ included by $\mathcal{L}_{\mathrm{tgt}}$. Such a constraint can be constructed by finitely many (non-symbolic) membership queries.

Example 2. Figure 1d illustrates a *timed* observation table. The equivalence between $p_1, p_2 \in P$ under S can be checked by comparing the cells in the rows indexed with p_1 and p_2 with renaming equations. For the cells in rows indexed by p_1 and p_2, their constraints are the same by replacing $\tau_0 + \tau_1$ with $\tau_1 + \tau_2$ and vice versa. Thus, p_1 and p_2 are equivalent with the current extensions S.

Once the learner obtains enough information, it constructs a DTA via the monoid-based representation of recognizable timed languages [21]. We show that for any recognizable timed language, our algorithm terminates and returns a DTA recognizing it. We also show that the number of the necessary queries is polynomial to the size of the equivalence class defined by the Nerode-style congruence if symbolic membership queries are allowed and, otherwise, exponential to it. Moreover, if symbolic membership queries are not allowed, the number of the necessary queries is at most doubly exponential to the number of the clock variable of a DTA recognizing the target language and singly exponential to the number of locations of a DTA recognizing the target language. This worst-case complexity is the same as the one-clock DTA learning algorithm in [30].

We implemented our DTA learning algorithm in a prototype library LearnTA. Our experiment results show that it is efficient enough for some benchmarks taken from practical applications, e. g., the FDDI protocol. This suggests the practical relevance of our algorithm.

The following summarizes our contribution.

- We characterize recognizable timed languages by a Nerode-style congruence.
- Using the above characterization, we give an active DTA learning algorithm.
- Our experiment results suggest its practical relevance.

Related Work. Among various characterization of timed languages [4,10–13,21], the characterization by *recognizability* [21] is closest to our Myhill-Nerode-style characterization. Both of them use finite sets of elementary languages for characterization. Their main difference is that [21] proposes a formalism to define a timed language by relating prefixes by a morphism, whereas we propose a technical gadget to define an equivalence relation over timed words with respect to suffixes using symbolic membership. This difference makes our definition suitable for an L*-style algorithm, where the original L* algorithm is based on Nerode's

congruence, which defines an equivalence relation over words with respect to suffixes using conventional membership.

As we have discussed so far, active TA learning [5,15–17,30] has been studied mostly for limited subclasses of TAs, where the number of the clock variables or the clock variables reset at each edge is fixed. In contrast, our algorithm infers both of the above information. Another difference is in the technical strategy. Most of the existing algorithms are related to the active learning of *symbolic automata* [9,14], enhancing the languages with clock valuations. In contrast, we take a more semantic approach via the Nerode-style congruence.

Another recent direction is to use a *genetic algorithm* to infer TAs in passive [27] or active [3] learning. This differs from our learning algorithm based on a formal characterization of timed languages. Moreover, these algorithms may not converge to the correct automaton due to a genetic algorithm.

2 Preliminaries

For a set X, its powerset is denoted by $\mathcal{P}(X)$. We denote the empty sequence by ε. For sets X, Y, we denote their symmetric difference by $X \triangle Y = \{x \mid x \in X \wedge x \notin Y\} \cup \{y \mid y \in Y \wedge y \notin X\}$.

2.1 Timed Words and Timed Automata

Definition 3 (timed word). *For a finite alphabet Σ, a* timed word *w is an alternating sequence $\tau_0 a_1 \tau_1 a_2 \dots a_n \tau_n$ of Σ and $\mathbb{R}_{\geq 0}$. The set of timed words over Σ is denoted by $\mathcal{T}(\Sigma)$. A* timed language *$\mathcal{L} \subseteq \mathcal{T}(\Sigma)$ is a set of timed words.*

For timed words $w = \tau_0 a_1 \tau_1 a_2 \dots a_n \tau_n$ and $w' = \tau'_0 a'_1 \tau'_1 a'_2 \dots a'_{n'} \tau'_{n'}$, their concatenation $w \cdot w'$ is $w \cdot w' = \tau_0 a_1 \tau_1 a_2 \dots a_n (\tau_n + \tau'_0) a'_1 \tau'_1 a'_2 \dots a'_{n'} \tau'_{n'}$. The concatenation is naturally extended to timed languages: for a timed word w and timed languages $\mathcal{L}, \mathcal{L}'$, we let $w \cdot \mathcal{L} = \{w \cdot w_{\mathcal{L}} \mid w_{\mathcal{L}} \in \mathcal{L}\}$, $\mathcal{L} \cdot w = \{w_{\mathcal{L}} \cdot w \mid w_{\mathcal{L}} \in \mathcal{L}\}$, and $\mathcal{L} \cdot \mathcal{L}' = \{w_{\mathcal{L}} \cdot w_{\mathcal{L}'} \mid w_{\mathcal{L}} \in \mathcal{L}, w_{\mathcal{L}'} \in \mathcal{L}'\}$. For timed words w and w', w is a *prefix* of w' if there is a timed word w'' satisfying $w \cdot w'' = w'$. A timed language $\mathcal{L}$ is *prefix-closed* if for any $w \in \mathcal{L}$, $\mathcal{L}$ contains all the prefixes of w.

For a finite set C of clock variables, a *clock valuation* is a function $\nu \in (\mathbb{R}_{\geq 0})^C$. We let $\mathbf{0}_C$ be the clock valuation satisfying $\mathbf{0}_C(c) = 0$ for any $c \in C$. For $\nu \in (\mathbb{R}_{\geq 0})^C$ and $\tau \in \mathbb{R}_{\geq 0}$, we let $\nu + \tau$ be the clock valuation satisfying $(\nu + \tau)(c) = \nu(c) + \tau$ for any $c \in C$. For $\nu \in (\mathbb{R}_{\geq 0})^C$ and $\rho \subseteq C$, we let $\nu[\rho := 0]$ be the clock valuation satisfying $(\nu[\rho := 0])(x) = 0$ for $c \in \rho$ and $(\nu[\rho := 0])(c) = \nu(c)$ for $c \notin \rho$. We let $\mathcal{G}_C$ be the set of constraints defined by a finite conjunction of inequalities $c \bowtie d$, where $c \in C$, $d \in \mathbb{N}$, and $\bowtie \in \{>, \geq, \leq, <\}$. We let $\mathcal{C}_C$ be the set of constraints defined by a finite conjunction of inequalities $c \bowtie d$ or $c - c' \bowtie d$, where $c, c' \in C$, $d \in \mathbb{N}$, and $\bowtie \in \{>, \geq, \leq, <\}$. We denote $\bigwedge \emptyset$ by $\top$. For $\nu \in (\mathbb{R}_{\geq 0})^C$ and $\varphi \in \mathcal{C}_C \cup \mathcal{G}_C$, we denote $\nu \models \varphi$ if ν satisfies φ.

Definition 4 (timed automaton). *A* timed automaton *(TA) is a 7-tuple $(\Sigma, L, l_0, C, I, \Delta, F)$, where: Σ is the finite alphabet, L is the finite set of locations, $l_0 \in L$ is the initial location, C is the finite set of clock variables, $I \colon L \to \mathcal{C}_C$ is the invariant of each location, $\Delta \subseteq L \times \mathcal{G}_C \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{P}(C) \times L$ is the set of edges, and $F \subseteq L$ is the accepting locations.*

A TA is *deterministic* if 1) for any $a \in \Sigma$ and $(l, g, a, \rho, l'), (l, g', a, \rho', l'') \in \Delta$, $g \wedge g'$ is unsatisfiable, or 2) for any $(l, g, \varepsilon, \rho, l') \in \Delta$, $g \wedge I(l)$ is at most a singleton. Figure 1c shows a deterministic TA (DTA).

The semantics of a TA is defined by a *timed transition system (TTS)*.

Definition 5 (semantics of TAs). *For a TA $\mathcal{A} = (\Sigma, L, l_0, C, I, \Delta, F)$, the* timed transition system (TTS) *is a 4-tuple $\mathcal{S} = (Q, q_0, Q_F, \to)$, where: $Q = L \times (\mathbb{R}_{\geq 0})^C$ is the set of* (concrete) states*, $q_0 = (l_0, \mathbf{0}_C)$ is the* initial state*, $Q_F = \{(l, \nu) \in Q \mid l \in F\}$ is the set of* accepting states*, and $\to \subseteq Q \times Q$ is the* transition relation *consisting of the following*[1]*.*

- *For each $(l, \nu) \in Q$ and $\tau \in \mathbb{R}_{>0}$, we have $(l, \nu) \xrightarrow{\tau} (l, \nu + \tau)$ if $\nu + \tau' \models I(l)$ holds for each $\tau' \in [0, \tau)$.*
- *For each $(l, \nu), (l', \nu') \in Q$, $a \in \Sigma$, and $(l, g, a, \rho, l') \in \Delta$, we have $(l, \nu) \xrightarrow{a} (l', \nu')$ if we have $\nu \models g$ and $\nu' = \nu[\rho := 0]$.*
- *For each $(l, \nu), (l', \nu') \in Q$, $\tau \in \mathbb{R}_{>0}$, and $(l, g, \varepsilon, \rho, l') \in \Delta$, we have $(l, \nu) \xrightarrow{\varepsilon, \tau} (l', \nu' + \tau)$ if we have $\nu \models g$, $\nu' = \nu[\rho := 0]$, and $\forall \tau' \in [0, \tau). \nu' + \tau' \models I(l')$.*

A *run* of a TA $\mathcal{A}$ is an alternating sequence $q_0, \to_1, q_1, \dots, \to_n, q_n$ of $q_i \in Q$ and $\to_i \in \to$ satisfying $q_{i-1} \to_i q_i$ for any $i \in \{1, 2, \dots, n\}$. A run $q_0, \to_1, q_1, \dots, \to_n, q_n$ is accepting if $q_n \in Q_F$. Given such a run, the associated timed word is the concatenation of the labels of the transitions. The timed *language* $\mathcal{L}(\mathcal{A})$ of a TA $\mathcal{A}$ is the set of timed words associated with some accepting run of $\mathcal{A}$.

2.2 Recognizable Timed Languages

Here, we review the *recognizability* [21] of timed languages.

Definition 6 (timed condition). *For a set $\mathbb{T} = \{\tau_0, \tau_1, \dots, \tau_n\}$ of ordered variables, a* timed condition *Λ is a finite conjunction of inequalities $\mathbb{T}_{i,j} \bowtie d$, where $\mathbb{T}_{i,j} = \sum_{k=i}^{j} \tau_k$, $\bowtie \in \{>, \geq, \leq, <\}$, and $d \in \mathbb{N}$.*

A timed condition Λ is *simple*[2] if for each $\mathbb{T}_{i,j}$, Λ contains $d < \mathbb{T}_{i,j} < d+1$ or $d \leq \mathbb{T}_{i,j} \wedge \mathbb{T}_{i,j} \leq d$ for some $d \in \mathbb{N}$. A timed condition Λ is *canonical* if we cannot strengthen or add any inequality $\mathbb{T}_{i,j} \bowtie d$ to Λ without changing its semantics.

---

[1] We use $\xrightarrow{\varepsilon, \tau}$ to avoid the discussion with an arbitrary small dwell time in [21].

[2] The notion of simplicity is taken from [15].

Definition 7 (elementary language). *A timed language $\mathcal{L}$ is* elementary *if there are $u = a_1, a_2, \ldots, a_n \in \Sigma^*$ and a timed condition Λ over $\{\tau_0, \tau_1, \ldots, \tau_n\}$ satisfying $\mathcal{L} = \{\tau_0 a_1 \tau_1 a_2 \ldots a_n \tau_n \mid \tau_0, \tau_1, \ldots, \tau_n \models \Lambda\}$, and the set of valuations of $\{\tau_0, \tau_1, \ldots, \tau_n\}$ defined by Λ is bounded. We denote such $\mathcal{L}$ by (u, Λ). We let $\mathcal{E}(\Sigma)$ be the set of elementary languages over Σ.*

For $p, p' \in \mathcal{E}(\Sigma)$, p is a *prefix* of p' if for any $w' \in p'$, there is a prefix $w \in p$ of w', and for any $w \in p$, there is $w' \in p'$ such that w is a prefix of w'. For any elementary language, the number of its prefixes is finite. For a set of elementary languages, *prefix-closedness* is defined based on the above definition of prefixes.

An elementary language (u, Λ) is *simple* if there is a simple and canonical timed condition Λ' satisfying $(u, \Lambda) = (u, \Lambda')$. We let $\mathcal{SE}(\Sigma)$ be the set of simple elementary languages over Σ. Without loss of generality, we assume that for any $(u, \Lambda) \in \mathcal{SE}(\Sigma)$, Λ is simple and canonical. We remark that any DTA cannot distinguish timed words in a simple elementary language, i. e., for any $p \in \mathcal{SE}(\Sigma)$ and a DTA $\mathcal{A}$, we have either $p \subseteq \mathcal{L}(\mathcal{A})$ or $p \cap \mathcal{L}(\mathcal{A}) = \emptyset$. We can decide if $p \subseteq \mathcal{L}(\mathcal{A})$ or $p \cap \mathcal{L}(\mathcal{A}) = \emptyset$ by taking some $w \in p$ and checking if $w \in \mathcal{L}(\mathcal{A})$.

Definition 8 (immediate exterior). *Let $\mathcal{L} = (u, \Lambda)$ be an elementary language. For $a \in \Sigma$, the* discrete immediate exterior $\mathrm{ext}^a(\mathcal{L})$ *of $\mathcal{L}$ is $\mathrm{ext}^a(\mathcal{L}) = (u \cdot a, \Lambda \cup \{\tau_{|u|+1} = 0\})$. The* continuous immediate exterior $\mathrm{ext}^t(\mathcal{L})$ *of $\mathcal{L}$ is $\mathrm{ext}^t(\mathcal{L}) = (u, \Lambda^t)$, where Λ^t is the timed condition such that each inequality $\mathbb{T}_{i,|u|} = d$ in Λ is replaced with $\mathbb{T}_{i,|u|} > d$ if such an inequality exists, and otherwise, the inequality $\mathbb{T}_{i,|u|} < d$ in Λ with the smallest index i is replaced with $\mathbb{T}_{i,|u|} = d$. The* immediate exterior *of $\mathcal{L}$ is $\mathrm{ext}(\mathcal{L}) = \bigcup_{a \in \Sigma} \mathrm{ext}^a(\mathcal{L}) \cup \mathrm{ext}^t(\mathcal{L})$.*

Example 9. For a word $u = \mathrm{a} \cdot \mathrm{a}$ and a timed condition $\Lambda = \{\mathbb{T}_{0,0} \in (1,2) \wedge \mathbb{T}_{0,1} \in (1,2) \wedge \mathbb{T}_{0,2} \in (1,2) \wedge \mathbb{T}_{1,2} \in (0,1) \wedge \mathbb{T}_{2,2} = 0\}$, we have $1.3 \cdot \mathrm{a} \cdot 0.5 \cdot \mathrm{a} \cdot 0 \in (u, \Lambda)$ and $1.7 \cdot \mathrm{a} \cdot 0.5 \cdot \mathrm{a} \cdot 0 \notin (u, \Lambda)$. The discrete and continuous immediate exteriors of (u, Λ) are $\mathrm{ext}^{\mathrm{a}}((u, \Lambda)) = (u \cdot \mathrm{a}, \Lambda^{\mathrm{a}})$ and $\mathrm{ext}^t((u, \Lambda)) = (u, \Lambda^t)$, where $\Lambda^{\mathrm{a}} = \{\mathbb{T}_{0,0} \in (1,2) \wedge \mathbb{T}_{0,1} \in (1,2) \wedge \mathbb{T}_{0,2} \in (1,2) \wedge \mathbb{T}_{1,2} \in (0,1) \wedge \mathbb{T}_{2,2} = \mathbb{T}_{3,3} = 0\}$ and $\Lambda^t = \{\mathbb{T}_{0,0} \in (1,2) \wedge \mathbb{T}_{0,1} \in (1,2) \wedge \mathbb{T}_{0,2} \in (1,2) \wedge \mathbb{T}_{1,2} \in (0,1) \wedge \mathbb{T}_{2,2} > 0\}$.

Definition 10 (chronometric timed language). *A timed language $\mathcal{L}$ is* chronometric *if there is a finite set $\{(u_1, \Lambda_1), (u_2, \Lambda_2), \ldots, (u_m, \Lambda_m)\}$ of disjoint elementary languages satisfying $\mathcal{L} = \bigcup_{i \in \{1,2,\ldots,m\}} (u_i, \Lambda_i)$.*

For any elementary language $\mathcal{L}$, its immediate exterior $\mathrm{ext}(\mathcal{L})$ is chronometric. We naturally extend the notion of exterior to chronometric timed languages, i. e., for a chronometric timed language $\mathcal{L} = \bigcup_{i \in \{1,2,\ldots,m\}} (u_i, \Lambda_i)$, we let $\mathrm{ext}(\mathcal{L}) = \bigcup_{i \in \{1,2,\ldots,m\}} \mathrm{ext}((u_i, \Lambda_i))$, which is also chronometric. For a timed word $w = \tau_0 a_1 \tau_1 a_2 \ldots a_n \tau_n$, we denote the valuation of $\tau_0, \tau_1, \ldots, \tau_n$ by $\kappa(w)$.

Chronometric relational morphism [21] relates any timed word to a timed word in a certain set P, which is later used to define a timed language. Intuitively, the tuples in Φ specify a mapping from timed words immediately out of P to timed words in P. By inductively applying it, any timed word is mapped to P.

Definition 11 (chronometric relational morphism). *Let P be a chronometric and prefix-closed timed language. Let $(u, \Lambda, u', \Lambda', R)$ be a 5-tuple such that $(u, \Lambda) \subseteq \mathrm{ext}(P)$, $(u', \Lambda') \subseteq P$, and R is a finite conjunction of equations of the form $\mathbb{T}_{i,|u|} = \mathbb{T}'_{j,|u'|}$, where $i \leq |u|$ and $j \leq |u'|$. For such a tuple, we let $[\![(u, \Lambda, u', \Lambda', R)]\!] \subseteq (u, \Lambda) \times (u', \Lambda')$ be the relation such that $(w, w') \in [\![(u, \Lambda, u', \Lambda', R)]\!]$ if and only if $\kappa(w), \kappa(w') \models R$. For a finite set Φ of such tuples, the* chronometric relational morphism *$[\![\Phi]\!] \subseteq \mathcal{T}(\Sigma) \times P$ is the relation inductively defined as follows: 1) for $w \in P$, we have $(w, w) \in [\![\Phi]\!]$; 2) for $w \in \mathrm{ext}(P)$ and $w' \in P$, we have $(w, w') \in [\![\Phi]\!]$ if we have $(w, w') \in [\![(u, \Lambda, u', \Lambda', R)]\!]$ for one of the tuples $(u, \Lambda, u', \Lambda', R) \in \Phi$; 3) for $w \in \mathrm{ext}(P)$, $w' \in \mathcal{T}(\Sigma)$, and $w'' \in P$, we have $(w \cdot w', w'') \in [\![\Phi]\!]$ if there is $w''' \in \mathcal{T}(\Sigma)$ satisfying $(w, w''') \in [\![\Phi]\!]$ and $(w''' \cdot w', w'') \in [\![\Phi]\!]$. We also require that all (u, Λ) in the tuples in Φ must be disjoint and the union of each such (u, Λ) is $\mathrm{ext}(P) \setminus P$.*

A chronometric relational morphism $[\![\Phi]\!]$ is *compatible* with $F \subseteq P$ if for each tuple $(u, \Lambda, u', \Lambda', R)$ defining $[\![\Phi]\!]$, we have either $(u', \Lambda') \subseteq F$ or $(u', \Lambda') \cap F = \emptyset$.

Definition 12 (recognizable timed language). *A timed language $\mathcal{L}$ is* recognizable *if there is a chronometric prefix-closed set P, a chronometric subset F of P, and a chronometric relational morphism $[\![\Phi]\!] \subseteq \mathcal{T}(\Sigma) \times P$ compatible with F satisfying $\mathcal{L} = \{w \mid \exists w' \in F, (w, w') \in [\![\Phi]\!]\}$.*

It is known that for any recognizable timed language $\mathcal{L}$, we can construct a DTA $\mathcal{A}$ recognizing $\mathcal{L}$, and vice versa [21].

2.3 Distinguishing Extensions and Active DFA Learning

Most DFA learning algorithms are based on *Nerode's congruence* [18]. For a (not necessarily regular) language $\mathcal{L} \subseteq \Sigma^*$, Nerode's congruence $\equiv_{\mathcal{L}} \subseteq \Sigma^* \times \Sigma^*$ is the equivalence relation satisfying $w \equiv_{\mathcal{L}} w'$ if and only if for any $w'' \in \Sigma^*$, we have $w \cdot w'' \in \mathcal{L} \iff w' \cdot w'' \in \mathcal{L}$.

Generally, we cannot decide if $w \equiv_{\mathcal{L}} w'$ by testing because it requires infinitely many membership checking. However, if $\mathcal{L}$ is regular, there is a finite set of suffixes $S \subseteq \Sigma^*$ called *distinguishing extensions* satisfying $\equiv_{\mathcal{L}} = \sim^S_{\mathcal{L}}$, where $\sim^S_{\mathcal{L}}$ is the equivalence relation satisfying $w \sim^S_{\mathcal{L}} w'$ if and only if for any $w'' \in S$, we have $w \cdot w'' \in \mathcal{L} \iff w' \cdot w'' \in \mathcal{L}$. Thus, the minimum DFA recognizing $\mathcal{L}_{\mathrm{tgt}}$ can be learned by[3]: i) identifying distinguishing extensions S satisfying $\equiv_{\mathcal{L}_{\mathrm{tgt}}} = \sim^S_{\mathcal{L}_{\mathrm{tgt}}}$ and ii) constructing the minimum DFA $\mathcal{A}$ corresponding to $\sim^S_{\mathcal{L}_{\mathrm{tgt}}}$.

The L^* *algorithm* [8] is an algorithm to learn the minimum DFA $\mathcal{A}_{\mathrm{hyp}}$ recognizing the target regular language $\mathcal{L}_{\mathrm{tgt}}$ with finitely many *membership* and *equivalence* queries to the teacher. In a membership query, the learner asks if $w \in \Sigma^*$ belongs to the target language $\mathcal{L}_{\mathrm{tgt}}$ i. e., $w \in \mathcal{L}_{\mathrm{tgt}}$. In an equivalence query, the learner asks if the hypothesis DFA $\mathcal{A}_{\mathrm{hyp}}$ recognizes the target language

[3] The distinguishing extensions S can be defined locally. For example, the TTT algorithm [19] is optimized with *local* distinguishing extensions for some prefixes $w \in \Sigma^*$. Nevertheless, we use the global distinguishing extensions for simplicity.

Algorithm 1: Outline of an L*-style active DFA learning algorithm

```
1  P ← {ε}; S ← {ε}
2  while ⊤ do
3      while the observation table is not closed or consistent do
4          update P and S so that the observation table is closed and consistent
5      A_hyp ← ConstructDFA(P, S, T)
6      switch eq_{L_tgt}(A_hyp) do
7          case ⊤ do
8              return A_hyp
9          case cex do
10             Update P and/or S using cex
```

$\mathcal{L}_{\mathrm{tgt}}$ i. e., $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$, where $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$ is the language of the hypothesis DFA $\mathcal{A}_{\mathrm{hyp}}$. When we have $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \neq \mathcal{L}_{\mathrm{tgt}}$, the teacher returns a counterexample $\mathit{cex} \in \mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \triangle \mathcal{L}_{\mathrm{tgt}}$. The information obtained via queries is stored in a 2-dimensional array called an *observation table*. See Fig. 1b for an illustration. For finite index sets $P, S \subseteq \Sigma^*$, for each pair $(p, s) \in (P \cup P \cdot \Sigma) \times S$, the observation table stores whether $p \cdot s \in \mathcal{L}_{\mathrm{tgt}}$. S is the current candidate of the distinguishing extensions, and P represents $\Sigma^* / {\sim^S_{\mathcal{L}_{\mathrm{tgt}}}}$. Since P and S are finite, one can fill the observation table using finite membership queries.

Algorithm 1 outlines an L*-style algorithm. We start from $P = S = \{\varepsilon\}$ and incrementally increase them. To construct a hypothesis DFA $\mathcal{A}_{\mathrm{hyp}}$, the observation table must be *closed* and *consistent*. An observation table is *closed* if, for each $p \in P \cdot \Sigma$, there is $p' \in P$ satisfying $p \sim^S_{\mathcal{L}_{\mathrm{tgt}}} p'$. An observation table is *consistent* if, for any $p, p' \in P \cup P \cdot \Sigma$ and $a \in \Sigma$, $p \sim^S_{\mathcal{L}_{\mathrm{tgt}}} p'$ implies $p \cdot a \sim^S_{\mathcal{L}_{\mathrm{tgt}}} p' \cdot a$.

Once the observation table becomes closed and consistent, the learner constructs a hypothesis DFA $\mathcal{A}_{\mathrm{hyp}}$ and checks if $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$ by an equivalence query. If $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$ holds, $\mathcal{A}_{\mathrm{hyp}}$ is the resulting DFA. Otherwise, the teacher returns $\mathit{cex} \in \mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \triangle \mathcal{L}_{\mathrm{tgt}}$, which is used to refine the observation table. There are several variants of the refinement. In the L* algorithm, all the prefixes of *cex* are added to P. In the Rivest-Schapire algorithm [20, 25], an extension s strictly refining S is obtained by an analysis of *cex*, and such s is added to S.

3 A Myhill-Nerode Style Characterization of Recognizable Timed Languages with Elementary Languages

Unlike the case of regular languages, any finite set of timed words cannot correctly distinguish recognizable timed languages due to the infiniteness of dwell time in timed words. Instead, we use a finite set of *elementary languages* to define a Nerode-style congruence. To define the Nerode-style congruence, we extend the notion of membership to elementary languages.

Definition 13 (symbolic membership). *For a timed language $\mathcal{L} \subseteq \mathcal{T}(\Sigma)$ and an elementary language $(u, \Lambda) \in \mathcal{E}(\Sigma)$, the* symbolic membership $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}}((u, \Lambda))$ *of (u, Λ) to $\mathcal{L}$ is the strongest constraint such that for any $w \in (u, \Lambda)$, we have $w \in \mathcal{L}$ if and only if $\kappa(w) \models \mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}}(\mathcal{L})$.*

We discuss how to obtain symbolic membership in Sect. 4.5. We define a Nerode-style congruence using symbolic membership. A naive idea is to distinguish two elementary languages by the equivalence of their symbolic membership. However, this does not capture the semantics of TAs. For example, for the DTA $\mathcal{A}$ in Fig. 1c, for any timed word w, we have $1.3 \cdot \mathrm{a} \cdot 0.4 \cdot w \in \mathcal{L}(\mathcal{A}) \iff 0.3 \cdot \mathrm{a} \cdot 1.0 \cdot \mathrm{a} \cdot 0.4 \cdot w \in \mathcal{L}(\mathcal{A})$, while they have different symbolic membership. This is because symbolic membership distinguishes the *position* in timed words where each clock variable is reset, which must be ignored. We use *renaming equations* to abstract such positional information in symbolic membership. Note that $\mathbb{T}_{i,n} = \sum_{k=i}^{n} \tau_k$ corresponds to the value of the clock variable reset at τ_i.

Definition 14 (renaming equation). *Let $\mathbb{T} = \{\tau_0, \tau_1, \dots, \tau_n\}$ and $\mathbb{T}' = \{\tau_0^{'}, \tau_1^{'}, \dots, \tau_{n'}^{'}\}$ be sets of ordered variables. A* renaming equation *R over $\mathbb{T}$ and $\mathbb{T}'$ is a finite conjunction of equations of the form $\mathbb{T}_{i,n} = \mathbb{T}^{'}_{i',n'}$, where $i \in \{0, 1, \dots, n\}$, $i' \in \{0, 1, \dots, n'\}$, $\mathbb{T}_{i,n} = \sum_{k=i}^{n} \tau_k$ and $\mathbb{T}^{'}_{i',n'} = \sum_{k=i'}^{n'} \tau'_k$.*

Definition 15 ($\sim^{S}_{\mathcal{L}}$). *Let $\mathcal{L} \subseteq \mathcal{T}(\Sigma)$ be a timed language, let $(u, \Lambda), (u', \Lambda'), (u'', \Lambda'') \in \mathcal{E}(\Sigma)$ be elementary languages, and let R be a renaming equation over $\mathbb{T}$ and $\mathbb{T}'$, where $\mathbb{T}$ and $\mathbb{T}'$ are the variables of Λ and Λ', respectively. We let $(u, \Lambda) \sqsubseteq^{(u'',\Lambda''),R}_{\mathcal{L}} (u', \Lambda')$ if we have the following: for any $w \in (u, \Lambda)$, there is $w' \in (u', \Lambda')$ satisfying $\kappa(w), \kappa(w') \models R$; $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}}((u, \Lambda) \cdot (u'', \Lambda'')) \land R \land \Lambda'$ is equivalent to $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}}((u', \Lambda') \cdot (u'', \Lambda'')) \land R \land \Lambda$. We let $(u, \Lambda) \sim^{(u'',\Lambda''),R}_{\mathcal{L}} (u', \Lambda')$ if we have $(u, \Lambda) \sqsubseteq^{(u'',\Lambda''),R}_{\mathcal{L}} (u', \Lambda')$ and $(u', \Lambda' \sqsubseteq^{(u'',\Lambda''),R}_{\mathcal{L}} (u, \Lambda)$. Let $S \subseteq \mathcal{E}(\Sigma)$. We let $(u, \Lambda) \sim^{S,R}_{\mathcal{L}} (u', \Lambda')$ if for any $(u'', \Lambda'') \in S$, we have $(u, \Lambda) \sim^{(u'',\Lambda''),R}_{\mathcal{L}} (u', \Lambda')$. We let $(u, \Lambda) \sim^{S}_{\mathcal{L}} (u', \Lambda')$ if $(u, \Lambda) \sim^{S,R}_{\mathcal{L}} (u', \Lambda')$ for some renaming equation R.*

Example 16. Let $\mathcal{A}$ be the DTA in Fig. 1c and let (u, Λ), (u', Λ'), and (u'', Λ'') be elementary languages, where $u = \mathrm{a}$, $\Lambda = \{\tau_0 \in (1, 2) \land \tau_0 + \tau_1 \in (1, 2) \land \tau_1 \in (0, 1)\}$, $u' = \mathrm{a} \cdot \mathrm{a}$, $\Lambda' = \{\tau'_0 \in (0, 1) \land \tau'_0 + \tau'_1 \in (1, 2) \land \tau'_1 + \tau'_2 \in (1, 2) \land \tau'_2 \in (0, 1)\}$, $u'' = \mathrm{a}$, and $\Lambda'' = \{\tau_0 \in (0, 1) \land \tau_1 = 0\}$. We have $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}(\mathcal{A})}((u, \Lambda) \cdot (u'', \Lambda'')) = \Lambda \land \Lambda'' \land \tau_1 + \tau''_0 \leq 1$ and $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}(\mathcal{A})}((u', \Lambda') \cdot (u'', \Lambda'')) = \Lambda' \land \Lambda'' \land \tau'_2 + \tau''_0 \leq 1$. Therefore, for the renaming equation $\mathbb{T}_{1,1} = \mathbb{T}^{'}_{2,2}$, we have $(u, \Lambda) \sim^{(u'',\Lambda''),\mathbb{T}_{1,1}=\mathbb{T}^{'}_{2,2}}_{\mathcal{L}} (u', \Lambda')$.

An algorithm to check if $(u, \Lambda) \sim^{S}_{\mathcal{L}} (u', \Lambda')$ is shown in Appendix B.2 of [29].

Intuitively, $(u, \Lambda) \sqsubseteq^{s,R}_{\mathcal{L}} (u', \Lambda')$ shows that any $w \in (u, \Lambda)$ can be "simulated" by some $w' \in (u', \Lambda')$ with respect to s and R. Such intuition is formalized as follows.

Theorem 17. *For any $\mathcal{L} \subseteq \mathcal{T}(\Sigma)$ and $(u, \Lambda), (u', \Lambda'), (u'', \Lambda'') \in \mathcal{E}(\Sigma)$ satisfying $(u, \Lambda) \sqsubseteq_{\mathcal{L}}^{(u'',\Lambda'')} (u', \Lambda')$, for any $w \in (u, \Lambda)$, there is $w' \in (u', \Lambda')$ such that for any $w'' \in (u'', \Lambda'')$, $w \cdot w'' \in \mathcal{L} \iff w' \cdot w'' \in \mathcal{L}$ holds.* □

By $\bigcup_{(u,\Lambda)\in\mathcal{E}(\Sigma)}(u, \Lambda) = \mathcal{T}(\Sigma)$, we have the following as a corollary.

Corollary 18. *For any timed language $\mathcal{L} \subseteq \mathcal{T}(\Sigma)$ and for any elementary languages $(u, \Lambda), (u', \Lambda') \in \mathcal{E}(\Sigma)$, $(u, \Lambda) \sim_{\mathcal{L}}^{\mathcal{E}(\Sigma)} (u', \Lambda')$ implies the following.*

- *For any $w \in (u, \Lambda)$, there is $w' \in (u', \Lambda')$ such that for any $w'' \in \mathcal{T}(\Sigma)$, we have $w \cdot w'' \in \mathcal{L} \iff w' \cdot w'' \in \mathcal{L}$.*
- *For any $w' \in (u', \Lambda')$, there is $w \in (u, \Lambda)$ such that for any $w'' \in \mathcal{T}(\Sigma)$, we have $w \cdot w'' \in \mathcal{L} \iff w' \cdot w'' \in \mathcal{L}$.* □

The following characterizes recognizable timed languages with $\sim_{\mathcal{L}}^{\mathcal{E}(\Sigma)}$.

Theorem 19. (Myhill-Nerode style characterization). *A timed language $\mathcal{L}$ is recognizable if and only if the quotient set $\mathcal{SE}(\Sigma)/{\sim_{\mathcal{L}}^{\mathcal{E}(\Sigma)}}$ is finite.* □

By Theorem 19, we always have a finite set S of distinguishing extensions.

Theorem 20. *For any recognizable timed language $\mathcal{L}$, there is a finite set S of elementary languages satisfying $\sim_{\mathcal{L}}^{\mathcal{E}(\Sigma)} = \sim_{\mathcal{L}}^{S}$.* □

4 Active Learning of Deterministic Timed Automata

We show our L*-style active learning algorithm for DTAs with the Nerode-style congruence in Sect. 3. We let $\mathcal{L}_{\mathrm{tgt}}$ be the target timed language in learning.

For simplicity, we first present our learning algorithm with a smart teacher answering the following three kinds of queries: membership query $\mathtt{mem}_{\mathcal{L}_{\mathrm{tgt}}}(w)$ asking whether $w \in \mathcal{L}_{\mathrm{tgt}}$, symbolic membership query asking $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}_{\mathrm{tgt}}}((u, \Lambda))$, and equivalence query $\mathtt{eq}_{\mathcal{L}_{\mathrm{tgt}}}(\mathcal{A}_{\mathrm{hyp}})$ asking whether $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$. If $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$, $\mathtt{eq}_{\mathcal{L}_{\mathrm{tgt}}}(\mathcal{A}_{\mathrm{hyp}}) = \top$, and otherwise, $\mathtt{eq}_{\mathcal{L}_{\mathrm{tgt}}}(\mathcal{A}_{\mathrm{hyp}})$ is a timed word $cex \in \mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \triangle \mathcal{L}_{\mathrm{tgt}}$. Later in Sect. 4.5, we show how to answer a symbolic membership query with finitely many membership queries. Our task is to construct a DTA $\mathcal{A}$ satisfying $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{\mathrm{tgt}}$ with finitely many queries.

4.1 Successors of Simple Elementary Languages

Similarly to the L* algorithm in Sect. 2.3, we learn a DTA with an observation table. Reflecting the extension of the underlying congruence, we use sets of simple elementary languages for the indices. To define the extensions, $P \cup (P \cdot \Sigma)$ in the L* algorithm, we introduce *continuous* and *discrete successors* for simple elementary languages, which are inspired by *regions* [4]. We note that immediate exteriors do not work for this purpose. For example, for $(u, \Lambda) = (a, \{\mathbb{T}_{0,1} < 2 \wedge \mathbb{T}_{1,1} < 1\})$ and $w = 0.7 \cdot a \cdot 0.9$, we have $w \in (u, \Lambda)$ and $\mathrm{ext}^t((u, \Lambda)) = (a, \{\mathbb{T}_{0,1} = 2 \wedge \mathbb{T}_{1,1} < 1\})$, but there is no $t > 0$ satisfying $w \cdot t \in \mathrm{ext}^t((u, \Lambda))$.

Algorithm 2: DTA construction from a timed observation table

Input : A cohesive timed observation table (P, S, T)
Output : A DTA $\mathcal{A}_{\mathrm{hyp}}$ row-faithful to the given timed observation table

```
1 Function MakeDTA(P, S, T):
2     Φ ← ∅; F ← {(u, Λ) ∈ P | T((u, Λ), (ε, τ'_0 = 0)) = {Λ ∧ τ'_0 = 0}}
3     for p ∈ P such that succ^t(p) ∉ P (resp. succ^a(p) ∉ P) do
          // Construct (u, Λ, u', Λ', R) for some p' ∈ P and R
          // Such R is chosen using an exhaustive search
4         pick p' ∈ P and R such that succ^t(p) ~^{S,R}_{L_tgt} p' (resp. succ^a(p) ~^{S,R}_{L_tgt} p')
5         add (u, Λ, u', Λ', R) to Φ, where (u, Λ) = ext^t(p) (resp. ext^a(p)) and (u', Λ') = p'
6     return the DTA A_hyp obtained from (P, F, Φ) by the construction in [21]
```

For any $(u, \Lambda) \in \mathcal{SE}(\Sigma)$, we let $\Theta_{(u,\Lambda)}$ be the total order over 0 and the fractional parts $\mathrm{frac}(\mathbb{T}_{0,n}), \mathrm{frac}(\mathbb{T}_{1,n}), \dots, \mathrm{frac}(\mathbb{T}_{n,n})$ of $\mathbb{T}_{0,n}, \mathbb{T}_{1,n}, \dots, \mathbb{T}_{n,n}$. Such an order is uniquely defined because Λ is simple and canonical (Proposition 36 of [29]).

Definition 21 (successor). *Let $p = (u, \Lambda) \in \mathcal{SE}(\Sigma)$ be a simple elementary language. The* discrete successor $\mathrm{succ}^a(p)$ *of p is* $\mathrm{succ}^a(p) = (u \cdot a, \Lambda \wedge \tau_{n+1} = 0)$*. The* continuous successor $\mathrm{succ}^t(p)$ *of p is* $\mathrm{succ}^t(p) = (u, \Lambda^t)$*, where Λ^t is defined as follows: if there is an equation $\mathbb{T}_{i,n} = d$ in Λ, all such equations are replaced with $\mathbb{T}_{i,n} \in (d, d+1)$; otherwise, for each greatest $\mathbb{T}_{i,n}$ in terms of $\Theta_{(u,\Lambda)}$, we replace $\mathbb{T}_{i,n} \in (d, d+1)$ with $\mathbb{T}_{i,n} = d+1$. We let* $\mathrm{succ}(p) = \bigcup_{a \in \Sigma} \mathrm{succ}^a(p) \cup \mathrm{succ}^t(p)$*. For $P \subseteq \mathcal{SE}(\Sigma)$, we let* $\mathrm{succ}(P) = \bigcup_{p \in P} \mathrm{succ}(p)$*.*

Example 22. Let $u = \mathrm{aa}$, $\Lambda = \{\mathbb{T}_{0,0} \in (1,2) \wedge \mathbb{T}_{0,1} \in (1,2) \wedge \mathbb{T}_{0,2} \in (1,2) \wedge \mathbb{T}_{1,1} \in (0,1) \wedge \mathbb{T}_{1,2} \in (0,1) \wedge \mathbb{T}_{2,2} = 0\}$. The order $\Theta_{(u,\Lambda)}$ is such that $0 = \mathrm{frac}(\mathbb{T}_{2,2}) < \mathrm{frac}(\mathbb{T}_{1,2}) < \mathrm{frac}(\mathbb{T}_{0,2})$. The continuous successor of (u, Λ) is $\mathrm{succ}^t((u, \Lambda)) = (u, \Lambda^t)$, where $\Lambda^t = \{\mathbb{T}_{0,0} \in (1,2) \wedge \mathbb{T}_{0,1} \in (1,2) \wedge \mathbb{T}_{0,2} \in (1,2) \wedge \mathbb{T}_{1,1} \in (0,1) \wedge \mathbb{T}_{1,2} \in (0,1) \wedge \mathbb{T}_{2,2} \in (0,1)\}$. The continuous successor of (u, Λ^t) is $\mathrm{succ}^t((u, \Lambda^t)) = (u, \Lambda^{tt})$, where $\Lambda^{tt} = \{\mathbb{T}_{0,0} \in (1,2) \wedge \mathbb{T}_{0,1} \in (1,2) \wedge \mathbb{T}_{0,2} = 2 \wedge \mathbb{T}_{1,1} \in (0,1) \wedge \mathbb{T}_{1,2} \in (0,1) \wedge \mathbb{T}_{2,2} \in (0,1)\}$.

4.2 Timed Observation Table for Active DTA Learning

We extend the observation table with (simple) elementary languages and symbolic membership to learn a *recognizable timed language*.

Definition 23 (timed observation table). *A* timed observation table *is a 3-tuple (P, S, T) such that: P is a prefix-closed finite set of simple elementary languages, S is a finite set of elementary languages, and T is a function mapping $(p, s) \in (P \cup \mathrm{succ}(P)) \times S$ to the symbolic membership* $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}_{\mathrm{tgt}}}(p \cdot s)$*.*

Figure 2 illustrates timed observation tables: each cell indexed by (p, s) show the symbolic membership $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}_{\mathrm{tgt}}}(p \cdot s)$. For timed observation tables, we extend the notion of closedness and consistency with $\sim^{S}_{\mathcal{L}_{\mathrm{tgt}}}$ we introduced in Sect. 3.

Algorithm 3: Counterexample analysis in our DTA learning algorithm

```
1  Function AnalyzeCEX(cex):
2      i ← 1; w_0 ← cex
3      while ∄p ∈ P.w_i ∈ p do
4          i ← i + 1
5          split w_{i−1} into w'_i · w''_i such that w'_i ∈ p'_i for some p'_i ∈ succ(P) \ P
6          let p_i ∈ P and R_i be such that p'_i ~^{S,R_i}_{L_tgt} p_i
7          let w̄_i ∈ p_i be such that κ(w'_i), κ(w̄_i) ⊨ R_i
8          w_i ← w̄_i · w''_i
9      find j ∈ {1, 2, ..., i} such that w_{j−1} ∈ L_tgt△L(A_hyp) and w_j ∉ L_tgt△L(A_hyp)
       // We use a binary search with membership queries for ⌈log(i)⌉ times.
10     return the simple elementary language including w''_j
```

We note that consistency is defined only for discrete successors. This is because a timed observation table does not always become "consistent" for continuous successors. See Appendix C of [29] for a detailed discussion. We also require *exterior-consistency* since we construct an exterior from a successor.

Definition 24 (closedness, consistency, exterior-consistency, cohesion). *Let $O = (P, S, T)$ be a timed observation table. O is* closed *if, for each $p \in \mathrm{succ}(P) \setminus P$, there is $p' \in P$ satisfying $p \sim^{S}_{\mathcal{L}_{\mathrm{tgt}}} p'$. O is* consistent *if, for each $p, p' \in P$ and for each $a \in \Sigma$, $p \sim^{S}_{\mathcal{L}_{\mathrm{tgt}}} p'$ implies $\mathrm{succ}^{a}(p) \sim^{S}_{\mathcal{L}_{\mathrm{tgt}}} \mathrm{succ}^{a}(p')$. O is* exterior-consistent *if for any $p \in P$, $\mathrm{succ}^{t}(p) \notin P$ implies $\mathrm{succ}^{t}(p) \subseteq \mathrm{ext}^{t}(p)$. O is* cohesive *if it is closed, consistent, and exterior-consistent.*

From a cohesive timed observation table (P, S, T), we can construct a DTA as outlined in Algorithm 2. We construct a DTA via a recognizable timed language. The main ideas are as follows: 1) we approximate $\sim^{\mathcal{E}(\Sigma),R}_{\mathcal{L}_{\mathrm{tgt}}}$ by $\sim^{S,R}_{\mathcal{L}_{\mathrm{tgt}}}$; 2) we decide the equivalence class of $\mathrm{ext}(p) \in \mathrm{ext}(P) \setminus P$ in $\mathcal{E}(\Sigma)$ from successors. Notice that there can be multiple renaming equations R showing $\sim^{S,R}_{\mathcal{L}_{\mathrm{tgt}}}$. We use one of them found by an exhaustive search in Appendix B.2 of [29].

The DTA obtained by `MakeDTA` is faithful to the timed observation table in *rows*, i. e., for any $p \in P \cup \mathrm{succ}(P)$, $\mathcal{L}_{\mathrm{tgt}} \cap p = \mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \cap p$. However, unlike the L* algorithm, this does not hold for each *cell*, i. e., there may be $p \in P \cup \mathrm{succ}(P)$ and $s \in S$ satisfying $\mathcal{L}_{\mathrm{tgt}} \cap (p \cdot s) \neq \mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \cap (p \cdot s)$. This is because we do not (and actually cannot) enforce consistency for continuous successors. See Appendix C of [29] for a discussion. Nevertheless, this does not affect the correctness of our algorithm partly by Theorem 26.

Theorem 25 (row faithfulness). *For any cohesive timed observation table (P, S, T), for any $p \in P \cup \mathrm{succ}(P)$, $\mathcal{L}_{\mathrm{tgt}} \cap p = \mathcal{L}(\mathtt{MakeDTA}(P, S, T)) \cap p$ holds.* □

Theorem 26. *For any cohesive timed observation table (P, S, T), $\sim^{S}_{\mathcal{L}_{\mathrm{tgt}}} = \sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}}$ implies $\mathcal{L}_{\mathrm{tgt}} = \mathcal{L}(\mathtt{MakeDTA}(P, S, T))$.* □

Algorithm 4: Outline of our L*-style algorithm for DTA learning

```
1  P ← {(ε, τ_0 = 0)}; S ← {(ε, τ'_0 = 0)}
2  while ⊤ do
3      while (P, S, T) is not cohesive do
4          if ∃p ∈ succ(P) \ P. ∄p' ∈ P. p ~^S_{L_tgt} p' then          // (P, S, T) is not closed
5              P ← P ∪ {p}                                               // Add such p to P
6          else if ∃p, p' ∈ P, a ∈ Σ. p ~^S_{L_tgt} p' ∧ succ^a(p) ≁^S_{L_tgt} succ^a(p') then
               // (P, S, T) is inconsistent due to a
7              let S' ⊆ S be a minimal set such that p ≁^{S∪{{a·w | w∈s} | s∈S'}}_{L_tgt} p'
8              S ← S ∪ {{a · w | w ∈ s} | s ∈ S'}
9          else                                          // (P, S, T) is not exterior-consistent
10             P ← P ∪ {p' ∈ succ^t(P) \ P | ∃p ∈ P. p' = succ^t(p) ∧ p' ⊈ ext^t(p)}
11         fill T using symbolic membership queries
12     A_hyp ← MakeDTA(P, S, T)
13     if cex = eq_{L_tgt}(A_hyp) then
14         add AnalyzeCEX(cex) to S
15     else return A_hyp                          // It returns A_hyp if eq_{L_tgt}(A_hyp) = ⊤.
```

4.3 Counterexample Analysis

We analyze the counterexample *cex* obtained by an equivalence query to refine the set S of suffixes in a timed observation table. Our analysis, outlined in Algorithm 3, is inspired by the Rivest-Schapire algorithm [20,25]. The idea is to reduce the counterexample *cex* using the mapping defined by the congruence $\sim^S_{\mathcal{L}_{\mathrm{tgt}}}$ (lines 5–7), much like Φ in recognizable timed languages, and to find a suffix s strictly refining S (line 9), i. e., satisfying $p \sim^S_{\mathcal{L}_{\mathrm{tgt}}} p'$ and $p \not\sim^{S \cup \{s\}}_{\mathcal{L}_{\mathrm{tgt}}} p'$ for some $p \in \mathrm{succ}(P)$ and $p' \in P$.

By definition of *cex*, we have $cex = w_0 \in \mathcal{L}_{\mathrm{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$. By Theorem 25, $w_n \notin \mathcal{L}_{\mathrm{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$ holds, where n is the final value of i. By construction of $\mathcal{A}_{\mathrm{hyp}}$ and w_i, for any $i \in \{1, 2, \dots, n\}$, we have $w_0 \in \mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) \iff w_i \in \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$. Therefore, there is $i \in \{1, 2, \dots, n\}$ satisfying $w_{i-1} \in \mathcal{L}_{\mathrm{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$ and $w_i \notin \mathcal{L}_{\mathrm{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$. For such i, since we have $w_{i-1} = w'_i \cdot w''_i \in \mathcal{L}_{\mathrm{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$, $w_i = \overline{w}_i \cdot w''_i \notin \mathcal{L}_{\mathrm{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\mathrm{hyp}})$, and $\kappa(w'_i), \kappa(\overline{w}_i) \models R_i$, such w''_i is a witness of $p'_i \not\sim^{\mathcal{E}(\Sigma), R_i}_{\mathcal{L}_{\mathrm{tgt}}} p_i$. Therefore, S can be refined by the simple elementary language $s \in \mathcal{SE}(\Sigma)$ including w''_i.

4.4 L*-Style Learning Algorithm for DTAs

Algorithm 4 outlines our active DTA learning algorithm. At line 1, we initialize the timed observation table with $P = \{(\varepsilon, \tau_0 = 0)\}$ and $S = \{(\varepsilon, \tau'_0 = 0)\}$. In the loop in lines 2–15, we refine the timed observation table until the hypothesis DTA $\mathcal{A}_{\mathrm{hyp}}$ recognizes the target language $\mathcal{L}_{\mathrm{tgt}}$, which is checked by equivalence queries. The refinement finishes when the equivalence relation $\sim^S_{\mathcal{L}_{\mathrm{tgt}}}$ defined by the suffixes S converges to $\sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}}$, and the prefixes P covers $\mathcal{SE}(\Sigma)/{\sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}}}$.

In the loop in lines 3–11, we make the timed observation table cohesive. If the timed observation table is not closed, we move the incompatible row in $\mathrm{succ}(P) \setminus P$ to P (line 5). If the timed observation table is inconsistent, we concatenate an

	$(\varepsilon, \tau'_0 = 0)$
$(\varepsilon, \tau_0 = 0)$	$\tau_0 = \tau'_0 = 0$
$(\varepsilon, \tau_0 \in (0,1))$	$0 = \tau'_0 < \tau_0 < 1$
$(\mathrm{a}, \tau_0 = \tau_1 = 0)$	$\tau_0 = \tau_1 = \tau'_0 = 0$

(a) The initial timed observation table $O_1 = (P_1, S_1, T_1)$

(b) DTA $\mathcal{A}^1_{\mathrm{hyp}}$ constructed from O_1

(c) DTA $\mathcal{A}^3_{\mathrm{hyp}}$ constructed from O_3

	$s_0 = (\varepsilon, \tau'_0 = 0)$	$s_1 = (\mathrm{a}, \tau'_1 = 0 < \tau'_0 < 1)$
$p_0 = (\varepsilon, \tau_0 = 0)$	$\top$	$\top$
$p_1 = (\varepsilon, \tau_0 \in (0,1))$	$\top$	$\tau_0 + \tau'_0 \in (0,1)$
$p_2 = (\mathrm{a}, \tau_0 = \tau_0 + \tau_1 = \tau_1 = 0)$	$\top$	$\top$

(d) Timed observation table $O_2 = (P_2, S_2, T_2)$ after processing *cex*

	$(\varepsilon, \tau'_0 = 0)$	$(\mathrm{a}, \tau'_1 = 0 < \tau'_0 < 1)$
$(\varepsilon, \tau_0 = 0)$	$\top$	$\top$
$(\varepsilon, \tau_0 \in (0,1))$	$\top$	$\tau_0 + \tau'_0 \in (0,1)$
$(\varepsilon, \tau_0 = 1)$	$\top$	$\bot$
$(\mathrm{a}, \tau_0 = \tau_0 + \tau_1 = 1 \land \tau_1 = 0)$	$\bot$	$\top$
$(\mathrm{a}, \tau_0 = 1 \land \tau_1 \in (0,1))$	$\bot$	$\tau_1 + \tau'_0 \in (0,1]$
$(\mathrm{a}, \tau_0 = \tau_1 = 1 \land \tau_0 + \tau_1 = 2)$	$\bot$	$\bot$
$(\mathrm{a}, \tau_0 = \tau_0 + \tau_1 = \tau_1 = 0)$	$\top$	$\top$
$(\mathrm{a}, \tau_0 = \tau_0 + \tau_1 \in (0,1) \land \tau_1 = 0)$	$\top$	$\tau_0 + \tau_1 + \tau'_0 \in (0,1)$
$(\varepsilon, \tau_0 \in (1,2))$	$\top$	$\bot$
$(\mathrm{aa}, \tau_0 = \tau_0 + \tau_1 = \tau_0 + \tau_1 + \tau_2 = 1 \land \tau_1 = \tau_2 = \tau_1 + \tau_2 = 0)$	$\top$	$\top$
$(\mathrm{aa}, \tau_0 = 1 \land \tau_1 = \tau_1 + \tau_2 \in (0,1) \land \tau_0 + \tau_1 = \tau_0 + \tau_1 + \tau_2 \in (1,2) \land \tau_2 = 0)$	$\top$	$\tau_1 + \tau_2 + \tau'_0 \in (0,1)$
$(\mathrm{a}, \tau_0 = 1 < \tau_1 < 2 < \tau_0 + \tau_1 < 3)$	$\bot$	$\bot$
$(\mathrm{aa}, \tau_0 = \tau_1 = \tau_1 + \tau_2 = 1 \land \tau_0 + \tau_1 = \tau_0 + \tau_1 + \tau_2 = 2 \land \tau_2 = 0)$	$\top$	$\bot$

(e) The final observation table $O_3 = (P_3, S_3, T_3)$

Fig. 2. Timed observation tables O_1, O_2, O_3, and the DTAs $\mathcal{A}^1_{\mathrm{hyp}}$ and $\mathcal{A}^3_{\mathrm{hyp}}$ made from O_1 and O_3, respectively. In O_2 and O_3, we only show the constraints non-trivial from p and s. The DTAs are simplified without changing the language. The use of clock assignments, which does not change the expressiveness, is from [21].

event $a \in \Sigma$ in front of some of the suffixes in S (line 8). If the timed observation table is not exterior-consistent, we move the boundary $\mathrm{succ}^t(p) \in \mathrm{succ}^t(P) \setminus P$ satisfying $\mathrm{succ}^t(p) \not\subseteq \mathrm{ext}^t(p)$ to P (line 10). Once we obtain a cohesive timed observation table, we construct a DTA $\mathcal{A}_{\mathrm{hyp}} = \texttt{MakeDTA}(P, S, T)$ and make an equivalence query (line 12). If we have $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$, we return $\mathcal{A}_{\mathrm{hyp}}$. Otherwise, we have a timed word *cex* witnessing the difference between the language of the hypothesis DTA $\mathcal{A}_{\mathrm{hyp}}$ and the target language $\mathcal{L}_{\mathrm{tgt}}$. We refine the timed observation table using Algorithm 3.

Example 27. Let $\mathcal{L}_{\mathrm{tgt}}$ be the timed language recognized by the DTA in Fig. 1c. We start from $P = \{(\varepsilon, \tau_0 = 0)\}$ and $S = \{(\varepsilon, \tau'_0 = 0\}$. Figure 2a shows the initial timed observation table O_1. Since the timed observation table O_1 in Fig. 2a is cohesive, we construct a hypothesis DTA $\mathcal{A}^1_{\mathrm{hyp}}$. The hypothesis recognizable timed language is (P_1, F_1, Φ_1) is such that $P_1 = F_1 = \{(\varepsilon, \tau_0 = 0)\}$ and $\Phi_1 = \{(\varepsilon, \tau_0 > 0, \varepsilon, \tau_0, \top), (\mathrm{a}, \tau_0 = \tau_0 + \tau_1 = \tau_1 = 0, \varepsilon, \tau_0, \top)\}$. Figure 2b shows the first hypothesis DTA $\mathcal{A}^1_{\mathrm{hyp}}$.

We have $\mathcal{L}(\mathcal{A}^1_{\mathrm{hyp}}) \neq \mathcal{L}_{\mathrm{tgt}}$, and the learner obtains a counterexample, e.g., $cex = 1.0 \cdot \mathrm{a} \cdot 0$, with an equivalence query. In Algorithm 3, we have $w_0 = cex$, $w_1 = 0.5 \cdot \mathrm{a} \cdot 0$, $w_2 = 0 \cdot \mathrm{a} \cdot 0$, and $w_3 = 0$. We have $w_0 \notin \mathcal{L}(\mathcal{A}^1_{\mathrm{hyp}}) \triangle \mathcal{L}_{\mathrm{tgt}}$ and $w_1 \in \mathcal{L}(\mathcal{A}^1_{\mathrm{hyp}}) \triangle \mathcal{L}_{\mathrm{tgt}}$, and the suffix to distinguish w_0 and w_1 is $0.5 \cdot \mathrm{a} \cdot 0$. Thus, we add $s_1 = (\mathrm{a}, \tau'_1 = 0 < \tau'_0 = \tau'_0 + \tau'_1 < 1)$ to S_1 (Fig. 2d).

In Fig. 2d, we observe that $T_2(p_1, s_1)$ is more strict than $T_2(p_0, s_1)$, and we have $p_1 \not\sim^{S_2}_{\mathcal{L}_{\mathrm{tgt}}} p_0$. To make (P_2, S_2, T_2) closed, we add p_1 to P_2. By repeating similar operations, we obtain the timed observation table $O_3 = (P_3, S_3, T_3)$ in Fig. 2e, which is cohesive. Figure 2c shows the DTA $\mathcal{A}^3_{\mathrm{hyp}}$ constructed from O_3. Since $\mathcal{L}(\mathcal{A}^3_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$ holds, Algorithm 4 finishes returning $\mathcal{A}^3_{\mathrm{hyp}}$.

By the use of equivalence queries, Algorithm 4 returns a DTA recognizing the target language if it terminates, which is formally as follows.

Theorem 28 (correctness). *For any target timed language $\mathcal{L}_{\mathrm{tgt}}$, if Algorithm 4 terminates, for the resulting DTA $\mathcal{A}_{\mathrm{hyp}}$, $\mathcal{L}(\mathcal{A}_{\mathrm{hyp}}) = \mathcal{L}_{\mathrm{tgt}}$ holds.* □

Moreover, Algorithm 4 terminates for any recognizable timed language $\mathcal{L}_{\mathrm{tgt}}$ essentially because of the finiteness of $\mathcal{SE}(\Sigma)/\sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}}$.

Theorem 29 (termination). *For any recognizable timed language $\mathcal{L}_{\mathrm{tgt}}$, Algorithm 4 terminates and returns a DTA $\mathcal{A}$ satisfying $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{\mathrm{tgt}}$.*

Proof (Theorem 29). By the recognizability of $\mathcal{L}_{\mathrm{tgt}}$ and Theorem 19, $\mathcal{SE}(\Sigma)/\sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}}$ is finite. Let $N = |\mathcal{SE}(\Sigma)/\sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}}|$. Since each execution of line 5 adds p to P, where p is such that for any $p' \in P$, $p \not\sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}} p'$ holds, it is executed at most N times. Since each execution of line 8 refines S, i. e., it increases $|\mathcal{SE}(\Sigma)/\sim^{S}_{\mathcal{L}_{\mathrm{tgt}}}|$, line 8 is executed at most N times. For any $(u, \Lambda) \in \mathcal{SE}(\Sigma)$, if Λ contains $\mathbb{T}_{i,|u|} = d$ for some $i \in \{0, 1, \dots, |u|\}$ and $d \in \mathbb{N}$, we have $\mathrm{succ}^t((u, \Lambda)) \subseteq \mathrm{ext}^t((u, \Lambda))$. Therefore, line 10 is executed at most N times. Since S is strictly refined in line 14, i. e., it increases $|\mathcal{SE}(\Sigma)/\sim^{S}_{\mathcal{L}_{\mathrm{tgt}}}|$, line 14 is executed at most N times. By Theorem 26, once $\sim^{S}_{\mathcal{L}_{\mathrm{tgt}}}$ saturates to $\sim^{\mathcal{E}(\Sigma)}_{\mathcal{L}_{\mathrm{tgt}}}$, `MakeDTA` returns the correct DTA. Overall, Algorithm 4 terminates. □

4.5 Learning with a Normal Teacher

We briefly show how to learn a DTA only with membership and equivalence queries. We reduce a symbolic membership query to finitely many membership queries, answerable by a normal teacher. See Appendix B.1 of [29] for detail.

Let (u, Λ) be the elementary language given in a symbolic membership query. Since Λ is bounded, we can construct a finite and disjoint set of simple and canonical timed conditions $\Lambda'_1, \Lambda'_2, \dots, \Lambda'_n$ satisfying $\bigvee_{1 \leq i \leq n} \Lambda'_i = \Lambda$ by a simple enumeration. For any simple elementary language $(u', \Lambda') \in \mathcal{SE}(\Sigma)$ and timed words $w, w' \in (u', \Lambda')$, we have $w \in \mathcal{L} \iff w' \in \mathcal{L}$. Thus, we can construct $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}}((u, \Lambda))$ by making a membership query $\mathtt{mem}_{\mathcal{L}}(w)$ for each such $(u', \Lambda') \subseteq (u, \Lambda)$ and for some $w \in (u', \Lambda')$. We need such an exhaustive search, instead of a binary search, because $\mathtt{mem}^{\mathtt{sym}}_{\mathcal{L}}((u, \Lambda))$ may be non-convex.

Assume Λ is a canonical timed condition. Let M be the size of the variables in Λ and I be the largest difference between the upper bound and the lower bound for some $\mathbb{T}_{i,j}$ in Λ. The size n of the above decomposition is bounded by $(2 \times I + 1)^{1/2 \times M \times (M+1)}$, which exponentially blows up with respect to M.

In our algorithm, we only make symbolic membership queries with elementary languages of the form $p \cdot s$, where p and s are simple elementary languages. Therefore, I is at most 2. However, even with such an assumption, the number of the necessary membership queries blows up exponentially to the size of the variables in Λ.

4.6 Complexity Analysis

After each equivalence query, our DTA learning algorithm strictly refines S or terminates. Thus, the number of equivalence queries is at most N. In the proof of Theorem 29, we observe that the size of P is at most $2N$. Therefore, the number $(|P| + |\mathrm{succ}(P)|) \times |S|$ of the cells in the timed observation table is at most $(2N + 2N \times (|\Sigma| + 1)) \times N = 2N^2|\Sigma| + 2$. Let J be the upper bound of i in the analysis of *cex* returned by equivalence queries (Algorithm 3). For each equivalence query, the number of membership queries in Algorithm 3 is bounded by $\lceil \log J \rceil$, and thus, it is, in total, bounded by $N \times \lceil \log J \rceil$. Therefore, if the learner can use symbolic membership queries, the total number of queries is bounded by a polynomial of N and J. In Sect. 4.5, we observe that the number of membership queries to implement a symbolic membership query is at most exponential to M. Since P is prefix-closed, M is at most N. Overall, if the learner cannot use symbolic membership queries, the total number of queries is at most exponential to N.

Table 1. Summary of the results for Random. Each row index $|L|_|\Sigma|_K_C$ shows the number of locations, the alphabet size, and the upper bound of the maximum constant in the guards, respectively. The row "count" shows the number of instances finished in 3 h. Cells with the best results are highlighted.

		# of Mem. queries			# of Eq. queries			Exec. time [sec.]			count
		max	mean	min	max	mean	min	max	mean	min	
3_2_10	LearnTA	35,268	14,241	2,830	**11**	**6**	**4**	2.32e+00	6.68e-01	**4.50e-02**	**10/10**
	OneSMT	**468**	**205**	**32**	13	8	5	**9.58e-01**	**2.89e-01**	6.58e-02	**10/10**
4_2_10	LearnTA	194,442	55,996	10,619	**14**	**7**	**4**	2.65e+01	7.98e+00	4.88e-01	**10/10**
	OneSMT	**985**	**451**	**255**	16	12	7	**3.53e-01**	**2.09e-01**	**1.27e-01**	**10/10**
4_4_20	LearnTA	1,681,769	858,759	248,399	**21**	**15**	**10**	8.34e+03	1.41e+03	3.23e+01	8/10
	OneSMT	**5,329**	**3,497**	**1,740**	42	32	26	**2.19e+00**	**1.42e+00**	**8.27e-01**	**10/10**
5_2_10	LearnTA	627,980	119,906	8,121	**19**	**8**	**5**	1.67e+02	2.28e+01	**1.96e-01**	**10/10**
	OneSMT	**1,332**	**876**	**359**	22	16	12	**5.20e-01**	**3.66e-01**	2.58e-01	**10/10**
6_2_10	LearnTA	555,939	106,478	2,912	**14**	**9**	**6**	2.44e+02	2.81e+01	**4.40e-02**	**10/10**
	OneSMT	**3,929**	**1,894**	**104**	35	20	11	**1.72e+00**	**8.01e-01**	1.73e-01	**10/10**

Let $\mathcal{A}_{\mathrm{tgt}} = (\Sigma, L, l_0, C, I, \Delta, F)$ be a DTA recognizing $\mathcal{L}_{\mathrm{tgt}}$. As we observe in the proof of Lemma 33 of [29], N is bounded by the size of the state space of the region automaton [4] of $\mathcal{A}_{\mathrm{tgt}}$, N is at most $|C|! \times 2^{|C|} \times \prod_{c \in C}(2K_c + 2) \times |L|$, where K_c is the largest constant compared with $c \in C$ in $\mathcal{A}_{\mathrm{tgt}}$. Thus, without symbolic

membership queries, the total number of queries is at most doubly-exponential to $|C|$ and singly exponential to $|L|$. We remark that when $|C| = 1$, the total number of queries is at most singly exponential to $|L|$ and K_c, which coincides with the worst-case complexity of the one-clock DTA learning algorithm in [30].

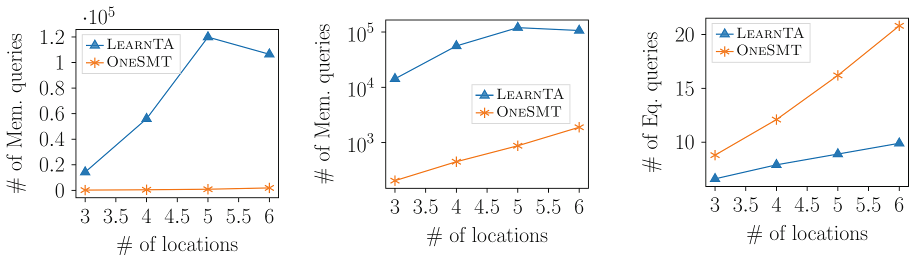
5 Experiments

We experimentally evaluated our DTA learning algorithm using our prototype library LearnTA[4] implemented in C++. In LearnTA, the equivalence queries are answered by a zone-based reachability analysis using the fact that DTAs are closed under complement [4]. We pose the following research questions.

RQ1 How is the scalability of LearnTA to the language complexity?
RQ2 How is the efficiency of LearnTA for practical benchmarks?

For the benchmarks with one clock variable, we compared LearnTA with one of the latest one-clock DTA learning algorithms [1,30], which we call OneSMT. OneSMT is implemented in Python with Z3 [23] for constraint solving.

For each execution, we measured the number of queries and the total execution time, including the time to answer the queries. For the number of queries, we report the number with memoization, i. e., we count the number of the queried timed words (for membership queries) and the counterexamples (for equivalence queries). We conducted all the experiments on a computing server with Intel Core i9-10980XE 125 GiB RAM that runs Ubuntu 20.04.5 LTS. We used 3 h as the timeout.



(a) Membership queries (b) Membership queries (log scale) (c) Equivalence queries

Fig. 3. The number of locations and the number of queries for $|L|$_2_10 in Random, where $|L| \in \{3, 4, 5, 6\}$

[4] LearnTA is publicly available at https://github.com/masWag/LearnTA. The artifact of the experiments is available at https://doi.org/10.5281/zenodo.7875383.

Table 2. Summary of the target DTAs and the results for Unbalanced. $|L|$ is the number of locations, $|\Sigma|$ is the alphabet size, $|C|$ is the number of clock variables, and K_C is the maximum constant in the guards in the DTA.

		$\|L\|$	$\|\Sigma\|$	$\|C\|$	K_C	# of Mem. queries	# of Eq. queries	Exec. time [sec.]
Unbalanced:1	LearnTA	5	1	1	2	51	2	2.00e-03
Unbalanced:2	LearnTA	5	1	2	4	576,142	3	3.64e+01
Unbalanced:3	LearnTA	5	1	3	4	403,336	4	2.24e+01
Unbalanced:4	LearnTA	5	1	4	6	4,142,835	5	2.40e+02
Unbalanced:5	LearnTA	5	1	5	6	10,691,400	5	8.68e+02

5.1 RQ1: Scalability with Respect to the Language Complexity

To evaluate the scalability of LearnTA, we used randomly generated DTAs from [5] (denoted as Random) and our original DTAs (denoted as Unbalanced). Random consists of five classes: 3_2_10, 4_2_10, 4_4_20, 5_2_10, and 6_2_10, where each value of $|L|_|\Sigma|_K_C$ is the number of locations, the alphabet size, and the upper bound of the maximum constant in the guards in the DTAs, respectively. Each class consists of 10 randomly generated DTAs. Unbalanced is our original benchmark inspired by the “unbalanced parentheses” timed language from [10]. Unbalanced consists of five DTAs with different complexity of timing constraints. Table 2 summarizes their complexity.

Table 1 and 3 summarize the results for Random, and Table 2 summarizes the results for Unbalanced. Table 1 shows that LearnTA requires more membership queries than OneSMT. This is likely because of the difference in the definition of prefixes and successors: OneSMT’s definitions are discrete (e. g., prefixes are only with respect to events with time elapse), whereas ours are both continuous and discrete (e. g., we also consider prefixes by trimming the dwell time in the end); Since our definition makes significantly more prefixes, LearnTA tends to require much more membership queries. Another, more high-level reason is that LearnTA learns a DTA without knowing the number of the clock variables, and many more timed words are potentially helpful for learning. Table 1 shows that LearnTA requires significantly many membership queries for 4_4_20. This is likely because of the exponential blowup with respect to K_C, as discussed in Sect. 4.6. In Fig. 3, we observe that for both LearnTA and OneSMT, the number of membership queries increases nearly exponentially to the number of locations. This coincides with the discussion in Sect. 4.6.

In contrast, Table 1 shows that LearnTA requires fewer equivalence queries than OneSMT. This suggests that the cohesion in Definition 24 successfully detected contradictions in observation before generating a hypothesis, whereas OneSMT mines timing constraints mainly by equivalence queries and tends to require more equivalence queries. In Fig. 3c, we observe that for both LearnTA and OneSMT, the number of equivalence queries increases nearly linearly to the number of locations. This also coincides with the complexity analysis in Sect. 4.6. Figure 3c also shows that the number of equivalence queries increases faster in OneSMT than in LearnTA.

Table 3. Summary of the target DTA and the results for practical benchmarks. The columns are the same as Table 2. Cells with the best results are highlighted.

		$\lvert L\rvert$	$\lvert\Sigma\rvert$	$\lvert C\rvert$	K_C	# of Mem. queries	# of Eq. queries	Exec. time [sec.]
AKM	LearnTA	17	12	1	5	12,263	**11**	**5.85e-01**
	OneSMT	17	12	1	5	**3,453**	49	7.97e+00
CAS	LearnTA	14	10	1	27	66,067	**17**	**4.65e+00**
	OneSMT	14	10	1	27	**4,769**	18	9.58e+01
Light	LearnTA	5	5	1	10	3,057	**7**	**3.30e-02**
	OneSMT	5	5	1	10	**210**	7	9.32e-01
PC	LearnTA	26	17	1	10	245,134	**23**	**6.49e+01**
	OneSMT	26	17	1	10	**10,390**	29	1.24e+02
TCP	LearnTA	22	13	1	2	11,300	**15**	**3.82e-01**
	OneSMT	22	13	1	2	**4,713**	32	2.20e+01
Train	LearnTA	6	6	1	10	13,487	**8**	**1.72e-01**
	OneSMT	6	6	1	10	**838**	13	1.13e+00
FDDI	LearnTA	16	5	7	6	**9,986,271**	**43**	**3.00e+03**

Table 2 also suggests a similar tendency: the number of membership queries rapidly increases to the complexity of the timing constraints; In contrast, the number of equivalence queries increases rather slowly. Moreover, LearnTA is scalable enough to learn a DTA with five clock variables within 15 min.

Table 1 also suggests that LearnTA does not scale well to the maximum constant in the guards, as observed in Sect. 4.6. However, we still observe that LearnTA requires fewer equivalence queries than OneSMT. Overall, compared with OneSMT, LearnTA has better scalability in the number of equivalence queries and worse scalability in the number of membership queries.

5.2 RQ2: Performance on Practical Benchmarks

To evaluate the practicality of LearnTA, we used seven benchmarks: AKM, CAS, Light, PC, TCP, Train, and FDDI. Table 3 summarizes their complexity. All the benchmarks other than FDDI are taken from [30] (or its implementation [1]). FDDI is taken from TChecker [2]. We use the instance of FDDI with two processes.

Table 3 summarizes the results for the benchmarks from practical applications. We observe, again, that LearnTA requires more membership queries and fewer equivalence queries than OneSMT. However, for these benchmarks, the difference in the number of membership queries tends to be much smaller than in Random. This is because these benchmarks have simpler timing constraints than Random for the exploration by LearnTA. In AKM, Light, PC, TCP, and Train, the clock variable can be reset at every edge without changing the language. For such a DTA, all simple elementary languages are equivalent in terms of the Nerode-style congruence if we have the same edge at their last event and the same dwell time after it. If two simple elementary languages are equivalent, LearnTA explores the successors of only one of them, and the exploration is

relatively efficient. We have a similar situation in CAS. Moreover, in many of these DTAs, only a few edges have guards. Overall, despite the large number of locations and alphabets, these languages' complexities are mild for LearnTA.

We also observe that, surprisingly, for all of these benchmarks, LearnTA took a shorter time for DTA learning than OneSMT. This is partly because of the difference in the implementation language (i. e., C++ vs. Python) but also because of the small number of equivalence queries and the mild number of membership queries. Moreover, although it requires significantly more queries, LearnTA successfully learned FDDI with seven clock variables. Overall, such efficiency on benchmarks from practical applications suggests the potential usefulness of LearnTA in some realistic scenarios.

6 Conclusions and Future Work

Extending the L* algorithm, we proposed an active learning algorithm for DTAs. Our extension is by our Nerode-style congruence for recognizable timed languages. We proved the termination and the correctness of our algorithm. We also proved that our learning algorithm requires a polynomial number of queries with a smart teacher and an exponential number of queries with a normal teacher. Our experiment results also suggest the practical relevance of our algorithm.

One of the future directions is to extend more recent automata learning algorithms (e. g., TTT algorithm [19] to improve the efficiency) to DTA learning. Another direction is constructing a *passive* DTA learning algorithm based on our congruence and an existing passive DFA learning algorithm. It is also a future direction to apply our learning algorithm for practical usage, e. g., identification of black-box systems and testing black-box systems with black-box checking [22, 24, 28]. Optimization of the algorithm, e. g., by incorporating clock information is also a future direction.

Acknowledgements. This work is partially supported by JST ACT-X Grant No. JPMJAX200U, JST PRESTO Grant No. JPMJPR22CA, JST CREST Grant No. JPMJCR2012, and JSPS KAKENHI Grant No. 22K17873.

References

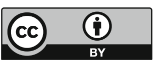
1. GitHub: Leslieaj/DOTALearningSMT. https://github.com/Leslieaj/DOTALearningSMT, (Accessed 10 Jan 2023)
2. Github: ticktac-project/tchecker. https://github.com/ticktac-project/tchecker, (Accessed 20 Jan 2023)
3. Aichernig, B.K., Pferscher, A., Tappler, M.: From passive to active: learning timed automata efficiently. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 1–19. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_1
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8

5. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. In: TACAS 2020. LNCS, vol. 12078, pp. 444–462. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_25
6. An, J., Wang, L., Zhan, B., Zhan, N., Zhang, M.: Learning real-time automata. Science China Inf. Sci. **64**(9), 1–17 (2021). https://doi.org/10.1007/s11432-019-2767-4
7. An, J., Zhan, B., Zhan, N., Zhang, M.: Learning nondeterministic real-time automata. ACM Trans. Embed. Comput. Syst. **20**(5s), 99:1–99:26 (2021). https://doi.org/10.1145/3477030,
8. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
9. Argyros, G., D'Antoni, L.: The learnability of symbolic automata. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 427–445. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_23
10. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM **49**(2), 172–206 (2002). https://doi.org/10.1145/506147.506151
11. Bersani, M.M., Rossi, M., San Pietro, P.: A logical characterization of timed regular languages. Theor. Comput. Sci. **658**, 46–59 (2017). https://doi.org/10.1016/j.tcs.2016.07.020
12. Bojańczyk, M., Lasota, S.: A machine-independent characterization of timed languages. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012. LNCS, vol. 7392, pp. 92–103. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31585-5_12
13. Bouyer, P., Petit, A., Thérien, D.: An algebraic characterization of data and timed languages. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 248–261. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44685-0_17
14. Drews, S., D'Antoni, L.: Learning symbolic automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 173–189. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_10
15. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. Theor. Comput. Sci. **411**(47), 4029–4054 (2010). https://doi.org/10.1016/j.tcs.2010.07.008
16. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006). https://doi.org/10.1007/11817949_29
17. Henry, L., Jéron, T., Markey, N.: Active learning of timed automata with unobservable resets. In: Bertrand, N., Jansen, N. (eds.) FORMATS 2020. LNCS, vol. 12288, pp. 144–160. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57628-8_9
18. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 3rd edn. Addison-Wesley, Pearson international edition (2007)
19. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
20. Isberner, M., Steffen, B.: An abstract framework for counterexample analysis in active automata learning. In: Clark, A., Kanazawa, M., Yoshinaka, R. (eds.) Proceedings of the 12th International Conference on Grammatical Inference, ICGI

2014, Kyoto, Japan, 17–19 September 2014. JMLR Workshop and Conference Proceedings, vol. 34, pp. 79–93. JMLR.org (2014). http://proceedings.mlr.press/v34/isberner14a.html
21. Maler, O., Pnueli, A.: On recognizable timed languages. In: Walukiewicz, I. (ed.) FoSSaCS 2004. LNCS, vol. 2987, pp. 348–362. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24727-2_25
22. Meijer, J., van de Pol, J.: Sound black-box checking in the learnlib. Innov. Syst. Softw. Eng. **15**(3–4), 267–287 (2019). https://doi.org/10.1007/s11334-019-00342-6
23. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
24. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX 1999, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), 5–8 October 1999, Beijing, China. IFIP Conference Proceedings, vol. 156, pp. 225–240. Kluwer (1999)
25. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021
26. Shijubo, J., Waga, M., Suenaga, K.: Efficient black-box checking via model checking with strengthened specifications. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 100–120. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_6
27. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Time to learn – learning timed automata from tests. In: André, É., Stoelinga, M. (eds.) FORMATS 2019. LNCS, vol. 11750, pp. 216–235. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29662-9_13
28. Waga, M.: Falsification of cyber-physical systems with robustness-guided black-box checking. In: Ames, A.D., Seshia, S.A., Deshmukh, J. (eds.) HSCC 2020: 23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, 21–24 April 2020, pp. 11:1–11:13. ACM (2020). https://doi.org/10.1145/3365365.3382193
29. Waga, M.: Active learning of deterministic timed automata with myhill-nerode style characterization. CoRR abs/ arXiv: 2305.17742 (2023). http://arxiv.org/abs/2305.17742
30. Xu, R., An, J., Zhan, B.: Active learning of one-clock timed automata using constraint solving. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, 25–28 October 2022, Proceedings. LNCS, vol. 13505, pp. 249–265. Springer (2022). https://doi.org/10.1007/978-3-031-19992-9_16
31. Zhang, H., Feng, L., Li, Z.: Control of black-box embedded systems by integrating automaton learning and supervisory control theory of discrete-event systems. IEEE Trans. Autom. Sci. Eng. **17**(1), 361–374 (2020). https://doi.org/10.1109/TASE.2019.2929563

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Automated Analyses of IOT Event Monitoring Systems

Andrew Apicelli[1], Sam Bayless[1], Ankush Das[1], Andrew Gacek[1], Dhiva Jaganathan[1], Saswat Padhi[2], Vaibhav Sharma[3](✉), Michael W. Whalen[1], and Raveesh Yadav[1]

[1] Amazon Web Services, Inc., Seattle, USA
{apicea,sabayles,daankus,gacek,dhivasj,mww,raveeshy}@amazon.com
[2] Google LLC, Mountain View, USA
spadhi@google.com
[3] Amazon.com Services LLC, Seattle, USA
svaib@amazon.com

Abstract. AWS IoT Events is an AWS service that makes it easy to respond to events from IoT sensors and applications. *Detector models* in AWS IoT Events enable customers to monitor their equipment or device fleets for failures or changes in operation and trigger actions when such events occur. If these models are incorrect, they may become out-of-sync with the actual state of the equipment causing customers to be unable to respond to events occurring on it.

Working backwards from common mistakes made when creating detector models, we have created a set of automated analyzers that allow customers to prove their models are free from six common mistakes. Our analyzers have been running in the AWS IoT Events production service since December 2021. Our analyzers check six correctness properties in the production service in real time. 93% of customers of AWS IoT Events have run our analyzers without needing to have any knowledge of them. Our analyzers have reported property violations in 22% of submitted detector models in the production service.

1 Introduction

AWS IoT Events is a managed service for managing fleets of IoT devices. Customers use AWS IoT Events in diverse use cases such as monitoring self-driving wheelchairs, monitoring a device's network connectivity, humidity, temperature, pressure, oil level, and oil temperature sensing. Customers use AWS IoT Events by creating a *detector model* that detects events occurring on IoT devices and notifies an external service so that a corrective action can be taken. An example is an industrial boiler which constantly reports its temperature to a detector. The detector tracks the boiler's average temperature over the past 90 min and notifies a human operator when it is running too hot.

S. Padhi–Work done while at Amazon Web Services, Inc.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 27–39, 2023.
https://doi.org/10.1007/978-3-031-37706-8_2

Each detector model is defined as a finite state machine with dynamically typed variables and timers, where timers allow detectors to keep track of state over time. A model processes inputs from IoT devices to update internal state and to notify other AWS services when events are detected. Customers can use a single detector model to instantaneously detect events in thousands of devices. Ensuring well-formedness of a detector model is crucial as ill-formed detector models can miss events in *every* monitored device.

Starting from a survey that identified sources of well-formedness problems in customer models, we identified most common mistakes made by customers and detect them using type- and model-checking. To use a model-checker for checking well-formedness of a detector model, we formalize the execution semantics of a detector model and translate this semantics into the source-language notation of the JKind model checker [1]. Model checking [2–9] verifies desirable properties over the behavior of a system by performing the equivalent of an exhaustive enumeration of all the states reachable from its initial state. Most model checking tools use *symbolic encodings* and some form of *induction* [6] to prove properties of very large finite or even infinite state spaces.

We have implemented type-checking and model-checking as an analysis feature in the production AWS IoT Events service. Our analyzers have reported well-formedness property violations in 22% of submitted detector models. 93% of customers of AWS IoT Events have checked their detector models using our analyzers. Our analyzers report property violations to customers with an average latency of 5.6 s (see Sect. 4).

Our contributions are as follows:

1. We formalize the semantics of AWS IoT Events detector models.
2. We identify six well-formedness properties whose violations detect common customer mistakes.
3. We create fast, push-button analyzers that report property violations to customers.

2 Overview

Consider a user of AWS IoT Events who wants to monitor the temperature of an industrial boiler. If the industrial boiler overheats, it can cause fires and endanger human lives. To detect an early warning of an overheating event, they want to automatically identify two different alarming events on the boiler's temperature. They want their first alarm to be triggered if the boiler's reported temperature is outside the normal range for more than 1 min. They want their second alarm to be triggered if the temperature is outside the normal range for another 5 min after the first alarm.

A user might try to implement these requirements by creating the (flawed) detector model shown in Fig. 1. This detector receives temperature data from the boiler and responds by sending a text message to the user. The detector model contains four states:

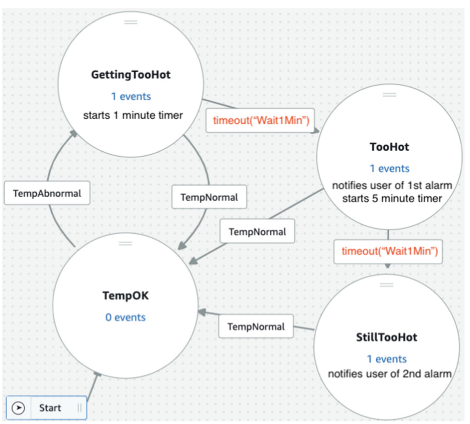


Fig. 1. AWS IoT Events detector model with two alarms (buggy version)

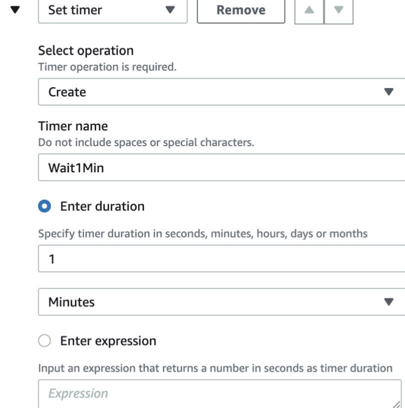


Fig. 2. An action in the detector model from Fig. 1

- TempOK: starting state of the detector model. The detector stays in this state as long as the boiler's temperature lies in a normal range. The detector transitions from TempOK to GettingTooHot on detecting a temperature outside normal range, indicated by TempAbnormal.
- GettingTooHot: detector starts a 1 min timer and transitions back to TempOK if the boiler cools down. When the timer expires, it transitions to TooHot.
- TooHot: detector first notifies the user of the 1st alarm. It then starts a 5 min timer and transitions back to TempOK if the boiler cools down. When the 5 min timer expires, it transitions to StillTooHot.
- StillTooHot: detector notifies user of the 2nd alarm.

Understanding the Bug: Every state in the detector model consists of *actions*. An action changes the internal state of a detector or triggers an external service. For example, the GettingTooHot state consists of an action that starts a timer. The user can edit these actions with an interface shown in Fig. 2. This action starts a one minute timer named Wait1Min. Note that timers are accessible from every state in the detector model. Even though the Wait1Min timer is created in the GettingTooHot state of Fig. 1, it can be checked for expiration in all the four states of Fig. 1.

The detector model in Fig. 1 has a fatal flaw based on a typo. The user has written timeout("Wait1Min") instead of timeout("Wait5Min") when transitioning out of TooHot. This is allowed as timers are globally referenceable. However, it is a bug because each global timer has a unique name and the Wait1Min timer has already been used and expired. This makes StillTooHot unreachable, meaning the 2nd alarm won't ever fire, since a timer can expire at most once.

Related Work. Languages such as IOTA [10], SIFT [11], and the system from Garcia et al. [12] use *trigger-condition-action rules* [13] to control the behavior of

internet of things applications. These languages have the benefit of being largely declarative, allowing users to specify desired actions under different environmental stimuli. Similar to our approach, SIFT [11] automatically removes common user mistakes as well as compiles specifications into controller implementations without user interaction, and IOTA [10] is a reasoning calculus that allows custom specifications to be written both about why something *should* or *should not* occur. AWS IoT Events is designed explicitly for monitoring, rather than control, and our approach is imperative, rather than declarative: detector models do not have the same inconsistencies as rule sets, as they are disambiguated using explicit priorities on transitions. On the other hand, customers may still construct machines that do not match their intentions, motivating the analyses described in this paper.

3 Technique

In this section, we present a formal execution semantics of an AWS IoT Events detector model and describe specifications for the correctness properties.

Formalization of Detector Models. Defining the alphabet and the transition relation for the state machine is perhaps the most interesting aspect of our formalization. Since detector models may contain global timers, *timed automata* [14] might seem like an apt candidate abstraction. However, AWS IoT Events users are not allowed to change the clock frequency of timers, nor specify arbitrary *clock constraints*. These observations allow us to formalize the detector models as a regular state machine, with timeout durations as additional state variables.

Formally, we represent the state machine for a detector model $\mathbf{M}$ as a tuple $\langle \mathbf{S}, \mathbf{S}_0, \mathbf{I}, \mathbf{G}, \mathbf{T}, \mathcal{E}_E, \mathcal{E}_X, \mathcal{E}_I \rangle$, where:

- $\mathbf{S}$: finite set of states in the FSM,
- $\mathbf{S}_0 \subseteq \mathbf{S}$: set of *initial* state(s),
- $\mathbf{I}$: set of input variables assigned by the environment
- $\mathbf{G}$: set of global variables assigned by the state machine
- $\mathbf{T}$: set of timer variables that are reset by the model and updated as time evolves in the environment
- $\mathcal{E}_E : \mathbf{S} \rightarrow \kappa\ \texttt{list}$: mapping from states to a (possibly empty) list of entry events to be performed when entering a state. κ describes an *event*, further explained in the description of the grammar.
- $\mathcal{E}_X : \mathbf{S} \rightarrow \kappa\ \texttt{list}$ is a mapping from states to a list of exit events to be performed when exiting a state.
- $\mathcal{E}_I : \mathbf{S} \rightarrow (\kappa\ \texttt{list} \times \mu\ \texttt{list})$: mapping from states to a list of input events, including transitions to other states.

It is assumed that the sets $\mathbf{I}$, $\mathbf{G}$, and $\mathbf{T}$ are pairwise disjoint, and we define the set $\mathbf{V} \triangleq \mathbf{I} \cup \mathbf{G}$ to represent input and global variables in the model.

We denote by $\mathbb{V}$ the set of values for global ($\mathbf{G}$) and input ($\mathbf{I}$) variables; $\mathbb{V}$ ranges over the values of primitive types: integers, decimals (rationals), booleans,

$$
\begin{aligned}
\tau &::= \texttt{int} \mid \texttt{dec} \mid \texttt{str} \mid \texttt{bool} \\
\epsilon &::= e_0 \ bop \ e_1 \mid uop \ e_0 \mid l \mid v \mid \texttt{timeout}(t) \mid \texttt{isundefined}(v) \mid \ldots \\
\alpha &::= \texttt{setTimer}(t, e) \mid \texttt{resetTimer}(t) \\
&\quad \mid \texttt{clearTimer}(t) \mid \texttt{setGlobal}(g, e) \\
\kappa &::= \texttt{event}(e, a*) \\
\mu &::= \texttt{transition}(e, a*, s) \\
\iota &::= \texttt{message}(i, v) \mid \texttt{timeout}(t)
\end{aligned}
$$

Fig. 3. Types, expressions, actions, and events in IoT Events Detector Models

and strings. Integers and rationals are assumed to be unbounded, and rationals are arbitrarily precise. We use $\mathbb{N}$ as the domain for time and timeout values. Sets $\mathbb{V}^{\perp}$ and $\mathbb{N}^{\perp}$ are extended with the value $\perp$ to represent an *uninitialized* variable.

The grammar for types (τ), expressions (ϵ), actions (α), events (κ), transitions (μ) and input triggers (ι) is shown in Fig. 3. In the grammar, metavariable e stands for an expression, l stands for a literal value in $\mathbb{V}$, v stands for any variable in $\mathbf{V}$, t is a timer variable in $\mathbf{T}$, a is an action, and i is an input in $\mathbf{I}$. The unary and binary operators include standard arithmetic, Boolean, and relational operators. The `timeout` expression is true at the instant timer t expires, and the `isundefined` expression returns true if the variable or timer in question has not been assigned. Actions (α) describe changes to the system state: `setTimer` starts a timer and sets the periodicity of the timer, while the `resetTimer` and `clearTimer` reset and clear a timer (without changing the periodicity of the timer). The `setGlobal` action assigns a global variable. Events (κ) describe conditions under which a sequence of actions occur.

We define configurations $\mathbf{C}$ for the state machine as:

$$\mathbf{C} \triangleq \mathbf{S} \times (\mathbf{I} \rightarrow \mathbb{V}^{\perp}) \times (\mathbf{T} \rightarrow (\mathbb{N}^{\perp} \times \mathbb{N}^{\perp})) \times (\mathbf{G} \rightarrow \mathbb{V}^{\perp})$$

Each configuration $C = \langle s, i, t, g \rangle$ tracks the following:

- a state $s \in \mathbf{S}$ in the detector model,
- the input valuation $i \in (\mathbf{I} \rightarrow \mathbb{V}^{\perp})$ containing the values of inputs,
- the timer valuation $t \in (\mathbf{T} \rightarrow (\mathbb{N}^{\perp} \times \mathbb{N}^{\perp}))$ for user-defined timers. Each timer has both a periodicity and (if active) a time remaining, and
- the global valuation $g \in (\mathbf{G} \rightarrow \mathbb{V}^{\perp})$ for global variables in the detector model.

Example 1. Consider a corrected version of our example detector model from Fig. 1 which has two timers, `Wait1Min` and `Wait5Min`, and no global variables. Some examples of configurations for this model are:

- $\langle \texttt{TempOK}, \{\texttt{temp} : \perp\}, \{\texttt{Wait1Min} : (\perp, \perp), \texttt{Wait5Min} : (\perp, \perp)\}, \{\}\rangle$ is the initial configuration. The model contains input `temp`, timers `Wait1Min` and `Wait5Min`, and no global variables. As no variables or timers have been assigned, all variables have value undefined ($\perp$).

$$\boxed{\begin{array}{ll} C \vdash_{\epsilon} e \rightarrow v & \\ C \vdash_{\alpha} a \rightarrow C' & C \vdash_{\alpha *} al \rightarrow C' \\ C \vdash_{\kappa} k \rightarrow C' & C \vdash_{\kappa *} kl \rightarrow C' \\ C \vdash_{\mu *} ml \rightarrow C' & C \vdash_{\mathcal{E}_I} \mathcal{E}_I \rightarrow C' \\ \vdash_{\iota} C \overset{i}{\mapsto} C' & \end{array}}
\qquad
\frac{\langle s, i, t, g \rangle \vdash_{\epsilon} e \rightarrow v}{\begin{array}{l}\langle s, i, t, g \rangle \vdash_{\alpha} \texttt{setTimer}(tr, e) \rightarrow \\ \langle s, i, t[tr \leftarrow (v, v)], g \rangle\end{array}}$$

$$\frac{t(tr) = (p, v)}{\begin{array}{l}\langle s, i, t, g \rangle \vdash_{\alpha} \texttt{resetTimer}(tr) \rightarrow \\ \langle s, i, t[tr \leftarrow (p, p)], g \rangle\end{array}}
\qquad
\frac{t(tr) = (p, v)}{\begin{array}{l}\langle s, i, t, g \rangle \vdash_{\alpha} \texttt{clearTimer}(tr) \rightarrow \\ \langle s, i, t[tr \leftarrow (p, \bot)], g \rangle\end{array}}$$

$$\frac{\langle s, i, t, g \rangle \vdash_{\epsilon} e \rightarrow v}{\begin{array}{l}\langle s, i, t, g \rangle \vdash_{\alpha} \texttt{setGlobal}(gv, e) \\ \rightarrow \langle s, i, t, g[gv \leftarrow v] \rangle\end{array}}
\qquad
\frac{C \vdash_{\epsilon} e \rightarrow \texttt{false}}{C \vdash_{\kappa} \texttt{event}(e, al) \rightarrow C}
\qquad
\frac{\begin{array}{l}C \vdash_{\epsilon} e \rightarrow \texttt{true} \\ C \vdash_{\alpha *} al \rightarrow C'\end{array}}{C \vdash_{\kappa} \texttt{event}(e, al) \rightarrow C'}$$

$$\frac{}{C \vdash_{\mu *} \texttt{nil} \rightarrow C}
\qquad
\frac{\begin{array}{l}C \vdash_{\epsilon} e \rightarrow \texttt{false} \\ C \vdash_{\mu *} tl \rightarrow C'\end{array}}{C \vdash_{\mu *} \texttt{transition}(e, al, s') :: tl \rightarrow C'}$$

$$\frac{\begin{array}{l}C \vdash_{\epsilon} e \rightarrow \texttt{true} \qquad C \vdash_{\alpha *} al \rightarrow C' \\ C' \vdash_{\kappa *} \mathcal{E}_X(C.s) \rightarrow C'' \\ C''[s \leftarrow s'], ti \vdash_{\kappa *} \mathcal{E}_E(s') \rightarrow C'''\end{array}}{C \vdash_{\mu *} \texttt{transition}(e, al, s') :: tl \rightarrow C'''}
\qquad
\frac{\begin{array}{l}C \vdash_{\kappa *} kl \rightarrow C' \\ C' \vdash_{\mu *} ml \rightarrow C''\end{array}}{C \vdash_{\mathcal{E}_I} (kl, ml) \rightarrow C''}$$

$$\frac{\begin{array}{l}\texttt{matchesEarliest}(C.t, ti) \wedge \texttt{subtractTimers}(C, ti) \rightarrow C' \\ C' \vdash_{\mathcal{E}_I} \mathcal{E}_I(C'.s) \rightarrow C'' \wedge \texttt{clearTimers}(C'') \rightarrow C'''\end{array}}{\vdash_{\iota} C \xmapsto{timeout(ti)} C'''}
\qquad
\frac{\langle s, i[iv \leftarrow v], t, g \rangle \vdash_{\mathcal{E}_I} \mathcal{E}_I(C.s) \rightarrow C'}{\vdash_{\iota} \langle s, i, t, g \rangle \xmapsto{message(iv, v)} C'}$$

Fig. 4. Rules describing behavior of the system

- $\langle \texttt{TooHot}, \{\texttt{temp} : 300\}, \{\texttt{Wait1Min} : (60, \bot), \ \texttt{Wait5Min} : (300, 260)\}, \{\} \rangle$ is the configuration at global time t if the temperature is still beyond the normal range and we transition to the `TooHot` detector model state. Note the `Wait1Min` timer is no longer set whereas the `Wait5Min` timer has a periodicity of 300 and is set to expire at $t + 260$.

To define the execution semantics, we create a structural operational semantics for each of the grammar rules and for the interaction with the external environment, as shown in Fig. 4. We distinguish semantic rules by decorating the turnstiles with the grammar type that they operate over ($\epsilon, \alpha, \kappa, \mu, \mathcal{E}_I$, and ι). The variables e, a, k, m, i stand in for elements of the appropriate syntactic class defined by the turnstile. For lists of elements, we decorate the syntactic class with * (e.g. $\vdash_{\alpha *}$), and the variables with 'l' (e.g. al). We use the following notation conventions: Given $C = \langle s, i, t, g \rangle$, we say $C.s = s$, and similarly with the other components of C. We also say $C[s \leftarrow s']$ is equivalent to $\langle s', i, t, g \rangle$, and similarly with the other components of C.

Expressions ($\vdash_{\epsilon}$) evaluate to values, given a configuration. We do not present expression rules (they are simple), but illustrate the other rule types in Fig. 4. For actions ($\vdash_{\alpha}$), the *setTimer* rule establishes the periodicity of a timer and also starts it. The *resetTimer* and *clearTimer* rules restart an existing timer given a periodicity p or clear it, respectively, and the *setGlobal* rule updates

the value of a global variable. *Events* (κ) are used by entry and exit events for states. The list rules for actions ($\alpha*$) and events ($\kappa*$) are not presented but are straightforward: they apply the relevant rule to the head of the list and pass the updated configuration to the remainder of the list, or return the configuration unchanged for `nil`. *Transition event lists* ($\mu*$) cause the system to change state, executing (only) the first transition from the list whose guard e evaluates to true. Finally, the top-level rule $\vdash_\iota$ describes how the system evolves according to external stimuli.

A *run* of the machine is any valid sequence of configurations produced by repeated applications of the $\vdash_\iota$ rule. Timeout inputs increment the time to the earliest active timeout as described by the *matchesEarliest* predicate:

$$\begin{aligned}\texttt{matchesEarliest}(t, x) \equiv \exists t_i, p_i.(p_i, x) = t(t_i) \wedge \\ \forall t_j, p_j, y.((p_j, y) = t(t_j) \implies y = \bot \vee y \geq x)\end{aligned}$$

The `subtractTimers` function subtracts t_i from each timer in C, and the `clearTimers` function, for any timers whose time remaining is equal to zero, calls the `clearTimer` action[1].

3.1 Well-formedness Properties

To find common issues with detector models, we surveyed (i) detector models across customer tickets submitted to AWS IoT Events, (ii) questions posted on internal forums like the AWS re:Post forum [15], and (iii) feedback submitted via the web-based console for AWS IoT Events. Based on this survey, we determined that the following correctness properties should hold over all detector models. For more details about this survey, please refer to Appendix A.

The Model does not Contain Type Errors: The AWS IoT Events expression language is *untyped*, and thus, may contain ill-typed expressions, e.g., performing arithmetic operations on Booleans. A large class of such bugs can be readily detected and prevented using a *type inference* algorithm. The algorithm follows the standard Hindley-Milner type unification approach [16–18] and generates (and solves) a set of type constraints or reports an error if no valid typing is possible. We use this type inference algorithm to detect type errors in the detector model. Every type error is reported as a warning to the customer. When our type inference successfully infers types for expressions, we use them to construct a well-typed abstract state machine using the formalization reported in Sect. 3.

For the remaining well-formedness properties we use model checking. We introduce one or more *indicator variables* in our global abstract state to track certain kinds of updates in the state machine, and then we assert temporal properties on these indicator variables. Because we use a model checker that

[1] In the interests of space, we do not cover the *batch* execution mode, where all variables used in expressions maintain their "pre-state" value until the step is completed; it is a straightforward extension.

checks only *safety properties*, in many cases we invert the property of interest and check that its negation is falsifiable, using the same mechanism often used for test-case generation [19].

Every Detector Model State is Reachable and Every Detector Model Transition and Event can be Executed: For each state $s \in \mathbf{S}$, we add a new Boolean *reachability indicator* variable v^s_{reached} to our abstract state that is initially `false` and assigned `true` when the state is entered (similarly for transitions and events). To encode the property in a safety property checker, we encode the following *unreachability* property expressed in LTL and check it is falsifiable. If it is provable, the tool warns the user.

$$\text{Unreachable}(s) \triangleq \Box\ (\neg\ v^s_{\text{reached}})$$

Every Variable is Set Before Use: In order to test that variables are properly initialized, first we identify the places where variables are assigned and used. In detector models, there are three places where variables are used: in the evaluation of conditions for events and transitions, and in the `setGlobal` action (which occurs because of an event or transition). We want to demonstrate that the variables used within these contexts are never equal to $\bot$ during evaluation. In this case, we can reuse the reachability variables that we have created for events and transitions to encode that variables should always have defined values when they are used.

We first define some functions to extract the set of variables used in expressions and action lists. The function $Vars(e) : \epsilon \rightarrow v$ `set` simply extracts the variables in the expression. For action lists, it is slightly more complex, because variables are both defined and used:

$$\begin{aligned}
Vars(\texttt{nil}) &= \{\} \\
Vars(\texttt{setTimer}(t, e) :: tl) &= Vars(e) \cup Vars(tl) \\
Vars(\texttt{resetTimer}(t) :: tl) &= Vars(tl) \\
Vars(\texttt{clearTimer}(t) :: tl) &= Vars(tl) \\
Vars(\texttt{setGlobal}(g, e) :: tl) &= Vars(e) \cup (Vars(tl) - \{g\}) \\
Vars(\texttt{event}(e, al)) &= Vars(e) \cup Vars(al) \\
Vars(\texttt{transition}(e, al, s')) &= Vars(e) \cup Vars(al)
\end{aligned}$$

Every event or transition can be executed at most once during a computation step, so we can use the execution indicator variables to determine when a variable might be used.

$$\begin{aligned}
&\forall a_i, v_j \in Vars(a_i)\ . \\
&\quad \text{SetBeforeUse}(a_i, v_j) \triangleq \Box\,(v^{a_i}_{\text{exec}} \implies v_j \neq \bot)
\end{aligned}$$

Input Read Only on Message Trigger: This property is covered in the previous property, with one small change. To enforce it, we modify the translation of the semantics slightly so that at the beginning of each step, prior to processing the input message, all input variables are assigned $\bot$.

Message Triggered Between Consecutive Timeouts: We conservatively approximate a liveness property (no infinite path consisting of only timeout events) with a safety property: the same timer should not timeout twice without an input message occurring in between the timeouts. This formulation may flag models that do not have infinite paths with no input events, but our customers consider it a reasonable indicator.

We begin by defining an indicator variable for each timer t_i (of type integer rather than Boolean): v^i_{timeout} and initialize it to zero. We modify the translation of `updateTimers` to increment this variable when its timer variable equals zero, and modify the translation of the *message* rule to reset all v^i_{timeout} variables to zero. The property of interest is then:

$$\text{NoConsecutiveTimeouts}(t_i) \triangleq \square \ \left(v^i_{\text{timeout}} < 2\right)$$

4 Experiments

In this section, we evaluate the performance of model-checking safety properties on detector models, with a focus on model checking latency. Low analysis latency is crucial because our tool warns customers of property violations while they are editing their detector model. Our type inference implementation runs with an average latency of 10 milliseconds on all the detector models in our experiments. Since type inference is much faster than model checking and can be successfully run on all detector models, we do not evaluate it in this section.

AWS IoT Events has a commercial feature [20] which uses the type checking and model checking described in Sect. 3. The feature's implementation first infers types using the type inference algorithm. Next, it translates the detector model into the Lustre language [21]. The translation of IoT Events into Lustre is straightforward and directly follows from the semantics presented in Sect. 3. The safety properties described in Sect. 3.1 are attached to the model, along with location information. Then the feature analyzes the model using the JKind [1] tool suite, an open-source industrial model-checker. If JKind invalidates a safety property, the feature decodes the location from the safety property and includes it in the warning.

To evaluate this implementation, we randomly selected 210 detector models previously analyzed by the commercial feature. We checked the five properties described in Sect. 3.1 in parallel on a c4.8xlarge EC2 instance running Amazon Linux 2 x86_64 using JKind version 4.4.1, with a timeout of 60 s.

Of the safety properties that we were able to translate to Lustre, JKind resolved 96% within our timeout of 60 s, with 80% completing in less than 10 s.

Table 1 shows that checking the *no-unreachable-action* safety property requires the most time to complete. The detector models analyzed in the evaluation include models for monitoring self-driving wheel chairs, monitoring device connectivity, humidity, temperature, pressure, oil level, oil temperature, doors, motion, refrigerator temperature, dough fermentation, and vehicle speed-sensing. They consisted of between 1–7 states and from 0–14 state changes. The *no-unreachable-action* safety property is checked on every action, generating an

Table 1. Performance of our model-checking tool against 210 detector models

safety property	avg. latency (milliseconds)	# completed	# translation failed	# timeout
no-unreachable-state	3544	176	28	6
no-unreachable-action	5586	171	28	11
var-always-set-before-use	2968	179	28	3
no-infinite-timer-expiration	2875	174	28	8
no-input-read-with-timer-expiration	5477	177	30	3

average of 17 safety properties per detector model, the most of any kind of safety property. This large number of properties to be checked on every detector model caused checking the *no-unreachable-action* safety property to have the highest average latency (5.6 s per analysis).

Table 1 shows that about 13% of the properties could not be translated to Lustre. In 2% of the detector models, translation failures arose due to type errors or incorrect use of the AWS IoT Events expression language in the detector model. The remaining translation failures occurred due to either: (1) use of operations not supported by Lustre, (2) no types being inferred for inputs or variables in the detector model, or (3) use of non-linear arithmetic, which is unsupported in JKind. Bitwise functions, strings, and array data types are supported in the AWS IoT Events expression language but not in Lustre. This language gap prevented us from translating 19 of the 210 detector models. Failing to infer a type for a variable in the detector model prevented translation of 6 of the 210 detector models. JKind's lack of support for non-linear arithmetic prevented model-checking 2 of the 210 detector models. We are actively working to support more functions, string and array data types, type annotations, and non-linear arithmetic in our model-checking of detector models.

5 Conclusion

Our analyzers have been running in the AWS IoT Events production service since December 2021. Since then, 93% of AWS IoT Events customers have used our implementation to check their detector models for well-formedness, without needing to have any knowledge of the underlying type checking and model checking. Our analyzers successfully complete for 85% of real-world detector models and we are actively working on improving this support as explained in Sect. 4. Overall, our implementation has reported well-formedness property violations in 22% of submitted detector models in the production service, with an average latency of 5.6 s. We find giving customers push-button access to fast verification without requiring any knowledge of the underlying techniques enables adoption of automated reasoning-based tools.

A Common Issues with Detector Models

Table 2. Issues seen in detector models from customer questions

#	Issue	# of instances
1	incorrectly scaling detector model	1
2	unreachable action	2
3	infinite loop	3
4	variable-used-before-set	3
5	input read on timer expiration	3
6	insufficient logging permissions	3
7	incorrectly typed expression	5
8	incorrect cross-service setup	8
9	missing simplifications	8
		36

As mentioned in Sect. 3.1, we surveyed customer detector models for generic correctness problems. We present the root causes of the problems from this study in Table 2. Incorrect scaling (#1) occurs when the customer does not set up their detector model to be instantiated correctly for every IoT device in their fleet. Infinite loop (#3) occurs when the detector model has an infinite execution path involving only timeout events and no external input messages. IoT models should be eventually quiescent if no external inputs occur.

Variable-used-before-set (#4) occurs when a variable in the detector's state is read from before being set to an initial value. AWS IoT Events does not require variables in detector models to be initialized.

A step through a detector can be triggered due to both a timer expiration or a new value being sent to the detector by the outside world. Input read on timer expiration (#5) occurs when a step, triggered by timer expiration, causes the detector to read from its input(s). This is a problem because customers often do not realize that such a read will return the last value sent to the detector by the outside world. Insufficient logging permissions (#6) occurs when a detector is not given sufficient permissions to produce logging output. Incorrect cross-service setup (#8) occurs when customers do not correctly set up data flow across services in AWS IoT. While unnecessarily complex detector models (#9) is not a correctness problem, it poses a significant difficulty to customers in maintaining their detector models, and so, we include it in Table 2.

Of these 9 root causes, we identified that type checking and model checking detected 5 root causes highlighted in green in Table 2. These 5 root causes were responsible for 44% of issues in our survey. Based on Table 2, we determined that the following correctness properties should hold over all detector models:

1. *Detector models must be well-typed*
2. *Every detector model state must be reachable*

3. *Every detector model action must be executable*
4. *Every variable must be set before being used*
5. *Input reads shall not happen on timer expiration*
6. *Detector model must not have infinite timer expirations*

We explain these properties further s in Sect. 3.1.

References

1. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 20–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_3
2. Cho, C.Y., D'Silva, V., Song, D.: Blitz: compositional bounded model checking for real-world programs. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 136–146 (2013)
3. Baranová, Z., et al.: Model checking of c and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC/SIGSOFT FSE -1999. LNCS, vol. 1687, pp. 146–162. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48166-4_10
5. Karamanolis, C., Giannakopoulou, D., Magee, J., Wheater, S.M.: Model checking of workflow schemas. In: Proceedings Fourth International Enterprise Distributed Objects Computing Conference, EDOC 2000, pp. 170–179 (2000)
6. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to model checking. In: Handbook of Model Checking, pp. 1–26. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_1
7. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model checking. cyber physical systems series (2018)
8. Bradley, A.R.: Incremental, inductive model checking. In: 2013 20th International Symposium on Temporal Representation and Reasoning, pp. 5–6 (2013)
9. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: from refutation to verification. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_2
10. Newcomb, J.L., Chandra, S., Jeannin, J.-B., Schlesinger, C., Sridharan, M.: IOTA: a calculus for internet of things automation. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ser. Onward! 2017. New York, NY, USA, pp. 119–133. Association for Computing Machinery (2017). https://doi.org/10.1145/3133850.3133860
11. Liang, C.-J. M., et al.: SIFT: building an internet of safe things. In: Proceedings of the 14th International Conference on Information Processing in Sensor Networks, ser. IPSN 2015. New York, NY, USA, pp. 298–309. Association for Computing Machinery (2015). https://doi.org/10.1145/2737095.2737115
12. García-Herranz del Olmo, M., Haya, P.A., Alamán, X.: Towards a ubiquitous end-user programming system for smart spaces. J. Univers. Comput. Sci. **16**(12), 1633–1649 (2010)

13. Ur, B., McManus, E., Pak Yong Ho, M., Littman, M.L.: Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI 2014. New York, NY, USA, pp. 803–812. Association for Computing Machinery (2014). https://doi.org/10.1145/2556288.2557420
14. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
15. Amazon Web Services, Inc., Find answers to AWS questions about AWS IoT Events, AWS (2021). https://repost.aws/tags/TANsxSwnClQ_Wfh-uklXi7hQ
16. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL 1982, New York, NY, USA, pp. 207–212. Association for Computing Machinery (1982). https://doi.org/10.1145/582153.582176
17. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://www.sciencedirect.com/science/article/pii/0022000078900144
18. Hindley, R.: The principal type-scheme of an object in combinatory logic. Trans. Am. Math. Soc. **146**, 29–60 (1969). http://www.jstor.org/stable/1995158
19. Gay, G., Staats, M., Whalen, M., Heimdahl, M.P.E.: The risks of coverage-directed test case generation. IEEE Trans. Softw. Eng. **41**(8), 803–819 (2015)
20. AWS IoT Events, Troubleshooting a detector model by running analyses (2021). https://docs.aws.amazon.com/iotevents/latest/developerguide/iotevents-analyze-api.html
21. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Learning Assumptions for Compositional Verification of Timed Automata

Hanyue Chen[1], Yu Su[1], Miaomiao Zhang[1](✉), Zhiming Liu[2], and Junri Mi[1]

[1] Tongji University, Shanghai, China
{2111285,lemon,miaomiao,18mjr}@tongji.edu.cn
[2] Southwest University, Chongqing, China
zhimingliu88@swu.edu.cn

Abstract. Compositional verification, such as the technique of assume-guarantee reasoning (AGR), is to verify a property of a system from the properties of its components. It is essential to address the state explosion problem associated with model checking. However, obtaining the appropriate assumption for AGR is always a highly mental challenge, especially in the case of timed systems. In this paper, we propose a learning-based compositional verification framework for deterministic timed automata. In this framework, a modified learning algorithm is used to automatically construct the assumption in the form of a deterministic one-clock timed automaton, and an effective scheme is implemented to obtain the clock reset information for the assumption learning. We prove the correctness and termination of the framework and present two kinds of improvements to speed up the verification. We discuss the results of our experiments to evaluate the scalability and effectiveness of the framework. The results show that the framework we propose can reduce state space effectively, and it outperforms traditional monolithic model checking for most cases.

1 Introduction

Model checking [9,19,33,36] is an important technique to automatically determine whether a system satisfies a specified property. However, it suffers from the state explosion problem since it needs to store the explored system states in memory, which is impossible for most realistic systems [21]. In timed systems, although symbolic representations and partial order reductions have greatly increased the size of the systems that can be verified, many realistic timed systems are still too large to be handled. In particular, if a system has several components, the number of global system states will grow exponentially with the number of components. Assume-guarantee reasoning (AGR) [20,25,29,35] is a promising method helpful to address the state explosion problem.

Consider a system M composed of two components M_1 and M_2 that synchronize on a given set of shared actions. Supposing we are to verify that M satisfies a property ϕ, the verification rule in AG states that if there exists an assumption

This work has been funded by NSFC under grant No.61972284 and No.62032019.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 40–61, 2023.
https://doi.org/10.1007/978-3-031-37706-8_3

A on the environment of M_2 such that 1) M_1 and A satisfy the property ϕ, and 2) M_2 is a refinement of A, then M satisfies ϕ.

A major challenge in verifying component-based systems using the AG rule is the need to obtain the appropriate assumption that requires non-trivial human effort [26]. Based on abstraction-refinement paradigm in [22], the assumption is computed as a conservative abstraction of some of the components, and it is then refined using counterexamples obtained from model checking it [15]. The algorithm presented in [24] is capable of generating the weakest possible assumption automatically, though it does not compute partial results. In the later work [23], a framework is proposed for the automatic generation of assumptions in an incremental fashion using the L^* learning algorithm [8]. Several improvements, e.g. [14,17,18,38], are proposed to further reduce the learning complexity. The work [6] by Alur et al. presents a symbolic implementation of the L^* algorithm where the required data structures are maintained compactly using ordered BDDs [16].

All the aforementioned work focuses on untimed systems. For timed systems, using assume-guarantee style proof rules, the work in [39] proves a refined representation is a correct implementation of an abstract one. To check Zeroconf, a protocol for dynamic configuration of IPv4 link-local addresses, Berendsen et al. [12] model the protocol as a network of timed automata (TAs) [3,4], and provide a proof that combines model checking with the application of a new abstraction relation that is compositional with respect to committed locations. However, the abstract models there are all provided manually. Compared to the manual methods, the compositional verification framework presented in [31,32] utilizes a learning algorithm for automatic construction of timed assumptions for AGR. The work considers event-recording automata [5], which are a subclass of timed automata. Sankur [37] gives compositional verification for the system composed by a deterministic finite automaton (DFA) and a timed automaton, where a DFA assumption is learned [27] to approximate the timed component. The framework can only check the untimed property of the system and it has the limitation that the TA size is relatively small.

The timed automaton is the most appreciated model for its simplicity and adequacy in expressiveness, and it is widely used for practical real-time systems [28,30]. However, to the best of our knowledge, though compositional verification for timed systems helps mitigate the state space explosion problem, there is still no work to tackle the problem of automatically inferring the timed assumptions based on AGR for timed automata. Therefore, we propose, in this paper, a learning-based framework for AG-based automatic verification of deterministic timed automata. The framework applies the compositional rule in an iterative fashion. Each iteration consists of three steps. In the first step, based on the work in [7], a modified L^* algorithm is presented to learn a timed assumption in the form of a deterministic one-clock timed automata (DOTAs) using membership queries. Then two further steps are conducted to check whether the learned assumption satisfies the two premises of the proof rule via candidate queries. We design an algorithm for model conversion with polynomial complexity, which

is executed as a step preceding the above iterative steps. It converts the input models M_1, M_2 and ϕ to the output ones, which contain the clock reset information for the assumption learning. Thus, the complexity of the learning step in the framework in total is polynomial. We show this conversion preserves the verification results.

We further prove the correctness and termination of the compositional verification. We would like to note that the framework we propose applies to verification of systems with a number of components. In other words, though the assumption learned is a DOTA, M_1 and M_2 can be compositions of several DOTAs. For this, we design a heuristic to transform multi-clock reset information to one-clock reset information, which enables the framework to handle learning-based compositional verification for multi-clock systems. We also propose two improvements to speed up the verification, which are shown to have different advantages in cases of experiments. Finally, we implement the framework and conduct comparative experiments with UPPAAL [10,11] on cases of the benchmark of AUTOSAR (Automotive Open System Architecture) [1]. The experiments show that the framework proposed in this paper performs better than that of UPPAAL provided the properties to be checked are satisfied.

The rest of the paper is organized as follows. In Sect. 2, we introduce background knowledge. We present in Sect. 3 our learning-based compositional verification framework, as well as the proofs of termination and correctness. In Sect. 4, we present the two improvements. We report the experimental results in Sect. 5. Finally, we discuss the conclusions of the paper in Sect. 6.

2 Preliminaries

We use $\mathbb{N}$ to denote the set of natural numbers, $\mathbb{R}_{\geq 0}$ the set of non-negative reals, and let $\mathbb{B} = \{\top, \bot\}$, where $\top$ and $\bot$ stand for true and false, respectively.

2.1 Timed Automata

Let X be a finite set of real-valued variables ranged over by x, y, etc. standing for clocks. A clock valuation for X is a function $\nu : X \mapsto \mathbb{R}_{\geq 0}$ which associates every clock x with a value $\nu(x) \in \mathbb{R}_{\geq 0}$. For $t \in \mathbb{R}_{\geq 0}$, let $\nu + t$ denote the clock valuation which maps every clock $x \in X$ to the value $\nu(x) + t$. For a set $\gamma \subseteq X$ and a valuation ν, we use $[\gamma \to 0]\nu$ to denote the valuation which resets all clock variables in γ to 0 and agrees with ν for other clocks in $X \backslash \gamma$.

We use $\Phi(X)$ to denote the set of clock constraints over X of the form $\varphi ::= \top \mid x_1 \bowtie m \mid x_1 - x_2 \bowtie m \mid \varphi \wedge \varphi$, where $x_1, x_2 \in X$, $m \in \mathbb{N}$ and $\bowtie \in \{=, <, >, \leq, \geq\}$. We use $\varphi(\nu) = \top$ to mean that the clock valuation ν for X satisfies the clock constraint φ over X, i.e. φ evaluates to true using the values given by ν.

Definition 1 (Timed Automata). *A timed automaton (TA) is a 6-tuple $M = (Q, q_0, \Sigma, F, X, \Delta)$, where Q is a finite set called the locations, $q_0 \in Q$ is the*

initial location, Σ is a finite set called the alphabet, $F \subseteq Q$ is the set of accepting locations, X is the finite set of clocks, and $\Delta \subseteq Q \times \Sigma \times \Phi(X) \times 2^X \times Q$ is a finite set called the transitions.

A transition $\delta \in \Delta$ is a 5-tuple $(q, \sigma, \varphi, \gamma, q')$, where $q, q' \in Q$ are respectively the source and target locations, $\sigma \in \Sigma$ is an action, φ is a clock constraint over X which is called the guard of the transition and specifies that the transition is enabled when it is true in the source state, and the set $\gamma \subseteq X$ gives the reset clocks by this transition. Thus, δ allows a jump from q to q' by performing an action σ if it is enabled, i.e. $\varphi(\nu) = \top$. We use $\delta[i]$ to denote the i'th element of the tuple $\delta = (q, \sigma, \varphi, \gamma, q')$ for $i = 1, \ldots, 5$. A *run* ρ of M is a finite sequence of transitions $\rho = (q_0, \nu_0) \xrightarrow{\sigma_1, t_1} (q_1, \nu_1) \xrightarrow{\sigma_2, t_2} \cdots \xrightarrow{\sigma_n, t_n} (q_n, \nu_n)$ where $\nu_0 = \{\nu(x) | \nu(x) = 0, x \in X\}$, and for all $1 \leq i \leq n$ there exists a transition $(q_{i-1}, \sigma_i, \varphi_i, \gamma_i, q_i) \in \Delta$ such that $\varphi_i(\nu_{i-1} + t_i) = \top$, and $\nu_i = [\gamma_i \rightarrow 0](\nu_{i-1} + t_i)$. If q_n is an accepting location, we say ρ is an *accepting run* of M. Each pair $(\sigma_i, t_i) \in \Sigma \times \mathbb{R}_{\geq 0}$ in the run ρ is called a *timed action* that indicates the action σ_i is applied after t_i time units since the occurrence of the previous action.

The *timed trace* of ρ is a *timed word trace*$(\rho) = (\sigma_1, t_1)(\sigma_2, t_2) \ldots (\sigma_n, t_n)$. Since time value t_i represents *delay time*, we also call such a timed trace a *delay-timed word*, denoted by ω. Adding the reset information along ω, we get the corresponding *reset-delay-timed word*, denoted by $\omega_r = trace_r(\rho) = (\sigma_1, t_1, \gamma_1)(\sigma_2, t_2, \gamma_2) \cdots (\sigma_n, t_n, \gamma_n)$. Notice that here γ_i is a clock set $\gamma_i \subseteq X$ which records the reset clocks in the corresponding transition when taking timed action (σ_i, t_i).

If ρ is an accepting run of M, *trace*(ρ) is called an *accepting timed word*. The *recognized timed language* of M is the set of its accepting delay-timed words, i.e. $\mathcal{L}(M) = \{trace(\rho) \mid \rho \text{ is an accepting run of } M\}$. The *recognized reset-delay-timed language* $\mathcal{L}_r(M)$ is defined as $\{trace_r(\rho) \mid \rho \text{ is an accepting run of } M\}$. A TA M is *deterministic* iff for any given delay-timed word ω, there is at most one run ρ in M having *trace*$(\rho) = \omega$.

For a run ρ, we define the corresponding *logical-timed word* $\omega_l = (\sigma_1, \mathbf{v}_1)(\sigma_2, \mathbf{v}_2) \cdots (\sigma_n, \mathbf{v}_n)$, where $\mathbf{v}_i \in \mathbb{R}_{\geq 0}^{|X|}$ is the vector which records the values for all clocks in X. Therefore, delay-timed words and logical-timed words describe the operations of the timed model M from different perspectives. The former describe M from the external perspective, recording the actions and time intervals between two consecutive actions. While the latter describe it from the internal perspective, recording the actions and the specific values of internal clocks when the actions occur. Both are necessary for the active learning algorithm described in Sect. 2.2.

Given the clock reset information γ_i along the run ρ over the delay-timed word $\omega = (\sigma_1, t_1)(\sigma_2, t_2) \ldots (\sigma_n, t_n)$, we can obtain ω's corresponding logical-timed word $\omega_l = (\sigma_1, \mathbf{v}_1)(\sigma_2, \mathbf{v}_2) \cdots (\sigma_n, \mathbf{v}_n)$ by taking

$$\mathbf{v}_i[j] = \begin{cases} t_i, & \text{if } \ i = 1 \ \text{ or } \ x_j \in \gamma_{i-1} \text{ for all } 2 \leq i \leq n; \\ \mathbf{v}_{i-1}[j] + t_i, & \text{otherwise.} \end{cases} \tag{1}$$

where $1 \leq j \leq |X|$ and $\mathbf{v}_i[j]$ is the j'th element in $\mathbf{v}_i$. We use Γ to denote the mapping from the delay-timed words to the logical-timed words, that is, $\Gamma(\omega) = \omega_l$. With the reset information along the run ρ, we have the *reset-logical-timed word* $\omega_{rl} = (\sigma_1, \mathbf{v}_1, \gamma_1)(\sigma_2, \mathbf{v}_2, \gamma_2) \ldots (\sigma_n, \mathbf{v}_n, \gamma_n)$. We can extend the mapping Γ to a mapping from the reset-delay-timed words to the reset-logical-timed words.

The *recognized logical-timed language* of M is given as $L(M) = \{\Gamma(\mathit{trace}(\rho)) \mid \rho \text{ is an accepting run of } M\}$, and the *recognized reset-logical-timed language* of M is $L_r(M) = \{\Gamma(\mathit{trace}_r(\rho)) \mid \rho \text{ is an accepting run of } M\}$.

Definition 2 (Projection of Delay-Timed Words). *Given a delay-timed word $\omega = (\sigma_1, t_1)\,(\sigma_2, t_2) \ldots (\sigma_n, t_n) \in (\Sigma_1 \times \mathbb{R}_{\geq 0})^*$ and an alphabet Σ_2, the projection of ω to Σ_2 is a delay-timed word, denoted by $\omega\downharpoonright_{\Sigma_2}$, and defined as follows:*

$$\omega\downharpoonright_{\Sigma_2} = \left(\sigma_{i_1}, \textstyle\sum_{j=1}^{i_1} t_j\right) \left(\sigma_{i_2}, \textstyle\sum_{j=i_1+1}^{i_2} t_j\right) \ldots \left(\sigma_{i_m}, \textstyle\sum_{j=i_{m-1}+1}^{i_m} t_j\right) \tag{2}$$

where $\sigma_{i_k} \in \Sigma_2$ is the i_k'th action in ω, $1 \leq k \leq m$.

Therefore, $\omega\downharpoonright_{\Sigma_2}$ restricts each action σ_{i_k} to be in Σ_2 and modifies the corresponding delay time of σ_{i_k} to be the time interval between $\sigma_{i_k - 1}$ and σ_{i_k} in ω. For instance, let $\omega = (a,1)(b,3)(a,1)(c,4)(a,2)$ and $\Sigma_2 = \{b, c\}$, then the corresponding $\omega\downharpoonright_{\Sigma_2} = (b,4)(c,5)$.

Definition 3 (Parallel Composition of Timed Automata). *Given two timed automata $M_1 = (Q_1, q_0^1, \Sigma_1, F_1, X_1, \Delta_1)$ and $M_2 = (Q_2, q_0^2, \Sigma_2, F_2, X_2, \Delta_2)$, assume that the clock sets X_1 and X_2 are disjoint. Their parallel composition is a TA $M_1 \| M_2 = (Q_1 \times Q_2, (q_0^1, q_0^2), \Sigma_1 \cup \Sigma_2, F_1 \times F_2, X_1 \cup X_2, \Delta)$ where the transitions Δ are as follows:*

- *for $\sigma \in \Sigma_1 \cap \Sigma_2$, for every $\delta_1 : (q_1, \sigma, \varphi_1, \gamma_1, q_1') \in \Delta_1$ and $\delta_2 : (q_2, \sigma, \varphi_2, \gamma_2, q_2') \in \Delta_2$, $((q_1, q_2), \sigma, \varphi_1 \wedge \varphi_2, \gamma_1 \cup \gamma_2, (q_1', q_2')) \in \Delta$.*
- *for $\sigma \in \Sigma_1 \setminus \Sigma_2$, for every $\delta_1 : (q_1, \sigma, \varphi_1, \gamma_1, q_1') \in \Delta_1$ and every $q \in Q_2$, $((q_1, q), \sigma, \varphi_1, \gamma_1, (q_1', q)) \in \Delta$.*
- *for $\sigma \in \Sigma_2 \setminus \Sigma_1$, for every $\delta_2 : (q_2, \sigma, \varphi_2, \gamma_2, q_2') \in \Delta_2$ and every $q \in Q_1$, $((q, q_2), \sigma, \varphi_2, \gamma_2, (q, q_2')) \in \Delta$.*

The language of the composition is the set of accepting delay-timed words and $\mathcal{L}(M_1 \| M_2) = \{\omega | \omega \in ((\Sigma_1 \cup \Sigma_2) \times \mathbb{R}_{\geq 0})^* \text{ and } \omega\downharpoonright_{\Sigma_i} \in \mathcal{L}(M_i), i \in \{1, 2\}\}$.

Definition 4 (Language Inclusion). *Given two timed automata M_1 and M_2, if $\mathcal{L}(M_1)\downharpoonright_{\Sigma_2} = \{\omega\downharpoonright_{\Sigma_2} | \omega \in \mathcal{L}(M_1)\}$ is a subset of $\mathcal{L}(M_2)$, we say M_1 satisfies M_2, denoted by $M_1 \models M_2$.*

Definition 5 (Deterministic One-Clock Timed Automata). *A one-clock timed automaton (OTA) is the timed automaton with only one clock. A deterministic OTA is denoted by DOTA.*

2.2 Learning Deterministic One-Clock Timed Automata

In this section, we briefly describe the active learning algorithm for a DOTA M. We refer to [7] for more details. Active learning of a DOTA assumes the existence of a *teacher* who can answer two kinds of queries: *membership* and *candidate queries* posed by a *learner*. A membership query asks the question if $\omega_l \in L(M)$ for a logical-timed word ω_l; and a candidate query asks if the learned DOTA A represents the assumption satisfies the equation $\mathcal{L}(A) = \mathcal{L}(M)$. The main challenge for learning the timed assumption is to obtain the reset information of the logical clocks for each transition. We consider two different settings, depending on whether the teacher also provides clock reset information along with answers to queries.

A *smart teacher* is one which provides clock reset information along with answers to queries. It accepts a logical-timed word ω_l as an input for the membership query from the learner. It then returns an answer about if the timed word is accepted or not together with reset information of each transition along the trace, that is, the reset-logical-timed word ω_{rl}.

When the *smart teacher* takes a candidate query from the learner, a counterexample is yielded and provided as a reset-delay-timed word. The algorithm maintains a *timed observation table* $\mathbf{T}$ to store answers from all previous queries. Once the learner has gained sufficient information, i.e. $\mathbf{T}$ is *closed* and *consistent*, an assumption A is constructed from the table. Then the learner poses a candidate query to the teacher to judge if $\mathcal{L}(A) = \mathcal{L}(M)$. If yes, the algorithm terminates with the learned model A. Otherwise, the teacher responds with a reset-delay-timed word ω_r as a counterexample. After processing ω_r, the algorithm starts a new round of learning. The whole procedure repeats until the teacher gives a positive answer to a candidate query. It is known that the complexity of the algorithm is polynomial in the size of the learned model. In practical applications, this corresponds to the case where some parts of the model (information of clock reset) are known by testing or watchdogs.

In the case when *normal teacher* is used, the learner needs to guess the reset information on each transition discovered in the observation table. At each iteration, the learner guesses all needed reset information and forms a number of table candidates. Due to the required guesses, the complexity of the algorithm is exponential in the size of the learned model. The following theorem which is presented in [7] shows that for both types of teachers, the algorithm converts the learning problem to that of learning the reset-logical-timed language.

Theorem 1. *Given two DOTAs M and A, if $L_r(M) = L_r(A)$, then $\mathcal{L}(M) = \mathcal{L}(A)$.*

3 Framework for Learning-Based Compositional Verification of Timed Automata

Consider a system $M = M_1 \| M_2$ consisting of two deterministic timed automata and a safety property ϕ represented as a deterministic timed automaton. We

devote this section to presenting our learning-based verification framework for automatically finding an appropriate assumption A in the AG rule to verify that M satisfies ϕ. Section 3.1 first describes the framework. Then, in Sects. 3.2, 3.3 and 3.4, the main algorithms of the framework are presented in detail. Finally, Sect. 3.5 shows the correctness and termination of the framework.

3.1 Verification Framework via Assumption Learning

Let Σ_1, Σ_2 and Σ_ϕ be the alphabets of the TAs M_1, M_2 and ϕ, respectively. We then have that the alphabet of the assumption A_0 is $\Sigma_{A_0} = (\Sigma_1 \cup \Sigma_\phi) \cap \Sigma_2$. The AG rule is stated as follow:

$$\frac{M_1 \| A_0 \models \phi,\ M_2 \models A_0}{M_1 \| M_2 \models \phi} \tag{3}$$

The rule converts the problem of verifying $M_1 \| M_2 \models \phi$ to that of finding an assumption A_0 which is a DOTA satisfying both $M_1 \| A_0 \models \phi$ and $M_2 \models A_0$. Here, we consider M_1 and M_2 as general TAs, which are either a DOTA or compositions of a number of DOTAs. Therefore, the framework we propose is not only applicable to verifying the composition of just two components. For a system composed of n components, where $n > 2$, we can partition the components into two parts. For instance, if a system consists of 4 components $M = \{H_1, H_2, H_3, H_4\}$, we can let $M_1 = H_1 \| H_3$ and $M_2 = H_2 \| H_4$. In order to automatically obtain the assumption, we use model learning algorithms. However, the current learning algorithm for DOTA [7] is not directly applicable. We thus design a "smart teacher" with heuristic to answer clock reset information for the learning. For this, we also need to design a model conversion algorithm. We illustrate the learning-based verification framework in Fig. 1. The inputs of the framework are M_1, M_2 and property ϕ and the verification process consists of four steps, which we describe below.

The First Step. This step converts the input models into TAs M_1', M_2' and ϕ' (ref to Sec. 3.2) without changing the verification results, i.e. checking $M_1' \| M_2'$ against ϕ' is equivalent to checking $M_1 \| M_2$ against ϕ. The output of this step is utilized to determine the clock reset information for the assumption learning in the second step. Then, the AG rule 3 is applied to M_1', M_2' and ϕ'. Thus, if there exists an assumption A such that $M_1' \| A \models \phi'$ and $M_2' \models A$, then $M_1' \| M_2' \models \phi'$. The weakest assumption A_w is the one with which the rule is guaranteed to return conclusive results and $M_1' \| A_w \models \phi'$.

Definition 6 (Weakest Assumption). *Let M_1', M_2' and ϕ' be the models mentioned above and $\Sigma_A = (\Sigma_1' \cup \Sigma_\phi') \cap \Sigma_2'$. The weakest assumption A_w of M_2' is a timed automaton such that the two conditions hold: 1) $\Sigma_{A_w} = \Sigma_A$, and 2) for any timed automaton E with $\Sigma_E = \Sigma_A$ and $M_2' \models E$, $M_1' \| E \models \phi'$ iff $E \models A_w$.*

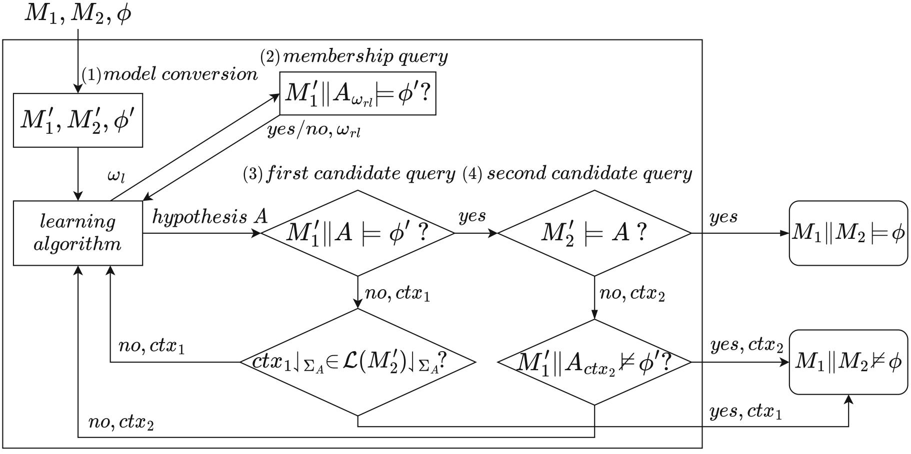


Fig. 1. Learning-based compositional verification framework for timed automata

The Second Step. A DOTA assumption A is learned through a number of membership queries in this step. The answer to each query involves gaining the definite clock reset information for each timed word, i.e. whether the clock of A is reset when an action is taken at a specific time. We design a heuristic to obtain such information from the clock reset information of the converted models M_1', M_2' and ϕ'. This allows the framework to handle learning-based compositional verification for multi-clock systems. We refer to Sect. 3.3 for more details.

The Third and the Fourth Steps. Once the assumption A is constructed, two candidate queries start for checking the compositional rule. The first is a subset query to check whether $M_1' \| A \models \phi'$. The second is a superset query to check whether $M_2' \models A$. If both candidate queries return true, the compositional rule guarantees that $M_1' \| M_2' \models \phi'$. Otherwise, a counterexample *ctx* (either ctx_1 or ctx_2 in Fig. 1) is generated and further analyzed to identify whether *ctx* is a witness of the violation of $M_1' \| M_2' \models \phi'$. If it does not show the violation, *ctx* is used to update A in the next learning iteration. The details about candidate queries are discussed in Sect. 3.4.

Therefore, $\mathcal{L}(A)$ is a subset of $\mathcal{L}(A_w)$ and a superset of $\mathcal{L}(M_2')\downharpoonright_{\Sigma_A}$. It is not guaranteed that a DOTA A can be learned to satisfy $\mathcal{L}(A) = \mathcal{L}(A_w)$. However, as shown later in Theorem 3, under the condition that $\mathcal{L}(A_w)$ is accepted by a DOTA, the learning process terminates when compositional verification returns a conclusive result often before $\mathcal{L}(A_w)$ is computed. This means that verification in the framework usually terminates earlier by finding either a counterexample that verifies that $M_1' \| M_2' \not\models \phi'$ or an assumption A that satisfies the two premises in the reasoning rule, indicating $M_1' \| M_2' \models \phi'$.

Algorithm 1: ConvertW(M_1, M_2, ϕ)

input : Two models M_1 and M_2 and the property ϕ to be verified
output: Converted timed automaton M_1', M_2' and property ϕ'
1 $M_1'', M_2'', \phi'' \leftarrow$ ConvertS(M_1, ϕ, M_2);
2 $\phi', M_1', M_2' \leftarrow$ ConvertS(ϕ'', M_1'', M_2'');
3 **return** M_1', M_2', ϕ';

3.2 Model Conversion

We use membership queries to learn the DOTA assumption. For a membership query with the input of a logical-timed word ω_l, an answer from the teacher is the clock reset information of the word, which is necessary for obtaining the reset-logical-timed word ω_{rl}. As shown in [7], the learning algorithm with a normal teacher can only generate the answer by guessing reset information and this is the cause of high complexity. We thus design a *smart teacher* in our framework scheme. The smart teacher generates the answer to a query with the input ω_l by directly making use of the available clock reset knowledge of $\mathcal{L}(A_w)$ (related with Σ_A, M_1 and ϕ). To this end, we implement the *model conversion* from the models M_1, M_2 and ϕ to the models M_1', M_2' and ϕ', respectively.

The model conversion algorithm is mainly to ensure that each action in Σ_A corresponds to unique clock reset information. Given an action σ having $\sigma \in \Sigma_A$ and $\sigma \in \Sigma_1$ (resp. Σ_ϕ), if there is only one transition by σ or all its different transitions have the same reset clocks, i.e. for any transitions δ_1 and δ_2, $\delta_1[4] = \delta_2[4]$ if $\delta_1[2] = \delta_2[2] = \sigma$, the reset information for the action σ is simply $\delta[4]$ of any particular transition by σ. If there are different transitions by σ, say δ_1 and δ_2, which have different reset clocks, i.e. $\delta_1[4] \neq \delta_2[4]$, we say that the *reset clocks of action σ are inconsistent.*

Reset clock inconsistency causes difficulty for the teacher to obtain the clock reset information of an action in a whole run. To deal with this difficulty, we design model conversion in Algorithm 1 to convert M_1, M_2 and ϕ into M_1', M_2' and ϕ'. In the algorithm, the conversion is implemented by calling Algorithm 2 twice to introduce auxiliary actions and transitions into M_1 and ϕ to resolve reset clock inconsistency in the two automata, respectively.

The converted models M_1', M_2' and ϕ' returned by the invocations to Algorithm 2 have the property that all transitions with the same action $\sigma \in \Sigma_A$ will have the same reset clocks, and thus M_1' and ϕ' do not have reset clock inconsistency. As shown later in Theorem 2, the verification of $M_1' \| M_2'$ against ϕ' is equivalent to that of $M_1 \| M_2$ against ϕ.

Algorithm 2, denoted by ConvertS($\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$), takes three deterministic TAs, namely $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$, as its input and convert them into three new TAs, namely $\mathcal{M}_1'$, $\mathcal{M}_2'$ and $\mathcal{M}_3'$, as the output. We explain the three main functionalities of the algorithm in the following three paragraphs.

Check Reset Information in $\mathcal{M}_1$ (Lines 1-6). Let $\boldsymbol{\Sigma} = (\Sigma_{\mathcal{M}_1} \cup \Sigma_{\mathcal{M}_2}) \cap \Sigma_{\mathcal{M}_3}$, f be a binary relation between $\boldsymbol{\Sigma}$ and 2^X, where X is the set of clocks of $\mathcal{M}_1$, and

Algorithm 2: ConvertS($\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$)

```
   input  : Three timed automata M1, M2, M3
   output: Converted timed automata M'1, M'2 and M'3
 1 M'1 ← M1, M'2 ← M2, M'3 ← M3;
 2 Σ ← (Σ_M1 ∪ Σ_M2) ∩ Σ_M3;
 3 f ← ∅;
 4 for δ ∈ Δ'1 do
 5     if δ[2] ∈ Σ and δ[2] ∉ dom(f) then
 6         put ⟨δ[2], δ[4]⟩ into f;
 7     else if δ[2] ∈ dom(f) and ⟨δ[2], δ[4]⟩ ∉ f then
 8         σ ← δ[2];
 9         σ_new ← introduce_new_action;
10         put σ_new into Σ_M'1, Σ_M'2, Σ_M'3;
11         for δ' ∈ {ω|ω ∈ Δ'1 and ω[2] = σ and ω[4] = δ[4]} do
12             δ'[2] ← σ_new;
13         for δ̄ ∈ {ω|ω ∈ Δ'2 and ω[2] = σ} do
14             δ̄' ←clone(δ̄);
15             δ̄'[2] ← σ_new;
16             put δ̄' into Δ'2;
17         for δ̄ ∈ {ω|ω ∈ Δ'3 and ω[2] = σ} do
18             δ̄' ←clone(δ̄);
19             δ̄'[2] ← σ_new;
20             put δ̄' into Δ'3;
21 return M'1, M'2, M'3;
```

$f = \emptyset$ initially. The transitions of $\mathcal{M}_1$ are checked one by one. For a transition δ, if its action $\delta[2]$ is in $\boldsymbol{\Sigma}$ but not in the domain of f (Line 5). Transition δ is the first transition by $\delta[2]$ found, and thus the pair $\langle \delta[2], \delta[4] \rangle$ is added to the relation f. If the action of δ is already in $dom(f)$ but the reset clocks $\delta[4]$ is inconsistent with the records in f, the algorithm proceeds to the next steps to handle the inconsistency of the reset clocks (Lines 7-20).

Introduce Auxiliary Actions in $\mathcal{M}_1$ (Lines 7-12). If $\delta[2] \in dom(f) \wedge \langle \delta[2], \delta[4] \rangle \notin f$ (Line 7), we need to introduce a *new action* (through the variable σ_{new}) and add it to the alphabets of the output models. Then the transition δ with action σ is modified to a new transition, say δ' by replacing action σ with the value of σ_{new} (Lines 11-12).

Add Auxiliary Transitions in $\mathcal{M}_2$ and $\mathcal{M}_3$ (Lines 13-20). Since new actions are introduced in $\mathcal{M}_1$, we need to add auxiliary transitions with each new action in $\mathcal{M}_2$ and $\mathcal{M}_3$ accordingly. Specifically, consider the case when $\mathcal{M}_1$ and $\mathcal{M}_2$ synchronize on action σ via transitions δ and $\overline{\delta}$ in the models, respectively. If δ in $\mathcal{M}_1$ is modified to δ' in $\mathcal{M}_1'$ by renaming its action σ to σ', a fresh co-transition $\overline{\delta'}$ should be added to $\mathcal{M}_2'$ which is a copy of $\overline{\delta}$ by changing σ to σ' so as for

the synchronisation in the composition of $\mathcal{M}_1'$ and $\mathcal{M}_2'$ (Lines 13-16). The same changes are made for $\mathcal{M}_3$ (Lines 17-20).

Example 1. Fig. 2 shows an example of the conversion. In M_1, there are two transitions that contain action a but only one has clock reset. To solve clock reset inconsistency of M_1, the new action a' is introduced, and M_1 is converted into M_1'' by changing action name a of one transition to a' marked as an orange dashed line. In M_2 and ϕ, by adding the corresponding new transitions, M_2'' and ϕ'' are achieved. In ϕ'', the transitions with a and a' still have different reset information, so it is further changed to ϕ' by adding a transition marked as a blue dotted line. Correspondingly, M_1'' and M_2'' are changed. Obviously, we can determine the reset information of the transition with a (a', a'', a''') in automata M_1' and ϕ'.

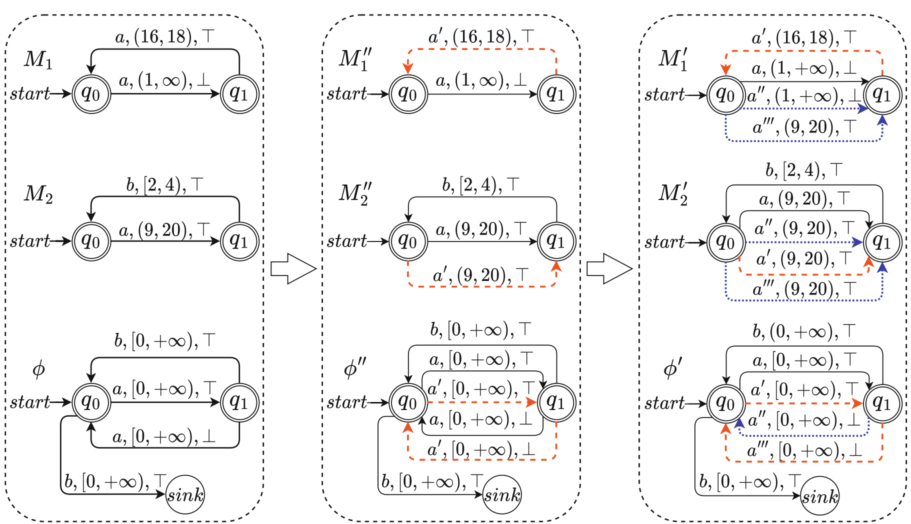


Fig. 2. M_1, M_2 and ϕ are converted into M_1', M_2' and ϕ'

We now show that the verification of $M_1'\|M_2'$ against ϕ' is equivalent to the original verification of $M_1\|M_2$ against ϕ.

Theorem 2. *Checking $M_1'\|M_2' \models \phi'$ is equivalent to checking $M_1\|M_2 \models \phi$.*

Proof. We prove $M_1\|M_2 \not\models \phi \Leftrightarrow M_1'\|M_2' \not\models \phi'$. This is equivalent to prove $\mathcal{L}(M_1\|M_2\|\overline{\phi}) \neq \emptyset \Leftrightarrow \mathcal{L}(M_1'\|M_2'\|\overline{\phi'}) \neq \emptyset$, where $\overline{\phi}$ and $\overline{\phi'}$ are the complements of ϕ and ϕ', respectively.

We first prove $\mathcal{L}(M_1\|M_2\|\overline{\phi}) \neq \emptyset \Rightarrow \mathcal{L}(M_1'\|M_2'\|\overline{\phi'}) \neq \emptyset$. The left hand side implies that $M_1\|M_2\|\overline{\phi}$ has at least one accepting run ρ. According to the construction of M_1', M_2' and ϕ', for the composed model $M_1'\|M_2'\|\overline{\phi'}$, compared

with $M_1 \| M_2 \| \overline{\phi}$, the locations and the guards of transitions remain the same, although some auxiliary transitions have been added to the model where actions are renamed. So we can construct a run ρ' in $M_1' \| M_2' \| \overline{\phi'}$, which visits the locations in the same order as ρ. Since ρ is an accepting run, its final location must be an accepting one, which implies ρ' is an accepting run of $M_1' \| M_2' \| \overline{\phi'}$, and $trace(\rho') \in \mathcal{L}(M_1' \| M_2' \| \overline{\phi'})$.

For $\mathcal{L}(M_1' \| M_2' \| \overline{\phi'}) \neq \emptyset \Rightarrow \mathcal{L}(M_1 \| M_2 \| \overline{\phi}) \neq \emptyset$, since $\mathcal{L}(M_1' \| M_2' \| \overline{\phi'}) \neq \emptyset$, there exists at least one accepting run ρ' in $M_1' \| M_2' \| \overline{\phi'}$. Still, by the construction of M_1', M_2' and ϕ', we can construct an accepting run ρ in $M_1 \| M_2 \| \overline{\phi}$, by replacing the newly introduced actions along ρ' with their original names, and $trace(\rho)$ is an evidence of $\mathcal{L}(M_1 \| M_2 \| \overline{\phi}) \neq \emptyset$. □

Complexity. For the model conversion, Algorithm 1 mainly consists of two invocations of Algorithm 2 which has a nested loop. In the worst case execution of Algorithm 2, the transitions of $\mathcal{M}_1$ in the outer loop and the transitions of $\mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{M}_3$ in the inner loops are traversed, so the time complexity is polynomial and quadratic in the number of transitions.

3.3 Membership Queries

After model conversion, a number of membership queries are used to learn the DOTA assumption A. For each membership query, the learner provides the teacher a logical-timed word $\omega_l = (\sigma_1, \mathbf{v}_1)(\sigma_2, \mathbf{v}_2) \cdots (\sigma_n, \mathbf{v}_n)$ to obtain clock reset information, where $\sigma_i \in \Sigma_A$ and $|\mathbf{v}_i| = 1$. Based on the converted model, the teacher supplements corresponding reset information γ_i for each σ_i in ω_l to construct the reset-logical-timed word $\omega_{rl} = (\sigma_1, \mathbf{v}_1, \gamma_1)(\sigma_2, \mathbf{v}_2, \gamma_2) \dots (\sigma_n, \mathbf{v}_n, \gamma_n)$. Though the learning algorithm we use is associated with one clock and the hypothesis we obtain is always a DOTA, the number of clocks in M_1' and ϕ' might be multiple since they are not necessarily DOTAs. This raises the question of how to transform the multi-clock reset information to the single-clock reset information. To solve this problem, we use a heuristic to generate the one-clock reset information γ_i for each action σ_i. Let X be the finite set of clocks of M_1' and ϕ', and x be the single clock of the learned assumption, where $|X| > 1$. For each action σ_i, we try four heuristics to determine whether x is reset: 1) random assignment, 2) γ_i is always $\{x\}$, 3) γ_i is always $\emptyset$, and 4) dynamic reset rule (if there exits a reset clock $y \in X$, then $\gamma_i = \{x\}$, otherwise $\gamma_i = \emptyset$). We use the fourth since the verification has the least checking time. After obtaining the logcial timed word ω_{rl}, the teacher further checks whether it satisfies ϕ' under the environment of M_1' by model checking if $M_1' \| A_{w_{rl}} \models \phi'$, where $A_{\omega_{rl}}$ is the automaton constructed from ω_{rl}.

As shown in Fig. 1, the step of model conversion is executed only once. It is then followed by the execution of the smart teacher we design, which only requires a polynomial number of membership queries for the assumption learning. Without the first step, the framework needs to turn to a normal teacher, in which case the reset information is obtained by guessing, and an exponential number of membership queries are required.

3.4 Candidate Queries

The candidate queries are to get answers about whether the learned hypothesis A satisfies the AG reasoning rule.

The First Candidate Query. This step checks whether $M_1' \| A \models \phi'$. If the answer is positive, we proceed to the second candidate query. Otherwise, a counterexample ctx_1, $ctx_1\rfloor_{\Sigma_A} \in \mathcal{L}(A)$ is generated and further analyzed by constructing a TA A_{ctx1} such that $M_1' \| A_{ctx1} \not\models \phi'$. We then check whether $ctx_1\rfloor_{\Sigma_A} \in \mathcal{L}(M_2')\rfloor_{\Sigma_A}$. If the result is positive, we have $M_1' \| M_2' \not\models \phi'$. Otherwise, $ctx_1\rfloor_{\Sigma_A} \in \mathcal{L}(A) \setminus \mathcal{L}(A_w)$ and ctx_1 serves as a negative counterexample to refine assumption A via the next round of membership queries.

The Second Candidate Query. This step checks whether $M_2' \models A$, i.e. $\mathcal{L}(M_2')\rfloor_{\Sigma_A} \subseteq \mathcal{L}(A)$. If yes, as $M_1' \| A \models \phi'$ and $M_2' \models A$, the verification algorithm terminates and we conclude $M_1' \| M_2' \models \phi'$. Otherwise, a counterexample ctx_2 is generated and a TA A_{ctx_2} is constructed from the timed word ctx_2. We check whether $M_1' \| A_{ctx_2} \not\models \phi'$. If yes, as $ctx_2\rfloor_{\Sigma_A} \in \mathcal{L}(M_2')\rfloor_{\Sigma_A}$, we conclude $M_1' \| M_2' \not\models \phi'$. Otherwise, $ctx_2\rfloor_{\Sigma_A} \in \mathcal{L}(A_w) \setminus \mathcal{L}(A)$ is a counterexample, indicating a new round learning is needed to refine and check A using membership and candidate queries until a conclusive result is obtained.

3.5 Correctness and Termination

We now show the correctness and termination of the framework.

Theorem 3. *Given two deterministic timed automata M_1 and M_2, and property ϕ, if there exists a DOTA that accepts the target language $\mathcal{L}(A_w)$, where A_w is the weakest assumption of the converted model M_2', the proposed learning-based compositional verification returns true if ϕ holds on $M_1 \| M_2$ and false otherwise.*

Proof. From Theorem 2, we only need to consider the converted models M_1', M_2' and ϕ'.

Termination. The proposed framework consists of the steps of model conversion, membership and candidate queries. We argue about the termination of the overall framework by showing the termination of each step.

By Algorithm 1 and Theorem 2, the step of model conversion terminates. Because the learning algorithm of DOTA terminates [7], assumption A will be obtained at last by membership queries. As to the candidate queries, they either conclude $M_1' \| M_2' \models \phi'$ and then terminate, or provide a positive or negative counterexample ctx, that is, $ctx\rfloor_{\Sigma_A} \in \mathcal{L}(A_w) \setminus \mathcal{L}(A)$ or $ctx\rfloor_{\Sigma_A} \in \mathcal{L}(A) \setminus \mathcal{L}(A_w)$, for the refinement of A.

For the weakest assumption A_w, since there exists a DOTA which accepts $\mathcal{L}(A_w)$, the framework eventually constructs A_w in some round to produce the positive answer $M_1' \| A_w \models \phi'$ to the first candidate query. As shown in Sect. 3.4, we can check whether $\mathcal{L}(M_2')\rfloor_{\Sigma_A} \subseteq \mathcal{L}(A)$. If the result is positive, we

have $M_1' \| M_2' \models \phi'$ and the framework terminates. Otherwise, a counterexample $ctx_2\downharpoonright_{\Sigma_A} \in \mathcal{L}(M_2')\downharpoonright_{\Sigma_A} \setminus \mathcal{L}(A_w)$ is generated. So $M_1' \| M_2' \not\models \phi'$, and ctx_2 is a witness to the fact that $M_1' \| M_2'$ violates ϕ'.

Correctness. Since there exists a DOTA that accepts the target language $\mathcal{L}(A_w)$, the framework always eventually terminates with a result which is either true or false. It is true only if both candidate queries return true and this means that ϕ' is held on $M_1' \| M_2'$. Otherwise, a counterexample $ctx\downharpoonright_{\Sigma_A} \notin \mathcal{L}(A_w)$ is generated. Since $M_1' \| A_{ctx} \not\models \phi'$ and $ctx\downharpoonright_{\Sigma_A} \in \mathcal{L}(M_2')\downharpoonright_{\Sigma_A}$, hence $M_1' \| M_2' \not\models \phi'$. □

It is possible, in some cases, there is no DOTA that can accept $\mathcal{L}(A_w)$, and the proposed verification framework cannot be guaranteed in these cases. However, the framework is still *sound*, meaning that for the cases when a DOTA assumption is learned and the verification terminates with a result, the result holds. Therefore, the framework is able to handle more flexible models such as multi-clock models. We will explore this with experiments in Sect. 5.

Theorem 4. *Given two deterministic timed automata M_1' and M_2' which might have multiple clocks, and property ϕ', even if there is no DOTA that accepts the target language $\mathcal{L}(A_w)$, the proposed verification framework is still sound.*

Proof. Given M_1' and M_2' which are multi-clock timed automata, suppose in some round if the learned DOTA assumption A satisfies $\mathcal{L}(A) \subseteq \mathcal{L}(A_w)$ and $\mathcal{L}(M_2')\downharpoonright_{\Sigma_A} \subseteq \mathcal{L}(A)$, we have that both results of the first and second candidate queries are positive. Hence, verification terminates and $M_1' \| M_2' \models \phi'$ holds. For the same reasoning, in the case of a counterexample ctx is generated, that is $M_1' \| A_{ctx} \not\models \phi'$ and $ctx\downharpoonright_{\Sigma_A} \in \mathcal{L}(M_2')\downharpoonright_{\Sigma_A}$, this implies that $M_1' \| M_2' \not\models \phi'$ and the verification terminates with the valid result. □

The framework is not *complete* though. For a M_1 with multiple clocks, it is not guaranteed to have a DOTA assumption A such that $\mathcal{L}(A) = \mathcal{L}(A_w)$. Thus, the framework is not guaranteed to terminate. Furthermore, for a M_2 with multiple clocks, the framework may not be able to learn a DOTA assumption A, such that $\mathcal{L}(M_2')\downharpoonright_{\Sigma_A} \subseteq \mathcal{L}(A)$ even though $M_1' \| M_2' \models \phi'$.

4 Optimization Methods

In this section, we give two improvements to the verification framework proposed in Sect. 3. The first one reduces state space and membership queries in terms of the given information of M_1' and ϕ'. The second one uses a smaller alphabet than $\Sigma_A = (\Sigma_1' \cup \Sigma_\phi') \cap \Sigma_2'$ to improve the verification speed.

4.1 Using Additional Information

In the process of learning assumption A with respect to M_1' and ϕ', we make better use of the available information of M_1' and ϕ'. It is clear that if there

are more actions taking place from a learned location, it is likely there are more successor locations from that location and more symbolic states are needed. It is, in general, that not all the actions are enabled in a location. Since the logical-timed words of the models M_1' or ϕ' are known beforehand, the sequence of actions that can be taken can be obtained. Therefore, we can use this information to remove those actions which do not take place from a certain location to reduce the number of successor states. Furthermore, the number of membership queries can be reduced by directly giving answers to these queries whose timed words violate the action sequences. This results in accelerating the learning process as well as speeding up the verification to some extent. The experiments in the next section also show these improvements.

For example, M_1' has two actions *read* and *write*. In addition, it is known that the *write* action can only be performed after the *read* has been executed. So, we add such information to the learning step of the verification framework. That is, *read* should take place before *write* in any timed word. Thus, for the membership queries with such word $\omega_l = \dots (write, \mathbf{v}_k) \dots (read, \mathbf{v}_m) \dots$, where *write* takes place before *read*, a negative answer is directly returned without the model checking steps for membership queries as shown in Sect. 3.3.

The additional information is usually derived from the design rules and other characteristics of the system under study. In the implementation, we provide some basic keywords to describe the rules, e.g. "beforeAfter" specifies the order of actions, and "startWith" specifies a certain action should be executed first. Therefore, the above example is encoded as "[beforeAfter]:(*read*,*write*)".

4.2 Minimizing the Alphabet of the Assumption

In our framework, the automated AG procedure uses a fixed assumption alphabet $\Sigma_A = (\Sigma_1' \cup \Sigma_\phi') \cap \Sigma_2'$. However, there may exist an assumption A_s over a smaller alphabet $\Sigma_s \subset \Sigma_A$ that satisfies the two premises of the AG rule. We thus propose and implement another improvement to build the timed assumption over a minimal alphabet. Smaller alphabet size can directly reduce the number of membership queries and thus speeds up the verification process.

Theorem 5. *Given $\Sigma_A = (\Sigma_1' \cup \Sigma_\phi') \cap \Sigma_2'$, if there exists an assumption A_s over non-empty alphabet $\Sigma_s \subset \Sigma_A$ satisfying $M_1' \| A_s \models \phi'$ and $M_2' \models A_s$, then there must exist an assumption A over Σ_A satisfying $M_1' \| A \models \phi'$ and $M_2' \models A$.*

Proof. Based on A_s, we can construct a timed assumption A over Σ_A as follows. For $A_s = (Q_s, q_0^s, \Sigma_s, F_s, X_s, \Delta_s)$, we first build $A = (Q, q_0, \Sigma_A, F, X, \Delta)$ where $Q = Q_s, q_0 = q_0^s, F = F_s, \Delta = \Delta_s$ and $X = X_s$. Then for $\forall q \in Q$ and $\forall \sigma \in \Sigma_A \setminus \Sigma_s$, we add $(q, \sigma, true, \emptyset, q)$ into Δ.

We now prove with such A, $M_1' \| A \models \phi'$ and $M_2' \models A$ still hold, that is, $M_1' \| M_2' \models \phi'$. Since the locations of A and A_s are the same, the locations of $M_1' \| A$ and $M_1' \| A_s$ are the same. For the composed model $M_1' \| A$, and the newly added transition $\delta_{new} = (q, \sigma, true, \emptyset, q)$ from state q in A, since $\sigma \in \Sigma_A \setminus \Sigma_s$, it will be synchronized with such transition taking the form $\delta_1 = (q_c, \sigma, \varphi_c, \gamma_c, q_c')$

in M_1'. So in $M_1' \| A$, the composed transition with respect to (q_c, q) and σ, is $((q_c, q), \sigma, \varphi_c, \gamma_c, (q_c', q))$. While in $M_1' \| A_s$, for such transition δ_1 in M_1', though there is no synchronized transition from state q in A_s, the composed transition is still $((q_c, q), \sigma, \varphi_c, \gamma_c, (q_c', q))$ in $M_1' \| A_s$. So $M_1' \| A \models \phi'$. According to the construction process of A from A_s, as $M_2' \models A_s$, i.e. $\mathcal{L}(M_2')\downharpoonright_{\Sigma_s} \subseteq \mathcal{L}(A_s)$, it follows that $M_2' \models A$. □

The main problem with smaller alphabet is that AG rule is no longer complete for deterministic finite automata [18]. The problem still exists for timed automata. If $\Sigma_s \subset \Sigma_A$, then there might not exist an assumption A_s over Σ_s that satisfies the two premises of AG even though $M_1' \| M_2' \models \phi'$. In this situation, we say Σ_s is incomplete and needs to be refined. So each time when we find Σ_s is incomplete, we select another $\Sigma_s' \subset \Sigma_A$ and restart the learning algorithm again. If a large number of round of refinement is needed, the speed of the verification is reduced significantly. To compensate for this speed reduction, we reuse the counterexamples that indicate the incompleteness of Σ_s in the previous loops and use a variable $List_c$ to store them. Before starting a new round of learning, we use $List_c$ to judge whether the current Σ_s' is appropriate in advance. We say Σ_s' is *appropriately selected* only if all the counterexamples of $List_c$ can not indicate Σ_s' is incomplete.

With a small alphabet $\Sigma_s \subset \Sigma_A$, we can not directly conclude the verification result if $M_1' \| M_2' \not\models \phi'$. The reason is that any given counterexample ctx maintaining $M_1' \| A_{ctx} \not\models \phi' \wedge ctx\downharpoonright_{\Sigma_s} \in \mathcal{L}(M_2')\downharpoonright_{\Sigma_s}$ will be used to illustrate the incompleteness of the Σ_s, though in some cases ctx indeed indicates that $M_1' \| M_2' \not\models \phi'$ over Σ_A. As a result, the treatment of ctxs will decrease the whole verification speed if $M_1' \| M_2' \not\models \phi'$. To solve this, we need to detect real counterexamples earlier. We will first check whether $M_1' \| A_{ctx} \not\models \phi' \wedge ctx\downharpoonright_{\Sigma_A} \in \mathcal{L}(M_2')\downharpoonright_{\Sigma_A}$ holds. If the result is yes, the verification concludes $M_1' \| M_2' \not\models \phi'$. Otherwise ctx is used to refine assumption over new Σ_s'.

5 Experimental Results

We implemented the proposed framework in Java. The membership queries and candidate queries are executed by calling the model checking tool UPPAAL. We evaluated the implementation on the benchmark of AUTOSAR (Automotive Open System Architecture) case studies. All the experiments were carried out on a 3.7GHz AMD Ryzen 5 5600X processor with 16GB RAM running 64-bit Windows 10. The source code of our tool and experiments is available in [2].

AUTOSAR is an open and standardized software architecture for automotive ECUs (Electronic Control Units). It consists of three layers, from top to bottom: AUTOSAR Software, AUTOSAR Runtime Environment (RTE), and Basic Software [1]. Its safety guarantee is very important [13,34,40]. A formal timed model of AUTOSAR architecture consists of several tasks and their corresponding runnables, different communication mechanisms of any two runnables, RTE communication controllers and task schedulers. In terms of different number of

tasks and runnables, we designed three kinds of composed models: the small-scale model AUTOSAR-1 (8 automata), the complex-scale composed models AUTOSAR-2 (14 automata) and AUTOSAR-3 (14 automata). The properties of the architecture to be checked are: 1) buffers between two runnables will never overflow or underflow, and 2) for a pair of sender runnable and receiver runnable, they should not execute the *write* action simultaneously. The checking methods we performed in the experiments are: 1) traditional monolithic model checking via UPPAAL, 2) compositional verification framework we propose (CV), 3) CV with the first improvement that uses additional information of M_2' and ϕ' (CV+A), 4) CV with the second improvement that minimizes assumption alphabet (CV+M), and 5) CV with both improvements (CV+A+M). Each experiment was conducted five times to calculate the average verification time. Tables 1-4 show the detailed verification results for each property using these methods, where Case IDs are given in the format *n-m-k-l*, denoting respectively the identifiers of the verified properties, the number of locations and clocks of M_2, and the alphabet size of M_2. The Boolean variable *Valid* denotes whether the property is satisfied. The symbols $|Q|$, $|\Sigma|$, R, and T_{mean} stand for the number of the locations and the alphabet size of the learned assumption, the number of alphabet refinements during learning and the average verification time in seconds, respectively.

1) AUTOSAR-1 Experiment. AUTOSAR-1 consists of 8 timed automata: 4 runnables, 2 buffers, and 2 schedulers used for scheduling the runnables. We partition the system into two parts, where M_1 is a DOTA and M_2 is composed of 7 DOTAs. The experimental results for this case are recorded in Table 1, where the proposed compositional verification (CV) outperforms the monolithic checking via UPPAAL except for cases 1-71424-7-8 and 3-71424-7-8 . This is because, for these two cases, the learning algorithm needs more than 30 rounds to refine assumptions using generated counterexamples. However, in terms of the first improvement (CV+A), i.e. CV with additional information of M_1', the verification time reduces drastically for these two cases. Similarly, by the use of the second improvement (CV+M), i.e. CV with a minimized alphabet, the verification time decreases due to fewer membership queries. With both improvements (CV+A+M), compared with single ones, the checking time varies depending on the actual case. As shown in Table 1, in the case of checking property 1 with CV+A, since the alphabet size of the learned assumption A is the largest one, i.e. 3, the second improvement can take effect. So the verification time using CV+A+M is less than that using CV+A. However, it is worse than CV+M.

We have discussed in Sect. 3.5 that the framework can handle models for M_1 which might be a multi-clock timed automaton, though termination is not guaranteed. So, we also repartition the AUTOSAR-1 system into two parts for verification, where M_1 is composed of 7 DOTAs. The results in Table 2 reveal that the proposed compositional method outperforms UPPAAL in most of the cases except the case 5-4-1-2. The reason is that UPPAAL might find a counterexample faster than the compositional approach because of the on-the-fly technique, which terminates the verification once a counterexample is found.

Table 1. Verification Results for AUTOSAR-1 where M_1 is a DOTA.

Case ID	Valid	UPPAAL	CV			CV+A			CV+M				CV+A+M			
		T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}
1-71424-7-8	Yes	37.862	62	3	1091.419	5	3	13.864	3	2	3	4.676	3	2	3	5.896
2-71424-7-8	Yes	46.215	1	3	0.237	4	3	11.030	1	1	0	0.163	1	1	0	0.273
3-71424-7-8	Yes	38.947	62	3	995.143	3	3	6.353	2	2	1	2.723	2	2	1	3.280
4-71424-7-8	Yes	38.783	1	3	0.234	2	3	3.859	1	1	0	0.164	1	1	0	0.341

In contrast, our framework needs to spend some time learning the assumption ahead of searching the counterexample, resulting in more time for the termination of the verification framework. In the experiments, we also observe that the time varies with the selection of M_1. Therefore, a proper selection of the components composed as M_1 or M_2 can lead to a faster verification, while ensuring termination of the framework.

Table 2. Verification Results for AUTOSAR-1 where M_1 is a composition of DOTAs

Case ID	Valid	UPPAAL	CV			CV+A			CV+M				CV+A+M			
		T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}
1-4-1-2	Yes	37.862	2	2	10.117	2	2	9.722	2	2	2	12.989	2	2	2	12.681
2-4-1-2	Yes	46.215	1	2	12.298	1	2	9.316	1	1	0	11.900	1	1	0	12.022
3-4-1-2	Yes	38.947	1	2	12.208	1	2	9.391	1	1	0	11.897	1	1	0	11.941
4-4-1-2	Yes	38.783	1	2	12.195	1	2	9.237	1	1	0	11.932	1	1	0	12.013
5-4-1-2	No	0.394	3	2	6.252	2	2	2.975	3	2	2	12.626	2	2	2	9.529
6-4-1-2	Yes	38.319	3	2	23.973	1	2	13.430	3	2	1	33.569	1	2	1	22.563

2) AUTOSAR-2 Experiment. AUTOSAR-2 is a more complex system with totally 14 automata, including 6 runnables and a task to which the runnables are mapped, 5 buffers, a RTE and a scheduler. In this experiment, we select M_1 as a composition of several DOTAs. The results in Table 3 show that in the cases of properties 1-4, UPPAAL fails to obtain checking results due to the large state space, whereas our compositional approach can finish the verification for all the properties in 300 seconds using the same memory size. This indicates that the framework can reduce the state space significantly in some cases.

Table 3. Verification Results for AUTOSAR-2

Case ID	Valid	UPPAAL	CV			CV+A			CV+M				CV+A+M			
		T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}
1-4-1-2	Yes	ROM	1	2	295.342	1	2	263.082	1	1	0	291.945	1	1	0	292.945
2-4-1-2	Yes	ROM	1	2	298.551	1	2	265.381	1	1	0	293.617	1	1	0	290.617
3-4-1-2	Yes	ROM	1	2	295.443	1	2	264.900	1	1	0	292.244	1	1	0	291.244
4-4-1-2	Yes	ROM	1	2	295.688	1	2	271.144	1	1	0	294.194	1	1	0	295.194

ROM: run out of memory.

3) AUTOSAR-3 Experiment. The system consists of 14 components, where both M_1 and M_2 are the compositions of several DOTAs. The checking results shown in Table 4 illustrate that the minimal alphabet improvement can obtain the smallest alphabet with size 1, thus reducing the verification time. However, the additional information improvement performs badly in most cases.

Table 4. Verification Results for AUTOSAR-3

Case ID	Valid	UPPAAL	CV			CV+A			CV+M				CV+A+M			
		T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}	$\|Q\|$	$\|\Sigma\|$	R	T_{mean}
1-30-1-7	Yes	1.354	1	3	0.910	1	3	0.298	1	1	0	0.808	1	1	0	0.801
2-30-1-7	Yes	1.313	1	6	0.351	3	6	2.839	1	1	0	0.152	1	1	0	0.150
3-30-1-7	Yes	1.363	1	6	0.348	3	6	2.838	1	1	0	0.161	1	1	0	0.156

6 Conclusion

Though in model checking, assume-guarantee reasoning can help alleviate state space explosion problem of a composite model, its practical impact has been limited due to the non-trivial human interaction to obtain the assumption. In this paper, we propose a learning-based compositional verification for deterministic timed automata, where the assumption is learned as a deterministic one-clock timed automaton. We design a model conversion algorithm to acquire the clock reset information of the learned assumption to reduce the learning complexity and prove this conversion preserves the verification results. To make the framework applicable to multi-clock systems, we design a smart teacher with heuristic to answer clock reset information. We also prove the correctness and termination of the framework. To speed up the verification, we further give two kinds of improvements to the learning process. We implemented the framework and performed experiments to evaluate our method. The results show that it outperforms monolithic model checking, and the state space can be effectively reduced. Moreover, the improvements also have positive effects on most studied systems.

References

1. AUTOSAR: Document search (2021). https://www.autosar.org/nc/document-search
2. The source code of our tool and experiments (2023). https://github.com/zeno-98/Tool-and-Experiments
3. Alur, R.: Timed automata. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 8–22. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_3
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)

5. Alur, R., Fix, L., Henzinger, T.A.: Event-clock automata: a determinizable class of timed automata. Theoret. Comput. Sci. **211**(1–2), 253–273 (1999)
6. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_52
7. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. In: TACAS 2020. LNCS, vol. 12078, pp. 444–462. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_25
8. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
9. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
10. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: International School on Formal Methods for the Design of Real-Time Systems. vol. 3185, pp. 200–236. Springer (2004)
11. Behrmann, G., et al.: Uppaal 4.0. Quantitative Evaluation of Systems, pp. 125–126 (2006)
12. Berendsen, J., Gebremichael, B., Vaandrager, F.W., Zhang, M.: Formal specification and analysis of zeroconf using uppaal. Transactions on Embedded Computing Systems **10**(3), 1–32 (2011)
13. Beringer, S., Wehrheim, H.: Verification of AUTOSAR software architectures with timed automata. In: ter Beek, M.H., Gnesi, S., Knapp, A. (eds.) FMICS/AVoCS -2016. LNCS, vol. 9933, pp. 189–204. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45943-1_13
14. Gheorghiu, M., Giannakopoulou, D., Păsăreanu, C.S.: Refining Interface Alphabets for Compositional Verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_23
15. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_14
16. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
17. Chaki, S., Strichman, O.: Optimized $L*$-based assume-guarantee reasoning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 276–291. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_22
18. Chaki, S., Strichman, O.: Three optimizations for assume-guarantee reasoning with L^*. Formal Methods Syst. Design **32**(3), 267–284 (2008)
19. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (1997)
20. Clarke, E.M., Long, D.E., Mcmillan, K.L.: Compositional model checking. In: Fourth Annual Symposium on Logic in Computer Science, pp. 353–362 (1989)
21. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
22. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15

23. Cobleigh, J.M., Giannakopoulou, D., PĂsĂreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_24
24. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Assumption generation for software component verification. In: 17th IEEE International Conference on Automated Software Engineering, pp. 3–12. IEEE (2002)
25. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Trans. Program. Lang. Syst. **16**(3), 843–871 (1994)
26. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: Methodology and case studies. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 440–451. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028765
27. Isberner, M., Howar, F., Steffen, B.: The TTT Algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
28. Jee, E., Wang, S., Kim, J.K., Lee, J., Sokolsky, O., Lee, I.: A safety-assured development approach for real-time software. In: 16th International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 133–142. IEEE (2010)
29. Jones, C.B.: Development methods for computer programs including a notion of interference. Oxford University Computing Laboratory (1981)
30. Kučera, P., Hynčica, O., Honzík, P.: Implementation of timed automata in a real-time operating system. In: World Congress on Engineering and Computer Science. vol. 1, pp. 56–60 (2010)
31. Lin, S.-W., André, É., Dong, J.S., Sun, J., Liu, Y.: An efficient algorithm for learning event-recording automata. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 463–472. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_35
32. Lin, S.W., André, E., Liu, Y., Sun, J., Dong, J.S.: Learning assumptions for compositionalverification of timed systems. IEEE Trans. Softw. Eng. **40**(2), 137–153 (2013)
33. Merz, S.: Model checking: a tutorial overview. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 3–38. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45510-8_1
34. Neumann, S., Kluge, N., Wätzoldt, S.: Automatic transformation of abstract AUTOSAR architectures to timed automata. In: 5th International Workshop on Model Based Architecting and Construction of Embedded Systems, pp. 55–60. ACM (2012)
35. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems. vol. 13, pp. 123–144. Springer (1985). https://doi.org/10.1007/978-3-642-82453-1_5
36. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
37. Sankur, O.: Timed automata verification and synthesis via finite automata learning. In: 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 13994, pp. 329–349. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_21

38. Sinha, N., Clarke, E.: SAT-based compositional verification using lazy learning. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 39–54. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_8
39. TaŞiran, S., Alur, R., Kurshan, R.P., Brayton, R.K.: Verifying abstractions of timed systems. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 546–562. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61604-7_75
40. Zhu, L., Liu, P., Shi, J., Wang, Z., Zhu, H.: A timing verification framework for AUTOSAR OS component development based on real-time maude. In: 7th International Symposium on Theoretical Aspects of Software Engineering, pp. 29–36. IEEE (2013)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Online Causation Monitoring of Signal Temporal Logic

Zhenya Zhang[1(✉)], Jie An[2], Paolo Arcaini[2], and Ichiro Hasuo[2]

[1] Kyushu University, Fukuoka, Japan
zhang@ait.kyushu-u.ac.jp
[2] National Institute of Informatics, Tokyo, Japan
{jiean,arcaini,hasuo}@nii.ac.jp

Abstract. Online monitoring is an effective validation approach for hybrid systems, that, at runtime, checks whether the (partial) signals of a system satisfy a specification in, e.g., *Signal Temporal Logic (STL)*. The classic STL monitoring is performed by computing a robustness interval that specifies, at each instant, how far the monitored signals are from violating and satisfying the specification. However, since a robustness interval monotonically shrinks during monitoring, classic online monitors may fail in reporting new violations or in precisely describing the system evolution at the current instant. In this paper, we tackle these issues by considering the *causation* of violation or satisfaction, instead of directly using the robustness. We first introduce a *Boolean causation monitor* that decides whether each instant is relevant to the violation or satisfaction of the specification. We then extend this monitor to a *quantitative causation monitor* that tells how far an instant is from being relevant to the violation or satisfaction. We further show that classic monitors can be derived from our proposed ones. Experimental results show that the two proposed monitors are able to provide more detailed information about system evolution, without requiring a significantly higher monitoring cost.

Keywords: online monitoring · Signal Temporal Logic · monotonicity

1 Introduction

Safety-critical systems require strong correctness guarantees. Due to the complexity of these systems, offline verification may not be able to guarantee their total correctness, as it is often very difficult to assess all possible system behaviors. To mitigate this issue, runtime verification [4,29,36] has been proposed as a

Z. Zhang is supported by JSPS KAKENHI Grant No. JP23K16865 and No. JP23H03372. J. An, P. Arcaini, and I. Hasuo are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST, Funding Reference number 10.13039/501100009024 ERATO. P. Arcaini is also supported by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project (Grant Number JPMJMI20B8), JST-Mirai.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 62–84, 2023.
https://doi.org/10.1007/978-3-031-37706-8_4

complementary technique that analyzes the system execution at runtime. Online monitoring is such an approach that checks whether the system execution (e.g., given in terms of signals) satisfies or violates a system specification specified in a temporal logic [28,34], e.g., *Signal Temporal Logic (STL)* [30].

Quantitative online monitoring is based on the STL *robust semantics* [17,21] that not only tells whether a signal satisfies or violates a specification φ (i.e., the classic Boolean satisfaction relation), but also assigns a value in $\mathbb{R} \cup \{\infty, -\infty\}$ (i.e., *robustness*) that indicates *how robustly* φ is satisfied or violated. However, differently from offline assessment of STL formulas, an online monitor needs to reason on *partial signals* and, so, the assessment of the robustness should be adapted. We consider an established approach [12] employed by *classic online monitors* (`ClaM` in the following). It consists in computing, instead of a single robustness value, a *robustness interval*; at each monitoring step, `ClaM` identifies an *upper bound* $[\mathrm{R}]^{\mathsf{U}}$ telling the maximal reachable robustness of any possible suffix signal (i.e., any continuation of the system evolution), and a *lower bound* $[\mathrm{R}]^{\mathsf{L}}$ telling the minimal reachable robustness. If, at some instant, $[\mathrm{R}]^{\mathsf{U}}$ becomes negative, the specification is violated; if $[\mathrm{R}]^{\mathsf{L}}$ becomes positive, the specification is satisfied. In the other cases, the specification validity is `unknown`.

Consider a simple example in Fig. 1. It shows the monitoring of the `speed` of a vehicle (in the upper plot); the specification requires the `speed` to be always below 10. The lower plot reports how the upper bound $[\mathrm{R}]^{\mathsf{U}}$ and the lower bound $[\mathrm{R}]^{\mathsf{L}}$ of the reachable robustness change over time. We observe that the initial value of $[\mathrm{R}]^{\mathsf{U}}$ is around 8 and gradually decreases.[1] The monitor allows to detect that the specification is violated at time $b = 20$ when the `speed` becomes higher than 10, and therefore $[\mathrm{R}]^{\mathsf{U}}$ goes below 0. After that, the violation severity progressively gets worse till time $b = 30$, when $[\mathrm{R}]^{\mathsf{U}}$ becomes -5. After that point, the monitor does not provide any additional useful information about the system evolution, as $[\mathrm{R}]^{\mathsf{U}}$ remains stuck at -5. However, if we observe the signal of the `speed` after $b = 30$, we notice that (i) the severity of the violation is mitigated, and the "1st violation episode" ends at time $b = 35$; however, the monitor `ClaM` does not report this type of information; (ii) a "2nd violation episode" occurs in the time interval $[40, 45]$; the monitor `ClaM` does not distinguish the new violation.

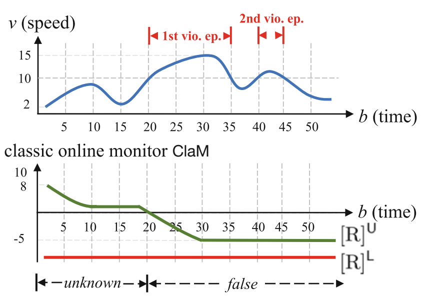


Fig. 1. `ClaM` – Robustness upper and lower bounds of $\Box_{[0,100]}(v < 10)$

The reason for the issues reported in the example is that the upper and lower bounds are monotonically decreasing and increasing; this has the consequence

[1] The value of lower bound $[\mathrm{R}]^{\mathsf{L}}$ is not shown in the figure, as not relevant. In the example, it remains constant before $b = 100$, and the value is usually set either according to domain knowledge about system signals, or to $-\infty$ otherwise.

that the robustness interval at a given step is "masked" by the history of previous robustness intervals, and, e.g., it is not possible to detect mitigation of the violation severity. Moreover, as an extreme consequence, as soon as the monitor `ClaM` assesses the violation of the specification (i.e., the upper bound $[\mathrm{R}]^{\mathsf{U}}$ becomes negative), or its satisfaction (i.e., the lower bound $[\mathrm{R}]^{\mathsf{L}}$ becomes positive), the Boolean status of the monitor does not change anymore. Such characteristic directly derives from the STL semantics and it is known as the *monotonicity* [9–11] of classic online monitors. Monotonicity has been recognized as a problem of these monitors in the literature [10,37,40], since it does not allow to detect specific types of information that are "masked". We informally define two types of *information masking* that can occur because of monotonicity:

***evolution masking*:** the monitor may not properly report the evolution of the system execution, e.g., mitigation of violation severity may not be detected;
***violation masking*:** as a special case of *evolution masking*, the first violation episode during the system execution "masks" the following ones.

The information not reported by `ClaM` because of information masking, is very useful in several contexts. First of all, in some systems, the first violation of the specification does not mean that the system is not operating anymore, and one may want to continue monitoring and detect all the succeeding violations; this is the case, e.g., of the monitoring approach reported by Selyunin et al. [37] in which all the violations of the SENT protocol must be detected. Moreover, having a precise description of the system evolution is important for the usefulness of the monitoring; for example, the monitoring of the `speed` in Fig. 1 could be used in a vehicle for checking the speed and notifying the driver whenever the speed is approaching the critical limit; if the monitor is not able to precisely capture the severity of violation, it cannot be used for this type of application.

Some works [10,37,40] try to mitigate the monotonicity issues, by "resetting" the monitor at specific points. A recent approach has been proposed by Zhang et al. [40] (called `ResM` in the following) that is able to identify each "violation episode" (i.e., it solves the problem of *violation masking*), but does not solve the *evolution masking* problem. For the example in Fig. 1, `ResM` is able to detect the two violation episodes in intervals $[20, 35]$ and $[40, 45]$, but it is not able to report that the speed decreases after $b = 10$ (in a non-violating situation), and that the severity of the violation is mitigated after $b = 30$.

Contribution. In this paper, in order to provide more information about the evolution of the monitored system, we propose to monitor the *causation* of violation or satisfaction, instead of considering the robustness directly. To do this, we rely on the notion of *epoch* [5]. At each instant, the *violation (satisfaction) epoch* identifies the time instants at which the evaluation of the atomic propositions of the specification φ causes the violation (satisfaction) of φ.

Based on the notion of epoch, we define a *Boolean causation monitor* (called `BCauM`) that, at runtime, not only assesses the specification violation/satisfaction, but also tells whether each instant is relevant to it. Namely, `BCauM` marks each current instant b as (i) a *violation causation instant*, if b is added to the violation epoch; (ii) a *satisfaction causation instant*, if b is added to the satisfaction epoch;

(iii) an *irrelevant instant*, if b is not added to any epoch. We show that `BCauM` is able to detect all the violation episodes (so solving the *violation masking* issue), as violation causation instants can be followed by irrelevant instants. Moreover, we show that the information provided by the classic Boolean online monitor can be derived from that of the Boolean causation monitor `BCauM`.

However, `BCauM` just tells us whether the current instant is a (violation or satisfaction) causation instant or not, but does not report *how far* the instant is from being a causation instant. To this aim, we introduce the notion of *causation distance*, as a quantitative measure characterizing the spatial distance of the signal value at b from turning b into a causation instant. Then, we propose the *quantitative causation monitor* (`QCauM`) that, at each instant, returns its causation distance. We show that using `QCauM`, besides solving the *violation masking* problem, we also solve the *evolution masking* problem. Moreover, we show that we can derive from `QCauM` both the classic quantitative monitor `ClaM`, and the Boolean causation monitor `BCauM`.

Experimental results show that the proposed monitors, not only provide more information, but they do it in an efficient way, not requiring a significant additional monitoring time w.r.t. the existing monitors.

Outline. Section 2 reports necessary background. We introduce `BCauM` in Sect. 3, and `QCauM` in Sect. 4. Experimental assessment of the two proposed monitors is reported in Sect. 5. Finally, Sect. 6 discusses some related work, and Sect. 7 concludes the paper.

2 Preliminaries

In this section, we review the fundamentals of *signal temporal logic (STL)* in Sect. 2.1, and then introduce the existing classic online monitoring approach in Sect. 2.2.

2.1 Signal Temporal Logic

Let $T \in \mathbb{R}_+$ be a positive real, and $d \in \mathbb{N}_+$ be a positive integer. A *d-dimensional signal* is a function $\mathbf{v}\colon [0,T] \to \mathbb{R}^d$, where T is called the *time horizon* of $\mathbf{v}$. Given an arbitrary time instant $t \in [0,T]$, $\mathbf{v}(t)$ is a d-dimensional real vector; each dimension concerns a *signal variable* that has a certain physical meaning, e.g., `speed`, `RPM`, `acceleration`, etc. In this paper, we fix a set **Var** of variables and assume that a signal $\mathbf{v}$ is *spatially bounded*, i.e., for all $t \in [0,T]$, $\mathbf{v}(t) \in \Omega$, where Ω is a d-dimensional hyper-rectangle.

Signal temporal logic (STL) is a widely-adopted specification language, used to describe the expected behavior of systems. In Definition 1 and Definition 2, we respectively review the syntax and the robust semantics of STL [17,21,30].

Definition 1 (STL syntax). In STL, the *atomic propositions* α and the *formulas* φ are defined as follows:

$$\alpha ::\equiv f(w_1,\ldots,w_K) > 0 \qquad \varphi ::\equiv \alpha \mid \bot \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Box_I \varphi \mid \Diamond_I \varphi \mid \varphi\, \mathcal{U}_I\, \varphi$$

Here f is a K-ary function $f : \mathbb{R}^K \to \mathbb{R}$, $w_1, \ldots, w_K \in \mathbf{Var}$, and I is a closed interval over $\mathbb{R}_{\geq 0}$, i.e., $I = [l, u]$, where $l, u \in \mathbb{R}$ and $l \leq u$. In the case that $l = u$, we can use l to stand for I. $\Box, \Diamond$ and $\mathcal{U}$ are temporal operators, which are known as *always*, *eventually* and *until*, respectively. The always operator $\Box$ and eventually operator $\Diamond$ are two special cases of the until operator $\mathcal{U}$, where $\Diamond_I \varphi \equiv \top \mathcal{U}_I\, \varphi$ and $\Box_I \varphi \equiv \neg \Diamond_I \neg \varphi$. Other common connectives such as $\vee, \rightarrow$ are introduced as syntactic sugar: $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$.

Definition 2 (STL robust semantics). Let $\mathbf{v}$ be a signal, φ be an STL formula and $\tau \in \mathbb{R}_+$ be an instant. The *robustness* $\mathrm{R}(\mathbf{v}, \varphi, \tau) \in \mathbb{R} \cup \{\infty, -\infty\}$ of $\mathbf{v}$ w.r.t. φ at τ is defined by induction on the construction of formulas, as follows.

$$\mathrm{R}(\mathbf{v}, \alpha, \tau) := f(\mathbf{v}(\tau)) \qquad \mathrm{R}(\mathbf{v}, \bot, \tau) := -\infty \qquad \mathrm{R}(\mathbf{v}, \neg\varphi, \tau) := -\mathrm{R}(\mathbf{v}, \varphi, \tau)$$

$$\mathrm{R}(\mathbf{v}, \varphi_1 \wedge \varphi_2, \tau) := \min\left(\mathrm{R}(\mathbf{v}, \varphi_1, \tau), \mathrm{R}(\mathbf{v}, \varphi_2, \tau)\right)$$

$$\mathrm{R}(\mathbf{v}, \Box_I \varphi, \tau) := \inf_{t \in \tau + I} \mathrm{R}(\mathbf{v}, \varphi, t) \qquad \mathrm{R}(\mathbf{v}, \Diamond_I \varphi, \tau) := \sup_{t \in \tau + I} \mathrm{R}(\mathbf{v}, \varphi, t)$$

$$\mathrm{R}(\mathbf{v}, \varphi_1 \,\mathcal{U}_I\, \varphi_2, \tau) := \sup_{t \in \tau + I} \min\left(\mathrm{R}(\mathbf{v}, \varphi_2, t), \inf_{t' \in [\tau, t)} \mathrm{R}(\mathbf{v}, \varphi_1, t')\right)$$

Here, $\tau + I$ denotes the interval $[l + \tau, u + \tau]$.

The original STL semantics is Boolean, which represents whether a signal $\mathbf{v}$ satisfies φ at an instant τ, i.e., whether $(\mathbf{v}, \tau) \models \varphi$. The robust semantics in Definition 2 is a quantitative extension that refines the original Boolean STL semantics, in the sense that, $\mathrm{R}(\mathbf{v}, \varphi, \tau) > 0$ implies $(\mathbf{v}, \tau) \models \varphi$, and $\mathrm{R}(\mathbf{v}, \varphi, \tau) < 0$ implies $(\mathbf{v}, \tau) \not\models \varphi$. More details can be found in [21, Proposition 16].

2.2 Classic Online Monitoring of STL

STL robust semantics in Definition 2 provides an offline monitoring approach for *complete signals*. *Online monitoring*, instead, targets a growing *partial signal* at runtime. Besides the verdicts $\top$ and $\bot$, an online monitor can also report the verdict `unknown` (denoted as ?), which represents a status when the satisfaction of the signal to φ is not decided yet. In the following, we formally define partial signals and introduce online monitors for STL.

Let T be the time horizon of a signal $\mathbf{v}$, and let $[a, b] \subseteq [0, T]$ be a subinterval in the time domain $[0, T]$. A *partial signal* $\mathbf{v}_{a:b}$ is a function which is only defined in the interval $[a, b]$; in the remaining domain $[0, T] \setminus [a, b]$, we denote that $\mathbf{v}_{a:b} = \epsilon$, where ϵ stands for a value that is not defined.

Specifically, if $a = 0$ and $b \in (a, T]$, a partial signal $\mathbf{v}_{a:b}$ is called a *prefix* (partial) signal; dually, if $b = T$ and $a \in [0, b)$, a partial signal $\mathbf{v}_{a:b}$ is called a *suffix* (partial) signal. Given a prefix signal $\mathbf{v}_{0:b}$, a *completion* $\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}$ of $\mathbf{v}_{0:b}$ is defined as the concatenation of $\mathbf{v}_{0:b}$ with a suffix signal $\mathbf{v}_{b:T}$.

Definition 3 (Classic Boolean STL online monitor). Let $\mathbf{v}_{0:b}$ be a prefix signal, and φ be an STL formula. An online monitor $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau)$ returns a

verdict in $\{\top, \bot, ?\}$ (namely, `true`, `false`, and `unknown`), as follows:

$$\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau) \ := \begin{cases} \top & \text{if } \forall \mathbf{v}_{b:T}.\, \mathrm{R}(\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}, \varphi, \tau) > 0 \\ \bot & \text{if } \forall \mathbf{v}_{b:T}.\, \mathrm{R}(\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}, \varphi, \tau) < 0 \\ ? & \text{otherwise} \end{cases}$$

Namely, the verdicts of $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau)$ are interpreted as follows:

- if any possible completion $\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}$ of $\mathbf{v}_{0:b}$ satisfies φ, then $\mathbf{v}_{0:b}$ satisfies φ;
- if any possible completion $\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}$ of $\mathbf{v}_{0:b}$ violates φ, then $\mathbf{v}_{0:b}$ violates φ;
- otherwise (i.e., there is a completion $\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}$ that satisfies φ, and there is a completion $\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}$ that violates φ), then $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau)$ reports `unknown`.

Note that, by Definition 3 only, we cannot synthesize a feasible online monitor, because the possible completions for $\mathbf{v}_{0:b}$ are infinitely many. A constructive online monitor is introduced in [12], which implements the functionality of Definition 3 by computing the *reachable* robustness of $\mathbf{v}_{0:b}$. We review this monitor in Definition 4.

Definition 4 (Classic Quantitative STL online monitor (`ClaM`)). Let $\mathbf{v}_{0:b}$ be a prefix signal, and let φ be an STL formula. We denote by $\mathtt{R}^{\alpha}_{\max}$ and $\mathtt{R}^{\alpha}_{\min}$ the possible *maximum* and *minimum bounds* of the robustness $\mathrm{R}(\mathbf{v}, \alpha, \tau)$[2]. Then, an *online monitor* $[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau)$, which returns a sub-interval of $[\mathtt{R}^{\alpha}_{\min}, \mathtt{R}^{\alpha}_{\max}]$ at the instant b, is defined as follows, by induction on the construction of formulas.

$$[\mathrm{R}](\mathbf{v}_{0:b}, \alpha, \tau) \ := \begin{cases} \left[f\left(\mathbf{v}_{0:b}(\tau)\right), f\left(\mathbf{v}_{0:b}(\tau)\right)\right] & \text{if } \tau \in [0, b] \\ \left[\mathtt{R}^{\alpha}_{\min}, \mathtt{R}^{\alpha}_{\max}\right] & \text{otherwise} \end{cases}$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \neg\varphi, \tau) \ := \ -[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau)$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) \ := \ \min\Big([\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathrm{R}](\mathbf{v}_{0:b}, \varphi_2, \tau)\Big)$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \Box_I \varphi, \tau) \ := \ \inf_{t \in \tau + I}\Big([\mathrm{R}](\mathbf{v}_{0:b}, \varphi, t)\Big)$$

$$[\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1 \,\mathcal{U}_I\, \varphi_2, \tau) \ := \ \sup_{t \in \tau + I} \min\Big([\mathrm{R}](\mathbf{v}_{0:b}, \varphi_2, t), \inf_{t' \in [\tau, t)} [\mathrm{R}](\mathbf{v}_{0:b}, \varphi_1, t')\Big)$$

Here, f is defined as in Definition 1, and the arithmetic rules over intervals $I = [l, u]$ are defined as follows: $-I := [-u, -l]$ and $\min(I_1, I_2) := [\min(l_1, l_2), \min(u_1, u_2)]$.

We denote by $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau)$ and $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau)$ the upper bound and the lower bound of $[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau)$ respectively. Intuitively, the two bounds together form the reachable robustness interval of the completion $\mathbf{v}_{0:b} \cdot \mathbf{v}_{b:T}$, under any possible suffix signal $\mathbf{v}_{b:T}$. For instance, in Fig. 2, the upper bound $[\mathrm{R}]^{\mathsf{U}}$ at $b = 20$ is 0, which indicates that the robustness of the completion of the signal `speed`, under any suffix, can never be larger than 0.

The quantitative online monitor `ClaM` in Definition 4 refines the Boolean one in Definition 3, and the Boolean monitor can be derived from `ClaM` as follows:

[2] $\mathrm{R}(\mathbf{v}, \alpha, \tau)$ is bounded because $\mathbf{v}$ is bounded by Ω. In practice, if Ω is not know, we set $\mathtt{R}^{\alpha}_{\max}$ and $\mathtt{R}^{\alpha}_{\min}$ to, respectively, ∞ and $-\infty$.

- if $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$, it implies that $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau) = \top$;
- if $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$, it implies that $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau) = \bot$;
- otherwise, if $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$ and $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$, $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau) = ?$.

The classic online monitors are *monotonic* by definition. In the Boolean monitor (Definition 3), with the growth of $\mathbf{v}_{0:b}$, $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau)$ can only turn from ? to $\{\bot, \top\}$, but never the other way around. In the quantitative one (Definition 4), as shown in Lemma 1, $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau)$ and $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau)$ are both monotonic, the former one decreasingly, the latter one increasingly. An example can be observed in Fig. 2.

Lemma 1 (Monotonicity of STL online monitor). Let $[\mathrm{R}](\mathbf{v}_{0:b}, \varphi, \tau)$ be the quantitative online monitor for a partial signal $\mathbf{v}_{0:b}$ and an STL formula φ. With the growth of the partial signal $\mathbf{v}_{0:b}$, the upper bound $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau)$ monotonically decreases, and the lower bound $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau)$ monotonically increases, i.e., for two time instants $b_1, b_2 \in [0, T]$, if $b_1 < b_2$, we have (i) $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b_1}, \varphi, \tau) \geq [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b_2}, \varphi, \tau)$, and (ii) $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b_1}, \varphi, \tau) \leq [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b_2}, \varphi, \tau)$.

Proof. This can be proved by induction on the structures of STL formulas. The detailed proof can be found in the full version [38]. □

3 Boolean Causation Online Monitor

As explained in Sect. 1, monotonicity of classic online monitors causes different types of *information masking*, which prevents some information from being delivered. In this section, we introduce a novel *Boolean causation (online) monitor* `BCauM`, that solves the *violation masking* issue (see Sect. 1). `BCauM` is defined based on *online signal diagnostics* [5,40], which reports the *cause* of violation or satisfaction of the specification at the atomic proposition level.

Definition 5 (Online signal diagnostics). Let $\mathbf{v}_{0:b}$ be a partial signal and φ be an STL specification. At an instant b, online signal diagnostics returns a *violation epoch* $\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$, under the condition $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$, as follows:

$$\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \alpha, \tau) := \begin{cases} \{\langle \alpha, \tau \rangle\} & \text{if } [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \alpha, \tau) < 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \neg\varphi, \tau) := \mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$$

$$\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) := \bigcup_{\substack{i \in \{1,2\} \text{ s.t.} \\ [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_i, \tau) < 0}} \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_i, \tau)$$

$$\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau) := \bigcup_{\substack{t \in \tau + I \text{ s.t.} \\ [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, t) < 0}} \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi, t)$$

$$\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \,\mathcal{U}_I\, \varphi_2, \tau) := \bigcup_{\substack{t \in \tau + I \text{ s.t.} \\ [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_1 \mathcal{U}_t \varphi_2, \tau) < 0}} \left(\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_2, t) \cup \bigcup_{t' \in [\tau, t)} \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, t') \right)$$

and a *satisfaction epoch* $\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$, under the condition $[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$, as follows:

$$\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \alpha, \tau) \ := \ \begin{cases} \{\langle \alpha, \tau \rangle\} & \text{if } [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \alpha, \tau) > 0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \neg\varphi, \tau) \ := \ \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$$

$$\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) \ := \bigcup_{\substack{i \in \{1,2\} \text{ s.t.} \\ [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_i, \tau) > 0}} \mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi_i, \tau)$$

$$\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau) \ := \bigcup_{\substack{t \in \tau + I \text{ s.t.} \\ [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, t) > 0}} \mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi, t)$$

$$\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi_1 \,\mathcal{U}_I\, \varphi_2, \tau) \ := \bigcup_{\substack{t \in \tau + I \text{ s.t.} \\ [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_1 \mathcal{U}_t \varphi_2, \tau) > 0}} \left(\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi_2, t) \cup \bigcup_{t' \in [\tau, t)} \mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi_1, t') \right)$$

If the conditions are not satisfied, $\mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$ and $\mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$ are both $\emptyset$. Note that the definition is recursive, thus the conditions should also be checked for computing the violation and satisfaction epochs of the sub-formulas of φ.

Computation for other operators can be inferred by the presented ones and the STL syntax (Definition 1).

Intuitively, when a partial signal $\mathbf{v}_{0:b}$ violates a specification φ, a violation epoch starts collecting the evaluations (identified by pairs of atomic propositions and instants) of the signal at the atomic proposition level, that cause the violation of the whole formula φ (which also applies to the satisfaction cases in a dual manner). This is done inductively, based on the semantics of different operators:

- in the case of an atomic proposition α, if α is violated at τ, it collects $\langle \alpha, \tau \rangle$;
- in the case of a negation $\neg\varphi$, it collects the satisfaction epoch of φ;
- in the case of a conjunction $\varphi_1 \wedge \varphi_2$, it collects the union of the violation epochs of the sub-formulas violated by the partial signal;
- in the case of an *always* operator $\Box_I \varphi$, it collects the epochs of the sub-formula φ at all the instants t where φ is evaluated as being violated.
- in the case of an *until* operator $\varphi_1 \,\mathcal{U}_I\, \varphi_2$, it collects the epochs of the sub-formula φ_2 at all the instants t and the epochs of φ_1 at the instants $t' \in [\tau, t)$, in the case where the clause "φ_1 until φ_2" is violated at t.

Example 1. The example in Fig. 2 illustrates how an epoch is collected. The specification requires that whenever the `speed` is higher than 10, the car should decelerate within 5 time units. As shown by the classic monitor, the specification is violated at $b = 25$, since v becomes higher than 10 at 20 but a remains positive during $[20, 25]$. Note that the specification can be rewritten as $\varphi \equiv \Box_{[0,100]}(\neg(v > 10) \vee \Diamond_{[0,5]}(a < 0))$. For convenience, we name the sub-formulas of φ as follows:

$$\varphi' \equiv \neg(v > 10) \vee \Diamond_{[0,5]}(a < 0) \qquad \varphi_1 \equiv \neg(v > 10) \qquad \varphi_2 \equiv \Diamond_{[0,5]}(a < 0)$$
$$\alpha_1 \equiv v > 10 \qquad \alpha_2 \equiv a < 0$$

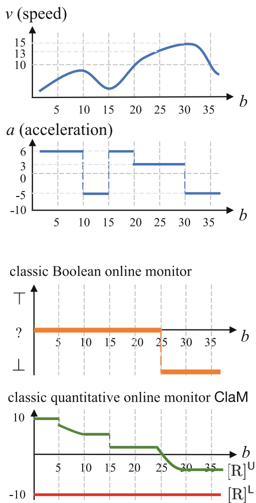


Fig. 2. Classic monitor (`ClaM`) result for the STL specification: $\square_{[0,100]}(v > 10 \rightarrow \Diamond_{[0,5]}(a < 0))$

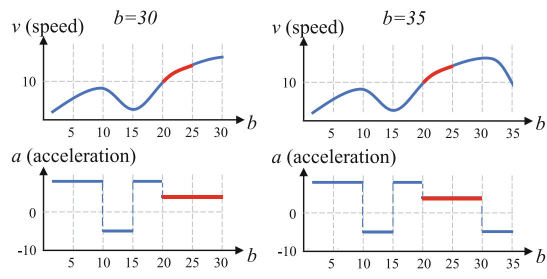


Fig. 3. The violation epochs (the red parts) respectively when $b = 30$ and $b = 35$

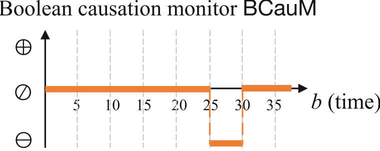


Fig. 4. Boolean causation monitor (`BCauM`) result

Figure 3 shows the violation epochs at two instants 30 and 35. First, at $b = 30$,

$$\begin{aligned} \mathrm{E}^{\ominus}(\mathbf{v}_{0:30}, \varphi, 0) &= \left(\bigcup_{t\in[20,25]} \mathrm{E}^{\oplus}(\mathbf{v}_{0:30}, \alpha_1, t)\right) \cup \left(\bigcup_{t\in[20,30]} \mathrm{E}^{\ominus}(\mathbf{v}_{0:30}, \alpha_2, t)\right) \\ &= \langle \alpha_1, [20, 25]\rangle \cup \langle \alpha_2, [20, 30]\rangle \end{aligned}$$

Similarly, the violation epoch $\mathrm{E}^{\ominus}(\mathbf{v}_{0:35}, \varphi, 0)$ at $b = 35$ is the same as that at $b = 30$. Intuitively, the epoch at $b = 30$ shows the cause of the violation of $\mathbf{v}_{0:30}$; then since signal $a < 0$ in $[30, 35]$, this segment is not considered as the cause of the violation, so the epoch remains the same at $b = 35$. $\triangleleft$

Definition 6 (Boolean causation monitor (`BCauM`)). Let $\mathbf{v}_{0:b}$ be a partial signal and φ be an STL specification. We denote by $\mathcal{A}$ the set of atomic propositions of φ. At each instant b, a *Boolean causation (online) monitor* `BCauM` returns a verdict in $\{\ominus, \oplus, \oslash\}$ (called *violation causation*, *satisfaction causation* and *irrelevant*), which is defined as follows,

$$\mathscr{M}(\mathbf{v}_{0:b}, \varphi, \tau) := \begin{cases} \ominus & \text{if } \exists \alpha \in \mathcal{A}.\ \langle \alpha, b\rangle \in \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau) \\ \oplus & \text{if } \exists \alpha \in \mathcal{A}.\ \langle \alpha, b\rangle \in \mathrm{E}^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau) \\ \oslash & \text{otherwise} \end{cases}$$

An instant b is called a *violation/satisfaction causation instant* if $\mathscr{M}(\mathbf{v}_{0:b}, \varphi, \tau)$ returns $\ominus/\oplus$, or an *irrelevant instant* if $\mathscr{M}(\mathbf{v}_{0:b}, \varphi, \tau)$ returns $\oslash$.

Intuitively, if the current instant b (with the related α) is included in the epoch (thus the signal value at b is relevant to the violation/satisfaction of φ), `BCauM` will

report a *violation/satisfaction causation* ($\ominus/\oplus$); otherwise, it will report *irrelevant* ($\oslash$). Notably BCauM is non-monotonic, in that even if it reports $\ominus$ or $\oplus$ at some instant b, it may still report $\oslash$ after b. This feature allows BCauM to bring more information, e.g., it can detect the end of a violation episode and the start of a new one (i.e., it solves the *violation masking* issue in Sect. 1); see Example 2.

Example 2. Based on the signal diagnostics in Fig. 3, the Boolean causation monitor BCauM reports the result shown as in Fig. 4.

Compared to the classic Boolean monitor in Fig. 2, BCauM brings more information, in the sense that it detects the end of the violation episode at $b = 30$, by going from $\ominus$ to $\oslash$, when the signal a becomes negative. $\triangleleft$

Theorem 1 states the relation of BCauM with the classic Boolean online monitor.

Theorem 1. The Boolean causation monitor BCauM in Definition 6 refines the classic Boolean online monitor in Definition 3, in the following sense:

- $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau) = \bot$ *iff.* $\bigvee_{t\in[0,b]} (\mathscr{M}(\mathbf{v}_{0:t}, \varphi, \tau) = \ominus)$
- $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau) = \top$ *iff.* $\bigvee_{t\in[0,b]} (\mathscr{M}(\mathbf{v}_{0:t}, \varphi, \tau) = \oplus)$
- $\mathrm{M}(\mathbf{v}_{0:b}, \varphi, \tau) = ?$ *iff.* $\bigwedge_{t\in[0,b]} (\mathscr{M}(\mathbf{v}_{0:t}, \varphi, \tau) = \oslash)$

Proof. The proof is based on Definitions 5 and 6, Lemma 1 about the monotonicity of classic STL online monitors, and two extra lemmas in the full version [38]. $\square$

4 Quantitative Causation Online Monitor

Although BCauM in Sect. 3 is able to solve the *violation masking* issue, it still does not provide enough information about the evolution of the system signals, i.e., it does not solve the *evolution masking* issue introduced in Sect. 1. To tackle this issue, we propose a *quantitative (online) causation monitor* QCauM in Definition 7, which is a quantitative extension of BCauM. Given a partial signal $\mathbf{v}_{0:b}$, QCauM reports a *violation causation distance* $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$ and a *satisfaction causation distance* $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$, which, respectively, indicate *how far* the signal value at the current instant b is from turning b into a violation causation instant and from turning b into a satisfaction causation instant.

Definition 7 (Quantitative causation monitor (QCauM)). Let $\mathbf{v}_{0:b}$ be a partial signal, and φ be an STL specification. At instant b, the quantitative causation monitor QCauM returns a *violation causation distance* $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$, as follows:

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \alpha, \tau) := \begin{cases} f(\mathbf{v}_{0:b}(\tau)) & \text{if } b = \tau \\ \mathrm{R}^{\alpha}_{\max} & \text{otherwise} \end{cases}$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \neg\varphi, \tau) := -[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) := \min\left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_2, \tau)\right)$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) := \min\begin{pmatrix} \max\left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_2, \tau)\right), \\ \max\left([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_2, \tau)\right) \end{pmatrix}$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau) := \inf_{t \in \tau + I} \left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, t) \right)$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \Diamond_I \varphi, \tau) := \inf_{t \in \tau + I} \left(\max\left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, t), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \Diamond_I \varphi, \tau) \right) \right)$$

$$[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \, \mathcal{U}_I \, \varphi_2, \tau) := \inf_{t \in \tau + I} \left(\max \left(\begin{array}{l} \min \left(\begin{array}{l} \inf_{t' \in [\tau, t)} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, t') \\ [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_2, t) \end{array} \right) \\ [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_1 \, \mathcal{U}_I \, \varphi_2, \tau) \end{array} \right) \right)$$

and a *satisfaction causation distance* $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$, as follows:

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \alpha, \tau) := \begin{cases} f(\mathbf{v}_{0:b}(\tau)) & \text{if } b = \tau \\ \mathtt{R}^{\alpha}_{\min} & \text{otherwise} \end{cases}$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \neg\varphi, \tau) := -[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau) := \max \left(\begin{array}{l} \min \left([\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_2, \tau) \right), \\ \min \left([\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_2, \tau) \right) \end{array} \right)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) := \max \left([\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_2, \tau) \right)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau) := \sup_{t \in \tau + I} \left(\min \left([\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, t), [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau) \right) \right)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \Diamond_I \varphi, \tau) := \sup_{t \in \tau + I} \left([\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, t) \right)$$

$$[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1 \, \mathcal{U}_I \, \varphi_2, \tau) := \sup_{t \in \tau + I} \left(\max \left(\begin{array}{l} \min \left(\begin{array}{l} \sup_{t' \in [\tau, t)} [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_1, t') \\ \inf_{t' \in [\tau, t)} [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_1, t') \\ [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_2, t) \end{array} \right) \\ \min \left(\begin{array}{l} \inf_{t' \in [\tau, t)} [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi_1, t') \\ [\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi_2, t) \end{array} \right) \end{array} \right) \right)$$

Intuitively, a violation causation distance $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau)$ is the spatial distance of the signal value $\mathbf{v}_{0:b}(b)$, at the current instant b, from turning b into a violation causation instant such that b is relevant to the violation of φ (also applied to the satisfaction case dually). It is computed inductively on the structure of φ:

- Case atomic propositions α: if $b = \tau$ (i.e., at which instant α should be evaluated), then the distance of b from being a violation causation instant is $f(\mathbf{v}_{0:b}(b))$; otherwise, if $b \neq \tau$, despite the value of $f(\mathbf{v}_{0:b}(b))$, b can never be a violation causation instant, according to Definition 5, because only $f(\mathbf{v}_{0:b}(\tau))$ is relevant to the violation of α. Hence, the distance will be $\mathtt{R}^{\alpha}_{\max}$;
- Case $\neg\varphi$: b is a violation causation instant for $\neg\varphi$ if b is a satisfaction causation instant for φ, so $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \neg\varphi, \tau)$ depends on $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau)$;
- Case $\varphi_1 \wedge \varphi_2$: b is a violation causation instant for $\varphi_1 \wedge \varphi_2$ if b is a violation causation instant for either φ_1 or φ_2, so $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \wedge \varphi_2, \tau)$ depends on the minimum between $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, \tau)$ and $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_2, \tau)$;

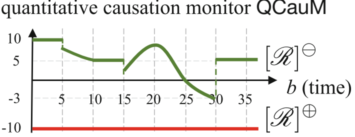


Fig. 5. Quantitative causation monitor (QCauM) result for Example 1

- Case $\varphi_1 \vee \varphi_2$: b is a violation causation instant for $\varphi_1 \vee \varphi_2$ if, first, $\varphi_1 \vee \varphi_2$ has been violated at b, and second, b is the violation causation instant for either φ_1 or φ_2. Hence, $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau)$ depend on both the violation status (measured by $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_i, \tau)$) of one sub-formula and the violation causation distance of the other sub-formula;
- Case $\Box_I \varphi$: b is a violation causation instant for $\Box_I \varphi$ if b is the violation causation instant for the sub-formula φ evaluated at any instant in $\tau + I$. So, $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \Box_I \varphi, \tau)$ depends on the infimum of the violation causation distances regarding φ evaluated at the instants in $\tau + I$;
- Case $\Diamond_I \varphi$: b is a violation causation instant for $\Diamond_I \varphi$ if, first, $\Diamond_I \varphi$ has been violated at b, and second, b is a violation causation instant for the sub-formula φ evaluated at any instant in $\tau + I$. So, $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \Diamond_I \varphi, \tau)$ depends on both the violation status of $\Diamond_I \varphi$ (measured by $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \Diamond_I \varphi, \tau)$) and the infimum of the violation causation distances of φ evaluated in $\tau + I$.
- Case $\varphi_1 \mathcal{U}_I \varphi_2$: $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \mathcal{U}_I \varphi_2, \tau)$ depends on, first, the violation status of the whole formula (measured by $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_1 \mathcal{U}_I \varphi_2, \tau)$), and also, the infimum of the violation causation distances regarding the evaluation of "φ_1 holds until φ_2" at each instant in $\tau + I$.

Similarly, we can also compute the satisfaction causation distance. We use Example 3 to illustrate the quantitative causation monitor for the signals in Example 1.

Example 3. Consider the quantitative causation monitor for the signals in Example 1. At $b = 30$, the violation causation distance is computed as:

$$
\begin{aligned}
&[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:30},\varphi,0) = \inf_{t\in[0,100]} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:30},\varphi',t)\\
&= \inf_{t\in[0,100]} \left(\min \begin{pmatrix} \max\left([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:30},\varphi_1,t), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:30},\varphi_2,t)\right), \\ \max\left([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:30},\varphi_1,t), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:30},\varphi_2,t)\right) \end{pmatrix} \right)\\
&= \inf_{t\in[0,100]} \left(\min \begin{pmatrix} \max\left(-[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:30},\alpha_1,t), \sup_{t'\in t+[0,5]} [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:30},\alpha_2,t')\right) \\ \max\left(-[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:30},\alpha_1,t), \max\begin{pmatrix} [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:30},\varphi_2,t), \\ \inf_{t'\in t+[0,5]} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:30},\alpha_2,t') \end{pmatrix}\right) \end{pmatrix} \right)\\
&= \max\left(-[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:30},\alpha_1,25), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:30},\varphi_2,25), \inf_{t'\in[25,30]} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:30},\alpha_2,t')\right)\\
&= \max(-3,-3,-5) = -3.
\end{aligned}
$$

Similarly, at $b = 35$, the violation causation distance $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:35}, \varphi, 0) = 5$. See the result of `QCauM` shown in Fig. 5. Compared to `ClaM` in Fig. 2, it is evident that `QCauM` provides much more information about the system evolution, e.g., it can report that, in the interval $[15, 20]$, the system satisfies the specification "more", as the `speed` decreases. $\lhd$

By using the violation and satisfaction causation distances reported by `QCauM` jointly, we can infer the verdict of `BCauM`, as indicated by Theorem 2.

Theorem 2. The quantitative causation monitor `QCauM` in Definition 7 refines the Boolean causation monitor `BCauM` in Definition 6, in the sense that:

- if $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$, it implies $\mathscr{M}(\mathbf{v}_{0:b}, \varphi, \tau) = \ominus$;
- if $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$, it implies $\mathscr{M}(\mathbf{v}_{0:b}, \varphi, \tau) = \oplus$;
- if $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi, \tau) > 0$ and $[\mathscr{R}]^{\oplus}(\mathbf{v}_{0:b}, \varphi, \tau) < 0$, it implies $\mathscr{M}(\mathbf{v}_{0:b}, \varphi, \tau) = \oslash$.

Proof. The proof is generally based on mathematical induction. First, by Definition 7 and Definition 5, it is straightforward that Theorem 2 holds for the atomic propositions.

Then, assuming that Theorem 2 holds for an arbitrary formula φ, we prove that Theorem 2 also holds for the composite formula φ' constructed by applying STL operators to φ. The complete proof for all three cases is shown in the full version [38].

As an instance, we show the proof for the first case with $\varphi' = \varphi_1 \vee \varphi_2$, i.e., we prove that $[\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) < 0$ implies $\mathscr{M}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) = \ominus$.

$$
\begin{aligned}
& [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) < 0 & \\
\Rightarrow & \max\Big([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, \tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_2, \tau)\Big) < 0 & \text{(by Def. 7 and w.l.o.g.)} \\
\Rightarrow & [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, \tau) < 0 & \text{(by def. of max)} \\
\Rightarrow & \mathscr{M}(\mathbf{v}_{0:b}, \varphi_1, \tau) = \ominus & \text{(by assumption)} \\
\Rightarrow & \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) \supseteq \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_1, \tau) & \text{(by Def. 5 and Thm. 1)} \\
\Rightarrow & \exists \alpha.\ \langle \alpha, b \rangle \in \mathrm{E}^{\ominus}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) & \text{(by def. of } \supseteq) \\
\Rightarrow & \mathscr{M}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau) = \ominus & \text{(by Def. 6)}
\end{aligned}
$$

$\square$

The relation between the quantitative causation monitor `QCauM` and the Boolean causation monitor `BCauM`, disclosed by Theorem 2, can be visualized by the comparison between Fig. 5 and Fig. 4. Indeed, when the violation causation distance reported by `QCauM` is negative in Fig. 5, `BCauM` reports $\ominus$ in Fig. 4.

Next, we present Theorem 3, which states the relation between the quantitative causation monitor `QCauM` and the classic quantitative monitor `ClaM`.

Theorem 3. The quantitative causation monitor `QCauM` in Definition 7 refines the classic quantitative online monitor `ClaM` in Definition 4, in the sense that, the monitoring results of `ClaM` can be reconstructed from the results of `QCauM`, as follows:

$$[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau) = \inf_{t \in [0,b]} [\mathscr{R}]^{\ominus} (\mathbf{v}_{0:t}, \varphi, \tau) \tag{1}$$

$$[\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau) = \sup_{t \in [0,b]} [\mathscr{R}]^{\oplus} (\mathbf{v}_{0:t}, \varphi, \tau) \tag{2}$$

Proof. The proof is generally based on mathematical induction. First, by Definition 7 and Definition 4, it is straightforward that Theorem 3 holds for the atomic propositions.

Then, we make the global assumption that Theorem 3 holds for an arbitrary formula φ, i.e., both the two cases $\inf_{t\in[0,b]} [\mathscr{R}]^{\ominus} (\mathbf{v}_{0:t}, \varphi, \tau) = [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau)$ and $\sup_{t\in[0,b]} [\mathscr{R}]^{\oplus} (\mathbf{v}_{0:t}, \varphi, \tau) = [\mathrm{R}]^{\mathsf{L}}(\mathbf{v}_{0:b}, \varphi, \tau)$ hold. Based on this assumption, we prove that Theorem 3 also holds for the composite formula φ' constructed by applying STL operators to φ.

As an instance, we prove $\inf_{t\in[0,b]} [\mathscr{R}]^{\ominus} (\mathbf{v}_{0:t}, \varphi', \tau) = [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi', \tau)$ with $\varphi' = \varphi_1 \vee \varphi_2$ as follows. The complete proof is presented in the full version [38].

First, if $b = \tau$, it holds that:

$$\begin{aligned}
&\inf_{t\in[0,b]} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:t},\varphi_1\vee\varphi_2,\tau) = [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:\tau},\varphi_1\vee\varphi_2,\tau)\\
&= \max\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:\tau},\varphi_1,\tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:\tau},\varphi_2,\tau)\Big) && \text{(by Def. 7 and global assump.)}\\
&= [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_1\vee\varphi_2,\tau) && \text{(by Def. 4)}
\end{aligned}$$

Then, we make a local assumption that, given an arbitrary b, it holds that $\inf_{t\in[0,b]} [\mathscr{R}]^{\ominus} (\mathbf{v}_{0:t}, \varphi_1 \vee \varphi_2, \tau) = [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi_1 \vee \varphi_2, \tau)$. We prove that, for b' which is the next sampling point to b, it holds that,

$$\begin{aligned}
&\inf_{t\in[0,b']} [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:t},\varphi_1\vee\varphi_2,\tau)\\
&= \min\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_1\vee\varphi_2,\tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_1\vee\varphi_2,\tau)\Big) && \text{(by local assump.)}\\
&= \min\begin{pmatrix} \max\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_1,\tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_2,\tau)\Big),\\ \max\Big([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_1,\tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b'},\varphi_2,\tau)\Big),\\ \max\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b'},\varphi_1,\tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_2,\tau)\Big) \end{pmatrix} && \text{(by Defs. 4 \& 7)}\\
&= \min\begin{pmatrix} \max\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_1,\tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_2,\tau)\Big),\\ \max\Big([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_1,\tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_2,\tau)\Big),\\ \max\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_1,\tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_2,\tau)\Big),\\ \max\big([\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_1,\tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_2,\tau)\big) \end{pmatrix} && \text{(by global assump.)}\\
&= \max\begin{pmatrix} \min\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_1,\tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_1,\tau)\Big),\\ \min\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b},\varphi_2,\tau), [\mathscr{R}]^{\ominus}(\mathbf{v}_{0:b'},\varphi_2,\tau)\Big) \end{pmatrix} && \text{(by def. of min, max)}\\
&= \max\Big([\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b'},\varphi_1,\tau), [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b'},\varphi_2,\tau)\Big) && \text{(by global assump.)}\\
&= [\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b'},\varphi_1\vee\varphi_2,\tau) && \text{(by Def. 4)}
\end{aligned}$$

□

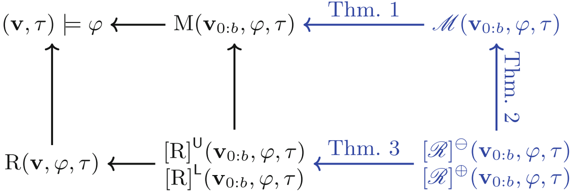


Fig. 6. Refinement among STL monitors

Theorem 3 shows that the result $[\mathrm{R}]^{\mathsf{U}}(\mathbf{v}_{0:b}, \varphi, \tau)$ of `ClaM` can be derived from the result of `QCauM` by applying $\inf_{t\in[0,b]} [\mathscr{R}]^{\ominus} (\mathbf{v}_{0:b}, \varphi, t)$. For instance, comparing the results of `QCauM` in Fig. 5 and the results of `ClaM` in Fig. 2, we can find that the results in Fig. 2 can be reconstructed by using the results in Fig. 5.

Remark 1. Figure 6 shows the refinement relations between the six STL monitoring approaches. The left column lists the offline monitoring approaches derived directly from the Boolean and quantitative semantics of STL respectively. The middle column shows the classic online monitoring approaches. Our two causation monitors, namely `BCauM` and `QCauM`, are given in the column on the right. Given a pair (A, B) of the approaches, $A \leftarrow B$ indicates that the approach B refines the approach A, in the sense that B can deliver more information than A, and the information delivered by A can be derived from the information delivered by B. It is clear that the refinement relation in the figure ensures transitivity. Note that blue arrows are contributed by this paper. As shown by Fig. 6, the relation between `BCauM` and `QCauM` is analogous to that between the Boolean and quantitative semantics of STL.

5 Experimental Evaluation

We implemented a tool[3] for our two causation monitors. It is built on the top of Breach [15], a widely used tool for monitoring and testing of hybrid systems [18]. Being consistent with Breach, the monitors target the output signals given by Simulink models, as an additional block. Experiments were executed on a MacOS machine, 1.4 GHz Quad-Core Intel Core-i5, 8 GB RAM, using Breach v1.10.0.

5.1 Experiment Setting

Benchmarks. We perform the experiments on the following two benchmarks. *Abstract Fuel Control (AFC)* is a powertrain control system from Toyota [27], which has been widely used as a benchmark in the hybrid system community [18–20]. The system outputs the *air-to-fuel* ratio `AF`, and requires that the deviation of `AF` from its reference value `AFref` should not be too large. Specifically, we consider the following properties from different perspectives:

- $\varphi_1^{\mathsf{AFC}} := \square_{[10,50]}(|\mathtt{AF} - \mathtt{AFref}| < 0.1)$: the deviation should always be small;

[3] Available at https://github.com/choshina/STL-causation-monitor, and Zenodo [39].

- $\varphi_2^{\mathsf{AFC}} := \Box_{[10,48.5]}\Diamond_{[0,1.5]}(|\mathtt{AF} - \mathtt{AFref}| < 0.08)$: a large deviation should not last for too long time;
- $\varphi_3^{\mathsf{AFC}} := \Box_{[10,48]}(|\mathtt{AF} - \mathtt{AFref}| > 0.08 \rightarrow \Diamond_{[0,2]}(|\mathtt{AF} - \mathtt{AFref}| < 0.08))$: whenever the deviation is too large, it should recover to the normal status soon.

Automatic transmission (AT) is a widely-used benchmark [18–20], implementing the transmission controller of an automotive system. It outputs the `gear`, `speed` and `RPM` of the vehicle, which are required to satisfy this safety requirement:

- $\varphi_1^{\mathsf{AT}} := \Box_{[0,27]}(\mathtt{speed} > 50 \rightarrow \Diamond_{[1,3]}(\mathtt{RPM} < 3000))$: whenever the `speed` is higher than 50, the `RPM` should be below 3000 in three time units.

Baseline and Experimental Design. In order to assess our two proposed monitors (the Boolean causation monitor `BCauM` in Definition 6, and the quantitative causation monitor `QCauM` in Definition 7), we compare them with two baseline monitors: the classic quantitative robustness monitor `ClaM` (see Definition 4); and the state-of-the-art approach *monitor with reset* `ResM` [40], that, once the signal violates the specification, resets at that point and forgets the previous partial signal.

Given a model and a specification, we generate input signals by randomly sampling in the input space and feed them to the model. The online output signals are given as inputs to the monitors and the monitoring results are collected. We generate 10 input signals for each model and specification. To account for fluctuation of monitoring times in different repetitions[4], for each signal, the experiment has been executed 10 times, and we report average results.

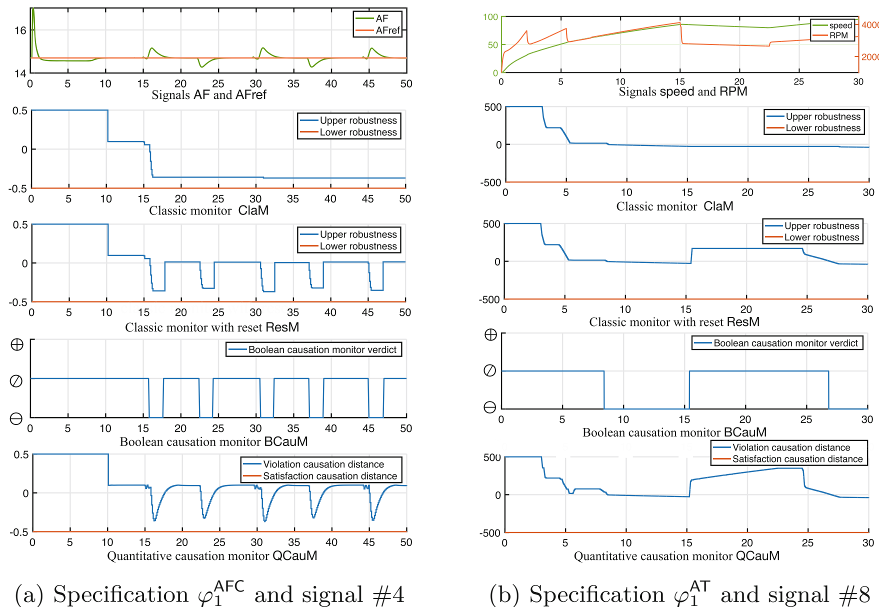
5.2 Evaluation

Qualitative Evaluation. We here show the type of information provided by the different monitors. As an example, Fig. 7 reports, for two specifications of the two models, the system output signal (in the top of the two sub-figures), and the monitoring results of the compared monitors. We notice that signals of both models (top plots) violate the corresponding specifications in multiple points. Let us consider monitoring results of φ_1^{AFC}; similar observations apply to φ_1^{AT}.

When using the `ClaM`, only the first violation right after time 15 is detected (the upper bound of robustness becomes negative); after that, the upper bound remains constant, without reporting that the system recovers from violation at around time 17, and that the specification is violated again four more times.

Instead, we notice that the monitor with reset `ResM` is able to detect all the violations (as the upper bound becomes greater than 0 when the violation episode ends), but it does not properly report the margin of robustness; indeed, during the violation episodes, it reports a constant value of around -0.4 for the upper bound, but the system violates the specification with different degrees of severity in these intervals; in a similar way, when the specification is satisfied around after time 17, the upper bound is just above 0, but actually the system

[4] Note that only the monitoring time changes across different repetitions; monitoring results are instead always the same, as monitoring is deterministic for a given signal.



(a) Specification φ_1^{AFC} and signal #4 (b) Specification φ_1^{AT} and signal #8

Fig. 7. Examples of the information provided by the different monitors

Table 1. Experimental results – Average (avg.) and standard deviation (stdv.) of monitoring and simulation times (ms)

	ClaM				ResM				BCauM				QCauM			
	monitor		total		monitor		total		monitor		total		monitor		total	
	avg.	stdv.	avg.	stdv.	avg.	stdv.	avg.	stdv.	avg.	stdv.	avg.	stdv.	avg.	stdv.	avg.	stdv.
φ_1^{AFC}	14.6	0.1	982.8	3.5	8.8	2.4	981.3	6.7	36.9	5.4	1009.7	16.5	15.1	0.1	981.9	4.4
φ_2^{AFC}	26.8	0.2	998.5	9.0	20.2	5.2	988.0	9.9	50.4	22.4	1023.9	25.1	27.4	0.2	999.5	8.2
φ_3^{AFC}	42.0	0.3	1016.5	8.9	45.5	4.8	1016.9	7.5	48.4	6.2	1021.2	7.9	81.0	1.2	1060.1	5.3
φ_1^{AT}	16.7	0.2	966.0	2.6	24.0	17.0	980.4	24.2	96.1	82.6	1065.2	93.4	31.2	0.6	985.0	7.5

satisfies the specification with different margins. As a consequence, `ResM` provides sharp changes of the robustness upper bound that do not faithfully reflect the system evolution.

We notice that the Boolean causation monitor `BCauM` only reports information about the violation episodes, but not on the degree of violation/satisfaction. Instead, the quantitative causation monitor `QCauM` is able to provide a very detailed information, not only reporting all the violation episodes, but also properly characterizing the degree with which the specification is violated or satisfied. Indeed, in `QCauM`, the violation causation distance smoothly increases from violation to satisfaction, so faithfully reflecting the system evolution.

Quantitative Assessment of Monitoring Time. We discuss the computation cost of doing the monitoring.

Table 2. Experimental results of the four monitoring approaches – Monitoring time (ms) – $\Delta A = {}^{(\texttt{QCauM}-A)}/_{A}$

φ_1^{AFC}	ClaM	ResM	BCauM	QCauM	QCauM stat. (%)		
					ΔClaM	ΔResM	ΔBCauM
#1	14.5	8.2	37.4	15.2	4.8	85.4	−59.4
#2	14.5	8.1	39.9	15.0	3.4	85.2	−62.4
#3	14.8	8.0	38.2	15.0	1.4	87.5	−60.7
#4	14.7	8.5	38.8	15.3	4.1	80.0	−60.6
#5	14.6	8.0	37.3	14.9	2.1	86.3	−60.1
#6	14.6	8.2	37.6	15.1	3.4	84.1	−59.8
#7	14.6	15.5	21.6	15.0	2.7	-3.2	−30.6
#8	14.7	7.9	39.5	15.0	2.0	89.9	−62.0
#9	14.6	7.8	39.9	15.1	3.4	93.6	−62.2
#10	14.5	8.0	38.4	15.1	4.1	88.8	−60.7

φ_2^{AFC}	ClaM	ResM	BCauM	QCauM	QCauM stat. (%)		
					ΔClaM	ΔResM	ΔBCauM
#1	26.8	19.8	45.9	27.4	2.2	38.4	−40.3
#2	27.1	27.3	27.6	27.8	2.6	1.8	0.7
#3	26.6	26.2	30.0	27.5	3.4	5.0	−8.3
#4	26.6	14.2	107.2	27.0	1.5	90.1	−74.8
#5	26.7	15.8	50.9	27.3	2.2	72.8	−46.4
#6	26.6	15.8	56.4	27.2	2.3	72.2	−51.8
#7	26.8	25.4	33.5	27.5	2.6	8.3	−17.9
#8	26.9	17.0	51.9	27.4	1.9	61.2	−47.2
#9	27.1	25.1	50.9	27.6	1.8	10.0	−45.8
#10	26.7	15.8	50.1	27.3	2.2	72.8	−45.5

φ_3^{AFC}	ClaM	ResM	BCauM	QCauM	QCauM stat. (%)		
					ΔClaM	ΔResM	ΔBCauM
#1	42.1	49.2	49.1	81.2	92.9	65.0	65.4
#2	42.5	42.2	42.2	82.1	93.2	94.5	94.5
#3	41.8	48.8	48.8	81.5	95.0	67.0	67.0
#4	42.0	34.9	63.4	78.8	87.6	125.8	24.3
#5	41.7	48.9	48.7	79.6	90.9	62.8	63.4
#6	41.7	48.5	48.7	79.7	91.1	64.3	63.7
#7	42.3	42.7	42.5	81.9	93.6	91.8	92.7
#8	42.1	42.2	42.0	81.6	93.8	93.4	94.3
#9	42.3	49.1	49.3	82.6	95.3	68.2	67.5
#10	41.6	48.6	49.1	80.8	94.2	66.3	64.6

φ_1^{AT}	ClaM	ResM	BCauM	QCauM	QCauM stat. (%)		
					ΔClaM	ΔResM	ΔBCauM
#1	16.9	30.7	29.6	32.1	89.9	4.6	8.4
#2	16.7	17.4	17.4	31.9	91.0	83.3	83.3
#3	16.7	16.8	253.4	31.0	85.6	84.5	−87.8
#4	16.9	69.7	70.2	31.8	88.2	−54.4	−54.7
#5	16.8	19.6	135.9	31.0	84.5	58.2	−77.2
#6	16.5	26.5	200.5	30.2	83.0	14.0	−84.9
#7	16.6	14.6	37.9	31.0	86.7	112.3	−18.2
#8	16.8	16.4	143.8	31.4	86.9	91.5	−78.2
#9	16.3	13.9	38.6	31.0	90.2	123.0	−19.7
#10	16.5	14.2	33.2	30.9	87.3	117.6	−6.9

In Table 1, we observe that, for all the monitors, the *monitor*ing time is much lower than the *total* time (system execution + monitoring). It shows that, for this type of systems, the monitoring overhead is negligible. Still, we compare the execution costs for the different monitors. Table 2 reports the monitoring times of all the monitors for each specification and each signal. Moreover, it reports the percentage difference between the quantitative causation monitor `QCauM` (the most informative one) and the other monitors.

We first observe that `ResM` and `BCauM` have, for the same specification, high variance of the monitoring times across different signals. `ClaM` and `QCauM`, instead, provide very consistent monitoring times. This is confirmed by the standard deviation results in Table 1. The consistent monitoring cost of `QCauM` is a good property, as the designers of the monitor can precisely forecast how long the monitoring will take, and design the overall system accordingly.

We observe that `QCauM` is negligibly slower than `ClaM` for φ_1^{AFC} and φ_2^{AFC}, and at most twice slower for the other two specifications. This additional monitoring cost is acceptable, given the additional information provided by `QCauM`. Compared to `ResM`, `QCauM` is usually slower (at most around the double); also in this case, as `QCauM` provides more information than `ResM`, the cost is acceptable.

Compared to the Boolean causation monitor `BCauM`, `QCauM` is usually faster, as it does not have to collect epochs, which is a costly operation. However, we observe that it is slower in φ_3^{AFC}, because, in this specification, most of the signals do not violate it (and so also `BCauM` does not collect epochs in this case).

To conclude, `QCauM` is a monitor able to provide much more information that exiting monitors, with an acceptable overhead in terms of monitoring time.

6 Related Work

Monitoring of STL. Monitoring can be performed either offline or online. Offline monitoring [16,30,33] targets complete traces and returns either `true` or `false`. In contrast, online monitoring deals with the partial traces, and thus a three-valued semantics was introduced for LTL monitoring [7,8], and in further for MTL and STL qualitative online monitoring [24,31], to handle the situation where neither of the conclusiveness can be made. In usual, the quantitative online monitoring provides a quantitative value or a robust satisfaction interval [12–14,25,26]. Based on them, several tools have been developed, e.g., AMT [32,33], Breach [15], S-Taliro [1], etc. We refer to the survey [3] for comprehensive introduction. Recently, in [35], Qin and Deshmukh propose clairvoyant monitoring to forecast future signal values and give probabilistic bounds on the specification validity. In [2], an online monitoring is proposed for perception systems with Spatio-temporal Perception Logic [23].

Monotonicity Issue. However, most of these works do not handle the monotonicity issue stated in this paper. In [10], Cimatti et al. propose an assumption-based monitoring framework for LTL. It takes the user expertise into account and allows the monitor *resettable*, in the sense that it can restart from any discrete time point. In [37], a recovery feature is introduced in their online monitor [25]. However, the technique is an application-specific approach, rather than a general framework. In [40], a reset mechanism is proposed for STL online monitor. However, as experimentally evaluated in Sect. 5, it essentially provides a solution for the Boolean semantics and still holds monotonicity between two resetting points.

Signal Diagnostics. Signal diagnostics [5,22,32] is originally used in an offline manner, for the purpose of fault localization and system debugging. In [22], the authors propose an approach to automatically address the single evaluations (namely, epochs) that account for the satisfaction/violation of an STL specification, for a complete trace. This information can be further used as a reference for detecting the root cause of the bugs in the CPS systems [5,6,32]. The online version of signal diagnostics, which is the basis of our Boolean causation monitor, is introduced in [40]. However, we show in Sect. 5 that the monitor based on this technique is often costly, and not able to deliver the quantitative runtime information compared to the quantitative causation monitor.

7 Conclusion and Future Work

In this paper, we propose a new way of doing STL monitoring based on causation that is able to provide more information than classic monitoring based on

STL robustness. Concretely, we propose two causation monitors, namely `BCauM` and `QCauM`. In particular, `BCauM` intuitively explains the concept of "causation" monitoring, and thus paves the path to `QCauM` that is more practically valuable. We further prove the relation between the proposed causation monitors and the classic ones.

As future work, we plan to improve the efficiency the monitoring, by avoiding some unnecessary computations for some instants. Moreover, we plan to apply it to the monitoring of real-world systems.

References

1. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
2. Balakrishnan, A., Deshmukh, J., Hoxha, B., Yamaguchi, T., Fainekos, G.: PerceMon: online monitoring for perception systems. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 297–308. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_18
3. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
4. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification. LNCS, vol. 10457. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5
5. Bartocci, E., Ferrère, T., Manjunath, N., Ničković, D.: Localizing faults in Simulink/Stateflow models with STL. In: HSCC 2018, pp. 197–206. ACM (2018). https://doi.org/10.1145/3178126.3178131
6. Bartocci, E., Manjunath, N., Mariani, L., Mateis, C., Ničković, D.: CPSDebug: automatic failure explanation in CPS models. Int. J. Softw. Tools Technol. Transfer **23**(5), 783–796 (2020). https://doi.org/10.1007/s10009-020-00599-4
7. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006). https://doi.org/10.1007/11944836_25
8. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 1–64 (2011). https://doi.org/10.1145/2000799.2000800
9. Ciccone, L., Dagnino, F., Ferrando, A.: Ain't no stopping us monitoring now. arXiv preprint arXiv:2211.11544 (2022)
10. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 165–184. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_10
11. Decker, N., Leucker, M., Thoma, D.: Impartiality and anticipation for monitoring of visibly context-free properties. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 183–200. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_11

12. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Formal Methods Syst. Des. **51**(1), 5–30 (2017). https://doi.org/10.1007/s10703-017-0286-7
13. Dokhanchi, A., Hoxha, B., Fainekos, G.: On-line monitoring for temporal logic robustness. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 231–246. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_19
14. Dokhanchi, A., Hoxha, B., Fainekos, G.: Metric interval temporal logic specification elicitation and debugging. In: MEMOCODE 2015, pp. 70–79. IEEE (2015). https://doi.org/10.1109/MEMCOD.2015.7340472
15. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
16. Donzé, A., Ferrère, T., Maler, O.: Efficient robust monitoring for STL. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 264–279. Springer, Cham (2013). https://doi.org/10.1007/978-3-642-39799-8_19
17. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15297-9_9
18. Ernst, G., et al.: ARCH-COMP 2021 category report: falsification with validation of results. In: Frehse, G., Althoff, M. (eds.) 8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21). EPiC Series in Computing, vol. 80, pp. 133–152. EasyChair (2021). https://doi.org/10.29007/xwl1
19. Ernst, G., et al.: ARCH-COMP 2020 category report: falsification. In: 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20). EPiC Series in Computing, vol. 74, pp. 140–152. EasyChair (2020). https://doi.org/10.29007/trr1
20. Ernst, G., et al.: ARCH-COMP 2022 category report: falsification with unbounded resources. In: Frehse, G., Althoff, M., Schoitsch, E., Guiochet, J. (eds.) Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22). EPiC Series in Computing, vol. 90, pp. 204–221. EasyChair (2022). https://doi.org/10.29007/fhnk
21. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoret. Comput. Sci. **410**(42), 4262–4291 (2009). https://doi.org/10.1016/j.tcs.2009.06.021
22. Ferrère, T., Maler, O., Ničković, D.: Trace diagnostics using temporal implicants. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 241–258. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_20
23. Hekmatnejad, M., Hoxha, B., Deshmukh, J.V., Yang, Y., Fainekos, G.: Formalizing and evaluating requirements of perception systems for automated vehicles using spatio-temporal perception logic (2022). https://doi.org/10.48550/arxiv.2206.14372
24. Ho, H.-M., Ouaknine, J., Worrell, J.: Online monitoring of metric temporal logic. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 178–192. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_15
25. Jakšić, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Ničkovié, D.: From signal temporal logic to FPGA monitors. In: MEMOCODE 2015, pp. 218–227. IEEE (2015). https://doi.org/10.1109/MEMCOD.2015.7340489

26. Jakšić, S., Bartocci, E., Grosu, R., Nguyen, T., Ničković, D.: Quantitative monitoring of STL with edit distance. Formal Methods Syst. Des. **53**(1), 83–112 (2018). https://doi.org/10.1007/s10703-018-0319-x
27. Jin, X., Deshmukh, J.V., Kapinski, J., Ueda, K., Butts, K.: Powertrain control verification benchmark. In: HSCC 2014, pp. 253–262. ACM (2014). https://doi.org/10.1145/2562059.2562140
28. Koymans, R.: Specifying real-time properties with metric temporal logic. Real Time Syst. **2**(4), 255–299 (1990). https://doi.org/10.1007/BF01995674
29. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
30. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
31. Maler, O., Ničković, D.: Monitoring properties of analog and mixed-signal circuits. Int. J. Softw. Tools Technol. Transf. **15**(3), 247–268 (2013). https://doi.org/10.1007/s10009-012-0247-9
32. Ničković, D., Lebeltel, O., Maler, O., Ferrère, T., Ulus, D.: AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. Int. J. Softw. Tools Technol. Transfer **22**(6), 741–758 (2020). https://doi.org/10.1007/s10009-020-00582-z
33. Nickovic, D., Maler, O.: AMT: a property-based monitoring tool for analog systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 304–319. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75454-1_22
34. Pnueli, A.: The temporal logic of programs. In: FOCS 1977, pp. 46–57. IEEE (1977). https://doi.org/10.1109/SFCS.1977.32
35. Qin, X., Deshmukh, J.V.: Clairvoyant monitoring for signal temporal logic. In: Bertrand, N., Jansen, N. (eds.) FORMATS 2020. LNCS, vol. 12288, pp. 178–195. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57628-8_11
36. Sánchez, C., et al.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019). https://doi.org/10.1007/s10703-019-00337-w
37. Selyunin, K., et al.: Runtime monitoring with recovery of the SENT communication protocol. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 336–355. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_17
38. Zhang, Z., An, J., Arcaini, P., Hasuo, I.: Online causation monitoring of signal temporal logic. arXiv (2023). https://doi.org/10.48550/arXiv.2305.17754
39. Zhang, Z., An, J., Arcaini, P., Hasuo, I.: Online causation monitoring of signal temporal logic (Artifact). Zenodo (2023). https://doi.org/10.5281/zenodo.7923888
40. Zhang, Z., Arcaini, P., Xie, X.: Online reset for signal temporal logic monitoring. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(11), 4421–4432 (2022). https://doi.org/10.1109/TCAD.2022.3197693

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Process Equivalence Problems as Energy Games

Benjamin Bisping(✉)

Technische Universität Berlin, Berlin, Germany
benjamin.bisping@tu-berlin.de
https://bbisping.de

Abstract. We characterize all common notions of behavioral equivalence by *one* 6-dimensional energy game, where energies bound capabilities of an attacker trying to tell processes apart. The defender-winning initial credits exhaustively determine which preorders and equivalences from the (strong) linear-time–branching-time spectrum relate processes.

The time complexity is exponential, which is optimal due to trace equivalence being covered. This complexity improves drastically on our previous approach for deciding groups of equivalences where exponential sets of distinguishing HML formulas are constructed on top of a super-exponential reachability game. In experiments using the VLTS benchmarks, the algorithm performs on par with the best similarity algorithm.

Keywords: Bisimulation · Energy games · Process equivalence spectrum

1 Introduction

Many verification tasks can be understood along the lines of "how equivalent" two models are. Figure 1 replicates a standard example, known for instance from the

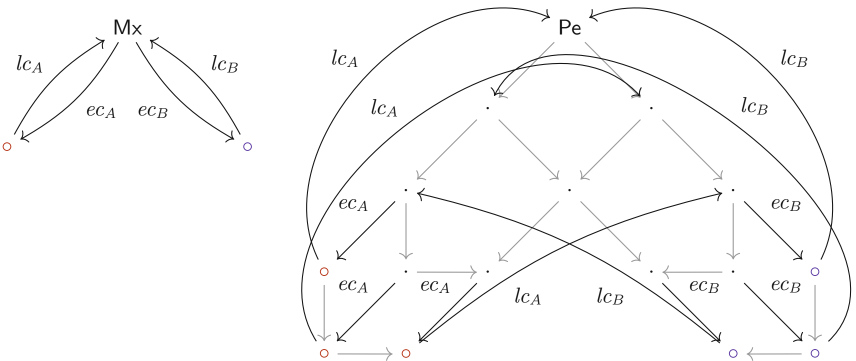


Fig. 1. A specification of mutual exclusion Mx, and Peterson's protocol Pe.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 85–106, 2023.
https://doi.org/10.1007/978-3-031-37706-8_5

textbook *Reactive Systems* [3]: A specification of mutual exclusion Mx as two alternating users A and B entering their critical section ec_A/ec_B and leaving lc_A/lc_B before the other may enter; and the transition system of Peterson's [28] mutual exclusion algorithm Pe, minimized by weak bisimilarity, with internal steps $\rightarrow$ due to the coordination that needs to happen. For Pe to faithfully implement mutual exclusion, it should behave somewhat similarly to Mx.

Semantics in concurrent models must take nondeterminism into account. Setting the degree to which nondeterminism counts induces equivalence notions with subtle differences: Pe and Mx *weakly simulate* each other, meaning that a tree of options from one process can be matched by a similar tree of the other. This implies that they have the same *weak traces*, that is, matching paths. However, they are not weakly *bi*-similar, which would require a higher degree of symmetry than mutual simulation, namely, matching absence of options. There are many more such notions. Van Glabbeek's *linear-time–branching-time spectrum* [21] (cf. Fig. 3) brings order to the hierarchy of equivalences. But it is notoriously difficult to navigate. In our example, one might wonder: Are there notions relating the two *besides* mutual simulation?

Our recent algorithm for *linear-time–branching-time spectroscopy* by Bisping, Nestmann, and Jansen [7,9] is capable of answering equivalence questions for finite-state systems by *deciding the spectrum of behavioral equivalences in one go*. In theory, that is. In practice, the algorithm of [7] runs out of memory when applied to the weak transition relation of even small examples like Pe. The reason for this is that saturating transition systems with the closure of weak steps adds a lot of nondeterminism. For instance, Pe may reach 10 different states by internal steps ($\rightarrow^*$). The spectroscopy algorithm of [7] builds a bisimulation game where the defender wins if the game starts at a pair of equivalent processes. To allow all attacks relevant for the spectrum, the [7]-game must consider partitionings of state sets reached through nondeterminism. There are 115,975 ways of partitioning 10 objects. As a consequence, the game graph of [7] comparing Pe and Mx has 266,973 game positions. On top of each postion, [7] builds sets of distinguishing formulas of Hennessy–Milner modal logic (HML) [21,24] with minimal expressiveness. These sets may grow exponentially. Game over!

Contributions. In this paper, we adapt the spectroscopy approach of [7,9] to render small verification instances like Pe/Mx feasible. The key ingredients that will make the difference are: understanding the spectrum purely through *depth-properties of HML formulas*; using *multidimensional energy games* [15] instead of reachability games; and exploiting the considered spectrum to drastically *reduce the branching-degree* of the game as well as the height of the energy lattice. Figure 2 lays out the algorithm with pointers to key parts of this paper.

- Subsection 2.2 explains how the *linear-time–branching-time spectrum can be understood in terms of six dimensions of HML expressiveness*, and Subsect. 3.1 introduces a class of *declining energy games* fit for our task.
- In Subsect. 3.2, we describe the novel *spectroscopy energy game*, and, in Subsect. 3.3, *prove it to characterize all notions of equivalence definable by the six dimensions.*

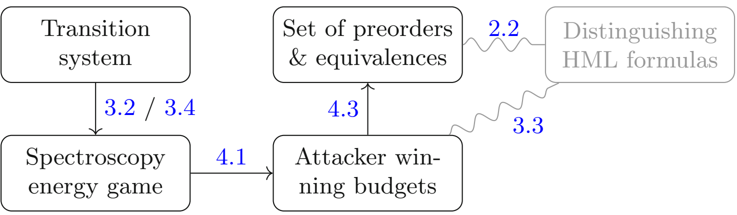


Fig. 2. Overview of the computations → and correspondences ∼ we will discuss.

- Subsection 3.4 shows that a *more clever game with only linear branching-factor* still covers the spectrum.
- Subsection 4.1 provides an algorithm to *compute winning initial energy levels for declining energy games with* $\texttt{min}_{\{\dots\}}$, which enables *decision of the whole considered spectrum in* $2^{\mathcal{O}(|\mathcal{P}|)}$ for systems with $|\mathcal{P}|$ processes (Subsect. 4.2).
- In Subsect. 4.3, we add fine print on *how to obtain equivalences and distinguishing formulas* in the algorithm.
- Section 5 compares to [7] and [29] through *experiments with the widely used VLTS benchmark suite* [18]. The experiments also reveal insights about the suite itself.

2 Distinctions and Equivalences in Transition Systems

Two classic concepts of system analysis form the background of this paper: *Hennessy–Milner logic* (HML) interpreted over *transition systems* goes back to Hennessy and Milner [24] investigating observational equivalence in operational semantics. Van Glabbeek's *linear-time–branching-time spectrum* [21] arranges all common notions of equivalence as a hierarchy of HML sublanguages.

2.1 Transition Systems and Hennessy–Milner Logic

Definition 1 (Labeled transition system). *A* labeled transition system *is a tuple* $\mathcal{S} = (\mathcal{P}, \Sigma, \rightarrow)$ *where* $\mathcal{P}$ *is the set of* processes*,* Σ *is the set of* actions*, and* $\rightarrow \subseteq \mathcal{P} \times \Sigma \times \mathcal{P}$ *is the* transition relation*.*

By $\mathcal{I}(p)$ *we denote the* actions enabled initially *for a process* $p \in \mathcal{P}$*, that is,* $\mathcal{I}(p) := \{a \in \Sigma \mid \exists p'.\, p \xrightarrow{a} p'\}$*. We lift the steps to sets with* $P \xrightarrow{a} P'$ *iff* $P' = \{p' \mid \exists p \in P.\, p \xrightarrow{a} p'\}$*.*

Hennessy–Milner logic expresses *observations* that one may make on such a system. The set of formulas true of a process offers a denotation for its semantics.

Definition 2 (Hennessy–Milner logic). *The* syntax of Hennessy–Milner logic *over a set* Σ *of actions,* $\mathsf{HML}[\Sigma]$*, is defined by the grammar:*

$$\begin{aligned} \varphi &::= \langle a \rangle \varphi && \text{with } a \in \Sigma \\ &\quad\mid \ \textstyle\bigwedge\{\psi, \psi, \dots\} \\ \psi &::= \neg\varphi \mid \varphi. \end{aligned}$$

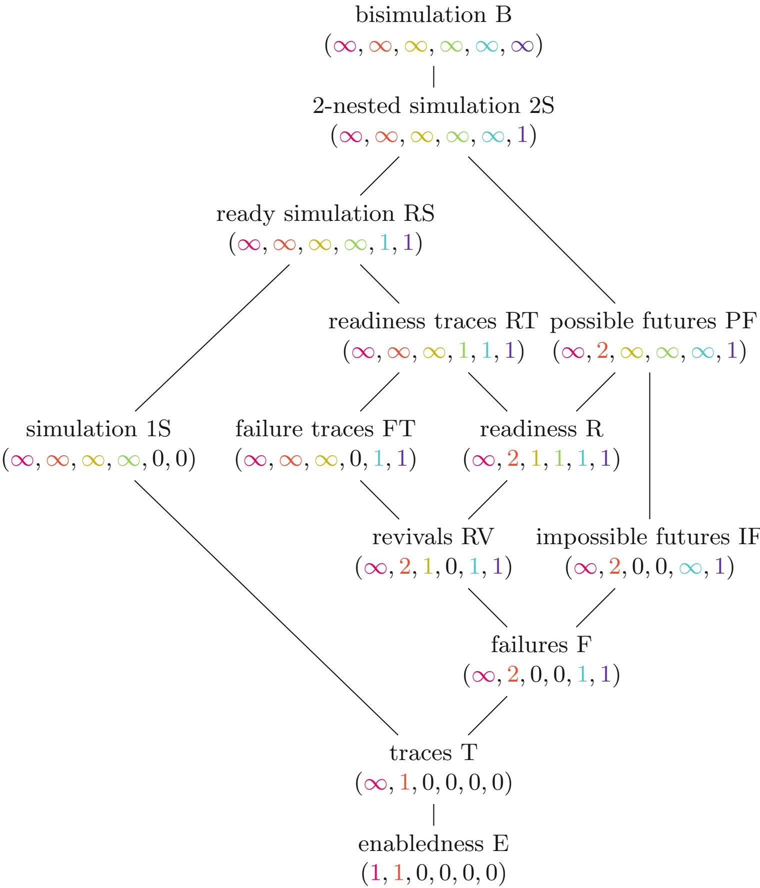


Fig. 3. Hierarchy of equivalences/preorders becoming finer towards the top.

Its semantics $[\![\cdot]\!]^{\mathcal{S}}$ *over a transition system* $\mathcal{S} = (\mathcal{P}, \Sigma, \rightarrow)$ *is given as the set of processes where a formula "is true" by:*

$$
\begin{aligned}
[\![\langle a \rangle \varphi]\!]^{\mathcal{S}} &:= \{p \in \mathcal{P} \mid \exists p' \in [\![\varphi]\!]^{\mathcal{S}}.\, p \xrightarrow{a} p'\} \\
[\![\bigwedge_{i \in I} \psi_i]\!]^{\mathcal{S}} &:= \bigcap \{[\![\psi_i]\!]^{\mathcal{S}} \mid i \in I \wedge \nexists \varphi.\, \psi_i = \neg\varphi\} \\
&\quad \setminus \bigcup \{[\![\varphi]\!]^{\mathcal{S}} \mid \exists i \in I.\, \psi_i = \neg\varphi\}.
\end{aligned}
$$

HML basically extends propositional logic with a modal observation operation. Conjunctions then bound trees of future behavior. Positive conjuncts mean lower bounds, negative ones impose upper bounds. For the scope of this paper, finite bounds suffice, i.e., conjunctions are finite-width. The empty conjunction $\top := \bigwedge \varnothing$ is usually omitted in writing.

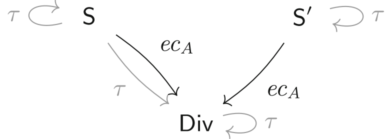


Fig. 4. Example system of internal decision $\xrightarrow{\tau}$ against an action $\xrightarrow{ec_A}$.

We use Hennessy–Milner logic to capture *differences* between program behaviors. Depending on how much of its expressiveness we use, different notions of equivalence are characterized.

Definition 3 (Distinguishing formulas and preordering languages). *A formula $\varphi \in \mathsf{HML}[\Sigma]$ is said to* distinguish *two processes $p, q \in \mathcal{P}$ iff $p \in [\![\varphi]\!]^{\mathcal{S}}$ and $q \notin [\![\varphi]\!]^{\mathcal{S}}$. A sublanguage of Hennessy–Milner logic, $\mathcal{O}_X \subseteq \mathsf{HML}[\Sigma]$, either distinguishes two processes, $p \not\preceq_X q$, if it contains a distinguishing formula, or preorders them otherwise. If processes are preordered in both directions, $p \preceq_X q$ and $q \preceq_X p$, then they are considered X-equivalent, $p \sim_X q$.*

Fig. 3 charts the *linear-time–branching-time spectrum.* If processes are preordered/equated by one notion of equivalence, they also are preordered/equated by every notion below. We will later formally characterize the notions through Proposition 1. For a thorough presentation, we point to [23]. For those familiar with the spectrum, the following example serves to refresh memories.

Example 1. Fig. 4 shows a tiny slice of the weak-step-saturated version of our initial example from Fig. 1 that is at the heart of why Pe and Mx are not bisimulation-equivalent. The difference between S and S' is that S can internally transition to Div (labeled $\xrightarrow{\tau}$) without ever performing an ec_A action. We can express this difference by the formula $\varphi_{\mathsf{S}} := \langle\tau\rangle\bigwedge\{\neg\langle ec_A\rangle\}$, meaning "after τ, ec_A may be impossible." It is true for S, but not for S'. Knowing a distinguishing formula means that S and S' cannot be bisimilar by the Hennessy–Milner theorem. The formula φ_{S} is called a *failure* (or *refusal*) as it specifies a set of actions that are disabled after a trace. In the other direction of comparison, the negation $\varphi_{\mathsf{S}'} := \bigwedge\{\neg\langle\tau\rangle\bigwedge\{\neg\langle ec_A\rangle\}\}$ distinguishes S' from S. The differences between the two processes cannot be expressed in HML without negation. Therefore the processes are simulation-equivalent, or *similar*, as similarity is characterized by the positive fragment of HML.

2.2 Price Spectra of Behavioral Equivalences

For algorithms exploring the linear-time–branching-time spectrum, it is convenient to have a representation of the spectrum in terms of numbers or "prices" of formulas as in [7]. We, here, use six dimensions to characterize the notions of equivalence depicted in Fig. 3. The numbers define the HML observation languages that characterize the very preorders/equivalences. Intuitively, the colorful

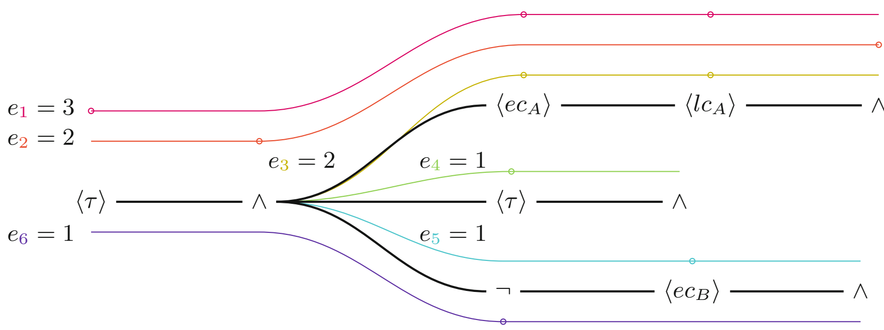


Fig. 5. Pricing e of formula $\langle\tau\rangle\bigwedge\{\langle ec_A\rangle\langle lc_A\rangle\mathsf{T}, \langle\tau\rangle\mathsf{T}, \neg\langle ec_B\rangle\mathsf{T}\}$.

numbers mean: (1) Formula modal depth of observations. (2) Formula nesting depth of conjunctions. (3) Maximal modal depth of deepest positive clauses in conjunctions. (4) Maximal modal depth of the other positive clauses in conjunctions. (5) Maximal modal depth of negative clauses in conjunctions. (6) Formula nesting depth of negations. More formally:

Definition 4 (Energies). *We denote as* energies, **En**, *the set of N-dimensional vectors* $(\mathbb{N})^N$, *and as* extended energies, $\mathbf{En}_\infty$, *the set* $(\mathbb{N} \cup \{\infty\})^N$.

We compare energies component-wise, i.e., $(e_1, \dots, e_N) \leq (f_1, \dots, f_N)$ iff $e_i \leq f_i$ for each i. Least upper bounds sup *are defined as usual as component-wise supremum, as are greatest lower bounds* inf.

Definition 5 (Formula prices). *The* expressiveness price $\mathsf{expr}\colon \mathsf{HML}[\Sigma] \to (\mathbb{N})^6$ *of a formula interpreted as 6-dimensional energies is defined recursively by:*

$$\mathsf{expr}(\langle a\rangle\varphi) := \begin{pmatrix} 1 + \mathsf{expr}_1(\varphi) \\ \mathsf{expr}_2(\varphi) \\ \mathsf{expr}_3(\varphi) \\ \mathsf{expr}_4(\varphi) \\ \mathsf{expr}_5(\varphi) \\ \mathsf{expr}_6(\varphi) \end{pmatrix} \qquad \mathsf{expr}(\neg\varphi) := \begin{pmatrix} \mathsf{expr}_1(\varphi) \\ \mathsf{expr}_2(\varphi) \\ \mathsf{expr}_3(\varphi) \\ \mathsf{expr}_4(\varphi) \\ \mathsf{expr}_5(\varphi) \\ 1 + \mathsf{expr}_6(\varphi) \end{pmatrix}$$

$$\mathsf{expr}(\bigwedge_{i\in I} \psi_i) := \sup\left(\left\{ \begin{pmatrix} 0 \\ 1 + \sup_{i\in I} \mathsf{expr}_2(\psi_i) \\ \sup_{i\in Pos} \mathsf{expr}_1(\psi_i) \\ \sup_{i\in Pos\setminus R} \mathsf{expr}_1(\psi_i) \\ \sup_{i\in Neg} \mathsf{expr}_1(\psi_i) \\ 0 \end{pmatrix} \right\} \cup \{\mathsf{expr}(\psi_i) \mid i \in I\}\right)$$

$$Neg := \{i \in I \mid \exists\varphi_i'.\, \psi_i = \neg\varphi_i'\}$$
$$Pos := I \setminus Neg$$
$$R := \begin{cases} \varnothing & \textit{if } Pos = \varnothing \\ \{r\} & \textit{for some } r \in Pos \textit{ where } \mathsf{expr}_1(\psi_r) \textit{ maximal for } Pos. \end{cases}$$

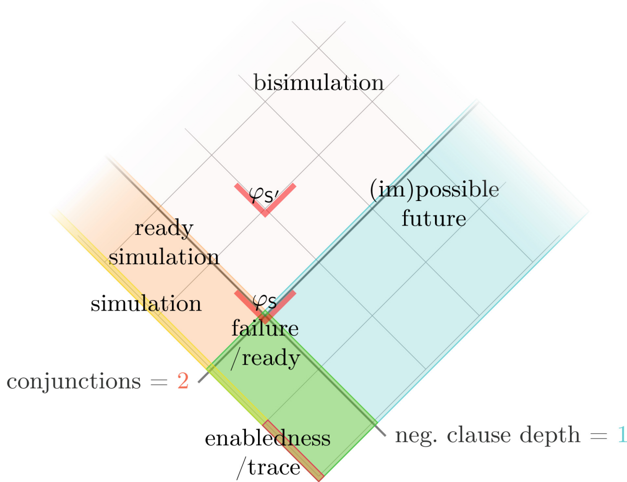


Fig. 6. Cut through the price lattice with dimensions 2 (conjunction nesting) and 5 (negated observation depth).

Figure 5 gives an example how the prices compound. The colors of the lines match those used for the dimensions and their updates in the other figures. Circles mark the points that are counted. The formula itself expresses a so-called *ready-trace* observation: We observe a trace $\tau \cdot ec_A \cdot lc_A$ and, along the way, may check what other options would have been enabled or disabled. Here, we check that τ is enabled and ec_B is disabled after the first τ-step. With the pricing, we can characterize all standard notions of equivalence:

Proposition 1. *On finite systems, the languages of formulas with prices below the coordinates given in Fig. 3 characterize the named notions of equivalence, that is, $p \preceq_X q$ with respect to equivalence X, iff no φ with $\mathsf{expr}(\varphi) \leq e_X$ distinguishes p from q.*

Example 2. The formulas of Example 1 have prices: $\mathsf{expr}(\langle\tau\rangle\bigwedge\{\neg\langle ec_A\rangle\}) = (2,2,0,0,1,1)$ for φ_{S} and $\mathsf{expr}(\bigwedge\{\neg\langle\tau\rangle\bigwedge\{\neg\langle ec_A\rangle\}\}) = (2,3,0,0,2,2)$ for $\varphi_{\mathsf{S}'}$. The prices of the two are depicted as red marks in Fig. 6. This also visualizes how $\varphi_{\mathsf{S}'}$ is a counterexample for bisimilarity and how φ_{S} is a counterexample for failure and finer preorders. Indeed the two preorders are coarsest ways of telling the processes apart. So, S and S' are equated by all preorders *below* the marks, i.e. similarity, $\mathsf{S} \sim_{\mathrm{1S}} \mathsf{S}'$, and coarser preorders ($\mathsf{S} \sim_{\mathrm{T}} \mathsf{S}'$, $\mathsf{S} \sim_{\mathrm{E}} \mathsf{S}'$). This carries over to the initial example of Peterson's mutex protocol from Fig. 1, where weak simulation, $\mathsf{Pe} \sim_{\mathrm{1WS}} \mathsf{Mx}$, is the most precise equivalence. Practically, this means that the specification Mx has liveness properties not upheld by the implementation Px.

Remark 1. Definition 5 deviates from our previous formula pricing of [7] in a crucial way: We only collect the *maximal depths of positive clauses*, whereas [7]

tracks *numbers of deep and flat positive clauses* (where a flat clause is characterized by an observation depth of 1). Our change to a purely "depth-guided" spectrum will allow us to characterize the spectrum by an energy game and to eliminate the Bell-numbered blow up from the game's branching-degree. The special treatment of the deepest positive branch is necessary to address revival, failure trace, and ready trace semantics, which are popular in the CSP community [17,31].

3 An Energy Game of Distinguishing Capabilities

Conventional equivalence problems ask whether a pair of processes is related by a specific equivalence. These problems can be abstracted into a more general "spectroscopy problem" to determine the set of equivalences from a spectrum that relate two processes as in [7]. This section captures the spectrum of Fig. 3 by one rather simple energy game.

3.1 Energy Games

Multidimensional energy games are played on graphs labeled by vectors to be added to (or subtracted from) a vector of "energies" where one player must pay attention to the energies not being exhausted. We plan to encode the distinction capabilities of the semantic spectrum as energy levels in an energy game enriched by $\mathtt{min}_{\{\ldots\}}$-operations that takes minima of components. This way, energy levels where the defender has a winning strategy will correspond to equivalences that hold. We will just need updates decrementing or maintaining energy levels.

Definition 6 (Energy updates). *The set of* energy updates*,* $\mathbf{Up}$*, contains vectors* $(u_1, \ldots, u_N) \in \mathbf{Up}$ *where each component is of the form*

- $u_k \in \{-1, 0\}$*, or*
- $u_k = \mathtt{min}_D$ *where* $D \subseteq \{1, \ldots, N\}$ *and* $k \in D$*.*

Applying an update to an energy, $\mathsf{upd}(e, u)$*, where* $e = (e_1, \ldots, e_N) \in \mathbf{En}$ *(or* $\mathbf{En}_\infty$*) and* $u = (u_1, \ldots, u_N) \in \mathbf{Up}$*, yields a new energy vector* e' *where* k*th components* $e'_k := e_k + u_k$ *for* $u_k \in \mathbb{Z}$ *and* $e'_k := \min_{d \in D} e_d$ *for* $u_k = \mathtt{min}_D$*. Updates that would cause any component to become negative are illegal.*

Definition 7 (Games). *An* N*-dimensional* declining energy game $\mathcal{G}[g_0, e_0] = (G, G_\mathrm{d}, \rightarrowtail, w, g_0, e_0)$ *is played on a directed graph uniquely labeled by energy updates consisting of*

- *a set of* game positions G*, partitioned into*
  - *a set of* defender positions $G_\mathrm{d} \subseteq G$
  - *a set of* attacker positions $G_\mathrm{a} := G \setminus G_\mathrm{d}$*,*
- *a relation of* game moves $\rightarrowtail \subseteq G \times G$*,*
- *a* weight function *for the moves* $w\colon (\rightarrowtail) \to \mathbf{Up}$*,*
- *an* initial position $g_0 \in G$*, and*

– *an* initial energy budget *vector* $e_0 \in \mathbf{En}_\infty$.

The notation $g \overset{u}{\rightarrowtail} g'$ *stands for* $g \rightarrowtail g'$ *and* $w(g, g') = u$.

Definition 8 (Plays, energies, and wins). *We call the (finite or infinite) paths* $\rho = g_0 g_1 \ldots \in G^\infty$ *with* $g_i \overset{u_i}{\rightarrowtail} g_{i+1}$ plays *of* $\mathcal{G}[g_0, e_0]$.

The energy level *of a play* ρ *at round* i, $\mathsf{EL}_\rho(i)$, *is recursively defined as* $\mathsf{EL}_\rho(0) \coloneqq e_0$ *and otherwise as* $\mathsf{EL}_\rho(i{+}1) \coloneqq \mathsf{upd}(\mathsf{EL}_\rho(i), u_i)$. *If we omit the index,* EL_ρ, *this refers to the final energy level of a finite run* ρ, *i.e.*, $\mathsf{EL}_\rho(|\rho| - 1)$.

Plays where energy levels become undefined (negative) are won by the defender. So are infinite plays. If a finite play is stuck (i.e., $g_0 \ldots g_n \not\rightarrowtail$*), the stuck player loses: The defender wins if* $g_n \in G_\mathrm{a}$, *and the attacker wins if* $g_n \in G_\mathrm{d}$.

Proposition 2. *In this model, energy levels can only decline.*

1. *Updates may only decrease energies,* $\mathsf{upd}(e, u) \leq e$.
2. *Energy level changes are monotonic: If* $\mathsf{EL}_{\rho g} \leq \mathsf{EL}_{\sigma g}$ *and* $g \rightarrowtail g'$ *then* $\mathsf{EL}_{\rho g g'} \leq \mathsf{EL}_{\sigma g g'}$.
3. *If* $e_0 \leq e_0'$ *and* $\mathcal{G}[g_0, e_0]$ *has non-negative play* ρ, *then* $\mathcal{G}[g_0, e_0']$ *also has non-negative play* ρ.

Definition 9 (Strategies and winning budgets). *An* attacker strategy *is a map from play prefixes ending in attacker positions to next game moves* $s \colon (G^* \times G_\mathrm{a}) \to G$ *with* $s(g_0 \ldots g_a) \in (g_a \rightarrowtail \cdot)$. *Similarly, a* defender strategy *names moves starting in defender states. If all plays consistent with a strategy* s *ensure a player to win,* s *is called a* winning strategy *for this player. The player with a winning strategy for* $\mathcal{G}[g_0, e_0]$ *is said to* win $\mathcal{G}$ *from position* g_0 *with initial energy budget* e_0. *We call* $\mathsf{Win}_\mathrm{a}(g) = \{e \mid \mathcal{G}[g, e]$ *is won by the attacker*$\}$ *the* attacker winning budgets.

Proposition 3. *The attacker winning budgets at positions are upward-closed with respect to energy, that is,* $e \in \mathsf{Win}_\mathrm{a}(g)$ *and* $e \leq e'$ *implies* $e' \in \mathsf{Win}_\mathrm{a}(g)$.

This means we can characterize the set of winning attacker budgets in terms of minimal winning budgets $\mathsf{Win}_\mathrm{a}^{\min}(g) = \mathrm{Min}(\mathsf{Win}_\mathrm{a}(g))$, where $\mathrm{Min}(S)$ selects minimal elements $\{e \in S \mid \nexists e' \in S.\, e' \leq e \wedge e' \neq e\}$. Clearly, $\mathsf{Win}_\mathrm{a}^{\min}$ must be an antichain and thus finite due to the energies being well-partially-ordered [26]. Dually, we may consider the *maximal* energy levels winning for the defender, $\mathsf{Win}_\mathrm{d}^{\max} \colon G \to \mathbf{2}^{\mathbf{En}_\infty}$ where we need extended energies to bound won half-spaces.

3.2 The Spectroscopy Energy Game

Let us now look at the "spectroscopy energy game" at the center of our contribution. Figure 7 gives a graphical representation. The intuition is that the attacker shows how to construct formulas that distinguish a process p from every q in a set of processes Q. The energies limit the expressiveness of the formulas. The first dimension bounds for how many turns the attacker may challenge observations

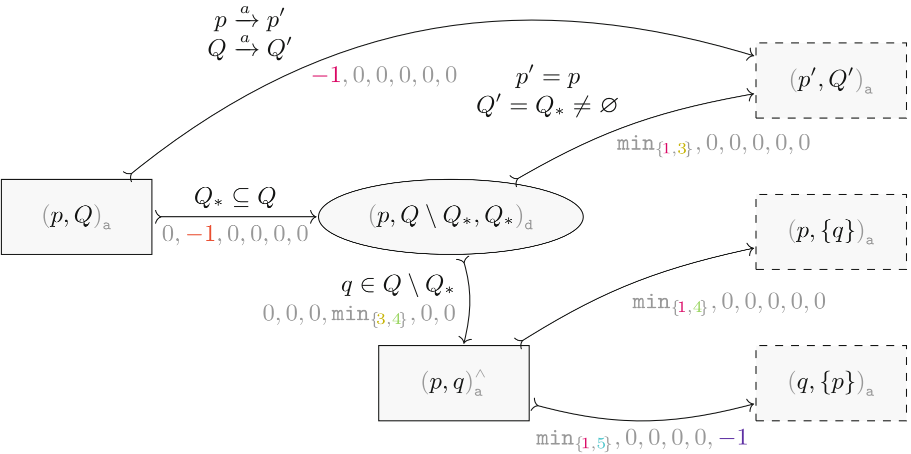


Fig. 7. Schematic spectroscopy game $\mathcal{G}_\triangle$ of Definition 10.

of actions. The second dimension limits how often they may use conjunctions to resolve nondeterminism. The third, fourth, and fifth dimensions limit how deeply observations may nest underneath a conjunction, where the fifth stands for negated clauses, the third for one of the deepest positive clauses and the fourth for the other positive clauses. The last dimension limits how often the attacker may use negations to enforce symmetry by swapping sides. The moves closely match productions in the grammar of Definition 2 and prices in Definition 5.

Definition 10. (Spectroscopy energy game). *For a system* $\mathcal{S} = (\mathcal{P}, \Sigma, \rightarrow)$, *the* 6*-dimensional* spectroscopy energy game $\mathcal{G}^{\mathcal{S}}_\triangle[g_0, e_0] = (G, G_{\mathrm{d}}, \rightarrowtail, w, g_0, e_0)$ *consists of*

- attacker positions $(p, Q)_{\mathrm{a}} \in G_{\mathrm{a}}$,
- attacker clause positions $(p, q)^{\wedge}_{\mathrm{a}} \in G_{\mathrm{a}}$,
- defender conjunction positions $(p, Q, Q_*)_{\mathrm{d}} \in G_{\mathrm{d}}$,

where $p, q \in \mathcal{P}$ *and* $Q, Q_* \in \mathbf{2}^{\mathcal{P}}$, *and six kinds of moves:*

- observation moves $(p, Q)_{\mathrm{a}} \xrightarrowtail{(-1,0,0,0,0,0)} (p', Q')_{\mathrm{a}}$ *if* $p \xrightarrow{a} p'$, $Q \xrightarrow{a} Q'$,
- conj. challenges $(p, Q)_{\mathrm{a}} \xrightarrowtail{(0,-1,0,0,0,0)} (p, Q \setminus Q_*, Q_*)_{\mathrm{d}}$ *if* $Q_* \subseteq Q$,
- conj. revivals $(p, Q, Q_*)_{\mathrm{d}} \xrightarrowtail{(\mathtt{min}_{\{1,3\}},0,0,0,0,0)} (p, Q_*)_{\mathrm{a}}$ *if* $Q_* \neq \varnothing$,
- conj. answers $(p, Q, Q_*)_{\mathrm{d}} \xrightarrowtail{(0,0,0,\mathtt{min}_{\{3,4\}},0,0)} (p, q)^{\wedge}_{\mathrm{a}}$ *if* $q \in Q$,
- positive decisions $(p, q)^{\wedge}_{\mathrm{a}} \xrightarrowtail{(\mathtt{min}_{\{1,4\}},0,0,0,0,0)} (p, \{q\})_{\mathrm{a}}$, *and*
- negative decisions $(p, q)^{\wedge}_{\mathrm{a}} \xrightarrowtail{(\mathtt{min}_{\{1,5\}},0,0,0,0,-1)} (q, \{p\})_{\mathrm{a}}$ *if* $p \neq q$.

The spectroscopy energy game is a bisimulation game in the tradition of Stirling [33].

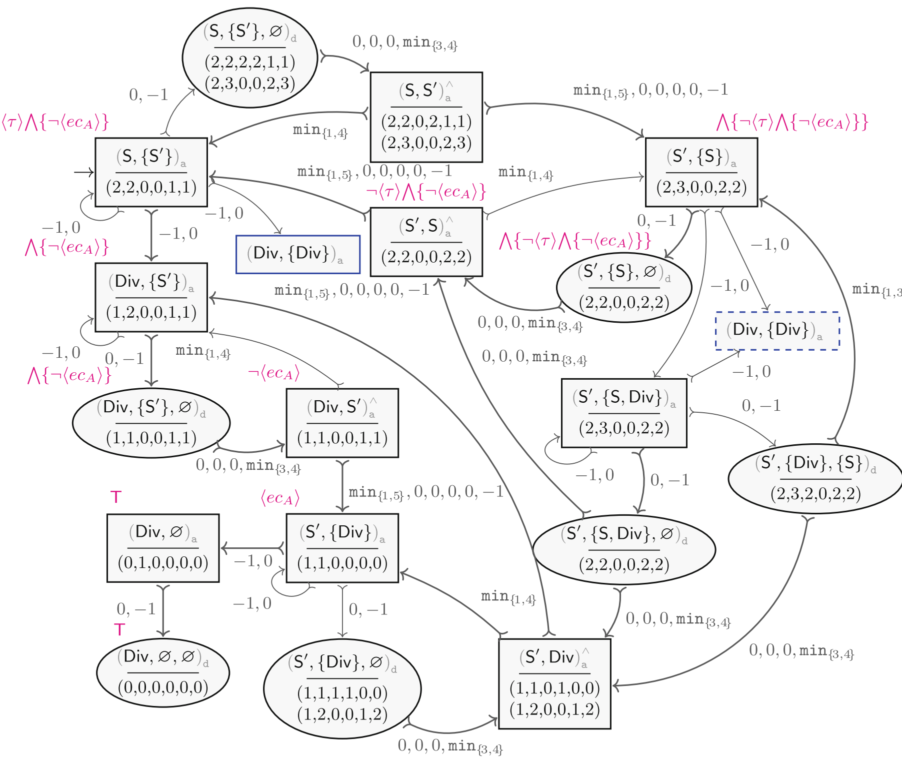


Fig. 8. Example 3 spectroscopy energy game, minimal attacker winning budgets, and distinguishing formulas/clauses. (In order to reduce visual load, only the first components of the updates are written next to the edges. The other components are 0.)

Lemma 1. (Bisimulation game, proof see [5]). *p_0 and q_0 are bisimilar iff the defender wins $\mathcal{G}_\triangle[(p_0, \{q_0\})_\mathrm{a}, e_0]$ for every initial energy budget e_0, i.e. if $(\infty, \infty, \infty, \infty, \infty, \infty) \in \mathsf{Win}_\mathrm{d}^{\max}((p_0, \{q_0\})_\mathrm{a})$.*

In other words, if there are initial budgets winning for the attacker, then the compared processes can be told apart. For $\mathcal{G}_\triangle$, the attacker "unknown initial credit problem" in energy games [34] coincides with the "apartness problem" [20] for processes.

Example 3. Figure 8 shows the spectroscopy energy game starting at $(\mathsf{S}, \{\mathsf{S}'\})_\mathrm{a}$ from Example 1. The lower part of each node displays the node's $\mathsf{Win}_\mathrm{a}^{\min}$. The magenta HML formulas illustrate distinctions relevant for the correctness argument of the following Subsect. 3.3. Section 4 will describe how to obtain attacker winning budgets and equivalences. The blue "symmetric" positions are definitely won by the defender—we omit the game graph below them. We also omit the move $(\mathsf{S}', \{\mathsf{S}, \mathsf{Div}\})_\mathrm{a} \xrightarrow{(0,-1,0,0,0,0)} (\mathsf{S}', \{\mathsf{S}\}, \{\mathsf{Div}\})_\mathrm{d}$—it can be ignored as will be discussed in Subsect. 3.4.

3.3 Correctness: Tight Distinctions

We will check that winning budgets indeed characterize what equivalences hold by constructing price-minimal distinguishing formulas from attacker budgets.

Definition 11 (Strategy formulas). *Given the set of winning budgets* Win_a*, the set of* attacker strategy formulas Strat *for a position with given energy level* e *is defined inductively as follows:*

$\langle b\rangle\varphi \in \mathsf{Strat}((p,Q)_a, e)$ *if* $(p,Q)_a \xrightarrow{u} (p',Q')_a$, $e' = \mathsf{upd}(e,u) \in \mathsf{Win}_a((p',Q')_a)$, $p \xrightarrow{b} p'$, $Q \xrightarrow{b} Q'$, *and* $\varphi \in \mathsf{Strat}((p',Q')_a, e')$,

$\varphi \in \mathsf{Strat}((p,Q)_a, e)$ *if* $(p,Q)_a \xrightarrow{u} (p,Q,Q_*)_d$, $e' = \mathsf{upd}(e,u) \in \mathsf{Win}_a((p,Q,Q_*)_d)$, *and* $\varphi \in \mathsf{Strat}((p,Q,Q_*)_d, e')$,

$\bigwedge_{q\in Q}\psi_q \in \mathsf{Strat}((p,Q,\varnothing)_d, e)$ *if* $(p,Q,\varnothing)_d \xrightarrow{u_q} (p,q)^\wedge_a$, $e_q = \mathsf{upd}(e,u_q) \in \mathsf{Win}_a((p,q)^\wedge_a)$ *and* $\psi_q \in \mathsf{Strat}((p,q)^\wedge_a, e_q)$ *for each* $q \in Q$,

$\bigwedge_{q\in Q\cup\{*\}}\psi_q \in \mathsf{Strat}((p,Q,Q_*)_d, e)$ *if* $(p,Q,Q_*)_d \xrightarrow{u_q} (p,q)^\wedge_a$, $e_q = \mathsf{upd}(e,u_q) \in \mathsf{Win}_a((p,q)^\wedge_a)$ *and* $\psi_q \in \mathsf{Strat}((p,q)^\wedge_a, e_q)$ *for each* $q \in Q$, *and if* $(p,Q,Q_*)_d \xrightarrow{u_*} (p,Q_*)_a$, $e_* = \mathsf{upd}(e,u_*) \in \mathsf{Win}_a((p,Q_*)_a)$, *and* $\psi_* \in \mathsf{Strat}((p,Q_*)_a, e_*)$ *is an observation,*

$\varphi \in \mathsf{Strat}((p,q)^\wedge_a, e)$ *if* $(p,q)^\wedge_a \xrightarrow{u} (p,\{q\})_a$, $e' = \mathsf{upd}(e,u) \in \mathsf{Win}_a((p,\{q\})_a)$ *and* $\varphi \in \mathsf{Strat}((p,\{q\})_a, e')$ *is an observation, and*

$\neg\varphi \in \mathsf{Strat}((p,q)^\wedge_a, e)$ *if* $(p,q)^\wedge_a \xrightarrow{u} (q,\{p\})_a$, $e' = \mathsf{upd}(e,u) \in \mathsf{Win}_a((q,\{p\})_a)$ *and* $\varphi \in \mathsf{Strat}((q,\{p\})_a, e')$ *is an observation.*

Because of the game structure, we actually know the u needed in each line of the definition. It is $u = (-1,0,0,0,0,0)$ in the first case; $(0,-1,0,0,0,0)$ in the second; $(0,0,0,\mathtt{min}_{\{3,4\}},0,0)$ in the third; $(0,0,0,\mathtt{min}_{\{3,4\}},0,0)$ and $(\mathtt{min}_{\{1,3\}},0,0,0,0,0)$ in the fourth; $(\mathtt{min}_{\{1,4\}},0,0,0,0,0)$ in the fifth; and $(\mathtt{min}_{\{1,5\}},0,0,0,0,-1)$ in last case. $\mathsf{Strat}((p,q)^\wedge_a,\cdot)$ can contain negative clauses, which form no proper formulas on their own.

Lemma 2 (Price soundness). $\varphi \in \mathsf{Strat}((p,Q)_a, e)$ *implies that* $\mathsf{expr}(\varphi) \leq e$ *and that* $\mathsf{expr}(\varphi) \in \mathsf{Win}_a((p,Q)_a)$.

Proof. By induction on the structure of φ with arbitrary p, Q, e, exploiting the alignment of the definitions of winning budgets and formula prices. Full proof in [5].

Lemma 3 (Price completeness). $e_0 \in \mathsf{Win}_a((p_0,Q_0)_a)$ *implies there are elements in* $\mathsf{Strat}((p_0,Q_0)_a, e_0)$.

Proof. By induction on the tree of winning plays consistent with some attacker winning strategy implied by $e_0 \in \mathsf{Win}_a((p_0,Q_0)_a)$. Full proof in [5].

Lemma 4 (Distinction soundness). *Every* $\varphi \in \mathsf{Strat}((p,Q)_a, e)$ *distinguishes* p *from every* $q \in Q$.

Proof. By induction on the structure of φ with arbitrary p, Q, e, exploiting that Strat can only construct formulas with the invariant that they are true for p and false for each $q \in Q$. Full proof in [5].

Lemma 5 (Distinction completeness). *If φ distinguishes p from every $q \in Q$, then* $\mathsf{expr}(\varphi) \in \mathsf{Win}_\mathrm{a}((p, Q)_\mathrm{a})$.

Proof. By induction on the structure of φ with arbitrary p, Q, exploiting the alignment of game structure and HML semantics and the fact that expr cannot "overtake" inverse updates. Full proof in [5].

Theorem 1 (Correctness). *For any equivalence X with coordinate e_X, $p \preceq_\mathrm{X} q$, precisely if all $e_{pq} \in \mathsf{Win}_\mathrm{a}^\mathrm{min}((p, \{q\})_\mathrm{a})$ are above or incomparable, $e_{pq} \not\leq e_X$.*

Proof. By contraposition, in both directions.

– Assume $p \not\preceq_X q$. This means some φ with $\mathsf{expr}(\varphi) \leq e_X$ distinguishes p from q. By Lemma 5, $\mathsf{expr}(\varphi) \in \mathsf{Win}_\mathrm{a}((p, \{q\})_\mathrm{a})$. Then either $\mathsf{expr}(\varphi)$ or a lower price $e_{pq} \leq \mathsf{expr}(\varphi)$ are minimal winning budgets, i.e. , $e_{pq} \in \mathsf{Win}_\mathrm{a}^\mathrm{min}((p, \{q\})_\mathrm{a})$ and $e_{pq} \leq e_X$.
– Assume there is $e_{pq} \in \mathsf{Win}_\mathrm{a}^\mathrm{min}((p, \{q\})_\mathrm{a})$ with $e_{pq} \leq e_X$. By Lemma 3, there is $\varphi \in \mathsf{Strat}((p, \{q\})_\mathrm{a}, e_{pq})$. Due to Lemma 4, φ must be distinguishing for p and q, and due to Lemma 2, $\mathsf{expr}(\varphi) \leq e_{pq} \leq e_X$.

The theorem basically means that by fixing an initial budget in $\mathcal{G}_\triangle$, we can obtain a characteristic game for any notion from the spectrum.

3.4 Becoming More Clever by Looking One Step Ahead

The spectroscopy energy game $\mathcal{G}_\triangle$ of Definition 10 may branch exponentially with respect to $|Q|$ at conjunction challenges after $(p, Q)_\mathrm{a}$. For the spectrum we are interested in, we can drastically limit the sensible attacker moves to four options by a little lookahead into the enabled actions $\mathcal{I}(q)$ of $q \in Q$ and $\mathcal{I}(p)$.

Definition 12 (Clever spectroscopy game). *The* clever spectroscopy game*, $\mathcal{G}_\blacktriangle$, is defined exactly like the previous spectroscopy energy game of Definition 10 with the restriction of the conjunction challenges*

$$(p, Q)_\mathrm{a} \quad \xrightarrow{(0,-1,0,0,0,0)}_\blacktriangle \quad (p, Q \setminus Q_*, Q_*)_\mathrm{d} \quad \text{with } Q_* \subseteq Q,$$

to situations where $Q_* \in \{\varnothing, \{q \in Q \mid \mathcal{I}(q) \subseteq \mathcal{I}(p)\}, \ \{q \in Q \mid \mathcal{I}(p) \subseteq \mathcal{I}(q)\}, \{q \in Q \mid \mathcal{I}(p) = \mathcal{I}(q)\}\}$.

Theorem 2 (Correctness of cleverness). *Assume modal depth of positive clauses $e_4 \in \{0, 1, \infty\}$, $e_4 \leq e_3$, and that modal depth of negative clauses $e_5 > 1$ implies $e_3 = e_4$. Then, the attacker wins $\mathcal{G}_\blacktriangle[(p_0, Q_0)_\mathrm{a}, e]$ precisely if they win $\mathcal{G}_\triangle[(p_0, Q_0)_\mathrm{a}, e]$.*

Proof. The implication from the clever spectroscopy game $\mathcal{G}_\blacktriangle$ to the full spectroscopy game $\mathcal{G}_\triangle$ is trivial as the attacker moves in $\rightarrowtail_\blacktriangle$ are a subset of those in $\rightarrowtail_\triangle$ and the defender has the same moves in both games. For the other direction, we have to show that any move $(p, Q)_\mathrm{a} \xrightarrow{(0,-1,0,0,0,0)}_\triangle (p, Q \setminus Q_*, Q_*)_\mathrm{d}$ winning at energy level e can be simulated by a winning move $(p, Q)_\mathrm{a} \xrightarrow{(0,-1,0,0,0,0)}_\blacktriangle (p, Q \setminus Q', Q')_\mathrm{d}$. Full proof in [5].

4 Computing Equivalences

The previous section has shown that attacker winning budgets in the spectroscopy energy game characterize distinguishable processes and, dually, that the defender's wins characterize equivalences. We now examine how to actually compute the winning budgets of both players.

4.1 Computation of Attacker Winning Budgets

The winning budgets of the attacker (Definition 9) are characterized inductively:

- Where the defender is stuck, $g \in G_\mathrm{d}$ and $g \not\rightarrowtail$, the attacker wins with any budget, $(0,0,0,0,0,0) \in \mathsf{Win}_\mathrm{a}^{\min}(g)$.
- Where the defender has moves, $g \in G_\mathrm{d}$ and $g \overset{u_i}{\rightarrowtail} g'_i$ (for some indexing $i \in I$ over all possible moves), the attacker wins if they have a budget equal or above to all budgets that might be necessary after the defender's move: If $\mathsf{upd}(e, u_i) \in \mathsf{Win}_\mathrm{a}(g'_i)$ for all $i \in I$, then $e \in \mathsf{Win}_\mathrm{a}(g)$.
- Where the attacker moves, $g \in G_\mathrm{a}$ and $g \overset{u}{\rightarrowtail} g'$, $\mathsf{upd}(e, u) \in \mathsf{Win}_\mathrm{a}(g')$ implies $e \in \mathsf{Win}_\mathrm{a}(g)$.

By Proposition 3, it suffices to find the finite set of minimal winning budgets, $\mathsf{Win}_\mathrm{a}^{\min}$. Turning this into a computation is not as straightforward as in other energy game models. Due to the $\mathtt{min}_D$-updates, the energy update function $\mathsf{upd}(\cdot, u)$ is neither injective nor surjective.

We must choose an inversion function upd^{-1} that picks minimal solutions and that minimally "casts up" inputs that are outside the image of $\mathsf{upd}(\cdot, u)$, i.e., such that $\mathsf{upd}^{-1}(e', u) = \inf\{e \mid e' \leq \mathsf{upd}(e, u)\}$. We compute it as follows:

Definition 13 (Inverse update). *The* inverse update *function is defined as* $\mathsf{upd}^{-1}(e', u) := \sup(\{e\} \cup \{m(i) \mid \exists D.\, u_i = \mathtt{min}_D\})$ *with* $e_i = e'_i - u_i$ *for all* i *where* $u_i \in \{0, -1\}$ *and* $e_i = e'_i$ *otherwise, and with* $(m(i))_j = e'_i$ *for* $u_i = \mathtt{min}_D$ *and* $j \in D$*, and* $(m(i))_j = 0$ *otherwise, for all* i, j*.*

Example 4. Let $u := (\mathtt{min}_{\{1,3\}}, \mathtt{min}_{\{1,2\}}, -1, -1)$. $(3,4,0,1) \notin \mathrm{img}(\mathsf{upd}(\cdot, u))$, but:

$$\mathsf{upd}^{-1}((3,4,0,1), u) = \sup\{(3,4,1,2), (3,0,3,0), (4,4,0,0)\} = (4,4,3,2)$$

$$\mathsf{upd}((4,4,3,2), u) = (3,4,2,1) \geq (3,4,0,1)$$

$$\mathsf{upd}^{-1}((3,4,2,1), u) = \sup\{(3,4,3,2), (3,0,3,0), (4,4,0,0)\} = (4,4,3,2)$$

With upd^{-1}, we only need to find the $\mathsf{Win}_\mathrm{a}^{\min}$ relation as a least fixed point of the inductive description. This is done by Algorithm 1. Every time a new way of winning a position for the attacker is discovered, this position is added to the todo. Initially, these are the positions where the defender is stuck. The update at an attacker position in Line 8 takes the inversely updated budgets (upd^{-1}) of successor positions to be tentative attacker winning budgets. At a defender position, the attacker only wins if they have winning budgets for all follow-up positions (Line 12). Any supremum of such budgets covering all follow-ups will be winning for the attacker (Line 13). At both updates, we only select the minima as a finite representation of the infinitely many attacker budgets.

```
1  def compute_winning_budgets(𝒢 = (G, G_d, ↣, w)):
2      attacker_win := [g ↦ {} | g ∈ G]
3      todo := {g ∈ G_d | g ↣̸}
4      while todo ≠ ∅:
5          g := some todo
6          todo := todo \ {g}
7          if g ∈ G_a :
8              new_attacker_win := Min(attacker_win[g] ∪ {upd⁻¹(e′, u) |
                   g ↣^u g′ ∧ e′ ∈ attacker_win[g′]})
9          else:
10             defender_post := {g′ | g ↣^u g′}
11             options := {(g′, upd⁻¹(e′, u)) | g ↣^u g′ ∧ e′ ∈ attacker_win[g′]}
12             if defender_post ⊆ dom(options) :
13                 new_attacker_win := Min({sup_{g′∈defender_post} strat(g′) |
                       strat ∈ (G → En) ∧ ∀g′. strat(g′) ∈ options(g′)})
14             else:
15                 new_attacker_win := ∅
16         if new_attacker_win ≠ attacker_win[g] :
17             attacker_win[g] := new_attacker_win
18             todo := todo ∪ {g_p | ∃u. g_p ↣^u g}
19     Win_a^min := attacker_win
20     return Win_a^min
```

Algorithm 1: Algorithm for computing attacker winning budgets of declining energy game $\mathcal{G}$.

4.2 Complexity and How to Flatten It

For finite games, Algorithm 1 is sure to terminate in exponential time of game graph branching degree and dimensionality.

Lemma 6 (Winning budget complexity, proof see [5]). *For an N-dimensional declining energy game with $\rightarrowtail$ of branching degree o, Algorithm 1 terminates in $\mathcal{O}(|\rightarrowtail| \cdot |G|^N \cdot (o + |G|^{(N-1)\cdot o}))$ time, using $\mathcal{O}(|G|^N)$ space for the output.*

Lemma 7 (Full spectroscopy complexity). *Time complexity of computing winning budgets for the full spectroscopy energy game $\mathcal{G}_\triangle$ is in $2^{\mathcal{O}(|\mathcal{P}|\cdot 2^{|\mathcal{P}|})}$.*

Proof. Out-degrees o in $\mathcal{G}_\triangle$ can be bounded in $\mathcal{O}(2^{|\mathcal{P}|})$, the whole game graph $|\rightarrowtail_\triangle| \in \mathcal{O}(|\dot{\rightarrow}| \cdot 2^{|\mathcal{P}|} + |\mathcal{P}|^2 \cdot 3^{|\mathcal{P}|})$, and game positions $|G_\triangle| \in \mathcal{O}(|\mathcal{P}| \cdot 3^{|\mathcal{P}|})$. Insert with $N = 6$ in Lemma 6. Full proof in [5].

We thus have established the approach to be double-exponential. The complexity of the previous spectroscopy algorithm [7] has not been calculated. One must presume it to be equal or higher as the game graph has Bell-numbered branching degree and as the algorithm computes formulas, which entails more options than

the direct computation of energies. This is what lies behind the introduction's observation that moderate nondeterminism already renders [7] unusable.

Our present energy game reformulation allows us to use two ingredients to do way better than double-exponentially when focussing on the common linear-time–branching-time spectrum:

First, Subsect. 3.4 has established that most of the partitionings in attacker conjunction moves can be disregarded by looking at the initial actions of processes.

Second, Fahrenberg et al. [15] have shown that considering just "capped" energies in a grid $\mathbf{En}_K = \{0, \ldots, K\}^N$ can reduce complexity. Such a *flattening of the lattice* turns the space of possible energies into constant factor $(K+1)^N$ (with $(K+1)^{N-1}$-sized antichains) independent of input size. For Algorithm 1, space complexity needed for `attacker_win` drops to $\mathcal{O}(|G|)$ and time complexity to $|{\rightarrowtail}| \cdot 2^{\mathcal{O}(o)}$. If we are only interested in finitely many notions of equivalence as in the case of Fig. 3, we can always bound the energies to range to the maximal appearing number plus one. The last number represents all numbers outside the bound up to infinity.

Lemma 8 (Clever spectroscopy complexity). *Time complexity of computing winning budgets for the clever spectroscopy energy game* $\mathcal{G}_{\blacktriangle}$ *with capped energies is in* $2^{\mathcal{O}(|\mathcal{P}|)}$.

Proof. Out-degrees o in $\mathcal{G}_{\blacktriangle}$ can be bounded in $\mathcal{O}(|\mathcal{P}|)$, the whole game graph $|{\rightarrowtail_{\blacktriangle}}| \in \mathcal{O}(|{\xrightarrow{\cdot}}| \cdot 2^{|\mathcal{P}|} + |\mathcal{P}|^2 \cdot 2^{|\mathcal{P}|})$, and game positions $|G_{\blacktriangle}| \in \mathcal{O}(|\mathcal{P}| \cdot 2^{|\mathcal{P}|})$. Inserting in the flattened version of Lemma 6 yields:

$$\begin{aligned}
\mathcal{O}(|{\rightarrowtail_{\blacktriangle}}| \cdot 2^{C_0 \cdot o}) &= \mathcal{O}((|{\xrightarrow{\cdot}}| \cdot 2^{|\mathcal{P}|} + |\mathcal{P}|^2 \cdot 2^{|\mathcal{P}|}) \cdot 2^{C_1 \cdot |\mathcal{P}|}) \\
&= \mathcal{O}((|{\xrightarrow{\cdot}}| + |\mathcal{P}|^2) \cdot 2^{C_2 \cdot |\mathcal{P}|}) \\
&= \mathcal{O}(|{\xrightarrow{\cdot}}| \cdot 2^{C_2 \cdot |\mathcal{P}|}).
\end{aligned}$$

Deciding trace equivalence in nondeterministic systems is PSPACE-hard and will thus take at least exponential time. Therefore, the exponential time of the "clever" spectroscopy algorithm restricted to a finite spectrum is about as good as it may get, asymptotically speaking.

4.3 Equivalences and Distinguishing Formulas from Budgets

For completeness, let us briefly flesh out how to actually obtain equivalence information from the minimal attacker winning budgets $\mathsf{Win}_{\mathrm{a}}^{\min}((p, \{q\})_{\mathrm{a}})$ we compute.

Definition 14. *For an antichain* $Mn \subseteq \mathbf{En}$ *characterizing an upper part of the energy space, the complement antichain* $\overline{Mn} := \mathrm{Min}\ (\mathbf{En}_\infty \cap (\{(\sup E') - (1, \ldots, 1) \mid E' \subseteq Mn\} \cup \{e(i) \in \mathbf{En}_\infty \mid (e(i))_i = (\inf Mn)_i - 1 \wedge \forall j \neq i.\, (e(i))_j = \infty\}))$ *has the complement energy space as its downset.*

$\mathsf{Win}_{\mathrm{d}}^{\max}((p,\{q\})_{\mathsf{a}}) = \overline{\mathsf{Win}_{\mathrm{a}}^{\min}((p,\{q\})_{\mathsf{a}})}$ characterizes *all* preordering formula languages and thus equivalences defined in terms of expressiveness prices for p and q. This might contain multiple, incomparable, notions from the spectrum. Taking both directions, $\overline{\mathsf{Win}_{\mathrm{a}}^{\min}((p,\{q\})_{\mathsf{a}}) \cup \mathsf{Win}_{\mathrm{a}}^{\min}((q,\{p\})_{\mathsf{a}})}$, will thus characterize the finest intersection of equivalences to equate p and q.

If we just wonder which of the equivalences from the spectrum hold, we may establish this more directly by checking which of them are not dominated by attacker wins.

From the information, we can also easily build witness relations to certify that we return sound equivalence results. In particular, the pairs won with arbitrary attacker budgets, $\{(p,q) \mid (\infty,\infty,\infty,\infty,\infty,\infty) \in \mathsf{Win}_{\mathrm{d}}^{\max}((p,\{q\})_{\mathsf{a}})\}$ are a bisimulation. Similarly, the strategy formulas of Definition 9 can directly be computed to explain inequivalence.

If we use symbolic winning budgets capped as proposed at the end of Subsect. 4.2, the formula reconstruction will be harder and the $\overline{\mathsf{Win}_{\mathrm{a}}^{\min}((p,\{q\})_{\mathsf{a}})}$ might be below the maximal defender winning budgets if these exceed the bound. But this will not matter as long as we choose a cap beyond the natural numbers that characterize our spectrum.

5 Exploring Minimizations

Our algorithm can be used to analyze the equivalence structure of moderately-sized real-world transition systems. In this section, we take a brief look at its performance on the VLTS ("very large transition systems") benchmark suite [18] and return to our initial Peterson example.

The energy spectroscopy algorithm has been added to the Linear-Time–Branching-Time Spectroscope of [7] and can be tried on transition systems at https://equiv.io/.

Table 1 reports the results of running the implementation of [7] and this paper's implementation in variants using the spectroscopy energy game $\mathcal{G}_{\triangle}$ and the clever spectroscopy energy game $\mathcal{G}_{\blacktriangle}$. We tested on the VLTS examples of up to 25,000 states and the Peterson example (Fig. 1). The table lists the $\mathcal{P}$-sizes of the input transition systems and of their bisimilarity quotient system $\mathcal{P}_{/\sim_{\mathrm{B}}}$. The spectroscopies have been performed on the bisimilarity quotient systems by constructing the game graph underneath positions comparing all pairs of enabledness-equivalent states. The middle three groups of columns list the resource usage for the three implementations using: the [7]-spectroscopy, the energy game $\mathcal{G}_{\triangle}$, and the clever game $\mathcal{G}_{\blacktriangle}$. For each group, the first column reports traversed game size, and the second gives the time the spectroscopy took in seconds. Where the tests ran out of memory or took longer than five minutes (in the Java Virtual Machine with 8 GB heap space, at 4 GHz, single-threaded), the cells are left blank. The last three columns list the output sizes of state spaces reduced with respect to enabledness $\sim_{\mathrm{E}}$, traces $\sim_{\mathrm{T}}$, and simulation $\sim_{\mathrm{1S}}$—as one would hope, all three algorithms returned the same results.

From the output, we learn that the VLTS examples, in a way, lack diversity: Bisimilarity $\sim_{\mathrm{B}}$ and trace equivalence $\sim_{\mathrm{T}}$ mostly coincide on the systems (third and penultimate column).

Concerning the algorithm itself, the experiments reveal that the computation time grows mostly linearly with the size of the game move graph. Our algorithm can deal with bigger examples than [7] (which fails at `peterson`, `vasy_10_56` and `cwi_1_2`, and takes more than 500 s for `vasy_8_24`). Even where [7] has a smaller game graph (e.g. `cwi_3_14`), the exponential formula construction renders it slower. Also, the clever game graph $\rightarrowtail_{\blacktriangle}$ indeed is much smaller than $\rightarrowtail_{\triangle}$ for examples with a lot of nondeterminism such as `peterson`.

Table 1. Sample systems, sizes, and benchmark results.

system	$\mathcal{P}$	$\mathcal{P}_{/\sim_{\mathrm{B}}}$	[7]-$\rightarrowtail$	t/s	$\rightarrowtail_{\triangle}$	t/s	$\rightarrowtail_{\blacktriangle}$	t/s	$\mathcal{P}_{/\sim_{\mathrm{E}}}$	$\mathcal{P}_{/\sim_{\mathrm{T}}}$	$\mathcal{P}_{/\sim_{\mathrm{1S}}}$
`peterson`	19	19			348,474	23.31	2,363	0.15	3	11	11
`vasy_0_1`	289	9	1,118	0.17	1,334	0.02	566	0.02	1	9	9
`vasy_1_4`	1,183	28	1,125	0.05	1,320	0.02	1,000	0.02	8	28	28
`vasy_5_9`	5,486	145	3,789	0.14	4,315	0.05	2,988	0.06	109	145	145
`vasy_8_24`	8,879	416	513,690	540.96	725,113	10.48	145,965	2.15	171	415	415
`vasy_8_38`	8,921	219	19,595	0.78	19,690	0.21	14,958	0.19	112	218	218
`vasy_10_56`	10,849	2,112					6,012,676	174.59	13	2,112	2,112
`vasy_18_73`	18,746	4,087									
`vasy_25_25`	25,217	25,217	100,866	1.15	0	0.32	0	0.33	25,217	25,217	25,217
`cwi_1_2`	1,952	1,132					22,723,369	384.13	9	1,132	1,132
`cwi_3_14`	3,996	62	14,761	2.48	25,666	0.28	18,350	0.3	2	62	62

Of those terminating, the heavily nondeterministic `cwi_1_2` is the most expensive example. As many coarse notions must record the nondeterministic options, this blowup is to be expected. If we compare to the best similarity algorithm by Ranzato and Tapparo [29], they report their algorithm SA to tackle `cwi_1_2` single-handedly. Like our implementation, the prototype of SA [29] ran out of memory while determining similarity for `vasy_18_73`. This is in spite of SA theoretically having optimal complexity and similarity being less complex (cubic) than trace equivalence, which we need to cover. The benchmarks in [29] failed at `vasy_10_56`, and `vasy_25_25`, which might be due to 2010's tighter memory requirements (they used 2 GB of RAM) or the degree to which bisimilarity and enabledness in the models is exploited.

6 Conclusion and Related Work

This paper has connected two strands of research in the field of system analysis: The strand of *equivalence games on transition systems* starting with Stirling's bisimulation game [7,12,32,33] and the strand of *energy games for systems of bounded resources* [2,10,11,14–16,27,30,34].

The connection rests on the insight that levels of equivalence correspond to resources available to an attacker who tries to tell two systems apart. This parallel is present in recent work within the security domain [25] just as much as in the first thoughts on observable nondeterminism by Hennessy and Milner [24].

The paper has not examined the precise relationship of the games of Sect. 3 to the whole zoo of VASS, energy, mean-payoff, monotonic [1], and counter games. The spectroscopy energy game deviates slightly from common multi-energy games due to $\mathtt{min}_D$-updates and due to the attacker being energy-bound (instead of the defender). As the energies cannot be exhausted by defender moves, the game can also be interpreted as a VASS game [2,10] where the attacker is stuck if they run out of energy. Our algorithm complexity matches that of general lower-bounded N-dimensional energy games [15]. Links between our declining energy games and other games from the literature might enable slight improvements of the algorithm. For instance, reachability in VASS games can turn polynomial [11].

In the strand of generalized game characterizations for equivalences [7,12,32], this paper extends applicability for real-world systems. The implementation performs on par with the most efficient similarity algorithm [29]. Given that among the hundreds of equivalence algorithms and tools most primarily address bisimilarity [19], a tool for coarser equivalences is a worthwhile addition. Although our previous algorithm [7] is able to solve the spectroscopy problem, its reliance on super-exponential partitions of the state space makes it ill-fit for transition systems with significant nondeterminism. In comparison, our new algorithm also needs one less layer of complexity because it determines equivalences without constructing distinguishing formulas.

These advances enable a spectroscopy of systems saturated by weak transitions. We can thus analyze weak equivalences such as in the example of Peterson's mutex. For special weak equivalences without a strong counterpart such as branching bisimilarity [22], deeper changes to the modal logic are necessary [6].

The increased applicability has allowed us to exhaustively consider equivalences on the smaller systems of the widely-used VLTS suite [18]. The experiments reveal that the spectrum between trace equivalence and bisimilarity mostly collapses for the examined systems. It may often be reasonable to specify systems in such a way that the spectrum collapses. In a benchmark suite, however, a lack of semantic diversity can be problematic: For instance, otherwise sensible techniques like polynomial-time reductions [13] will not speed up language inclusion testing, and nuances of the weak equivalence spectrum [8] will falsely seem insignificant. One may also overlook errors and performance degradations that appear only for transition systems where equal traces do not imply equivalent branching behavior. We hope this blind spot does not affect the validity of any of the numerous studies relying on VLTS benchmarks.

Acknowledgments. This work benefited from discussion with Sebastian Wolf, with David N. Jansen, with members of the LFCS Edinburgh, and with the MTV research group at TU Berlin, as well as from reviewer comments.

Data Availibility. Proofs and updates are to be found in the report version of this paper [5]. The Scala source is on GitHub: https://github.com/benkeks/equivalence-fiddle/. A webtool implementing the algorithm runs on https://equiv.io/. An artifact including the benchmarks is archived on Zenodo [4].

References

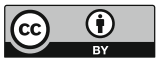
1. Abdulla, P.A., Bouajjani, A., D'orso, J.: Monotonic and downward closed games. J. Log. Comput. **18**(1), 153–169 (2008). https://doi.org/10.1093/logcom/exm062
2. Abdulla, P.A., Mayr, R., Sangnier, A., Sproston, J.: Solving parity games on integer vectors. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 106–120. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_9
3. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification, chap. Modelling mutual exclusion algorithms, pp. 142–158. Cambridge University Press (2007). https://doi.org/10.1017/CBO9780511814105.008
4. Bisping, B.: Linear-time-branching-time spectroscope v0.3.0 (2023). https://doi.org/10.5281/zenodo.7870252, archived on Zenodo
5. Bisping, B.: Process equivalence problems as energy games. Tech. rep., Technische Universität Berlin (2023). https://doi.org/10.48550/arXiv.2303.08904
6. Bisping, B., Jansen, D.N.: Linear-time–branching-time spectroscopy accounting for silent steps (2023). https://doi.org/10.48550/arXiv.2305.17671
7. Bisping, B., Jansen, D.N., Nestmann, U.: Deciding all behavioral equivalences at once: a game for linear-time-branching-time spectroscopy. Logical Methods Comput. Sci. **18**(3) (2022). https://doi.org/10.46298/lmcs-18(3:19)2022
8. Bisping, B., Nestmann, U.: Computing coupled similarity. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 244–261. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_14
9. Bisping, B., Nestmann, U.: A game for linear-time–branching-time spectroscopy. In: TACAS 2021. LNCS, vol. 12651, pp. 3–19. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_1
10. Brázdil, T., Jančar, P., Kučera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14162-1_40
11. Chaloupka, J.: Z-reachability problem for games on 2-dimensional vector addition systems with states is in P. In: Kučera, A., Potapov, I. (eds.) RP 2010. LNCS, vol. 6227, pp. 104–119. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15349-5_7
12. Chen, X., Deng, Y.: Game characterizations of process equivalences. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 107–121. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_8
13. Clemente, L., Mayr, R.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Logical Methods Comput. Sci. **15**(1) (2019). https://doi.org/10.23638/LMCS-15(1:12)2019
14. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. Int. J. Game Theory **8**(2), 109–113 (1979). https://doi.org/10.1007/BF01768705

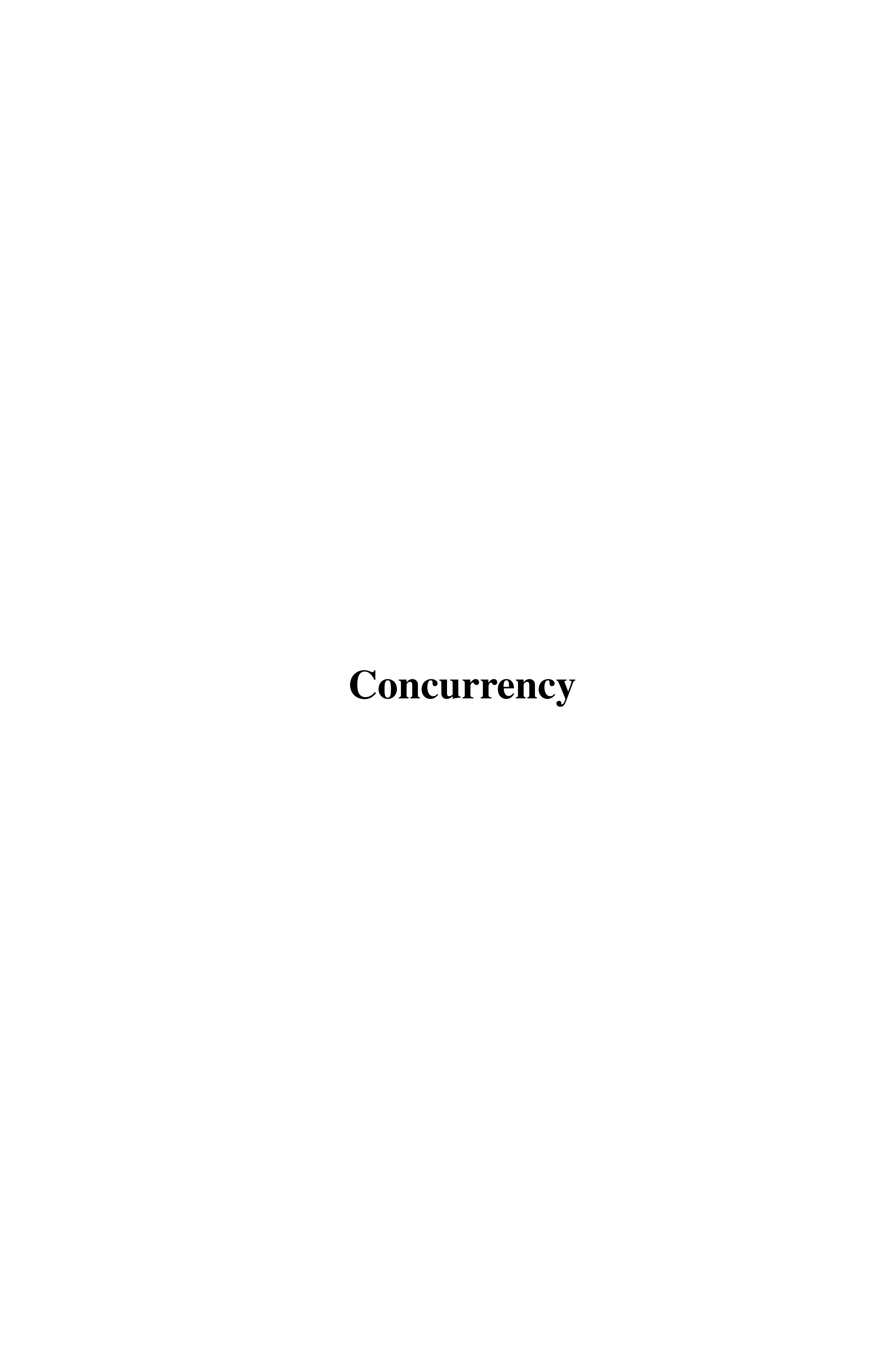
15. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy games in multiweighted automata. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 95–115. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23283-1_9
16. Filiot, E., Hamel-de le Court, E.: Two-player boundedness counter games. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory (CONCUR 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 243, pp. 21:1–21:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.21
17. Fournet, C., Hoare, T., Rajamani, S.K., Rehof, J.: Stuck-free conformance. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 242–254. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_19
18. Garavel, H.: The VLTS benchmark suite (2017). https://doi.org/10.18709/perscido.2017.11.ds100, jointly created by CWI/SEN2 and INRIA/VASY as a CADP resource
19. Garavel, H., Lang, F.: Equivalence checking 40 years after: A review of bisimulation tools. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday, pp. 213–265. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-15629-8_13
20. Geuvers, H., Jacobs, B.: Relating apartness and bisimulation. Logical Methods Comput. Sci. **17**(3) (2021). https://doi.org/10.46298/LMCS-17(3:15)2021
21. Glabbeek, R.J.: The linear time - branching time spectrum. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 278–297. Springer, Heidelberg (1990). https://doi.org/10.1007/BFb0039066
22. Glabbeek, R.J.: The linear time — Branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_6
23. van Glabbeek, R.J.: The linear time-branching time spectrum I: The semantics of concrete, sequential processes. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, chap. 1, pp. 3–99. Elsevier, Amsterdam (2001). https://doi.org/10.1016/B978-044482830-9/50019-9
24. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: de Bakker, J., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 299–309. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10003-2_79
25. Horne, R., Mauw, S.: Discovering ePassport Vulnerabilities using Bisimilarity. Logical Methods Comput. Sci. **17**(2) (2021). https://doi.org/10.23638/LMCS-17(2:24)2021
26. Kruskal, J.B.: The theory of well-quasi-ordering: A frequently discovered concept. J. Combinatorial Theory, Ser. A **13**(3), 297–305 (1972). https://doi.org/10.1016/0097-3165(72)90063-5
27. Kupferman, O., Shamash Halevy, N.: Energy games with resource-bounded environments. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory (CONCUR 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 243, pp. 19:1–19:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.19
28. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**, 115–116 (1981). https://doi.org/10.1016/0020-0190(81)90106-X

29. Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Inf. Comput. **208**(1), 1–22 (2010). https://doi.org/10.1016/j.ic.2009.06.002
30. Reichert, J.: Reachability games with counters: Decidability and algorithms. Theses, École normale supérieure de Cachan - ENS Cachan (2015). https://tel.archives-ouvertes.fr/tel-01314414
31. Roscoe, A.W.: Revivals, stuckness and the hierarchy of CSP models. J. Logic Algebraic Program. **78**(3), 163–190 (2009). https://doi.org/10.1016/j.jlap.2008.10.002
32. Shukla, S.K., Hunt, H.B., Rosenkrantz, D.J.: HORNSAT, model checking, verification and games. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 99–110. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_61
33. Stirling, C.: Modal and temporal logics for processes. Department of Computer Science, University of Edinburgh, Tech. rep. (1993)
34. Velner, Y., Chatterjee, K., Doyen, L., Henzinger, T.A., Rabinovich, A., Raskin, J.F.: The complexity of multi-mean-payoff and multi-energy games. Inf. Comput. **241**, 177–196 (2015). https://doi.org/10.1016/j.ic.2015.03.001

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Concurrency

Commutativity for Concurrent Program Termination Proofs

Danya Lette(✉) and Azadeh Farzan

University of Toronto, Toronto, Canada
danya@cs.toronto.edu, azadeh@cs.toronto.edu

Abstract. This paper explores how using commutativity can improve the efficiency and efficacy of algorithmic termination checking for concurrent programs. If a program run is terminating, one can conclude that all other runs equivalent to it up-to-commutativity are also terminating. Since reasoning about termination involves reasoning about infinite behaviours of the program, the equivalence class for a program run may include infinite words with lengths strictly larger than ω that capture the intuitive notion that some actions may soundly be postponed indefinitely. We propose a sound proof rule which exploits these as well as classic bounded commutativity in reasoning about termination, and devise a way of algorithmically implementing this sound proof rule. We present experimental results that demonstrate the effectiveness of this method in improving automated termination checking for concurrent programs.

1 Introduction

Checking termination of concurrent programs is an important practical problem and has received a lot of attention [3,29,35,37]. A variety of interesting techniques, including thread-modular reasoning [10,34,35,37], causality-based reasoning [29], and well-founded proof spaces [15], among others, have been used to advance the state of the art in reasoning about concurrent program termination. Independently, it has been established that leveraging *commutativity* in proving safety properties can be a powerful tool in improving automated checkers [16–19]. There are many instances of applications of Lipton's reductions [32] in program reasoning [14,28]. Commutativity has been used to simultaneously search for a program with a simple proof and its safety proof [18,19] and to improve the efficiency and efficacy of assertion checking for concurrent programs [16]. Recently [17], *abstract commutativity* relations are formalized and combined to increase the power of commutativity in algorithmic verification.

This paper investigates how using commutativity can improve the efficiency and efficacy of proving the termination of concurrent programs as an enhancement to existing techniques. The core idea is simple: if we know that a program run $\rho ab\rho'$ is terminating, and we know that a and b commute, then we can conclude that $\rho ba\rho'$ is also terminating. Let us use an example to make this idea concrete for termination proofs of concurrent programs. Consider the two thread

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 109–131, 2023.
https://doi.org/10.1007/978-3-031-37706-8_6

Producer Thread:

```
while (i < producer_limit){
   C++; // produce content
   i++
}
barrier++;
```

Consumer Thread:

```
assume(barrier >= producer_num);
while (j < consumer_limit){
   j--;
   C--; // consume content
}
```

Fig. 1. Producer/Consumer Template

templates in Fig. 1: one for a producer thread and one for a consumer thread, where `i` and `j` are local variables. The assumption is that `barrier` and the local counters `i` and `j` are initialized to 0. The producer generates content (modelled by incrementing of a global counter `C++`) up to a limit and then, using `barrier`, signals the consumer to start consuming. Independent of the number of producers and consumers, this synchronization mechanism ensures that the consumers wait for all producers to finish before they start consuming. Note that the producer threads fully commute—each statement in a producer commutes with each statement in another. A producer and consumer only partially commute.

In a program with only two producers, a human would argue at the high level that the *independence* of producer loops implies that their parallel composition is equivalent, up to *commutativity*, to their sequential composition. Therefore, it suffices to prove that the sequential program terminates. In other words, it should suffice to prove that each producer terminates. Let us see how this high level argument can be formalized using commutativity reasoning. Let λ_1 and λ_2 stand for the loop bodies of the two producers. Among others, consider the (syntactic) concurrent program run $(\lambda_1\lambda_2)^\omega$; this run may or may not be feasible. Since λ_1 and λ_2 commute, we can transform this run, by making *infinitely many* swaps, to the run $\lambda_1^\omega\lambda_2^\omega$. The model checking expert would consider this transformation rather misguided: it appears that we are indefinitely postponing λ_2 in favour of λ_1. Moreover, a word with a length strictly larger than ω, called a *transfinite* word, does not have an appropriate representation in language theory because it does not belong to Σ^ω. Yet, the observation that $(\lambda_1\lambda_2)^\omega \equiv \lambda_1^\omega\lambda_2^\omega$ is the key to a powerful proof rule for termination of concurrent programs: If λ_1^ω is terminating and λ_1 commutes against λ_2, then we can conclude that $(\lambda_1\lambda_2)^\omega$ is terminating. In other words, the termination proof for the first producer loop implies that all interleaved executions of two producers terminate, without the need for a new proof. Note that the converse is not true; termination of $\lambda_1^\omega\lambda_2^\omega$ does not necessarily imply the termination of λ_2^ω. So, even if we were to replace the second producer with a forever loop, our observation would stand as is. Hence, for the termination of the entire program (and not just the run $(\lambda_1\lambda_2)^\omega$), one needs to argue about the termination of both λ_1^ω and λ_2^ω, matching the high level argument. In Sect. 3, we formally state and prove this proof rule, called the *omega-prefix* proof rule, and show how it can be incorporated into an algorithmic verification framework. Using this proof rule, the program consisting of N producers can be proved terminating by proving precisely N single-thread loops terminating.

Now, consider adding a consumer thread to our two producer threads. The consumer loop is independent of the producer threads but the consumer thread,

as a whole, is not. In fact, part of the work of a termination prover is to prove that any interleaved execution of a consumer loop with either producer is *infeasible* due to the *barrier* synchronization and therefore terminating. Again, a human would argue that two such cases need to be considered: the consumer crosses the barrier with 0 or 1 producers having terminated. Each case involves several interleavings, but one should not have to prove them correct individually. Ideally, we want a mechanism that can take advantage of commutativity for both cases.

Before we explore this further, let us recall an algorithmic verification template which has proven useful in incorporating commutativity into safety reasoning [16–19] and in proving termination of sequential [25] and parameterized concurrent programs [15]. The work flow is illustrated in Fig. 2. The program and the proof are represented using (Büchi) automata, and module (d) (and consequently module (a)) are implemented as inclusion checks between the languages of these automata. The iteratively refined proof—a language of infeasible syntactic program runs—can be annotated Floyd-Hoare style and generalized using interpolation as in [25]. For module (b), any known technique for reasoning about the termination of simple sequential programs can be used on lassos.

The straightforward way to account for commutativity in this refinement loop would involve module (c): add to Π all program runs equivalent to the existing ones up to commutativity without having a proof for them. In the safety context, it is well-known that checking whether a program is subsumed by the *commutativity closure* of a proof is *undecidable*. We show (in Sect. 3) that the same hurdle exists when doing inclusion checks for program termination.

In the context of safety [16–19], *program reductions* were proposed as an antidote to this undecidability problem: rather than enlarging the proof, one reduces the program and verifies a new program with a subset of the original program runs while maintaining (at least) one representative for each commutativity equivalence class. These representatives are the lexicographically least members of their equivalence classes, and are algorithmically computed based on the idea of the *sleep set* algorithm [22] to construct the automaton for the reduced program. However, using this technique is not possible in termination reasoning where *lassos*, and not finite program runs, are the basic objects.

To overcome this problem, we propose a different class of reductions, called *finite-word reduction*. Inspired by the classical result that an ω-regular language can be faithfully captured as a finite-word language for the purposes of certain

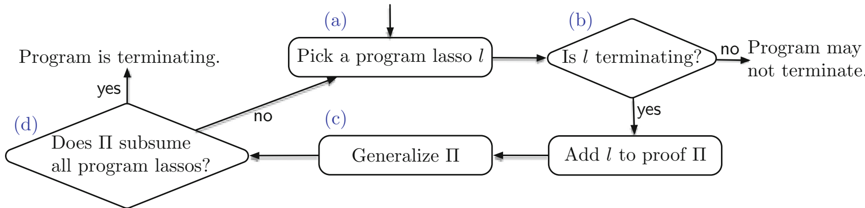


Fig. 2. Refinement Loop For Proving Termination.

checks such as inclusion checks [4], we propose a novel way of translating both the program and the proof into finite-word languages. The classical result is based on an exponentially sized construction and does not scale. We propose a polynomial construction that has the same properties for the purpose of our refinement loop. This contribution can be viewed as an efficient translation of termination analysis to safety analysis and is useful independent of the commutativity context. For the resulting finite-word languages, we propose a novel variation of the *persistent set* algorithm to *reduce* the finite-word program language. This reduction technique is aware of the lasso structure in finite words.

Used together, finite-word reductions and omega-prefix generalization provide an approximation of the undecidable commutativity-closure idea discussed above. They combine the idea of closures, from proof generalization schemes like [15] and reductions from safety [16], into one uniform proof rule that both *reduces* the program and *generalizes* the proof up to commutativity to take as much advantage as possible. Neither the reductions nor the generalizations are ideal, which is a necessity to maintain algorithmic computability. Yet, together, they can perform in a near optimal way in practice: for example, with 2 producers and one consumer, the program is proved terminating by sampling precisely 3 terminating lassos (1 for each thread) and 2 infeasible lassos (one for each barrier failure scenario).

Finally, mostly out of theoretical interest, we explore a class of infinite word reductions that have the same theoretical properties as safety reductions, that is, they are *optimal* and their regularity (in this case, ω-regularity) is guaranteed. We demonstrate that if one opts for the *Foata Normal Form (FNF)* instead of lexicographical normal form, one can construct optimal program reductions in the style of [16,18,19] for termination checking. To achieve this, we use the notion of the FNF of infinite words from (infinite) trace theory [13], and prove the ω-regular analogue of the classical result for regular languages: a reduction consisting of only program runs in FNF is ω-regular, optimal, and can be soundly proved terminating in place of the original program (Sect. 3).

To summarize, this paper proposes a way of improving termination checking for concurrent programs by exploiting commutativity to boost existing algorithmic verification techniques. We have implemented our proposed solution in a prototype termination checker for concurrent programs called TerMute, and present experimental results supporting the efficacy of the method in Sect. 6

2 Preliminaries

2.1 Concurrent Programs

In this paper, programs are *languages* over an alphabet of program statements Σ. The control flow graph for a *sequential* program with a set of locations Loc, and distinct entry and exit locations, naturally defines a finite automaton $(\mathsf{Loc}, \Sigma, \delta, \mathsf{entry}, \{\mathsf{exit}\})$. Without loss of generality, we assume that this automaton is deterministic and has a single exit location. This automaton recognizes *a language* of finite-length words. This is the set of all syntactic program runs that may or may not correspond to an actual program execution.

For the purpose of termination analysis, we are also interested in infinite-length program runs. Given a deterministic finite automaton $\mathcal{A}_L = (Q, \Sigma, \delta, q_0, F)$ with no dead states, where $\mathcal{L}(A_L) = L \subseteq \Sigma^*$ is a regular language of finite-length syntactic program runs, we define $\texttt{Büchi}(\mathcal{A}_L) = (Q, \Sigma, \delta, q_0, Q)$, a Büchi automaton recognizing the language $L^\omega = \{u \in \Sigma^\omega : \forall v \in \mathit{pref}(u).v \in \mathit{pref}(L)\}$, where $\mathit{pref}(u)$ denotes $\{w \in \Sigma^* : \exists w' \in \Sigma^* \cup \Sigma^\omega . w \cdot w' = u\}$ and $\mathit{pref}(L) = \bigcup_{v \in L} \mathit{pref}(v)$. These are all syntactic infinite program runs that may or may not correspond to an actual program execution.

We represent concurrency via interleaving semantics. A concurrent program is a parallel composition of a fixed number of threads, where each thread is a sequential program. Each thread P_i is recognized by an automaton $\mathcal{A}^i_{\mathsf{P}} = (\mathsf{Loc}_i, \Sigma_i, \delta_i, \mathsf{entry}_i, \{\mathsf{exit}_i\})$. We assume the Σ_i's are disjoint. The DFA recognizing $\mathsf{P} = \mathsf{P}_1 || \ldots || \mathsf{P}_n$ is constructed using the standard product construction for a DFA $\mathcal{A}_{\mathsf{P}}$ recognizing the *shuffle* of the languages of the individual thread DFA's.

The language of infinite runs of this concurrent program, denoted P^ω, is the language recognized by $\texttt{Büchi}(\mathcal{A}_{\mathsf{P}})$. Note that a word in the language P^ω may not necessarily be the shuffle of infinite runs of its individual threads.

$$\mathsf{P}^\omega = \{u \in \Sigma^\omega | \ \exists i : \ u|_{\Sigma_i} \in \mathsf{P}^\omega_i \wedge \forall j : \ u|_{\Sigma_j} \in \mathit{pref}(\mathsf{P}_j) \cup \mathsf{P}^\omega_j\}$$

In the rest of the paper, we will simply write P when we mean P^ω for brevity. Note that P^ω includes *unfair* program runs, for example those in which individual threads can be indefinitely starved. As argued in [15], this can be easily fixed by intersecting P^ω with the set of all fair runs.

2.2 Termination

Let $\mathbb{X}$ the domain of the program state, Σ a set of statements, and denote $[\![.]\!] : \Sigma^* \to \mathcal{P}(\mathbb{X} \times \mathbb{X})$ a function which maps a sequence of statements to a relation over the program state, satisfying $[\![s_1]\!][\![s_2]\!] = [\![s_1 \cdot s_2]\!]$ for all $s_1, s_2 \in \Sigma^*$. Define sequential composition of relations in the usual fashion: $r_1 r_2 = \{(x, y) : \exists z.(x, z) \in r_1 \wedge (z, y) \in r_2\}$. We write $s(x)$ to denote $\{y : (x, y) \in [\![s]\!]\} \subseteq \mathbb{X}$.

We say that an infinite sequence of statements $\tau \in \Sigma^\omega$ is infeasible if and only if $\forall x \in \mathbb{X} \ \exists k \in \mathbb{N} \ s_1 \ldots s_k(x) = \emptyset$, where s_i is the ith statement in the run τ. A program—an ω-regular language $P \subseteq \Sigma^\omega$—is terminating if all of its *infinite* runs are infeasible.

$$\frac{\forall \tau \in P, \ \tau \text{ is infeasible}}{P \text{ is terminating}} \qquad (\textsc{Term})$$

Lassos. It is not possible to effectively represent all infinite program runs, but we can opt for a slightly more strict rule by restricting our attention to ultimately periodic runs $UP \subseteq \Sigma^\omega$. That is, runs that are of the form uv^ω for some finite words $u, v \in \Sigma^*$. These are also typically called *lassos*.

It is *unsound* to replace *all runs* with *all ultimately periodic runs* in rule Term. P may be non-terminating while all its ultimately periodic runs are

terminating. Assume that our program P is an ω-regular language and there is a universe $\mathcal{T}$ of known *terminating* programs that are all omega-regular languages. Then, we get the following *sound* rule instead:

$$\frac{\exists \Pi \in \mathcal{T}. P \subseteq \Pi}{P \text{ is terminating}} \qquad (\text{TermUP})$$

If the inclusion $P \subseteq \Pi$ does not hold, then it is witnessed by an ultimately periodic run [4]. In a refinement loop in the style of Fig. 2, one can iteratively expand Π based on this ultimately periodic witness (a.k.a. a lasso), and hence have a termination proof construction scheme in which ultimately periodic runs (lassos) are the only objects of interest. Note that if P includes unfair runs of a concurrent program, rather than fixing it, one can instead initialize Π with all the unfair runs of the concurrent program, which is an ω-regular language. This way, the rule becomes a fair termination rule.

2.3 Commutativity and Traces

An *independence* (or commutativity) relation $I \subseteq \Sigma \times \Sigma$ is a symmetric, anti-reflexive relation that captures the commutativity of a program's statements: $(s_1, s_2) \in I \implies [\![s_1 s_2]\!] = [\![s_2 s_1]\!]$. In what follows, assume such an I is fixed.

Finite Traces. Two finite words w_1 and w_2 are *equivalent* whenever we can apply a finite sequences of swaps of adjacent independent program statements to transform w_1 into w_2. Formally, an independence relation I on statements gives rise to an equivalence relation $\equiv_I$ on words by defining $\equiv_I$ to be the reflexive and transitive closure of the the relation $\sim_I$, defined as $us_1s_2v \sim_I us_2s_1v \iff (s_1, s_2) \in I$. A Mazurkiewicz trace $[u]_I = \{v \in \Sigma^* : v \equiv_I u\}$ is the corresponding equivalence class; we use "trace" exclusively to denote Mazurkiewicz traces.

Infinite Traces. Traces may also be defined in terms of dependence graphs (or partial orders). Given a word $\tau = s_1 s_2 ...$, the dependence graph corresponding to τ is a labelled, directed, acyclic graph $G = (V, E)$ with labelling function $L : V \to \Sigma$ and vertices $V = \{1, 2, \dots\}$, where $L(i) = s_i$, and $(i, i') \in E$ whenever $i < i'$ and $(L(i), L(i')) \notin I$. Then, $[\tau]_I^\infty$, the equivalence class of the infinite word τ, is precisely the set of *linear extensions* of G. Therefore, $\tau' \equiv_I \tau$ iff τ' is a linear extension of G.

For example, Fig. 3(i) illustrates the Hasse diagram of the finite trace $[abcba]_I$, and Fig. 3(ii), the Hasse diagram of the infinite trace $[abc(ab)^\omega]_I^\infty$, where $I = \{(a, b), (b, a)\}$.

For an infinite word τ, the *infinite trace* $[\tau]_I^\infty$ may contain linear extensions that do not

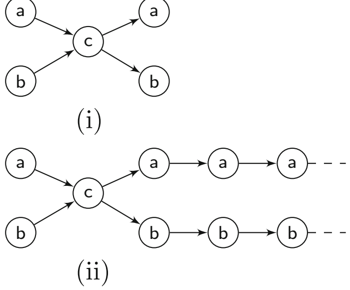


Fig. 3. Hasse diagrams.

correspond to any word in Σ^ω. For example, if $I = \{(a,b),(b,a)\}$, then the trace $[(ab)^\omega]_I^\infty$ includes a member (infinite word) in which all as appear before all bs. We use $a^\omega b^\omega$ to denote this word and call such words **transfinite**. This means that $[\tau]_I^\infty \nsubseteq \Sigma^\omega$, even for an ultimately periodic τ.

Normal Forms. A trace, as an equivalence class, may be represented unambiguously by one of its member words. *Lexicographical* normal forms [13] are the most commonly used normal forms, and the basis for the commonly known *sleep set* algorithm in partial order reduction [22]. *Foata Normal Forms* (FNF) are less well-known and are used in the technical development of this paper:

Definition 1 (Foata Normal Form of a finite trace [13]). *For a finite trace t, define FNF(t) as a sequence of sets $S_1 S_2 ... S_k$ (for some $k \in \mathbb{N}$) where $t = \Pi_i^k S_i$ and for all i:*

$$\forall a, b \in S_i \;\; a \neq b \implies (a,b) \in I \qquad \textit{(no dependencies in } S_i \textit{)}$$
$$\forall b \in S_{i+1} \; \exists a \in S_i \;\; (a,b) \notin I \qquad \textit{(} S_i \textit{ dependent on } S_{i+1} \textit{)}$$

Given a trace's dependence graphs, the FNF can be constructed by repeatedly removing sets of minimal elements, that is, sets with no incoming edges. Although we have defined a trace's FNF as a sequence of sets, we will generally refer to a trace's FNF as a word in which the elements in each set are assumed to be ordered lexicographically. For example, $\text{FNF}([abcba]_I) = ab \cdot c \cdot ab$, where $I = \{(a,b),(b,a)\}$. We overload this notation by writing $\text{FNF}([u]_I)$ as $\text{FNF}(u)$, and, for a language L, $\text{FNF}(L) = \{\text{FNF}(u) : u \in L\}$.

Theorem 1 ([13]). *L is a regular language iff the set of its Foata (respectively Lexicographical) normal forms is a regular language.*

3 Closures and Reductions

Commutativity defines an equivalence relation $\equiv_I$ which preserves the termination of a program run.

Proposition 1. *For $\tau, \tau' \in \Sigma^\omega$ and $\tau' \equiv_I \tau$, τ is terminating iff τ' is terminating.*

In the context of a refinement loop in the style of Fig. 2, Proposition 1 suggests one can take advantage of commutativity by including all runs that are equivalent to the ones in Π (which are already proved terminating) in module (c). We formally discuss this strategy next.

Given a language L and an independence relation I, define $[L]_I^\infty = \cup_{\tau \in L} [\tau]_I^\infty$. Recall from Sect. 2 that, in general, $[\tau]_I^\infty \nsubseteq \Sigma^\omega$. Since programs are represented

by ω-regular languages in our formalism, it is safe for us to exclude transfinite words from $[\tau]_I^\infty$ from commutativity closures computation. Define:

$$[L]_I^\omega = \cup_{\tau \in L}[\tau]_I^\infty \cap \Sigma^\omega \qquad (\omega\text{-closure})$$

The following sound proof rule is a straightforward adaptation of Rule TermUP that takes advantage of commutativity-based proof generalization:

$$\frac{\exists \Pi \subseteq \mathcal{T}.P \subseteq [\Pi]_I^\omega}{P \text{ is terminating}} \qquad (\text{TermClosure})$$

Recall the example from Sect. 1 with two producers. The *transfinite* program run $\lambda_1^\omega \lambda_2^\omega$ that is the sequential compositions of the two producers looping forever back to back does not belong to the ω-closure of any ω-regular language. We generalize the notion of ω-closure to incorporate the idea of such runs in a new proof rule.

Let τ a transfinite word (like $a^\omega b^\omega$). Let τ' a prefix of τ. If $|\tau'| = \omega$, we say that τ' is an ω-prefix of τ, or $\tau' \in \mathit{pref}_\omega(\tau)$. A direct definition for when a transfinite word τ is terminating would be rather contrived, since a word such as $a^\omega b^\omega$ does not correspond to a program execution in the usual sense. However, a very useful property arises when considering the ω-words of $\mathit{pref}_\omega(\tau)$: If an ω-prefix τ' of a transfinite word τ is terminating, then all words in $[\tau]_I^\omega$ are terminating.

Theorem 2 (Omega Prefix Proof Rule). *Let $\tau'', \tau' \in \Sigma^\omega, \tau$ a transfinite word, if $\tau \equiv_I \tau''$ and $\tau' \in \mathit{pref}_\omega(\tau)$, τ' terminates $\Rightarrow \tau''$ terminates.*

Remark that $[\tau]_I^\omega \subseteq \Sigma^\omega$, so the former theorem uses the usual definition of termination, i.e. termination of ω-words; however; this theorem implicitly defines a notion of termination for some transfinite words.

Define $[\tau]_I^{p\omega}$, the *omega-prefix closure* of τ as

$$[\tau]_I^{p\omega} = [\tau]_I^\omega \cup \bigcup_{\tau'.\tau \in \mathit{pref}_\omega(\tau')} [\tau']_I^\omega.$$

Theorem 2 guarantees that, if τ terminates, then all of $[\tau]_I^{p\omega}$ terminates. The converse, however, does not necessarily hold: $[\tau]_I^{p\omega}$ is not an equivalence class.

Example 1. Continuing the example in Fig. 1, recall that λ_1 and λ_2 are independent. Let us assume we have a proof that λ_1^ω is terminating. The class $[\lambda_1^\omega]_I^\omega = \{\lambda_1^\omega\}$ does not include any other members and therefore we cannot conclude the termination status of any other program runs based on it. On the other hand, since $\lambda_1^\omega \in \mathit{pref}_\omega(\lambda_1^\omega \lambda_2^\omega)$ and $[(\lambda_1\lambda_2)^\omega]_I^\omega = [\lambda_1^\omega \lambda_2^\omega]_I^\omega$, $(\lambda_1\lambda_2)^\omega \in [\lambda_1^\omega]_I^{p\omega}$. Therefore, we can conclude that $(\lambda_1\lambda_2)^\omega$ is also terminating. Note that λ_2 can be non-terminating and the argument still stands.

One can replace the closure in Rule TermClosure with omega-prefix closure and produce a new, more powerful, sound proof rule. There is, however, a major obstacle in the way of an algorithmic implementation of Rule TermClosure with either closure scheme: the inclusion check in the premise is not decidable.

Proposition 2. *$[L]_I^\omega$ and $[L]_I^{p\omega}$ for an ω-regular language L may not be ω-regular. Moreover, it is undecidable to check the inclusions $L_1 \subset [L_2]_I^\omega$ and $L_1 \subset [L_2]_I^{p\omega}$ for ω-regular languages L_1 and L_2.*

3.1 The Compromise: A New Proof Rule

In the context of safety verification, with an analogous problem, a dual approach was proposed as a way forward [18] based on *program reductions*.

Definition 2 (ω-Reduction and ωp-Reduction). *A language $R \subseteq P$ is an ω-reduction (resp. ωp-reduction of P) of program P under independence relation I iff for all $\tau \in P$ there is some $\tau' \in R$ such that $\tau \in [\tau']_I^\omega$ (resp. $\tau \in [\tau']_I^{p\omega}$).*

The idea is that a program reduction can be soundly proven in place of the original program but, with strictly fewer behaviours to prove correct, less work has to be done by the prover.

Proposition 3. *Let P be a concurrent program and Π be ω-regular. We have:*

- *$P \subseteq [\Pi]_I^\omega$ iff there exists an ω-reduction R of P under I such that $R \subseteq \Pi$.*
- *$P \subseteq [\Pi]_I^{p\omega}$ iff there exists an ωp-reduction R of P under I such that $R \subseteq \Pi$.*

An $\omega/\omega p$-reduction R may not always be ω-regular. However, Proposition 3 puts forward a way for us to make a compromise to rule TermClosure for the sake of algorithmic implementability. Consider a *universe* of program reductions $\mathsf{Red}(P)$, which does not include *all* reductions. This gives us a new proof rule:

$$\frac{\exists \Pi \in \mathcal{T}. \exists R \in \mathsf{Red}(P). R \subseteq \Pi}{P \text{ is terminating}} \qquad (\text{TermReduc})$$

If $\mathsf{Red}(P)$ is the set of *all* ω-reductions (resp. ωp-reductions), then Rule TermReduc becomes logically equivalent to Rule TermClosure (resp. with $[\Pi]_I^{p\omega}$). By choosing a strict subset of all reductions for $\mathsf{Red}(P)$, we trade the undecidable premise check of the proof rule TermClosure with a new decidable premise check for Rule TermReduc. The specific algorithmic problem that this paper solves is then the following: What are good candidates for $\mathsf{Red}(P)$ such that an effective and efficient algorithmic implementation of Rule TermReduc exists? Moreover, we want this implementation to show significant advantages over the existing algorithms that implement the Rule TermUP.

In Sect. 5, we propose *Foata Reduction* as a theoretically clean option for $\mathsf{Red}(P)$ in the universe of all ω-reductions. In particular, they have the algorithmically essential property that the reductions do not include any transfinite words. In the universe of ωp-reductions, which does account for transfinite words, such a theoretically clean notion does not exist. This paper instead proposes the idea of mixing both closures and reductions as a best algorithmic solution for the undecidable Rule TermClosure in the form of the following new proof rule:

$$\frac{\exists \Pi \subseteq \mathcal{T}. \exists R \in \mathsf{Red}(P). R \subseteq [\Pi]_I^{opg}}{P \text{ is terminating}} \qquad (\textsc{TermOP})$$

In Sect. 3.2, we introduce $[\Pi]_I^{opg}$ as an underapproximation of $[\Pi]_I^{p\omega}$ that is guaranteed to be ω-regular and computable. Then, in Sect. 4, we discuss how, through a representation shift from infinite words to finite words, an appropriate class of reductions for $\mathsf{Red}(P)$ can be defined and computed.

3.2 Omega Prefix Generalization

We can implement the underapproximation of $[\Pi]_I^{p\omega}$ by generalizing the proof of termination of each individual lasso in the refinement loop of Fig. 2. Let $u_1, ...u_m, v_1, ...v_{m'} \in \Sigma$ and consider the lasso uv^ω, where $u = u_1...u_m, v = v_1...v_{m'}$, and $m' > 0$. Let $\mathcal{A}_{uv^\omega} = (Q, \Sigma, \delta, q_0, \{q_m\})$ a Büchi automaton consisting of a stem and a loop, with a single accepting state q_m at the head of the loop, recognizing the ultimately periodic word uv^ω—in [25], this automaton is called a *lasso module* of uv^ω. Let $\Sigma_{I_{loop}} \subseteq \Sigma = \{a : \{v_1, ..., v_{m'}\} \times \{a\} \subseteq I\}$ the statements that are independent with the statements $v_1, \ldots, v_{m'}$ of the loop, and $\Sigma_{I_{stem}} \subseteq \Sigma_{I_{loop}} = \{a : \{u_1, \ldots, u_m, v_1, \ldots, v_{m'}\} \times \{a\} \subseteq I\}$ the statements that are independent of all statements appearing in uv^ω.

Define $\mathcal{OPG}(\mathcal{A}_\tau) = (Q \cup \{q'\}, \Sigma, \delta_{\mathcal{OPG}}, q_0, \{q_m\})$ for a lasso $\tau = uv^\omega$ where

$$\delta_{\mathcal{OPG}}(q, a) = \begin{cases} q & \text{if } q \in \{q_0, ..., q_{m-1}\} \wedge a \in \Sigma_{I_{stem}} \\ & \text{ or if } q \in \{q_{m+1}, ..., q_{m+m'}\} \cup \{q'\} \wedge a \in \Sigma_{I_{loop}} \\ q' & \text{if } q = q_m \wedge a \in \Sigma_{I_{loop}} \text{ or } m' = 1 \text{ and } a = v_1 \\ \delta(q_m, v_1) & \text{if } q = q' \wedge a = v_1 \\ \delta(q, a) & \text{o.w.} \end{cases}$$

We refer to the language $\mathcal{L}(\mathcal{OPG}(\mathcal{A}_\tau))$ recognized by this automaton as $[\tau]_I^{opg}$ for short. Note that this construction is given for individual lassos; we may generalize this to a (finite) set of lassos by simply taking their union. For a lasso $\tau = uv^\omega$, $\mathcal{OPG}(\mathcal{A}_\tau)$ is a linearly-sized Büchi automaton whose language satisfies the following:

Proposition 4. $[\tau]_I^{opg} \subseteq [\tau]_I^{p\omega}$.

Intuitively, this holds because this automaton simply allows us to intersperse the statements of uv^ω with independent statements; when considering the Mazurkiewicz trace arising from a word interspersed as described, these added independent statements may all be ordered after uv^ω, resulting in a transfinite word with ω-prefix uv^ω.

Theorem 3. *If τ is terminating, then every run in $[\tau]_I^{opg}$ is terminating.*

This follows directly from Theorem 2 and Proposition4, and concludes the soundness and algorithmic implementability of Rule TermOP if $\mathsf{Red}(P) = \{P\}$.

4 Finite-Word Reductions

In this section, inspired by the program reductions used in safety verification, we propose a way of using those families of reductions to implement $\mathsf{Red}(P)$ in Rule TermReduc. This method can be viewed as a way of translating the liveness problem into an equivalent safety problem.

In [4], a finite-word encoding of ω-regular languages was proposed that can be soundly used for checking inclusion in the premise of rules such as Rule TermReduc:

Definition 3 (\$-language [4]). *Let $L \in \Sigma^\omega$. Define the \$-language of L as*

$$\$(L) = \{u\$v|\ u, v \in \Sigma^* \wedge uv^\omega \in L\}.$$

If L is ω-regular, then $\$(L)$ is regular [4]. This is proved by construction, but the one given in [4] is exponential. Since the Büchi automaton for a concurrent program P is already large, an exponential blowup to construct $\$(P)$ can hardly be tolerated. We propose an alternative polynomial construction.

4.1 Efficient Reduction to Safety

Our polynomial construction, denoted by fast\$, consists of linearly many copies of the Büchi automaton recognizing the program language.

Definition 4 (fast\$). *Given a Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, define fast\$$(\mathcal{A}) = (Q_\$, \Sigma \cup \{\$\}, \delta_\$, q_0, F_\$)$ with $Q_\$ = Q \cup (Q \times Q \times \{0, 1\})$, $F_\$ = \{(q, q, 1) : q \in Q\}$, and for $q, r \in Q$, $i \in \{0, 1\}$:*

$$\delta_\$(q, a) = \begin{cases} \{(q, q, 0)\} & \text{if } a = \$ \\ \delta(q, a) & o.w. \end{cases}$$

$$\delta_\$((q, r, i), a) = \begin{cases} \{(q, r', 1) : r' \in \delta(r, a)\} & \text{if } i = 0 \text{ and } r \in F \\ \{(q, r', i) : r' \in \delta(r, a)\} & o.w. \end{cases}$$

Let L be an ω-regular language and $\mathcal{A}$ be a Büchi automaton recognizing L. We overload the notation and use fast\$$(L)$ to denote the language recognized by fast\$$(\mathcal{A})$. Note that fast\$$(L)$, unlike $\$(L)$, is a construction parametric on the Büchi automaton recognizing the language, rather than the language itself. In general, fast\$$(L)$ under-approximates $\$(L)$. But, under the assumption that all alphabet symbols of Σ label at most one transition in the Büchi automaton $\mathcal{A}$ (recognizing L), then fast\$$(L) = \$(L)$. This condition is satisfied for any Büchi automaton that is constructed from the control flow graph of a (concurrent) program since we may treat each statement appearing on the graph as unique, and these graph edges correspond to the transitions of the automaton.

Theorem 4. *For any ω-regular language L, we have fast\$$(L) \subseteq \$(L)$. If P is a concurrent program then fast\$$(P) = \$(P)$.*

First, let us observe that in Rule TermUP, we can replace P with $\mathsf{fast\$}(P)$ and Π with $\mathsf{fast\$}(\Pi)$ (and hence the universe $\mathcal{T}$ with a correspondingly appropriate universe) and derive a new *sound* rule.

Theorem 5. *The finite word version of Rule* TermUP *using* $\mathsf{fast\$}$ *is sound.*

The proof of Theorem 5 follows from Theorem 4. Using $\mathsf{fast\$}$, the program is precisely represented and the proof is under-approxiamted, therefore the inclusion check implies the termination of the program.

4.2 Sound Finite Word Reductions

With a finite word version of the Rule TermUP, the natural question arises if one can adopt the analogue of the sound proof rule used for safety [18] by introducing an appropriate class of reductions for program termination in the following proof rule:

$$\frac{\exists \Pi \in \mathcal{T}. \exists R \in \mathsf{Red}(\$(P)). R \subseteq \mathsf{fast\$}(\Pi)}{P \text{ is terminating}} \qquad \text{(FiniteTermReduc)}$$

A language R is a *sound reduction* of $\$(P)$ if the termination of all ultimately periodic words uv^ω, where $u\$v \in R$, implies the termination of all ultimately periodic words of P. Since, in $u\$v$, the word u represents the stem of a lasso and the word v represents its loop, it is natural to define equivalence, considering the two parts separately, that is: $u\$v \equiv_I u'\$v'$ iff $u' \equiv_I u \wedge v' \equiv_I v$. One can use any technique for producing reductions for safety, for example *sleep sets* for lexicographical reductions [18], in order to produce a *sound* reduction that includes representatives from this equivalence relation. Assume that \$ does not commute with any other letter in an extension $I_\$$ of I over $\Sigma \cup \{\$\}$ and observe that the standard finite-length word Mazurkiewicz equivalence relation of $u\$v \equiv_{I_\$} u'\$v'$ coincides with $u\$v \equiv_I u'\$v'$ as defined above. Let $\mathit{FRed}(\$(P))$ be the set of all such reductions. An algorithmic implementation of Rule FiniteTermReduc with $\mathsf{Red}(\$(P)) = \mathit{FRed}(\$(P))$ may then be taken straightforwardly from the safety literature.

Note, however, that reductions in $\mathit{FRed}(\$(P))$ are more restrictive than their infinite analogues; for example, $uv\$v \notin [u\$v]_I$, whereas $uvv^\omega = uv^\omega$ and therefore $uvv^\omega \equiv_I uv^\omega$ for any I. By treating $\$(P)$'s \$-word as a a finite word without recognizing its underlying lasso structure, every word uv^ω in the program necessarily engenders an infinite family of representatives in R—one for each \$-word $\{u\$v, uv\$v, u\$vv, ...\} \subseteq \$(P)$ corresponding to $uv^\omega \in P$.

We define *dollar closure* as variant of classic closure that is sensitive to the termination equivalence of the corresponding infinite words:

$$[u\$v]_I^\$ = \{x\$y : uv^\omega \in [xy^\omega]_I^{p\omega}\}$$

The termination of uv^ω is implied by the termination of any xy^ω such that $x\$y$ is a member of $[u\$v]_I^\$$ (see Theorem 2). However, the converse does not necessarily

hold. Therefore, like omega-prefix closure, $[u\$v]^\$_I$ is not an equivalence class. It suggests a more relaxed condition (than the one used for $\mathit{FRed}(\$(P))$) for the soundness of a reduction:

Definition 5 (Sound \$-Program Reduction). *A language $R \subseteq P$ is called a* sound *\$-program reduction of $\$(P)$ under independence relation I iff for all $uv^\omega \in P$ we have $[u\$v]^\$_I \cap R \neq \emptyset$.*

A \$-reduction R satisfying the above condition is obviously sound: It must contain a \$-representative $x\$y \in [u\$v]^\$_I$ for each word uv^ω in the program. If R is terminating, then xy^ω is terminating, and therefore so is uv^ω. Moreover, these sound \$-program reductions can be quite parsimonious, since one word can be an omega-prefix corresponding to many classes of program behaviours.

Under this soundness condition, we may now include one representative of $[u\$v]^\$_I$ for each $uv^\omega \in P$ in a sound reduction of P. For example, $R = \{\$a, \$b\}$ is a sound \$-program reduction of $P = a^\omega || b^\omega$ when $(a, b) \in I$. To illustrate, note that the only traces of P are the three depicted as Hasse diagrams in Fig. 4; the distinct program words $(ab)^\omega, (aba)^\omega, (abaa)^\omega, ...$ all correspond to the same infinite trace shown in Fig. 4(iii). A salient feature of Fig. 4(iii) is that a^ω and b^ω correspond to *disconnected* components of this dependence graph. The omega-prefix rule of Theorem 2 can be interpreted in this graphical context as follows: if any *connected component* of the trace is terminating, then the entire class is terminating.

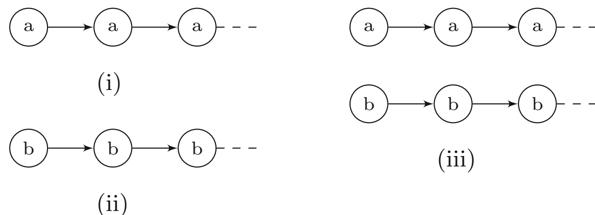


Fig. 4. The only three traces in $P = a^\omega || b^\omega$ when $(a, b) \in I$.

Recall that module (d) of the refinement loop of Fig. 2 may naturally be implemented as the inclusion check $P \subseteq \Pi$, or one of its variations that appear in the proof rules proposed throughout this paper. In a typical inclusion check, a product of the program and the complement of the proof automata are explored for the reachability of an accept state. Therefore, classic reduction techniques that operate on the program by pruning transitions/states during this exploration are highly desirable in this context. We propose a repurposing of such techniques that shares the simplicity and efficiency of constructing reductions from $\mathit{FRed}(\$(P)))$ (in the style of safety) and yet takes advantage of the weaker soundness condition in Definition 5 and performs a more aggressive reduction. In short, a reduced program may be produced by pruning transitions while performing an on-the-fly exploration of the program automaton. In pruning, our

goal is to discard transitions that would necessarily form words whose suffixes lead us into the disconnected components of the program traces underlying the program words that have been explored so far. This selective pruning technique is provided by a straightforward adaptation of the well-known safety reduction technique of persistent sets [22]. Consider the program illustrated in Fig. 5(a). In the graph in Fig. 5(b), the green states are explored and the dashed transitions are pruned. This amounts to proving two lassos terminating in the refinement loop of Fig. 2, where each lasso corresponds to one connected component of a program trace.

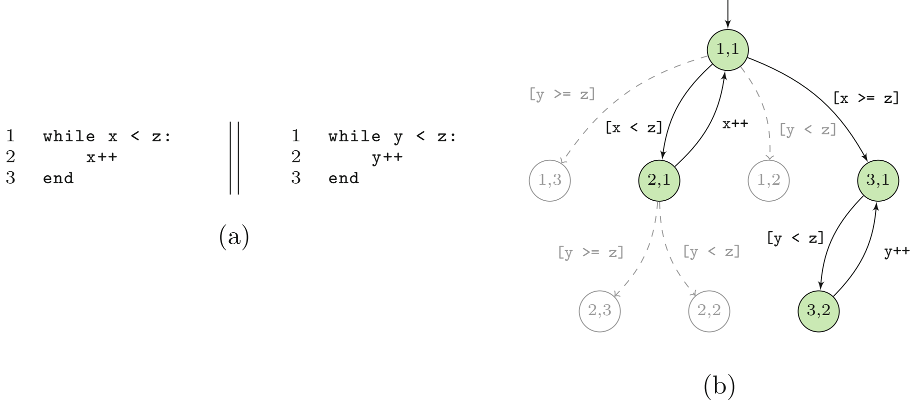


Fig. 5. Example of persistent set selective search.

We compute persistent sets using a variation of Algorithm 1 in Chap. 4 of [22]. In brief, $a \in Persistent_{\prec}(q)$ if a is the lexicographically least enabled state at q according to thread order $\prec$, if a is an enabled statement from the same thread as another statement $a' \in Persistent_{\prec}(q)$, or if a is dependent on some statement $a' \in Persistent_{\prec}(q)$ from a different thread than a. In addition, \$ is also persistent whenever it is enabled. This set may be computed via a fixed-point algorithm; whenever a statement that is not enabled is added to $Persistent_{\prec}(q)$, then $Persistent_{\prec}(q)$ is simply the set of all enabled states. Intuitively, this procedure works because transitions are ignored only when they are necessarily independent from all the statements that will be explored imminently; these may soundly be ignored indefinitely or deferred. Transitions that are

Algorithm 1: PersistentSS

```
   Input: fast$(A_P) = (Q, Σ, δ, q_0, F)
   Output: x$y
1  H ← ∅, S ← {(q_0, “”)}
2  while (q, w) = S.pop() do
3  |  if q ∉ H then
4  |  |  if q ∈ F then
5  |  |  |  return w
6  |  |  for a ∈ Σ ∩ Persistent(q) do
7  |  |  |  S.push(δ(q, a), w · a)
8  |  |  H ← H ∪ {q}
9  return “EMPTY”
```

deferred indefinitely are precisely those that would lead into a disconnected component of a program traces.

The reduced program that arises from the persistent set selective search of $\mathsf{fast\$}(\mathcal{A}_P)$ based on thread order $\prec$ is denoted by $\mathtt{PersistentSS}_{\prec}(\$(P))$. Figure 5(b) illustrates a reduced program; note that \$-transitions are omitted for simplicity. The reduced program corresponds to the states shown in green. The other program states are unreachable because the only persistent transitions correspond to statements from the least enabled thread; the transitions shown with dashed lines are not persistent.

Theorem 6 (soundness of finite word reductions). *Rule* FINITETERMREDUC *is a sound proof rule when* $\mathsf{Red}(\$(P)) = \{\forall \prec : \mathtt{PersistentSS}_{\prec}(\$(P))\}$.

The theorem holds under the condition that the set $\mathcal{T}$ from Rule FINITETERMREDUC is the set of all terminating ω-regular languages, and the under the assumption that the program is fair (or, equivalently, that the proof includes the unfair runs of P, as discussed in Sect. 2.2), where a fair run is one where no enabled thread action is indefinitely deferred. The proof of soundness appears in the extended version of this paper [31]. Intuitively, it relies on the fact that $\mathtt{PersistentSS}_{\prec}(\$(P))$ is a *\$-program reduction* for all the fair runs in P.

Example 2. Recall the producer-consumer in Fig. 1, and consider the program with two producers P_1 and P_2 and one consumer C. Let λ_1 denote the loop body of P_1, and λ_2 that of P_2. Concretely, $\lambda_1 =$ `[i < producer_limit] ; C++ ; i++` where `[...]` is an *assume* statement, and similarly for λ_2. In addition, each loop has an exit statement, which we denote by ι_1 and ι_2. For instance, $\iota_1 =$ `[i >= producer_limit]`. Let $\prec$ such that $P_1 \prec P_2 \prec C$.

In $\mathcal{A} = \mathtt{PersistentSS}_{\prec}(\$(P))$, P_1 is the *first* thread and therefore persistent; that is, the word $\$\lambda_1$—the \$-word corresponding to λ_1^ω – is in the reduction. Since λ_1 is independent of all statements in P_2 and C, any run in which P_1 enters the loop (and does not exit via ι_1) will not be included in the reduction. In effect, this means that λ_1^ω is the only representative of $[\lambda_1^\omega]_I^{p\omega} = [\lambda_1^\omega]_I^\omega \cup [\lambda_1^\omega \cdot (P_2 + C)^\omega]_I^\omega$ in the program reduction.

Even though P_2 seems identical to P_1, the same is not true for P_2 because it appears later in the thread order. In this case, $[\lambda_2]_I^{p\omega}$ is represented by the family of words $(\lambda_1)^* \iota_1 \lambda_2^\omega$.

5 Omega Regular Reductions

In the classic implementation of Rule TERMUP [25], ω-regular languages are used to represent the program P and the proof Π. It is therefore natural to ask if $\mathsf{Red}(P)$ in Rule TERMREDUC can be a family of ω-regular program reductions. For *finite* program reductions [16–19], and also for classic POR, lexicographical normal forms are almost always the choice. *Infinite traces* have lexicographic

normal forms that are analogous to their finite counterparts [13]. However, these normal forms are not suitable for defining $\mathsf{Red}(P)$. For example, if $(a, b) \in I$, then the lexicographic normal form of the trace $[(ab)^\omega]_I^\infty$ is $a^\omega b^\omega$ if $a < b$ or $b^\omega a^\omega$ otherwise; both transfinite words. Fortunately, Foata normal forms do not share the same problem.

Definition 6 (Foata Normal Form of an infinite trace[13]). *Foata Normal Form FNF(t) of an infinite trace t is a sequence of non-empty sets $S_1 S_2 ...$ such that $t = \Pi_{i \leq \omega} S_i$ and for all i:*

$$\forall a, b \in S_i \; a \neq b \implies (a, b) \in I \qquad \text{(no dependencies in } S_i \text{)}$$
$$\forall b \in S_{i+1} \; \exists a \in S_i \; (a, b) \notin I \qquad \text{(} S_i \text{ dependent on } S_{i+1} \text{)}$$

For example, $\mathrm{FNF}([(ab)^\omega]_I^\infty) = (ab)^\omega$ if $(a, b) \in I$. To define a reduction based on FNF, we need a mild assumption about the program language.

Definition 7 (Closedness). *A language $L \subseteq \Sigma^\infty$ is* closed *under the independence relation I iff $[L]_I^\infty \subseteq L$ and is ω-*closed *under I iff $[L]_I^\omega \subseteq L$.*

It is straightforward to see that any concurrent program P (as defined in Sect. 2.1), and any valid dependence relation I, we have that P is ω-closed. This means that for any (infinite) program run τ, any other ω-word τ that is equivalent to τ is also in the language of the program.

The key result that makes Foata normal forms amenable to automation in the automaton-theoretic framework is the following theorem.

Theorem 7. *If $L \subseteq \Sigma^\omega$ is ω-regular and closed, FNF(L) is ω-regular.*

The proof of this theorem provides a construction for the Büchi automaton that recognizes the language $\mathrm{FNF}(L)$; see [31] for more detail. However, this construction is not efficient since, for a program P, of size $\Theta(n)$, the Büchi automaton recognizing $\mathrm{FNF}(P)$ can be as large as $\mathcal{O}(n2^n)$. Finally, Foata reductions are *minimal* in the same exact sense that lexicographical reductions of finite-word languages are minimal:

Theorem 8 *[Theorem 11.2.15 [13]]. If $L \subseteq \Sigma^\omega$ is ω-regular and closed, then for all $\tau \in L$, $\tau' \in$ FNF$(L) \cap [\tau]_I^\omega \implies \tau' = \tau$.*

Our experimental results in Sect. 6 suggest that this complexity is a big bottleneck in practical benchmarks. Therefore, despite the fact that Foata normal forms put forward an algorithmic solution for the implementation of Rule TermReducTermReduc, the inefficiency of the solution makes it unsuitable for practical termination checkers.

6 Experimental Results

The techniques presented in this paper have been implemented in a prototype tool called TerMute written in Python and C++. The inputs are concurrent integer programs written in a C-like language. TerMute may output “Terminating”, or “Unknown”, in the latter case also returning a lasso whose termination could not be proved. Ranking functions and invariants are produced using the method described in [24], which is restricted to linear ranking functions of linear lassos. Interpolants are generated using SMTInterpol [6] and MathSAT [7]; the validity of Hoare triples are checked using CVC4 [2].

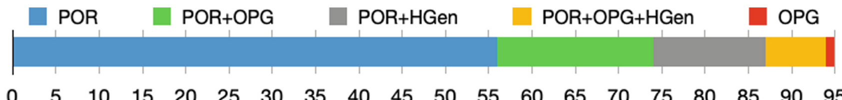
TerMute may be run in several different modes. FOATA is an implementation of the algorithm described in Sect. 5. The baseline is the core counterexample-guided refinement algorithm of [25], which has been adapted to the finite-word context in order to operate on the automata $\mathsf{fast\$}(P)$ and $\mathsf{fast\$}(\Pi)$ of Sect. 4.1. All other modes are modifications of this baseline, maintaining the same refinement scheme, so that we can isolate the impact of adding commutativity reasoning. Hoare triple generalization (“HGen”) augments the baseline by making solver calls after each refinement round in order to determine if edges may soundly be added to the proof for any valid Hoare triples not produced as part of the original proof. “POR” implements the persistent set technique of Sect. 4.2 and “OPG” is the finite-word analogue of the ω-prefix generalization in Sect. 3.2. TerMute can also be run on any combinations of these techniques. In what follows, we use TerMute to refer to the portfolio winner among all algorithms that employ *commutativity reasoning*, namely POR, OPG, POR + HGen, POR + OPG, and POR + OPG + HGen.

See [31] for more detail regarding our experimental setup and results.

Benchmarks. Our benchmarks include 114 terminating concurrent linear integer programs that range from 2 to 12 threads and cover a variety of patterns commonly used for synchronization, including the use of locks, barriers, and monitors. Some are drawn from the literature on termination verification of concurrent programs, specifically [29,34,37], and the rest were created by us, some of which are based on sequential benchmarks from The Termination Problem Database [38], modified to be multi-threaded. We include programs whose threads display a wide range of independence—from complete independence (e.g. the producer threads in Fig. 1), all the way to complete dependence—and demonstrate a range of complexity with respect to control flow.

Results. Our experiments have a timeout of 300 s and a memory cap of 32 GB, and were run on a 12th Gen Intel Core i7-12700K with 64 GB of RAM running Ubuntu 22.04. We experimented with both interpolating solvers and the reported times correspond to the winner of the two. The results are depicted in Fig. 6(a) as a *quantile* plot that compares the algorithms. The total number of benchmarks solved is noted on each curve. FOATA times out on all but the simplest benchmarks, and therefore is omitted from the plot.

The portfolio winner, TerMute, solves 101 benchmarks in total. It solves any benchmark otherwise solved by algorithms without commutativity reasoning (namely, the baseline or HGen). It is also faster on 95 out of 101 benchmarks it solves. The figure below illustrates how often each of the portfolio algorithms emerges as the fastest among these 95 benchmarks.



HGen aggressively generalizes the proof and consequently, it forces convergence in many fewer refinement rounds. This, however, comes at the cost of a time overhead per round. Therefore, HGen helps in solving more benchmarks, but whenever a benchmarks is solvable without it, it is solved much faster. The scatter plot in Fig. 6(b) illustrates this phenomenon when HGen is added to POR+OPG. The plot compares the times of benchmarks solved by both algorithms on a logarithmic scale, and the overhead caused by HGen is significant in the majority of the cases.

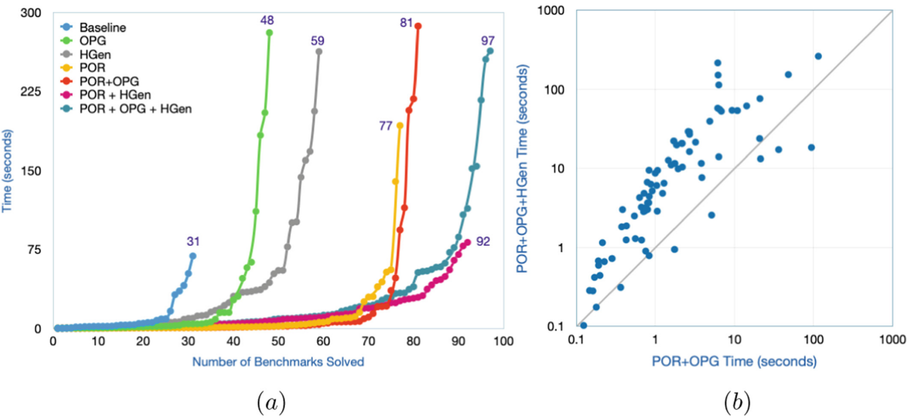


Fig. 6. Experimental results for TerMute: (a) quantile plot for the throughput of each algorithm, and (b) scatter plot for the impact of thread order on efficiency.

Recall, from Sect. 4, that the persistent set algorithm is parametrized on an order over the participating threads. The choice of order centrally affects the way the persistent set algorithm works, by influencing which transitions may be explored and, by extension, which words appear in the reduced program. Experimentally, we have observed that the chosen order plays a significant role in how well the algorithms work, but to varying degrees. For instance, for POR, the worst thread order times out on 16% of the benchmarks that the best order solves. For POR+OPG+HGen, the difference is more modest at 7%. In practice, it is sensible then to instantiate a few instances of the TerMute with a few different random orders to increase the chances of getting better performance.

7 Related Work

The contribution of this paper builds upon sequential program termination provers to produce termination proofs for concurrent programs. As such, any progress in the state of the art in sequential program termination can be used to produce proofs for more lassos, and is, therefore, complementary to our approach. So, we only position this paper in the context of algorithmic concurrent program termination, and the use of commutativity for verification in general, and skip the rich literature on sequential program termination [11,36] or model checking liveness [8,9,26,33].

Concurrent Program Termination. The thread-modular approach to proving termination of concurrent programs [10,34,35,37] aims to prove a thread's termination without reasoning directly about its interactions with other threads, but rather by inferring facts about the thread's environment. In [37], this approach is combined with compositional reasoning about termination arguments. Our technique can also be viewed as modular in the sense that lassos – which, like isolated threads, are effectively sequential programs – are dealt with independently of the broader program in which they appear; however, this is distinct from thread-modularity insofar as we reason directly about behaviours arising from the interaction of threads. Whenever a thread-modular termination proof can be automatically generated for the program, that proof is the most efficient in terms of scalability with the number of threads. However, for a thread-modular proof to always exist, local thread states have to be exposed as auxiliary information. The modularity in our technique does not rely on this information at all. Commutativity can be viewed as a way of observing and taking advantage of some degree of non-interference, different from that of thread modularity.

Causal dependence [29] presents an abstraction refinement scheme for proving concurrent programs terminating that takes advantage of the equivalence between certain classes of program runs. These classes of runs are determined by partial orders that capture the causal dependencies between transitions, in a manner reminiscent of the commutativity-based partial orders of Mazurkiewicz traces. The key to scalability of this method is that they forgo a containment check in the style of module (d) of Fig. 2. Instead, they cover the space of program behaviour by splitting it into cases. Therefore, for the producer-only instance of the example in Sect. 1, this method can scale to many many thread easily, while our commutativity-based technique cannot. Similar to thread-modular approach, this technique cannot be beaten in scalability for the programs that can be split into linearly many cases. However, there is no guarantee (none given in [29]), that a bounded complete causal trace tableau for a terminating program must exist; for example, when there is a dependency between loops in different threads that would cause the program to produce unboundedly many (Mazurkiewicz) traces that have to be analyzed for termination. The advantage of our method is that, once consumers are added to the example in Sect. 1, we can still take advantage of all the existing commutativity to gain more efficiency.

Similar to safety verification, context bounding [3] has been used as a way of under-approximating concurrent programs for termination analysis as well.

Commutativity in Verification. Program reductions have been used as a means of simplifying proofs of concurrent and distributed programs before. Lipton's movers [32] have been used to simplify programs for verification. CIVL [27,28] uses a combination of abstraction and reduction to produce *layered programs*; in an interactive setup, the programmer can prove that an implementation satisfies a specification by moving through these layered programs to increasingly more abstract programs. In the context of message-passing distributed systems [12,21], commutativity is used to produce a synchronous (rather than sequential) program with a simpler proof of correctness.

In [16–19] program reductions are used in a refinement loop in the same style as this paper to prove safety properties of concurrent programs. In [18,19], an unbounded class of lexicographical reductions are enumerated with the purpose of finding a simple proof for at least one of the reductions; the thesis being that there can be a significant variation in the simplicity of the proof for two different reductions. In [19], the idea of contextual commutativity—i.e. considering two statements commutative in some context yet not all contexts—is introduced and algorithmically implemented. In [16,17], only one reduction at a time is explored, in the same style as this paper. In [16], a persistent-set-based algorithm is used to produce space-efficient reductions. In [17] the idea of abstract commutativity is explored. It is shown that no *best* abstraction exists that provides a maximal amount of commutativity and, therefore, the paper proposes a way to combine the benefits of different commutativity relations in one verification algorithm. The algorithm in this paper can theoretically take advantage of all of these (orthogonal) findings to further increase the impact of commutativity in proving termination.

Non-termination. The problem of detecting non-termination has also been directly studied [1,5,20,23,30]. Presently, our technique does not accommodate proving the non-termination of a program. However, it is relatively straightforward to adapt any such technique (or directly use one of these tools) to accommodate this; in particular, when we fail to find a termination proof for a particular lasso, sequential methods for proving non-termination may be employed to determine if the lasso is truly a non-termination witness. However, it is important to note that a program may be non-terminating while all its lassos are terminating, and the refinement loop in Fig. 2 may just diverge without producing a counterexample in this style; this is a fundamental weakness of using lassos as modules to prove termination of programs.

8 Conclusion

In the literature on the usage of commutativity in safety verification, sound program reductions are constructed by selecting lexicographical normal forms of equivalence classes of concurrent program runs. These are not directly applicable in the construction of sound program reductions for termination checking,

since the lexicographical normal forms of infinite traces may not be ω-words. In this paper, we take this apparent shortcoming and turn it into an effective solution. First, these transfinite words are used in the design of the *omega prefix proof rule* (Theorem 2). They also inform the design of the *termination aware* persistent set algorithm described in Sect. 4.2. Overall, this paper contributes mechanisms for using commutativity-based reasoning in termination checking, and demonstrates that, using these mechanisms, one can efficiently check the termination of concurrent programs.

References

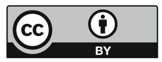
1. Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multithreaded programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 210–226. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_19
2. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Baumann, P., Majumdar, R., Thinniyam, R.S., Zetzsche, G.: Context-bounded verification of liveness properties for multithreaded shared-memory programs. Proc. ACM Program. Lang. **5**(POPL), 1–31 (2021)
4. Calbrix, H., Nivat, M., Podelski, A.: Ultimately periodic words of rational ?-languages. In: Brookes, S., Main, M., Melton, A., Mislove, M., Schmidt, D. (eds.) MFPS 1993. LNCS, vol. 802, pp. 554–566. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58027-1_27
5. Chatterjee, K., Goharshady, E.K., Novotnỳ, P., Žikelić, Đ.: Proving non-termination by program reversal. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 1033–1048 (2021)
6. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
8. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_15
9. Cook, B., Koskinen, E., Vardi, M.: Temporal property verification as a program analysis task. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 333–348. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_26
10. Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 320–330 (2007)
11. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM **54**(5), 88–98 (2011)

12. Desai, A., Garg, P., Madhusudan, P.: Natural proofs for asynchronous programs using almost-synchronous reductions. SIGPLAN Not. **49**(10), 709–725 (2014). https://doi.org/10.1145/2714064.2660211
13. Diekert, V., Rozenberg, G.: The Book of Traces. World scientific (1995)
14. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, 21–23 January 2009, pp. 2–15. ACM (2009)
15. Farzan, A., Kincaid, Z., Podelski, A.: Proving liveness of parameterized programs. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, pp. 185–196. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2933575.2935310
16. Farzan, A., Klumpp, D., Podelski, A.: Sound sequentialization for concurrent program verification. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 506–521 (2022)
17. Farzan, A., Klumpp, D., Podelski, A.: Stratified commutativity in verification algorithms for concurrent programs. Proc. ACM Program. Lang. **7**(POPL), 1426–1453 (2023)
18. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 200–218. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_11
19. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proc. ACM Program. Lang. **4**(POPL), 1–28 (2019)
20. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. arXiv preprint arXiv:1905.11187 (2019)
21. Gleissenthall, K.V., Kıcı, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: Synchronous verification of asynchronous distributed programs. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290372
22. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60761-7_31
23. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 147–158 (2008)
24. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 365–380. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_26
25. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_53
26. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014, Vienna, Austria, 14–18 July 2014, pp. 59:1–59:10. ACM (2014)
27. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, 15–20 June 2020, pp. 227–242. ACM (2020)

28. Kragl, B., Qadeer, S.: Layered concurrent programs. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 79–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_5
29. Kupriyanov, A., Finkbeiner, B.: Causal termination of multi-threaded programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 814–830. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_54
30. Le, T.C., Antonopoulos, T., Fathololumi, P., Koskinen, E., Nguyen, T.: Dynamite: dynamic termination and non-termination proofs. Proc. ACM Program. Lang. **4**(OOPSLA), 189:1–189:30 (2020)
31. Lette, D., Farzan, A.: Commutativity for concurrent program termination proofs (extended version). https://www.cs.toronto.edu/~azadeh/resources/papers/cav23-extended.pdf
32. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
33. Liu, Y.C., et al.: Proving LTL properties of bitvector programs and decompiled binaries. In: Oh, H. (ed.) APLAS 2021. LNCS, vol. 13008, pp. 285–304. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-89051-3_16
34. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 218–232. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_14
35. Pani, T., Weissenbacher, G., Zuleger, F.: Rely-guarantee bound analysis of parameterized concurrent shared-memory programs: with an application to proving that non-blocking algorithms are bounded lock-free. Formal Methods Syst. Des. **57**(2), 270–302 (2021)
36. Podelski, A., Rybalchenko, A.: Transition invariants. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), Turku, Finland, 14–17 July 2004, Proceedings, pp. 32–41. IEEE Computer Society (2004)
37. Popeea, C., Rybalchenko, A.: Compositional termination proofs for multi-threaded programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 237–251. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_17
38. The termination problem database (2023). https://github.com/TermCOMP/TPDB

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Fast Termination and Workflow Nets

Piotr Hofman[1(✉)], Filip Mazowiecki[1], and Philip Offtermatt[1,2]

[1] University of Warsaw, Warsaw, Poland
piotr.hofman@uw.edu.pl, f.mazowiecki@mimuw.edu.pl
[2] Université de Sherbrooke, Sherbrooke, Canada
Philip.Offtermatt@usherbrooke.ca

Abstract. Petri nets are an established model of concurrency. A Petri net is terminating if for every initial marking there is a uniform bound on the length of all possible runs. Recent work on the termination of Petri nets suggests that, in general, practical models should terminate fast, *i.e.* in polynomial time. In this paper we focus on the termination of workflow nets, an established variant of Petri nets used for modelling business processes. We partially confirm the intuition on fast termination by showing a dichotomy: workflow nets are either non-terminating or they terminate in linear time.

The central problem for workflow nets is to verify a correctness notion called soundness. In this paper we are interested in generalised soundness which, unlike other variants of soundness, preserves desirable properties like composition. We prove that verifying generalised soundness is coNP-complete for terminating workflow nets.

In general the problem is PSPACE-complete, thus intractable. We utilize insights from the coNP upper bound to implement a procedure for generalised soundness using MILP solvers. Our novel approach is a semi-procedure in general, but is complete on the rich class of terminating workflow nets, which contains around 90% of benchmarks in a widely-used benchmark suite. The previous state-of-the-art approach for the problem is a different semi-procedure which is complete on the incomparable class of so-called free-choice workflow nets, thus our implementation improves on and complements the state-of-the-art.

Lastly, we analyse a variant of termination time that allows parallelism. This is a natural extension, as workflow nets are a concurrent model by design, but the prior termination time analysis assumes sequential behavior of the workflow net. The sequential and parallel termination times can be seen as upper and lower bounds on the time a process represented as a workflow net needs to be executed. In our experimental section we show that on some benchmarks the two bounds differ significantly, which agrees with the intuition that parallelism is inherent to workflow nets.

Keywords: Workflow · Soundness · Fast termination · generalised Soundness · Polynomial time

This work was partially supported by ERC grant INFSYS: 501-D110-60-0196287. P. Offtermatt is now at Informal Systems, Munich, Germany.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 132–155, 2023.
https://doi.org/10.1007/978-3-031-37706-8_7

1 Introduction

Petri nets are a popular formalism to model problem in software verification [22], business processes [1] and many more (see [42] for a survey). One of the fundamental problems for such models is the *termination problem*, *i.e.* whether the lengths of all runs are universally bounded. There are two natural variants of this problem. First, if the initial configuration is fixed then the problem is effectively equivalent to the boundedness problem, known to be EXPSPACE-complete for Petri nets [36,41]. Second, if termination must hold for all initial configurations the problem known to be in polynomial time [30], and such nets are known as *structurally terminating*. In this paper we are interested in the latter variant.

Termination time is usually studied for *vector addition system with states* (VASS), an extension of Petri nets that allows the use of control states. In particular, the aforementioned EXPSPACE and polynomial time bounds work for VASS. In 2018, a deeper study of the termination problem for VASS was initiated [12]. This study concerns the asymptotics of the function $f(n)$ bounding the length of runs, where n bounds the size of the initial configuration. The focus is particularly on classes where $f(n)$ is a polynomial function, suggesting that such classes are more relevant for practical applications. This line of work was later continued for variants of VASS involving probabilities [11] and games [31].

For VASS the function $f(n)$ can asymptotically be as big as $F_i(n)$ in the Grzegorczyk hierarchy for any finite i (recall that $F_3(n)$ is nonelementary and $F_\omega(n)$ is Ackermann) [35,43]. It was known that for terminating Petri nets many problems are considerably simpler [40]. However, to the best of our knowledge, the asymptotic behaviour of $f(n)$ was not studied for Petri nets.

Our Contributions. In this paper we focus on *workflow nets*, a class of Petri nets widely used to model business processes [1]. Our first result is the following dichotomy: any workflow net is either non-terminating or $f(n)$ is linear. This confirms the intuition about fast termination of practical models [12]. In our proof, we follow the intuition of applying linear algebra from [40] and rely on recent results on workflow nets [9]. We further show that the optimal constant $a_{\mathcal{N}}$ such that $f(n) = a_{\mathcal{N}} \cdot n$ can be computed in polynomial time. The core of this computation relies on a reduction to continuous Petri nets [19], a well known relaxation of Petri nets. Then we can apply standard tools from the theory of continuous Petri nets, where many problems are in polynomial time [7,19].

For workflow nets, the central decision problems are related to soundness. There are many variants of this problem (see [2] for a survey). For example *k-soundness* intuitively verifies that k started processes eventually properly terminate. We are interested in *generalised soundness*, which verifies whether k-soundness holds for all k [25–27]. The exact complexity of most popular soundness problems was established only recently in 2022 [9]. Generalised soundness is surprisingly PSPACE-complete. Other variants, like k-soundness, are EXPSPACE-complete, thus computationally harder, despite having a seemingly less complex definition. Moreover, unlike k-soundness and other variants, generalised soundness preserves desirable properties like composition [26].

Building on our first result, *i.e.* the dichotomy between non-terminating and linearly terminating workflow nets, our second result is that generalised soundness is coNP-complete for terminating workflow nets.

Finally, we observe that the asymptotics of $f(n)$ are defined with the implicit assumption that transitions are fired sequentially. Since workflow nets are models for parallel executions it is natural to expect that runs would also be performed in parallel. Our definition of parallel executions is inspired by similar concepts for time Petri nets, and can be seen as a particular case [5]. We propose a definition of the optimal running time of runs exploiting parallelism and denote this time $g(n)$, where n bounds the size of the initial marking. We show that the asymptotic behaviour of $g(n)$, similar to $f(n)$, can be computed in polynomial time, for workflow nets with mild assumptions. Together, the two functions $f(n)$ and $g(n)$ can be seen as (pessimistic) upper bound and (optimistic) lower bound on the time needed for the workflow net to terminate.

Experiments. Based on our insights, we implement several procedures for problems related to termination in workflow nets. Namely, we implement our algorithms for checking termination, for deciding generalised soundness of workflow nets and for computing the asymptotic behaviour of $f(n)$. We additionally implement procedures to compute $f(k), g(k)$ and decide k-soundness for terminating workflow nets. To demonstrate the efficacy of our procedures, we test our implementation on a popular and well-studied benchmark suite of 1382 workflow nets, originally introduced in [18]. It turns out that the vast majority of instances (roughly 90%) is terminating, thus the class of terminating workflow nets seems highly relevant in practice. Further, we positively evaluate our algorithm for generalised soundness against a recently proposed state-of-art approach [10] which semi-decides the property in general, and is further exact on the class of *free-choice workflow nets* [3]. Interestingly, our novel approach for generalised soundness is also a semi-procedure in general, but precise on terminating workflow nets. The approach from [10] is implemented as an $\exists\forall$-formula from $\mathrm{FO}(\mathbb{Q}, <, +)$, while our approach manages to avoid any quantifier alternations. It turns out that our approach is faster on over 95% of benchmark instances, and thus significantly improves upon the state-of-art. The mean analysis time for our approach is just 12.8 ms, while it is about 2 s for the previous state-of-the-art. In addition, the classes of free-choice and terminating workflow nets are incomparable, thus our approach complements the state-of-the-art.

Related Work. For general Petri nets and VASS the most well-known problem is reachability, recently shown to be Ackermann-complete [14,33,34]. Despite its high complexity, there are tools for the problem [16,45], including some based on integer and continuous relaxations [6,8,21]. Reachability was also studied in the context of terminating models. In particular, it is PSPACE-complete for structurally terminating Petri nets [40] and EXPSPACE-complete for polynomially terminating VASS [32].

Most algorithms for soundness are based on reductions to reachability [1], this is the case for the first algorithms for generalised soundness [25,27]. However, such reductions only imply Ackermannian upper bounds on the problem, while a direct study yielded elementary complexities [9].

A different class of approaches for soundness relies on *reduction rules*, which can be applied iteratively to reduce the size of a net while exactly preserving soundness [4,39]. These approaches are not precise in general, but can be for subclasses, *e.g.* for *live and bounded* free-choice workflow nets [15]. We use a certain set of reduction rules [13] for generalised soundness in our experimental evaluation.

There exist many established tools and frameworks for the analysis of workflow nets, for example Woflan [44], WoPeD [20], and ProM [17]. However, when it comes to soundness problems, these tools typically focus on k-soundness, with a particular focus on $k = 1$ (except for the discussed tool in [10]).

Organisation. In Sect. 2 we define the models, problems and basic notation. In Sect. 3 we prove the dichotomy between non-terminating and linear workflow nets. Then, we show how to compute the linear constants for terminating workflow nets in Sect. 4. Building on the dichotomy we show that generalised soudness is coNP-complete in Sect. 5. In Sect. 6 we define and compute a variant of termination time that takes into account parallelism. We present our experimental results in Sect. 7. Most proofs can be found in the appendix.

2 Preliminaries

We write $\mathbb{N}, \mathbb{N}_{>0}, \mathbb{Z}, \mathbb{Q}$ and $\mathbb{Q}_{\geq 0}$ for the naturals (including 0), the naturals without 0, the integers, the rationals, and the nonnegative rationals, respectively.

Let N be a set of numbers, *e.g.* $N = \mathbb{N}$. For $d, d_1, d_2 \in \mathbb{N}_{>0}$ we write N^d for the set of vectors with elements from N in dimension d. Similarly, $N^{d_1 \times d_2}$ is the set of matrices with d_1 rows and d_2 columns and elements from N. We use bold font for vectors and matrices. For $a \in \mathbb{Q}$ and $d \in \mathbb{N}_{>0}$, we write $\boldsymbol{a}^d := (a, a, \ldots, a) \in \mathbb{Q}^d$ (or $\boldsymbol{a}$ if d is clear from context). In particular $\mathbf{0}^d = \mathbf{0}$ is the zero vector.

Sometimes it is more convenient to have vectors with coordinates in a finite set. Thus, for a finite set S, we write $\mathbb{N}^S$, $\mathbb{Z}^S$, and $\mathbb{Q}^S$ for the set of vectors over naturals, integers and rationals. Given a vector $\boldsymbol{v}$ and an element $s \in S$, we write $\boldsymbol{v}(s)$ for the value $\boldsymbol{v}$ assigns to s.

Given $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{Q}^S$, we write $\boldsymbol{v} \leq \boldsymbol{w}$ if $\boldsymbol{v}(s) \leq \boldsymbol{w}(s)$ for all $s \in S$, and $\boldsymbol{v} < \boldsymbol{w}$ if $\boldsymbol{v} \leq \boldsymbol{w}$ and $\boldsymbol{v}(s) < \boldsymbol{w}(s)$ for some $s \in S$. The *size* of S, denoted $|S|$, is the number of elements in S. We define the *norm* of a vector $\|\boldsymbol{v}\| := \max_{s \in S} |\boldsymbol{v}(s)|$, and the norm of a matrix $\boldsymbol{A} \in \mathbb{Q}^{m \times n}$ as $\|\boldsymbol{A}\| := \max_{1 \leq j \leq m, 1 \leq i \leq n} |A(i, j)|$. For a set $S \in \mathbb{Q}^d$, we denote by $\overline{S} \in \mathbb{R}^d$ the closure of S in the euclidean space.

2.1 (Integer) Linear Programs

Let $n, m \in \mathbb{N}_{>0}$, $\boldsymbol{A} \in \mathbb{Z}^{m \times n}$, and $\boldsymbol{b} \in \mathbb{Z}^m$. We say that $G := \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ is a *system of linear inequalities* with m inequalities and n variables. The *norm* of a system G is defined as $\|G\| := \|\boldsymbol{A}\| + \|\boldsymbol{b}\| + m + n$. An $(m \times n)$-*ILP*, short for *integer linear program*, is a system of linear inequalities with m inequalities and n variables, where we are interested in the integer solutions. An $(m \times n)$-*LP* is such a system where we are interested in the rational solutions. We use the term MILP, short for *mixed integer linear program*, for a system where some variables are allowed to take on rational values, while others are restricted to integer values.

We allow syntactic sugar in ILPs and LPs, such as allowing constraints $x \geq y$, $x = y$, $x < y$ (in the case of ILPs). Sometimes we are interested in finding optimal solutions. This means we have a objective function, formally a linear function on the variables of the system, and look for a solution that either maximizes or minimizes the value of that function. For LPs, finding an optimal solution can be done in polynomial time, while this is NP-complete for ILPs and MILPs.

2.2 Petri Nets

A *Petri net* $\mathcal{N}$ is a triple (P, T, F), where P is a finite set of *places*; T is a finite set of *transitions* such that $T \cap P = \emptyset$; and $F\colon ((P \times T) \cup (T \times P)) \to \mathbb{N}$ is a function describing its *arcs*. A *marking* is a vector $\boldsymbol{m} \in \mathbb{N}^P$. We say that $\boldsymbol{m}(p)$ is the number of *tokens* in place $p \in P$ and p is *marked* if $\boldsymbol{m}(p) > 0$. To write markings, we list only non-zero token amounts. For example, $\boldsymbol{m} = \{p_1\colon 2, p_2\colon 1\}$ is the marking $\boldsymbol{m}$ with $\boldsymbol{m}(p_1) = 2, \boldsymbol{m}(p_2) = 1$ and $\boldsymbol{m}(p) = 0$ for all $p \in P \setminus \{p_1, p_2\}$.

Let $t \in T$. We define the vector ${}^\bullet t \in \mathbb{N}^P$ by ${}^\bullet t(p) := F(p, t)$ for $p \in P$. Similarly, the vector $t^\bullet \in \mathbb{N}^P$ is defined by $t^\bullet(p) := F(t, p)$ for $p \in P$. We write the *effect* of t as $\Delta(t) := t^\bullet - {}^\bullet t$. A transition t is *enabled* in a marking $\boldsymbol{m}$ if $\boldsymbol{m} \geq {}^\bullet t$. If t is enabled in the marking $\boldsymbol{m}$, we can *fire* it, which leads to the marking $\boldsymbol{m}' := \boldsymbol{m} + \Delta(t)$, which we denote $\boldsymbol{m} \to^t \boldsymbol{m}'$. We write $\boldsymbol{m} \to \boldsymbol{m}'$ if there exists some $t \in T$ such that $\boldsymbol{m} \to^t \boldsymbol{m}'$.

A sequence of transitions $\pi = t_1 t_2 \ldots t_n$ is called a *run*. We denote the *length* of π as $|\pi| := n$. A run π is *enabled* in a marking $\boldsymbol{m}$ iff $\boldsymbol{m} \to^{t_1} \boldsymbol{m}_1 \to^{t_2} \boldsymbol{m}_2 \to^{t_3} \ldots \boldsymbol{m}_{n-1} \to^{t_n} \boldsymbol{m}'$ for some markings $\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}' \in \mathbb{N}^P$. The set of all runs is denoted $\mathrm{Runs}_{\mathcal{N}}^{\boldsymbol{m}}$, *i.e.* $\pi \in \mathrm{Runs}_{\mathcal{N}}^{\boldsymbol{m}}$ if π is enabled in $\boldsymbol{m}$. The *effect* of π is $\Delta(\pi) := \sum_{i \in [1..n]} \Delta(t_i)$. *Firing* π from $\boldsymbol{m}$ leads to a marking $\boldsymbol{m}'$, denoted $\boldsymbol{m} \to^\pi \boldsymbol{m}'$, iff $\boldsymbol{m} \in \mathrm{Runs}_{\mathcal{N}}^{\boldsymbol{m}}$ and $\boldsymbol{m}' = \boldsymbol{m} + \Delta(\pi)$. We denote by $\to^*$ the reflexive, transitive closure of $\to$. Given two runs $\pi = t_1 t_2 \ldots t_n$ and $\pi' = t_1' t_2' \ldots t_{n'}'$ we denote $\pi\pi' := t_1 t_2 \ldots t_n t_1' t_2' \ldots t_{n'}'$.

The size of a Petri net is defined as $|\mathcal{N}| = |P| + |T| + |F|$. We define the *norm* of $\mathcal{N}$ as $\|\mathcal{N}\| := \|F\| + 1$, where we view F as a vector in $\mathbb{N}^{(P \times T) \cup (T \times P)}$.

We also consider several variants of the firing semantics of Petri nets which we will need throughout the paper. In the *integer semantics*, we consider markings over $\mathbb{Z}^P$, and transitions can be fired without being enabled. To denote the firing and reachability relations, we use the notations $\to_{\mathbb{Z}}$ and $\to_{\mathbb{Z}}^*$. In the *continuous semantics* [19], we consider markings over $\mathbb{Q}_{\geq 0}^P$. Given $t \in T$ and a *scaling factor*

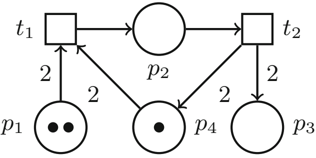


Fig. 1. A Petri net with places p_1, p_2, p_3, p_4 and transitions t_1, t_2. Marking $\{p_1 \colon 2, p_4 \colon 1\}$ is drawn. No transition is enabled.

$\beta \in \mathbb{Q}_{\geq 0}$[1], the effect of firing βt is $\Delta(\beta t) \coloneqq \beta \cdot \Delta(t)$. Further, βt is enabled in a marking $\boldsymbol{m}$ iff $\beta \cdot {}^\bullet t \leq \boldsymbol{m}$. We use $\rightarrow_{\mathbb{Q}_{\geq 0}}$ for the continuous semantics, that is, $\boldsymbol{m} \rightarrow_{\mathbb{Q}_{\geq 0}}^{\beta t} \boldsymbol{m}'$ means βt is enabled in $\boldsymbol{m}$ and $\boldsymbol{m}' = \boldsymbol{m} + \Delta(\beta t)$. A *continuous run* π is a sequence of factors and transitions $\beta_1 t_1 \beta_2 t_2 \dots \beta_n t_n$. Enabledness and firing are extended to continuous runs: $\boldsymbol{m} \rightarrow_{\mathbb{Q}_{\geq 0}}^{\pi} \boldsymbol{m}'$ holds iff there exist $\boldsymbol{m}_1, \dots, \boldsymbol{m}_{n-1}$ such that $\boldsymbol{m} \rightarrow_{\mathbb{Q}_{\geq 0}}^{\beta_1 t_1} \boldsymbol{m}_1 \rightarrow_{\mathbb{Q}_{\geq 0}}^{\beta_2 t_2} \cdots \boldsymbol{m}_{n-1} \rightarrow_{\mathbb{Q}_{\geq 0}}^{\beta_n t_n} \boldsymbol{m}'$. The *length* of π is $|\pi|_c \coloneqq \sum_{i=1}^{n} \beta_i$. Given $\alpha \in \mathbb{Q}_{\geq 0}$ and a run $\pi = \beta_1 t_1 \beta_2 t_2 \dots \beta_n t_n$ we write $\alpha\pi$ to denote the run $(\alpha\beta_1)t_1(\alpha\beta_2)t_2 \dots (\alpha\beta_n)t_n$. We introduce a lemma stating that continuous runs can be rescaled.

Lemma 1 (Lemma 12(1) in [19]**).** *Let* $\alpha \in \mathbb{Q}_{\geq 0}$. *Then* $\boldsymbol{m} \rightarrow_{\mathbb{Q}_{\geq 0}}^{\pi} \boldsymbol{m}'$ *if and only if* $\alpha\boldsymbol{m} \rightarrow_{\mathbb{Q}_{\geq 0}}^{\alpha\pi} \alpha\boldsymbol{m}'$.

Each run under normal semantics or integer semantics is *equivalent* to a continuous run i.e. $t_1 t_2 \dots t_n \approx 1t_1 1t_2 \dots 1t_2$. Given $\pi \in \text{Runs}_{\mathcal{N}}^{\boldsymbol{m}}$ (*i.e.* a standard run) we define $\alpha\pi = \alpha\pi_c$ where $\pi_c \approx \pi$ is a continuous run. If $\pi_c = \beta_1 t_1 \dots \beta_n t_n$ with $\beta_i \in \mathbb{N}$ for all $i \in \{1, \dots, n\}$ then we also call π a (standard) run, *i.e.* the run where every transition t_i is repeated β_i times.

We define the set of continuous runs enabled from $\boldsymbol{m} \in \mathbb{N}^P$ in $\mathcal{N}$ as $\text{CRuns}_{\mathcal{N}}^{\boldsymbol{m}}$. The *Parikh image* of a continuous run $\pi = \beta_1 t_1 \beta_2 t_2 \dots \beta_n t_n$ is the vector $\boldsymbol{R}_\pi \in \mathbb{Q}^T$ such that $\boldsymbol{R}_\pi(t) = \sum_{i | t_i = t} \beta_i$. For a (standard) run π we define its Parikh image $\boldsymbol{R}_\pi \coloneqq \boldsymbol{R}_{\pi_c}$ where $\pi_c \approx \pi$. Given a vector $\boldsymbol{R} \in \mathbb{Q}_{\geq 0}^T$, we define $\Delta(\boldsymbol{R}) \coloneqq \sum_{t \in T} \boldsymbol{R}(t) \cdot \Delta(t)$, ${}^\bullet\boldsymbol{R} \coloneqq \sum_{t \in T} {}^\bullet t \cdot \boldsymbol{R}(t)$, $\boldsymbol{R}^\bullet \coloneqq \sum_{t \in T} t^\bullet \cdot \boldsymbol{R}(t)$. Note that $\boldsymbol{R}$ is essentially a run without imposing an order on the transitions. For ease of notation, we define $\Delta(T)$ as a matrix with columns indexed by T and rows indexed by P, where $\Delta(T)(t)(p) \coloneqq \Delta(t)(p)$. Then $\Delta(\boldsymbol{R}) = \Delta(T)\boldsymbol{R}$.

Example 1. Consider the Petri net drawn in Fig. 1. Marking $\boldsymbol{m} \coloneqq \{p_1 \colon 2, p_4 \colon 1\}$ enables no transitions. However, we have $\boldsymbol{m} \rightarrow_{\mathbb{Z}}^{t_1 t_2} \{p_3 \colon 2\}$. We also have $\boldsymbol{m} \rightarrow_{\mathbb{Z}}^{t_2 t_1} \{p_3 \colon 2\}$, since the transition order does not matter under the integer semantics. Thus, when we take $R = \{t_1 \colon 1, t_2 \colon 1\}$, we have $\boldsymbol{m} \rightarrow_{\mathbb{Z}}^{R} \{p_3 \colon 2\}$.

Under the continuous semantics we can fire $1/2 t_1$, which is impossible under the normal semantics. For example, we have $\boldsymbol{m} \rightarrow_{\mathbb{Q}_{\geq 0}}^{1/2 t_1} \{p_1 \colon 1, p_2 \colon 1/2\} \rightarrow_{\mathbb{Q}_{\geq 0}}^{1/2 t_2} \{p_1 \colon 1, p_3 \colon 1, p_4 \colon 1\} \rightarrow_{\mathbb{Q}_{\geq 0}}^{1/3 t_1} \{p_1 \colon 1/3, p_2 \colon 1/3, p_3 \colon 1, p_4 \colon 2/3\}$.

[1] Sometimes scaling factors are defined to be at most 1. The definitions are equivalent: Scaling larger than 1 can be done by firing the same transition multiple times.

2.3 Workflow Nets

A *workflow net* is a Petri net $\mathcal{N}$ such that:

- There exists an *initial* place i with $F(t, \mathsf{i}) = 0$ for all $t \in T$ (*i.e.* no tokens can be added to i);
- there exists a *final* place f with $F(\mathsf{f}, t) = 0$ for all $t \in T$ (*i.e.* no tokens can be removed from f); and
- in the graph (V, E) with $V = P \cup T$ and $(u, v) \in E$ iff $F(u, v) \neq 0$, each $v \in V$ lies on at least one path from i to f.

We say that $\mathcal{N}$ is *k-sound* iff for all $\boldsymbol{m}$, $\{\mathsf{i}\colon k\} \to^* \boldsymbol{m}$ implies $\boldsymbol{m} \to^* \{\mathsf{f}\colon k\}$. Further, we say $\mathcal{N}$ is *generalised sound* iff it is k-sound for all k.

A place $p \in P$ is *nonredundant* if $\{\mathsf{i}\colon k\} \to^* \boldsymbol{m}$ for some $k \in \mathbb{N}$ and marking $\boldsymbol{m}$ with $\boldsymbol{m}(p) > 0$, and *redundant* otherwise. We accordingly say that $\mathcal{N}$ is *nonredundant* if all $p \in P$ are nonredundant, otherwise $\mathcal{N}$ is *redundant*. A redundant workflow net can be made nonredundant by removing each redundant place $p \in P$ and all transitions such that ${}^\bullet t(p) > 0$ or $t^\bullet(p) > 0$. Note that this does not impact behaviour of the workflow, as the discarded transitions could not be fired in the original net. A polynomial-time saturation procedure can identify redundant places, see [27, Thm. 8, Def. 10, Sect. 3.2] and [9, Prop. 5.2].

If $\mathcal{N}$ is a workflow net, we write $\text{Runs}^k_{\mathcal{N}}$ for the set of runs that are enabled from the marking $\{\mathsf{i}\colon k\}$, and $\text{CRuns}^k_{\mathcal{N}}$ for the same for continuous runs. Lemma 1 implies that if $\pi \in \text{Runs}^k_{\mathcal{N}}$ then $\frac{1}{k}\pi \in \text{CRuns}^1_{\mathcal{N}}$. The converse does not need to hold as the rescaled continuous run need not have natural coefficients.

Example 2. The Petri net in Fig. 1 can be seen as a workflow net with initial place p_1 and final place p_3. The workflow is not k-sound for any k. Further, the net is redundant: $\{\mathsf{i}\colon k\}$ is a deadlock for every k, so places p_2, p_3 and p_4 are redundant. $\lhd$

2.4 Termination Complexity

Let $\mathcal{N}$ be a workflow net. Let us define as $MaxTime_{\mathcal{N}}(k)$ the supremum of lengths among runs enabled in $\{\mathsf{i}\colon k\}$, that is, $MaxTime_{\mathcal{N}}(k) = \sup\{|\pi| \mid \pi \in \text{Runs}^k_{\mathcal{N}}\}$. We say that $\mathcal{N}$ is *terminating* if $MaxTime_{\mathcal{N}}(k) \neq \infty$ for all $k \in \mathbb{N}_{>0}$, otherwise it is *non-terminating*.

We say that $\mathcal{N}$ has *polynomial termination time* if there exist $d \in \mathbb{N}$, $\ell \in \mathbb{R}$ such that for all k,

$$MaxTime_{\mathcal{N}}(k) \leq \ell \cdot k^d. \quad (1)$$

Further $\mathcal{N}$ has *linear termination time* if Eq. (1) holds with $d = 1$. Even more fine-grained, $\mathcal{N}$ has a-linear *termination time* if Eq. (1) holds for $\ell = a$ and $d = 1$. Note that any net with a-linear termination time also has $(a+b)$-linear termination time for all $b \geq 0$. For ease of notation, we call workflow nets that have linear termination time *linear workflow nets*, and similarly for a-linear.

We define $a_{\mathcal{N}} := \inf\{a \in \mathbb{R} \mid \mathcal{N} \text{ is } a\text{ -linear}\}$. Note that in particular $\mathcal{N}$ is $a_{\mathcal{N}}$-linear (because the inequality in Eq. (1) is not strict) and that $a_{\mathcal{N}}$ is the smallest constant with this property.

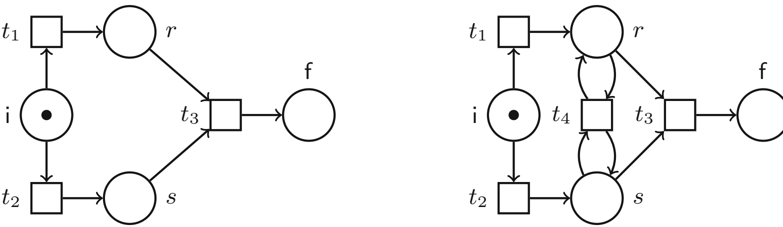


Fig. 2. Two workflow nets with the initial marking $\{\mathsf{i}: 1\}$. The workflow net on the left-hand side is terminating in linear time. The workflow net on the right-hand side is the same as the one on the left, but with one extra transition t_4. It is non-terminating.

Example 3. The net on the left-hand side of Fig. 2 is terminating. For example, from $\{\mathsf{i}: 2\}$ all runs have length at most 3. It is easy to see that from $\{\mathsf{i}: k\}$ all runs have length at most $\frac{3}{2}k$ (*e.g.* the run $(t_1t_2t_3)^{\lfloor \frac{k}{2} \rfloor}$). The net has $a_{\mathcal{N}} = 3/2$.

The net on the right-hand side is non-terminating. From $\{\mathsf{i}: 2\}$, all runs of the form $t_1t_2t_4^*$ are enabled. Note that while the net is non-terminating, all runs from $\{\mathsf{i}: 1\}$ have length at most 1 (because t_3 and t_4 are never enabled). ◁

Our definition of termination time is particular to workflow nets, as there it is natural to have only i marked initially. It differs from the original definition of termination complexity in [12]. In [12] VASS are considered instead of Petri nets, and the initial marking is arbitrary. The termination complexity is measured in the size of the encoding of $\boldsymbol{m}$. The core difference is that in [12] it is possible to have a fixed number of tokens in some places, but arbitrarily many tokens in other places. In Sect. 3 we show an example that highlights the difference between the two notions. Our definition is a more natural fit for workflow nets, and will allow us to reason about soundness. Indeed, our particular definition of termination time allows us to obtain the coNP-completeness result of generalised soundness for linear workflow nets in Sect. 5.

3 A Dichotomy of Termination Time in Workflow Nets

Let us exhibit behaviour in Petri nets that cannot occur in workflow nets. Consider the net drawn in black in Fig. 3 and a family of initial markings $\{\{p_1 : 1, s_1 : 1, b : n\} \mid n \in \mathbb{N}\}$. From the marking $\{p_1 : 1, s_1 : 1, b : n\}$, all runs have finite length, yet a run has length exponential in n. From the marking $\{p_1 : k, s_1 : 1, b : n\}$, the sequence $(t_1t_2)^k t_4 (t_3)^{2k} t_5$ results in the marking $\{p_1 : 2k, s_1 : 1, b : n-1\}$. Thus, following this pattern n times leads from $\{p_1 : 1, s_1 : 1, b : n\}$ to $\{p_1 : 2^n, s_1 : 1\}$. This behaviour crucially requires us to keep a single token in s_1, while having n tokens in b.

We can transform the net into a workflow net, as demonstrated by the colored part of Fig. 3. However, observe that then

$$\{\mathsf{i} : 2\} \rightarrow^{t_\mathsf{i} t_\mathsf{i} t_4} \{p_1 : 2, s_1 : 1, s_2 : 1, b : 1\} \rightarrow^{t_1t_2t_3} \{p_1 : 2, s_1 : 1, s_2 : 1, b : 1, p_3 : 1\}.$$

Note that the sequence $t_1 t_2 t_3$ strictly increased the marking. It can thus be fired arbitrarily many times, and the workflow net is non-terminating.

It turns out that, contrary to standard Petri nets, there exist no workflow nets with exponential termination time.[2] Instead, there is a dichotomy between non-termination and linear termination time.

Theorem 1. *Every workflow net $\mathcal{N}$ is either non-terminating or linear. Moreover, $MaxTime_{\mathcal{N}}(k) \leq ak$ for some $a \leq \|\mathcal{N}\|^{poly(|\mathcal{N}|)}$.*

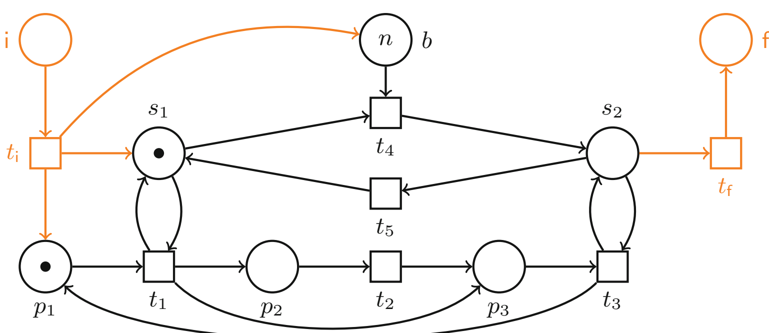


Fig. 3. In black: A Petri net $\mathcal{N}$ adapted from [28, Lemma 2.8]. It enables a run with length exponential in n from marking $\{p_1 : 1, s_1 : 1, b : n\}$. In color: Additional places and transitions, which make $\mathcal{N}$ a workflow net.

As explained in Sect. 2.3 we can assume that $\mathcal{N}$ is nonredundant, *i.e.* for all $p \in P$ there exists $k \in \mathbb{N}$ such that $\{\mathsf{i}\colon k\} \to^* \boldsymbol{m}$ with $\boldsymbol{m}(p) > 0$. The first important ingredient is the following lemma.

Lemma 2. *Let $\mathcal{N} = (P, T, F)$ be a nonredundant workflow net. Then $\mathcal{N}$ is non-terminating iff there exists a nonzero $\boldsymbol{R} \in \mathbb{N}^T$ such that $\Delta(\boldsymbol{R}) \geq \mathbf{0}$.*

Proof (sketch). The first implication follows from the fact that if we start from a big initial marking, then it is possible to fill every place with arbitrarily many tokens. In such a configuration any short run is enabled, so if there is a run with non-negative effect then it is further possible to repeat it infinitely many times. For the other implication we reason as follows. If there is an infinite run then by Dickson's lemma there are $\boldsymbol{m}, \boldsymbol{m}' \in \mathbb{N}^P$ such that for some k, it holds that $\{\mathsf{i}\colon k\} \to^{\pi} \boldsymbol{m} \to^{\rho} \boldsymbol{m}'$ and $\boldsymbol{m}' \geq \boldsymbol{m}$. But then $\Delta(\boldsymbol{R}_\rho) = \boldsymbol{m}' - \boldsymbol{m} \geq \mathbf{0}$. □

We define $\text{ILP}_{\mathcal{N}}$ with a $|T|$ dimensional vector of variables $\boldsymbol{x}$ as the following system of inequalities: $\boldsymbol{x} \geq \mathbf{0}$ and $\Delta(T)\boldsymbol{x} \geq \mathbf{0} - \{\mathsf{i}\colon \infty\}$.[3] The next lemma follows immediately from the definition of $\to_{\mathbb{Z}}$.

[2] This is caused by the choice of the family of initial configurations. Fixing the number of initial tokens in some places can be simulated by control states in the VASS model.

[3] This ∞ is syntactic sugar to omit the inequality for the place i. Formally $\Delta(T)$ and $\boldsymbol{x}$ should be projected to ignore i.

Lemma 3. *[Adapted from Claim 5.7 in [9]] For every $k \in \mathbb{N}$, $\boldsymbol{m} \in \mathbb{N}^P$, and a run π, it holds that $\{\mathsf{i}\colon k\} \rightarrow^{\pi}_{\mathbb{Z}} \boldsymbol{m}$ iff $\boldsymbol{R}_\pi$ is a solution to ILP$_{\mathcal{N}}$ with the additional constraint $\sum_{i=1}^{|T|} \Delta(t_i)(\mathsf{i}) \cdot \boldsymbol{R}_\pi(t_i) \geq -k$.*

Proof (Sketch for Theorem 1). Because of Lemma 3 the Parikh image of every run (in $\bigcup_{k\in\mathbb{N}} \text{Runs}^k_{\mathcal{N}}$) is a solution $\boldsymbol{R} \in \mathbb{N}^T$ of $\Delta(T)\boldsymbol{R} \geq -\{\mathsf{i}\colon \infty\}$. So, we consider a set of solutions of the system of inequalities $\Delta(T)\boldsymbol{R} \geq -\{\mathsf{i}\colon \infty\}$. It is a linear set, so the sum of two solutions is again a solution and any solution can be written as a sum of small solutions with norm smaller than some $c \in \mathbb{N}$. For such small solutions, the length of any corresponding run is at most $|T| \cdot c$. Now observe that if the workflow is terminating then there is no $\boldsymbol{R} \in \mathbb{N}^T$ such that $\Delta(T)\boldsymbol{R} \geq \boldsymbol{0}$, because of Lemma 2. But it holds that $\Delta(\boldsymbol{R})(\mathsf{i}) \leq -1$ for any solution $\boldsymbol{R}$, so in particular for all small solutions. Let us take a run $\pi \in \text{Runs}^k_{\mathcal{N}}$. We decompose $\boldsymbol{R}_\pi$ as a finite sum $\sum_i^{\ell} \boldsymbol{R}_i$ where $\boldsymbol{R}_i$ are from the set of small solutions. We have $-k \leq \Delta(\boldsymbol{R}_i)(\mathsf{i}) = \sum_i^{\ell} \Delta(\boldsymbol{R}_i)(\mathsf{i}) \leq \sum_i^{\ell} -1 = -\ell$. Recall that the norm of small solutions is bounded by c. It follows that the length of the run π is bounded by $\ell \cdot |T| \cdot c \leq k \cdot |T| \cdot c$. So the workflow is $|T| \cdot c$-linear.

4 Refining Termination Time

Recall that $a_{\mathcal{N}}$ is the smallest constant such that $\mathcal{N}$ is $a_{\mathcal{N}}$-linear. In this section, we are interested in computing $a_{\mathcal{N}}$. This number is interesting, as it can give insights into the shape and complexity of the net, *i.e.* a large $a_{\mathcal{N}}$ implies complicated runs firing transitions several times, while a small $a_{\mathcal{N}}$ implies some degree of choice, where not all transitions can be fired for each initial token.

The main goal of this section is to show an algorithm for computing $a_{\mathcal{N}}$. Our algorithm handles the more general class of *aggregates* on workflow nets, and we can compute $a_{\mathcal{N}}$ as such an aggregate. More formally, let $\mathcal{N} = (P, T, F)$ be a workflow net. An aggregate is a linear map $f : \mathbb{Q}^T \to \mathbb{Q}$. The aggregate of a (continuous) run is the aggregate of its Parikh image, that is $f(\pi) \coloneqq f(\boldsymbol{R}_\pi)$.

Example 4. Consider the aggregate $f_{all}(\pi) \coloneqq \sum_{t\in T} \boldsymbol{R}_\pi(t) = |\pi|$, which computes the number of occurrences of all transitions. Let us consider two other natural aggregates. The aggregate $f_t(\pi) \coloneqq \boldsymbol{R}_\pi(t)$ computes the number of occurrences of transition t, and $f_p(\pi) \coloneqq \sum_{t\in T} \Delta(t)(p) \cdot \boldsymbol{R}_\pi(t)$ computes the number of tokens added to place p. Another use for aggregates is counting transition, but with different weights for each transition, thus simulating *e.g.* different costs. $\triangleleft$

Given a workflow net $\mathcal{N}$ and an aggregate f we define

$$\sup_{f,\mathcal{N}} = \sup\left\{\frac{f(\pi)}{k} \mid k \in \mathbb{N}_{>0},\ \pi \in \text{Runs}^k_{\mathcal{N}}\right\}. \tag{2}$$

Let us justify the importance of this notion by relating it to $a_{\mathcal{N}}$.

Proposition 1. *Let $\mathcal{N}$ be a linear workflow net. Then $a_{\mathcal{N}} = sup_{f_{all},\mathcal{N}}$.*

Proof. Recall that $a_{\mathcal{N}}$ is the smallest number a such that $|\pi| \leq a \cdot k$ for all $k \in \mathbb{N}_{>0}$ and $\pi \in \text{Runs}^k_{\mathcal{N}}$. Equivalently, $\frac{|f_{all}(\pi)|}{k} \leq a$. Thus by definition $\sup_{f_{all},\mathcal{N}} \leq a_{\mathcal{N}}$, and the inequality cannot be strict since $a_{\mathcal{N}}$ is the smallest number with this property. □

Theorem 2. *Consider a workflow net $\mathcal{N}$ and an aggregate f. The value $sup_{f,\mathcal{N}}$ can be computed in polynomial time.*

Corollary 1. *Let $\mathcal{N} = (P, T, F)$ be a linear workflow net. The constant $a_{\mathcal{N}}$ can be computed in polynomial time.*

In practice, we can use an LP solver to compute the constant $a_{\mathcal{N}}$. The algorithm is based on the fact that continuous reachability for Petri nets is in polynomial time [7,19]. We formulate a lemma that relates the values of aggregates under the continuous and standard semantics.

Lemma 4. *Let $\mathcal{N}$ be a Petri net and f be an aggregate.*

1. *Let $\pi \in Runs^k_{\mathcal{N}}$. Then $1/k \cdot \pi \in CRuns^1_{\mathcal{N}}$ and $f(1/k \cdot \pi) = f(\pi)/k$.*
2. *Let $\pi_c \in CRuns^1_{\mathcal{N}}$. There are $k \in \mathbb{N}$ and $\pi \in Runs^k_{\mathcal{N}}$ with $f(\pi_c) = f(\pi)/k$.*

Proof. Both items are simple consequences of Lemma 1 and the linearity of aggregates. Note that for (2), if $\pi_c = \beta_1 t_1 \dots \beta_n t_n$ then it suffices to define k such that $\beta_i \cdot k \in \mathbb{N}$ for all $i \in \{1, \dots, n\}$. □

From the above lemma we immediately conclude the following.

Corollary 2. *It holds that $sup_{f,\mathcal{N}} = sup\{f(\pi_c) \mid \pi_c \in CRuns^1_{\mathcal{N}}\}$.*

Proof (The proof of Theorem 2). We use Corollary 2 and conclude that we have to compute $sup\{f(\pi_c) \mid \pi_c \in \text{CRuns}^1_{\mathcal{N}}\}$. Let $S = \{\boldsymbol{R}_{\pi_c} \mid \pi_c \in \text{CRuns}^1_{\mathcal{N}}\}$. As $f(\pi)$ is defined as $f(\boldsymbol{R}_\pi)$, we reformulate our problem to compute $sup\{f(\boldsymbol{v}) \mid \boldsymbol{v} \in S\}$. Since f is a continuous function, it holds that $sup\{f(\boldsymbol{v}) \mid \boldsymbol{v} \in S\} = sup\{f(\boldsymbol{v}) \mid \boldsymbol{v} \in \overline{S}\}$. Let us define $\text{LP}_{\text{f},\mathcal{N}}$ as an LP with variables $\boldsymbol{x} := x_1, \dots, x_{|T|}$ and constraints $\Delta(T)\boldsymbol{x} \geq -\{\mathsf{i}\colon 1\}$ and $\boldsymbol{x} \geq \boldsymbol{0}$.

Claim 1. It holds that $\boldsymbol{v} \in \overline{S}$ if and only if $\boldsymbol{v}$ is a solution to $\text{LP}_{\text{f},\mathcal{N}}$.

We postpone the proof of Claim 1. Claim 1 allows us to rephrase the computation of $sup\{f(\boldsymbol{v}) \mid \boldsymbol{v} \in \overline{S}\}$ as an $\text{LP}_{\text{f},\mathcal{N}}$ where we want to maximise $f(\boldsymbol{v})$, which can be done in polynomial time. □

What remains is the proof of Claim 1. It constitutes the remaining part of this Section. The claim is a special case of the forthcoming Lemma 8. Its formulation and proof require some preparation.

Definition 1. *A workflow net is* good *for a set of markings $M \subseteq \mathbb{Q}^P_{\geq 0}$ if for every place p there are markings $\boldsymbol{m}, \boldsymbol{m}'$ and continuous runs π and π' such that $\boldsymbol{m}(p) > 0$, $\boldsymbol{m}' \in M$, and $\{\mathsf{i}\colon 1\} \to^{\pi}_{\mathbb{Q}_{\geq 0}} \boldsymbol{m} \to^{\pi'}_{\mathbb{Q}_{\geq 0}} \boldsymbol{m}'$.*

The notion of being good for a set of markings is a refined concept of nonredundancy. The nonredundancy allow us to mark every place. But if, after marking the place, we want to continue the run and reach a marking in a specific set of markings $M \subseteq \mathbb{Q}^P_{\geq 0}$, then we don't know if the given place can be marked. This motivates Definition 1.

Example 5. Let us consider a workflow net depicted on Fig. 4. It is nonredundant, as every place can be marked. But it is not good for $\{\mathsf{f}\colon 1\}$ as there is no continuous run to the marking $\{\mathsf{f}\colon 1\}$. In the initial marking the only enabled transition is t_1 but firing βt_1 for any $\beta \in \mathbb{Q}_{\geq 0}$ reduce the total amount of tokens in the net. The lost tokens can not be recrated so it is not possible to reach $\{\mathsf{f}\colon 1\}$.

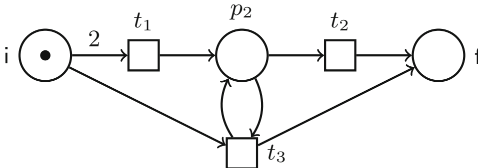


Fig. 4. A Petri net with places p_1, p_2, p_3 and transitions t_1, t_2, t_3. Marking $\{\mathsf{i}\colon 1\}$ is drawn.

The important fact is as follows:

Lemma 5. *Let $M \subseteq \mathbb{Q}^P_{\geq 0}$ be a set of solutions of some LP. Then testing if a net is good for M can be done in polynomial time.*

Lemma 6. *Suppose a workflow net $\mathcal{N}$ is good for $M \subseteq \mathbb{Q}^P_{\geq 0}$ and M is a convex set. Then there is a marking $\boldsymbol{m_+}$ such that $\boldsymbol{m_+}(p) > 0$ for every $p \in P$ and there are continuous runs π, π', and a marking $\boldsymbol{m_f} \in M$ such that $\{\mathsf{i}\colon 1\} \rightarrow^{\pi}_{\mathbb{Q}_{\geq 0}} \boldsymbol{m_+} \rightarrow^{\pi'}_{\mathbb{Q}_{\geq 0}} \boldsymbol{m_f}$.*

Informally, we prove it by taking a convex combination of a $|P|$ runs one for each $p \in P$. The last bit needed for the proof of Lemma 8 is the following lemma, shown in [19].

Lemma 7 ([19], Lemma 13). *Let $\mathcal{N}$ be a Petri net. Consider $\boldsymbol{m_0}$, $\boldsymbol{m} \in \mathbb{N}^P$ and $\boldsymbol{v} \in \mathbb{Q}^T_{\geq 0}$ such that:*

- $\boldsymbol{m} = \boldsymbol{m_0} + \Delta(\boldsymbol{v})$;
- $\forall p \in {}^\bullet\boldsymbol{v}\colon\ \boldsymbol{m_0}(p) > 0$;
- $\forall p \in \boldsymbol{v}^\bullet\colon\ \boldsymbol{m}(p) > 0$.

Then there exists a finite continuous run π such that $\boldsymbol{m_0} \rightarrow^{\pi}_{\mathbb{Q}_{\geq 0}} \boldsymbol{m}$ and $\boldsymbol{R}_\pi = \boldsymbol{v}$.

Lemma 8. *Suppose M is a convex set of markings over $\mathbb{Q}_{\geq 0}^P$ and that the workflow net is good for M. Let S be the set of Parikh images of continuous runs that start in $\{\mathsf{i}\colon 1\}$ and end in some marking $\boldsymbol{m}' \in M$ i.e.*

$$S := \{\boldsymbol{R}_\pi \mid \exists_{\pi \in CRuns^1_{\mathcal{N}}} \exists_{\boldsymbol{m}' \in M} \text{ such that } \{\mathsf{i}\colon 1\} \rightarrow^{\pi}_{\mathbb{Q}_{\geq 0}} \boldsymbol{m}'\}.$$

Then $\boldsymbol{v} \in \overline{S}$ if and only if there is a marking $\boldsymbol{m} \in M$ such that $\Delta(T)\boldsymbol{v} = \boldsymbol{m} - \{\mathsf{i}\colon 1\}$.

In one direction the proof of the lemma is trivial, in the opposite direction, intuitively, we construct a sequence of runs with Parikh images converging to $\boldsymbol{v}$. The Lemma 6 is used to put ε in every place (for $\varepsilon \to 0$) and Lemma 7 to show that there are runs with the Parihk image equal $\varepsilon\boldsymbol{x} + (1-\varepsilon)\boldsymbol{v}$ for some $\boldsymbol{x}$ witnessing Lemma 6. We are ready to prove Claim 1.

Claim 1. It holds that $\boldsymbol{v} \in \overline{S}$ if and only if $\boldsymbol{v}$ is a solution to $\mathrm{LP}_{\mathrm{f},\mathcal{N}}$.

Proof. Let M be the set of all markings over $\mathbb{Q}_{\geq 0}^P$, which clearly is convex. As $\mathcal{N}$ is nonredundant we know that every place can be marked via a continuous run, and because M is the set of all markings we conclude that $\mathcal{N}$ is good for M according to Definition 1. Thus M satisfies the prerequisites of Lemma 8. It follows that $\overline{S}$ is the set of solutions of a system of linear inequalities. Precisely, $\boldsymbol{v} \in \overline{S}$ if and only if there is $\boldsymbol{m} \in \mathbb{Q}_{\geq 0}^P$ such that $\Delta(T)\boldsymbol{v} \geq \boldsymbol{m} - \{\mathsf{i}\colon 1\}$ and $\boldsymbol{v} \geq 0$, which is equivalent to $\Delta(T)\boldsymbol{v} \geq -\{\mathsf{i}\colon 1\}$ and $\boldsymbol{v} \geq 0$, as required. □

5 Soundness in Terminating Workflow Nets

The dichotomy between linear termination time and non-termination shown in Sect. 3 yields an interesting avenue for framing questions in workflow nets. We know that testing generalised soundness is PSPACE-complete, but the lower bound in [9] relies on a reset gadget which makes the net non-terminating. Indeed, it turns out that the problem is simpler for linear workflow nets.

Theorem 3. *Generalised soundness is coNP-complete for linear workflow nets.*

A marking $\boldsymbol{m}$ is called a *deadlock* if $\mathrm{Runs}^{\boldsymbol{m}}_{\mathcal{N}} = \emptyset$. To help prove the coNP upper bound, let us introduce a lemma.

Lemma 9. *Let $\mathcal{N}$ be a terminating nonredundant workflow net. Then $\mathcal{N}$ is not generalised sound iff there exist $k \in \mathbb{N}$ and a marking $\boldsymbol{m} \in \mathbb{N}^P$ such that $\{\mathsf{i}\colon k\} \rightarrow^*_{\mathbb{Z}} \boldsymbol{m}$, $\boldsymbol{m}$ is a deadlock and $\boldsymbol{m} \neq \{\mathsf{f}\colon k\}$. Moreover, if $\|\mathcal{N}\| \leq 1$ then $\{\mathsf{i}\colon k\} \rightarrow^*_{\mathbb{Z}} \boldsymbol{m}$ can be replaced with $\{\mathsf{i}\colon k\} \rightarrow^*_{\mathbb{Q}} \boldsymbol{m}$.*

The last part of the lemma is not needed for the theoretical results, but it will speed up the implementation in Sect. 7. We can now show Theorem 3.

Proof (of the coNP upper bound in Theorem 3). Let $\mathcal{N} = (P, T, F)$ and denote $T = \{t_1, \ldots, t_n\}$. By Lemma 9 $\mathcal{N}$ is not generalised sound iff there are $k \in \mathbb{N}$ and $\boldsymbol{m} \in \mathbb{N}^P$ such that $\{\mathsf{i}\colon k\} \rightarrow^*_{\mathbb{Z}} \boldsymbol{m}$, $\boldsymbol{m}$ is a deadlock and $\boldsymbol{m} \neq \{\mathsf{f}\colon k\}$. We can reduce the property to an ILP. First, the procedure guesses $|T|$ places $p_1, \ldots, p_n \in P$ (one for each transition). For each transition t_i, place p_i will prohibit firing t_i by not being marked with enough tokens. We create $\text{ILP}_{\mathcal{N},p_1,\ldots,p_n}$, which is very similar to $\text{ILP}_{\mathcal{N}}$ (see Sect. 3), but adds additional constraints. They state that $(\Delta(T)\boldsymbol{x})(p_j) - {}^{\bullet}t_j(p_j) < 0$ for every $1 \leq j \leq n$.

Let us show that there are $p_1, \ldots, p_n$ such that $\text{ILP}_{\mathcal{N},p_1,\ldots,p_n}$ has a solution iff there exist k and a deadlock $\boldsymbol{m}$ such that $\{\mathsf{i}\colon k\} \rightarrow^*_{\mathbb{Z}} \boldsymbol{m}$. Indeed, let $x_1, \ldots, x_n$ be a solution of $\text{ILP}_{\mathcal{N},p_1,\ldots,p_n}$. We denote $k = -\sum_{i=1}^{n} \Delta(t_i)(\mathsf{i}) \cdot x_i$ and $\boldsymbol{m} = \{\mathsf{i}\colon k\} + \sum_{i=1}^{n} \Delta(t_i) \cdot x_i$. It is clear that $\{\mathsf{i}\colon k\} \rightarrow^*_{\mathbb{Z}} \boldsymbol{m}$. The new constraints ensure that for each $t_i \in T$ there exists $p_i \in P$ such that ${}^{\bullet}t_i(p_i) > \boldsymbol{m}(p_i)$, thus $\boldsymbol{m}$ is a deadlock.

To encode the requirement that $\boldsymbol{m} \neq \{\mathsf{f}\colon k\}$, note that there are three cases, either $\boldsymbol{m}(k) \leq k-1$, $\boldsymbol{m}(k) \geq k+1$, or $\boldsymbol{m}(k) = k$ but $\boldsymbol{m} - \{\mathsf{f}\colon k\} \geq 0$. We guess which case occurs, and add the constraint for that case to $\text{ILP}_{\mathcal{N},p_1,\ldots,p_n}$. □

The lower bound can be proven using a construction presented in [10, Theorem 2] to show a problem called continuous soundness on acyclic workflow nets is coNP-hard. We say that a workflow net is *continuously sound* iff for all $\boldsymbol{m}$ such that $\{\mathsf{i}\colon 1\} \rightarrow^*_{\mathbb{Q}_{\geq 0}} \boldsymbol{m}$, it holds that $\boldsymbol{m} \rightarrow^*_{\mathbb{Q}_{\geq 0}} \{\mathsf{f}\colon 1\}$. The reduction can be used as is to show that generalised soundness of nets with linear termination time is coNP-hard, but the proof differs slightly. See the appendix for more details.

6 Termination Time and Concurrent Semantics

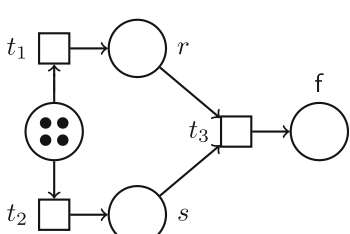
Note that in Petri nets, transitions may be fired concurrently. Thus, in a sense, our definition of termination time may overestimate the termination time.

In this section we investigate parallel executions for workflow nets. Whereas the termination time is focused on the worst case sequential execution, now we are interested in finding the best case parallel executions. Thus, we provide an optimistic lower bound on the execution time to contrast the pessimistic upper bound investigated in Sect. 3 and Sect. 4.

Definition 2. *Given a Petri net $\mathcal{N}$ let $\pi = t_1 t_2 \ldots t_n \in Runs^k_{\mathcal{N}}$ for some $k \in \mathbb{N}$. A* block *in π is a subsequence of π, i.e. $t_a, \ldots, t_b$ for some $1 \leq a \leq b \leq n$. We define the* parallel execution *of π with respect to k as a decomposition of π into blocks $\pi = \pi_1 \pi_2 \ldots \pi_\ell$ such that*

1. *all transitions are pairwise different in a single block; and*
2. ${}^{\bullet}\boldsymbol{R}_{\pi_i} \leq \{\mathsf{i}\colon k\} + \sum_{j<i} \Delta(\pi_j)$ *for every* $1 \leq i \leq \ell$.

The execution time *of a parallel execution is denoted as* $exec(\pi_1 \pi_2 \ldots \pi_\ell) := \ell$.



Example 6.
We consider parallel executions of the run $t_1t_2t_1t_2t_3t_3$ with respect to 4 initial tokens. The run can be decomposed into $(t_1t_2)(t_1t_2)(t_3)(t_3)$ but also into $(t_1)(t_2t_1)(t_2t_3)(t_3)$. Both executions have execution time 4. The parallel execution $(t_1t_2)(t_1t_2t_3)(t_3)$ has execution time 3. $\lhd$

We are interested in finding the parallel executions of a run that minimise the execution time. It turns out that the so-called *greedy parallel execution* is such a minimal parallel execution. Given π and k it is defined inductively on the prefix of π. Suppose we already have some blocks $\pi_1 \ldots \pi_{i-1}$. To construct block π_i, we simply choose the maximal sequence of transitions immediately following the last block π_{i-1} that satisfies the two conditions of Definition 2. In particular the last partition in Example 6 is the greedy parallel execution.

Lemma 10. *Consider a run π and $k \in \mathbb{N}$. The greedy parallel execution of π has the smallest execution time among all parallel executions of π with respect to k.*

Consider a workflow net $\mathcal{N}$ with the initial marking $\{\mathsf{i}\colon k\}$. Let $S_k := \{\pi \mid \{\mathsf{i}\colon k\} \to^{\pi} \{\mathsf{f}\colon k\}\}$. We define $MinTime_{\mathcal{N}}(k)$ as the minimal execution time among parallel executions of runs in S_k. If $S_k = \emptyset$ then $MinTime_{\mathcal{N}}(k) = +\infty$.

Lemma 11. *Let $\mathcal{N}$ be a workflow net and let $k, x \in \mathbb{N}$. Deciding whether $MinTime_{\mathcal{N}}(k) \leq x$ is PSPACE-hard even if we fix $k = 1$.*

As computing $MinTime_{\mathcal{N}}(k)$ is computationally hard, we modify the question and ask about the asymptotic behaviour (similarly to Sect. 4). Thus, we are interested in computing $\lim_{k\to\infty} \frac{MinTime_{\mathcal{N}}(k)}{k}$. The problem is well defined as the limit exists. This is interesting as $\lim_{k\to\infty} \frac{MinTime_{\mathcal{N}}(k)}{k}$ corresponds to the average processing time of a single token when the workflow runs (informally speaking) on its maximal efficiency.

Theorem 4. *For a given nonredundant, generalised sound workflow net*[4] *$\mathcal{N}$ we can compute $\lim_{k\to\infty} \frac{MinTime_{\mathcal{N}}(k)}{k}$ in polynomial time.*

Proof (A sketch of the proof). The main idea relies on the continuous semantics, similarly to the proof of Theorem 2. We show that the limit is equal to the infimum over execution times[5] of continuous runs $\{\mathsf{i}\colon 1\} \to_{\mathbb{Q}_{\geq 0}} \{\mathsf{f}\colon 1\}$. Then we prove the following claim.

[4] These assumptions can be relaxed to *a net good for* $\{\mathsf{f}\colon 1\}$, see Definition 1.
[5] For a suitably defined parallel execution and execution time of continuous runs.

Claim 2. Let $\boldsymbol{v} \in \mathbb{Q}_{\geq 0}^T$. Let $S_{\boldsymbol{v}} = \{\pi \mid \{\mathsf{i}\colon 1\} \rightarrow^{\pi}_{\mathbb{Q}_{\geq 0}} \{\mathsf{f}\colon 1\} \text{ and } \boldsymbol{R}_\pi = \boldsymbol{v}\}$. If $S \neq \emptyset$ then the infimum over optimal execution time of runs in $S_{\boldsymbol{v}}$ equals $\|\boldsymbol{v}\|$.

Let S be the set of Parikh images of continuous runs from $\{\mathsf{i}\colon 1\}$ to $\{\mathsf{f}\colon 1\}$. We define $f : \overline{S} \rightarrow \mathbb{Q}_{\geq 0}$ such that $f(\boldsymbol{v}) = \|\boldsymbol{v}\|$. Thus we can reformulate the problem as computing $inf\{f(\boldsymbol{v}) \mid \boldsymbol{v} \in S\}$. The function f is continuous, thus we can reformulate further as compute $inf\{f(\boldsymbol{v}) \mid \boldsymbol{v} \in \overline{S}\}$. The function f is not linear on $\overline{S}$, but it is piecewise linear. We define $\overline{S_t} \subseteq \overline{S}$ for $t \in T$ as follows $\overline{S_t} = \{\boldsymbol{v} \mid \boldsymbol{v} \in \overline{S} \text{ and } \boldsymbol{v}(t) \geq \boldsymbol{v}(t') \text{ for all } t' \in T\}$. Observe that f is linear over $\overline{S_t}$ for every $t \in T$ and that $\overline{S} = \bigcup_{t \in T} \overline{S_t}$. Thus we can rephrase our problem as computing the minimum over the set $\{inf\{\boldsymbol{v}(t) \mid \boldsymbol{v} \in \overline{S_t}\} \mid t \in T\}$.

Thus it is sufficient to show that $inf\{\boldsymbol{v}(t) \mid \boldsymbol{v} \in \overline{S_t}\}$ can be computed in polynomial time for any $t \in T$. Lemma 8 allows us to characterize $\overline{S}$ as follows: $\boldsymbol{v} \in \overline{S}$ iff $\Delta(T)\boldsymbol{v} = \{\mathsf{f}\colon 1\} - \{\mathsf{i}\colon 1\}$ and $\boldsymbol{v} \geq \boldsymbol{0}$. In consequence, $\overline{S_t}$ can be characterized as the set of solutions of the following system of inequalities

$$\Delta(T)\boldsymbol{v} = \{\mathsf{f}\colon 1\} - \{\mathsf{i}\colon 1\} \text{ and } \boldsymbol{v} \geq \boldsymbol{0} \text{ and } \boldsymbol{v}(t) \geq \boldsymbol{v}(t') \text{ for all } t' \in T.$$

This allows us to capture $\{inf\{\boldsymbol{v}(t) \mid \boldsymbol{v} \in \overline{S_t}\} \mid t \in T\}$ as an LP problem which can be solved in polynomial time. □

7 Experimental Evaluation

We have implemented prototypes of several procedures outlined in the paper, namely procedures to 1) decide termination; 2) decide soundness for terminating nets; 3) compute $a_{\mathcal{N}}$ for terminating nets; and 4) compute $MinTime_{\mathcal{N}}(1)$, $MaxTime_{\mathcal{N}}(1)$, and decide 1-soundness for nets with known $a_{\mathcal{N}}$. The idea behind all procedures is to use our results to encode the properties in LPs/ILPs. To solve these programs, we utilize the MILP solver *Gurobi* [24].

For 1), recall Lemma 2, which states that non-termination of a workflow net $\mathcal{N}$ is equivalent to the existence of a Parikh image $\boldsymbol{R} \in \mathbb{N}^T$ with $\Delta(\boldsymbol{R}) \geq \boldsymbol{0}$. We can instead search for $\boldsymbol{R} \in \mathbb{Q}^T$, as any solution could be scaled up to an integral one. Thus, we can encode this condition as an LP in a straightforward manner, and decide termination in polynomial time.[6]

For 2), we essentially use $\text{ILP}_{\mathcal{N},p_1,\ldots,p_n}$, as defined in the proof of Theorem 3. A solution to $\text{ILP}_{\mathcal{N},p_1,\ldots,p_n}$ yields a run π such that there exists $k \in \mathbb{N}$ with $\{\mathsf{i}\colon k\} \rightarrow^{\pi}_{\mathbb{Z}} \boldsymbol{m}$, where $\boldsymbol{m}$ is a deadlock.

We also consider continuous instead of integral variables. Then solutions relate to runs over $\rightarrow^*_{\mathbb{Q}}$ instead. As hinted at in the last sentence of Lemma 9, both variants yield equivalent results on nets without arc weights, *i.e.* $\|\mathcal{N}\| \leq 1$. However, continuous variables are generally easier to handle for MILP solvers. For brevity, by *integer deadlocks* we refer to the approach using integer variables, and by *continuous deadlocks* to the approach with continuous variables.

[6] This observation and the general approach comes from [30].

For 3), recall the LP given in Claim 1. We can use it to compute $\sup_{f,\mathcal{N}}$ for any aggregate $\mathcal{N}$, so in particular we can use it to compute $\sup_{f_{all},\mathcal{N}}$, which is equal to $a_{\mathcal{N}}$ by Equation (2). Here, it only remains to mention that Gurobi allows not only checking feasibility of systems of linear inequalities, but further allows optimizing an objective value, as required by the LP.

For 4), note that if we have the bound $a_{\mathcal{N}}$ on the length of runs from $\{\mathsf{i}\colon 1\}$, we can check properties by unrolling runs. The intuition is as follows. We have $a_{\mathcal{N}} \cdot |T|$ integer variables. For step j of the run, we have variables $x_{1,j}, x_{2,j}, \ldots, x_{|T|,j}$. The variables for a step encode which transition(s) are fired in that step. We ensure that we encode a run by requiring $\sum_{i=1}^{|T|} x_{i,j} \leq 1$ for all $j \in [1..a_{\mathcal{N}}]$. We use integer variables, so either one or no transition is fired in each step.

Alternatively, we encode a parallel execution by imposing the requirements of Definition 2 on steps. By further specifying that for all $j \in [1..a_{\mathcal{N}}]$, it holds that $\{\mathsf{i}\colon 1\} + \sum_{j'=0}^{j} \sum_{i=1}^{|T|} \Delta(t_i) x_{i,j'} \geq \mathbf{0}$, thus the marking reached so far after each step is nonnegative. To compute $MinTime_{\mathcal{N}}(1)/MaxTime_{\mathcal{N}}(1)$, we minimise/maximise the number of blocks/steps with non-zero transition variables. For 1-soundness, we require reaching a deadlock different from $\{\mathsf{f}\colon 1\}$.

Our prototype is implemented in C#. All experiments were run on an 8-Core Intel® Core™ i7-7700 CPU @ 3.60 GHz with Ubuntu 18.04. We limited memory to ∼8 GB. The time was limited to 60 s for checking termination and generalised soundness as well as for computing $a_{\mathcal{N}}$. It was limited to 15 s for computing $MinTime_{\mathcal{N}}(1), MaxTime_{\mathcal{N}}(1)$ and for checking 1-soundness.

7.1 Benchmark Suite

We use a popular benchmark suite of 1386 free-choice nets originating from models created in the IBM WebSphere Business Modeler. The instances were originally introduced in [18] and have frequently been studied since, see [13,37,38]. The nets use a slightly different formalisation of workflow nets that allow multiple final places, which can be transformed to standard workflow nets using a technique from [29]. This technique adds transitions, thus can increase $a_{\mathcal{N}}$, $MinTime_{\mathcal{N}}$ and $MaxTime_{\mathcal{N}}$. Unfortunately, 4 instances cannot be transformed to workflow nets with this technique, so we remove them. We also apply a set of well-known reduction rules from [13] that reduce the size of instances while keeping all types of soundness intact, and remove instances that are trivially sound after reduction. These rules never increase $a_{\mathcal{N}}$. While they in theory could increase $MinTime_{\mathcal{N}}$, this does not happen on our benchmarks. Due to the nature of the reduction rules, it may not be appropriate to run them before analyzing $MinTime_{\mathcal{N}}, MaxTime_{\mathcal{N}}(1)$ and $a_{\mathcal{N}}$, since these numbers then give no information about the original workflow. Thus we only run experiments on the reduced instances when we check soundness and termination.

In total, we are left with 1382 unreduced and 740 non-trivial reduced instances. Statistics about the sizes of the workflow nets can be seen in the columns under Net Size in Fig. 5. The reduced nets are much smaller than the unreduced ones, even when the nets are not reduced to the trivial net.

		Net Size		Analysis Time (in ms)			
		$\lvert P \rvert$	$\lvert T \rvert$	Termination	Continuous Deadlock	Integer Deadlock	Continuous Soundness [10]
Unreduced instances	Mean	48.78	33.07	4.09	7.17	12.8	2022.54
	Median	37	26	3	5	11	88
	Max	274	285	23	85	88	55707
Reduced instances	Mean	7.43	5.49	2.99	2.3	8.88	44.51
	Median	6	5	3	2	8	33
	Max	33	22	5	18	39	99

		Total	Deadlocking (Not generalised sound)
Unreduced instances	Terminating	1262	523
	Nonterm.	120	53
Reduced instances	Terminating	694	536
	Nonterm.	46	23

Fig. 5. Top: Statistics on the net size, and analysis times for deciding termination, and checking generalised soundness via deadlocks and continuous soundness. Bottom: Statistics on the number of terminating/non-terminating and deadlocking/non-deadlocking (thus generalised unsound/generalised sound) nets.

7.2 Termination and Deadlocks

The time taken to decide termination is shown in the column labelled "Termination" in the top table of Fig. 5. The numbers of nets that are terminating and non-terminating are shown in the bottom table of Fig. 5. Among both the unreduced and reduced instances, the vast majority are terminating (about 90%). Note that the reduction rules can remove nontermination, even when they do not make the net nontrivial, thus the prevalence of terminating instances is even stronger among the reduced instances. In terms of analysis time, termination can be decided in under 25 ms for all instances, with a median of 3 ms.

The top of Fig. 5 shows the analysis times for generalised soundness. We use three algorithms. Columns "Continuous Deadlock" and "Integer Deadlock" show results for our two proposed approaches, and column "Continuous Soundness" shows the performance of a state-of-art approach [10] for deciding generalised soundness. Note that both approaches may claim an unsound workflow net to be sound, but they are precise on different classes of nets. The absence of integer deadlocks is equivalent to generalised soundness on terminating nets, see Lemma 9. Similarly, continuous soundness is equivalent to generalised soundness on free-choice nets [10].

In practice, it turns out that our approach for checking the absence of integer deadlocks is faster than the existing approach using continuous soundness on every single instance. Continuous soundness times out on 215 of the unreduced instances (not listed in the table), but neither of the approaches utilizing deadlocks times out on any instance. The performance of continuous soundness is

not surprising: continuous soundness is checked by passing an $\exists\forall$-formula from $FO(\mathbb{Q}, <, +)$ to an SMT solver. Quantifier alternation increases the complexity of validating such formulas [23]. In comparison, our check for integer deadlocks is implemented using standard ILP techniques, and thus an existential formula.

The bottom shows how many nets are non-terminating, as well as how many are deadlocking (thus not generalised sound). Recall that integer deadlocks and continuous deadlocks are equivalent for nets without arc weights, which all of our nets are. Both types of deadlocks are fast to compute, taking less than $90ms$ on each instance. In practice, checking for continuous deadlocks may be useful even for nets with arc weights, since their absence also proves the absence of integer deadlocks. About 50% of the unreduced instances and roughly 75% of the reduced instances are deadlocking. Note that the reduction rules can only make sound instances trivial, which are by definition not able to reach a deadlock.

7.3 $a_{\mathcal{N}}$, $MinTime_{\mathcal{N}}(1)$ and $MaxTime_{\mathcal{N}}(1)$

The top of Fig. 6 the distribution of $a_{\mathcal{N}}$. This number depends on the number of transitions, so is hard to put into context. We instead display $\mathfrak{L} := {}^{a_{\mathcal{N}}}/_{|T|}$. Intuitively, that number is an upper bound on the average of how many times each transition can be fired per initial tokens. For example, a net with $\mathfrak{L} = 1$ likely is linear, *i.e.* each transition can be fired only once per initial token, while nets with $\mathfrak{L} >> 1$ may exhibit more complex behaviour, and nets with $\mathfrak{L} << 1$ may exhibit high degrees of choice, where runs only visit a part of the net. We group nets with similar $\mathfrak{L}$ to give an idea of the distribution of the values of $\mathfrak{L}$ across instances. Our computation of $a_{\mathcal{N}}$ ran out of memory on 8 nets, so the figure displays only 1254 nets. Most nets have $\mathfrak{L} \leq 1$, with a significant number having in particular $\mathfrak{L} = 1$. The maximal $\mathfrak{L}$ is 5.83 among unreduced and 4.33 among reduced instances, while the minimal $\mathfrak{L}$ is 0.17 and 0.11 respectively.

To display $MinTime_{\mathcal{N}}(1)$ and $MaxTime_{\mathcal{N}}(1)$, we also divide them by the number of transitions, as we did for $a_{\mathcal{N}}$. We write $\mathfrak{T}_{Min} := {}^{MinTime_{\mathcal{N}}(1)}/_{|T|}$ and $\mathfrak{T}_{Max} := {}^{MaxTime_{\mathcal{N}}(1)}/_{|T|}$. We are mostly interested in their difference $\mathfrak{D} := \mathfrak{T}_{Max} - \mathfrak{T}_{Min}$. For nets with large $\mathfrak{D}$, the difference between the pessimistic sequential and optimistic parallel execution time is large, thus they might allow a high degree of parallelism. On the contrary, if nets have very small $\mathfrak{D}$, they have a sequential structure. We again group nets with similar $\mathfrak{D}$, as we did for $\mathfrak{L}$ above. The results of the analysis are shown in the middle table of Fig. 6.

As we divide by $|T|$ in the definition of $\mathfrak{D}$, it would be unusual for it to take on huge values, and indeed all nets have $\mathfrak{D} < 1$. Note that even $\mathfrak{D} = 0.5$ is significant, as it means that $MinTime_{\mathcal{N}}(1)$ and $MaxTime_{\mathcal{N}}(1)$ differ by half the number of transitions. The table totals only 700 nets. On 111 nets, computing $MinTime_{\mathcal{N}}(1)$ times out, while on 32 nets computing $MaxTime_{\mathcal{N}}(1)$ times out, and both time out on 51 nets. On the remaining 360 nets, there is no execution from $\{\mathsf{i}\colon 1\}$ to $\{\mathsf{f}\colon 1\}$, thus $MinTime_{\mathcal{N}}(1) = \infty$.

The analysis times for computing $a_{\mathcal{N}}$, $MinTime_{\mathcal{N}}(1)$ and $MaxTime_{\mathcal{N}}(1)$ are shown in the bottom table of Fig. 6. We group nets by their size $|\mathcal{N}| = |P| + |T|$ to show how the analysis times depend on the instance size. We only list 1060 nets,

	Buckets B				
	$[0, 0.75)$	$[0.75, 1)$	$[1, 1]$	$(1, 1.75)$	$[1.75, \infty)$
Count with $\mathfrak{L} \in B$	303	274	422	173	82

	Buckets B				
	$[0, 0.05)$	$[0.05, 0.15)$	$[0.15, 0.3)$	$[0.3, 0.5)$	$[0.5, 1)$
Count with $\mathfrak{D} \in B$	29	222	295	120	34

		Buckets B			
		$[0, 20)$	$[20, 60)$	$[60, 150)$	$[150, 405)$
Count with $\lvert\mathcal{N}\rvert \in B$		241	391	388	40
Analysis time for computing $a_{\mathcal{N}}$ (in ms)	Mean	11.9	9.56	9.65	9.8
	Median	7	7	8	8
	Max	714	246	289	33
Analysis time for computing $MinTime_{\mathcal{N}}(1)$ (in ms)	Mean	8.29	120.52	1610.44	2128.83
	Median	8	36	307	1454
	Max	14	6599	14905	12160
Analysis time for computing $MaxTime_{\mathcal{N}}(1)$ (in ms)	Mean	3.99	44.23	669.66	5305.5
	Median	4	29	173	4934
	Max	8	2561	12370	14954

Fig. 6. Top: Statistics on the distribution of $\mathfrak{L}$. Middle: Statistics on the distribution of $\mathfrak{D}$. Bottom: Statistics on the analysis times for $a_{\mathcal{N}}$, $\mathfrak{I}_{Min}$ and $\mathfrak{I}_{Max}$.

as we omit those where the computation of $MinTime_{\mathcal{N}}(1)$ or $MaxTime_{\mathcal{N}}(1)$ timed out. One interesting observation is that for most instances, particularly small ones, $MinTime_{\mathcal{N}}(1)$ is harder to compute than $MaxTime_{\mathcal{N}}(1)$. However, both are very slow to compute compared to $a_{\mathcal{N}}$, which indeed never times out on our instances. In fact, $a_{\mathcal{N}}$ takes at most $714ms$ to compute for any instance. It is interesting that the time for computing $a_{\mathcal{N}}$ does not seem to depend highly on the net size. We suspect this might be partly due to the fact that $a_{\mathcal{N}}$ tends to be proportionally smaller for larger instances: Bucket $[0, 20)$ has a mean $\mathfrak{L}$ of 1.04, while the mean is 0.86 for bucket $[150, 405)$.

7.4 1-Soundness

Lastly, we briefly comment on the time for deciding 1-soundness via unrolling for nets with known $a_{\mathcal{N}}$. The procedure times out for 71 instances, among which $a_{\mathcal{N}}$ has a mean of 133.88 and a maximum of 256. It takes a mean of $612.66ms$ and a maximum of $14431ms$ to decide 1-soundness in this way. Unlike in the case for generalised soundness, our procedure for 1-soundness does not seem to be able to compete with the state-of-the-art. In [18], 1-soundness is decided for many of our instances in a few milliseconds per instance, which our approach does so only for instances with small $a_{\mathcal{N}}$ (up to about 25).

References

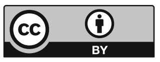
1. Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63139-9_48
2. van der Aalst, W.M.P., et al.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects Comput. **23**(3), 333–363 (2011). https://doi.org/10.1007/s00165-010-0161-4
3. van der Aalst, W.M.: A class of petri net for modeling and analyzing business processes. Comput. Sci. Rep. **95**(26), 1–25 (1995)
4. van der Aalst, W.M.P., Hirnschall, A., Verbeek, H.M.W.: An alternative way to analyze workflow graphs. In: Pidduck, A.B., Ozsu, M.T., Mylopoulos, J., Woo, C.C. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 535–552. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47961-9_37
5. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of different semantics for time petri nets. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 293–307. Springer, Heidelberg (2005). https://doi.org/10.1007/11562948_23
6. Blondin, M., Finkel, A., Haase, C., Haddad, S.: Approaching the coverability problem continuously. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 480–496. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_28
7. Blondin, M., Finkel, A., Haase, C., Haddad, S.: The logical view on continuous Petri nets. ACM Trans. Comput. Logic (TOCL) **18**(3), 24:1–24:28 (2017). https://doi.org/10.1145/3105908
8. Blondin, M., Haase, C., Offtermatt, P.: Directed reachability for infinite-state systems. In: TACAS 2021. LNCS, vol. 12652, pp. 3–23. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_1
9. Blondin, M., Mazowiecki, F., Offtermatt, P.: The complexity of soundness in workflow nets. In: Proceedings of the 37th Symposium on Logic in Computer Science (LICS) (2022). https://doi.org/10.1145/3531130.3533341
10. Blondin, M., Mazowiecki, F., Offtermatt, P.: Verifying generalised and structural soundness of workflow nets via relaxations. In: Shoham, S., Vizel, Y. (eds.) CAV. LNCS, vol. 13372, pp. 468–489. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_23
11. Brázdil, T., Chatterjee, K., Kučera, A., Novotný, P., Velan, D.: Deciding fast termination for probabilistic VASS with nondeterminism. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 462–478. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_27
12. Brázdil, T., Chatterjee, K., Kucera, A., Novotný, P., Velan, D., Zuleger, F.: Efficient algorithms for asymptotic bounds on termination time in VASS. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, 09–12 July 2018, pp. 185–194. ACM (2018). https://doi.org/10.1145/3209108.3209191
13. Bride, H., Kouchnarenko, O., Peureux, F.: Reduction of workflow nets for generalised soundness verification. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 91–111. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_6
14. Czerwinski, W., Orlikowski, L.: Reachability in vector addition systems is Ackermann-complete. In: Proceedings 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS) (2021)

15. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press (1995). https://doi.org/10.1017/CBO9780511526558
16. Dixon, A., Lazić, R.: KReach: a tool for reachability in Petri nets. In: TACAS 2020. LNCS, vol. 12078, pp. 405–412. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_22
17. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM framework: a new era in process mining tool support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005). https://doi.org/10.1007/11494744_25
18. Fahland, D., et al.: Instantaneous soundness checking of industrial business process models. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 278–293. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03848-8_19
19. Fraca, E., Haddad, S.: Complexity analysis of continuous Petri nets. Fundamenta Informaticae **137**(1), 1–28 (2015). https://doi.org/10.3233/FI-2015-1168
20. Freytag, T., Allgaier, P., Burattin, A., Danek-Bulius, A.: WoPeD-a "proof-of-concept" platform for experimental BPM research projects. In: 15th International Conference on Business Process Management (BPM 2017) (2017)
21. Geffroy, T., Leroux, J., Sutre, G.: Occam's razor applied to the Petri net coverability problem. Theor. Comput. Sci. **750**, 38–52 (2018). https://doi.org/10.1016/j.tcs.2018.04.014
22. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992). https://doi.org/10.1145/146637.146681
23. Grigoriev, D.: Complexity of deciding Tarski algebra. J. Symb. Comput. **5**(1/2), 65–108 (1988). https://doi.org/10.1016/S0747-7171(88)80006-3
24. Gurobi Optimization, LLC: Gurobi optimizer reference manual (2023). https://www.gurobi.com
25. van Hee, K., Oanea, O., Sidorova, N., Voorhoeve, M.: Verifying generalized soundness of workflow nets. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 235–247. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70881-0_21
26. van Hee, K., Sidorova, N., Voorhoeve, M.: Soundness and separability of workflow nets in the stepwise refinement approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44919-1_22
27. van Hee, K., Sidorova, N., Voorhoeve, M.: Generalised soundness of workflow nets is decidable. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 197–215. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27793-4_12
28. Hopcroft, J., Pansiot, J.J.: On the reachability problem for 5-dimensional vector addition systems. Theoret. Comput. Sci. **8**(2), 135–159 (1979)
29. Kiepuszewski, B., ter Hofstede, A.H.M., van der Aalst, W.M.P.: Fundamentals of control flow in workflows. Acta Informatica **39**(3), 143–209 (2003). https://doi.org/10.1007/s00236-002-0105-4
30. Kosaraju, S.R., Sullivan, G.F.: Detecting cycles in dynamic graphs in polynomial time (preliminary version). In: Simon, J. (ed.) Proceedings of the 20th annual ACM symposium on theory of computing, 2–4 May 1988, Chicago, Illinois, USA, pp. 398–406. ACM (1988). https://doi.org/10.1145/62212.62251

31. Kucera, A., Leroux, J., Velan, D.: Efficient analysis of VASS termination complexity. In: Hermanns, H., Zhang, L., Kobayashi, N., Miller, D. (eds.) LICS 2020: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, 8–11 July 2020, pp. 676–688. ACM (2020). https://doi.org/10.1145/3373718.3394751
32. Leroux, J.: Polynomial vector addition systems with states. In: Chatzigiannakis, I., Kaklamanis, C., Marx, D., Sannella, D. (eds.) 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, 9-13 July 2018, Prague, Czech Republic. LIPIcs, vol. 107, pp. 134:1–134:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ICALP.2018.134
33. Leroux, J.: The reachability problem for Petri nets is not primitive recursive. In: Proceedings 62nd Annual IEEE Symposium on Foundations of Computer Science (FOCS) (2021)
34. Leroux, J., Schmitz, S.: Reachability in vector addition systems is primitive-recursive in fixed dimension. In: Proceedings 34th Symposium on Logic in Computer Science (LICS) (2019). https://doi.org/10.1109/LICS.2019.8785796
35. Leroux, J., Schnoebelen, P.: On functions weakly computable by petri nets and vector addition systems. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) RP 2014. LNCS, vol. 8762, pp. 190–202. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11439-2_15
36. Lipton, R.: The reachability problem requires exponential space. Department of Computer Science, Yale University, vol. 62 (1976)
37. Meyer, P.J., Esparza, J., Offtermatt, P.: Computing the expected execution time of probabilistic workflow nets. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 154–171. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_9
38. Meyer, P.J., Esparza, J., Völzer, H.: Computing the concurrency threshold of sound free-choice workflow nets. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 3–19. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_1
39. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). https://doi.org/10.1109/5.24143
40. Praveen, M., Lodaya, K.: Analyzing reachability for some petri nets with fast growing markings. Electron. Notes Theor. Comput. Sci. **223**, 215–237 (2008). https://doi.org/10.1016/j.entcs.2008.12.041
41. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. **6**, 223–231 (1978). https://doi.org/10.1016/0304-3975(78)90036-1
42. Schmitz, S.: The complexity of reachability in vector addition systems. ACM SIGLOG News **3**(1), 4–21 (2016). https://doi.org/10.1145/2893582.2893585
43. Valk, R., Vidal-Naquet, G.: Petri nets and regular languages. J. Comput. Syst. Sci. **23**(3), 299–325 (1981). https://doi.org/10.1016/0022-0000(81)90067-2
44. Verbeek, E., van der Aalst, W.M.P.: Woflan 2.0 a petri-net-based workflow diagnosis tool. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 475–484. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44988-4_28
45. Wolf, K.: Petri net model checking with LoLA 2. In: Khomenko, V., Roux, O.H. (eds.) PETRI NETS 2018. LNCS, vol. 10877, pp. 351–362. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91268-4_18

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM

Nikita Koval[1(✉)], Alexander Fedorov[1,2], Maria Sokolova[1], Dmitry Tsitelov[3], and Dan Alistarh[2]

[1] JetBrains, Prague, Czech Republic
`ndkoval@ya.ru`
[2] IST Austria, Klosterneuburg, Austria
[3] Devexperts, Munich, Germany

Abstract. This paper presents `Lincheck`, a new practical and user-friendly framework for testing concurrent algorithms on the Java Virtual Machine (JVM). `Lincheck` provides a simple and declarative way to write concurrent tests: instead of describing *how* to perform the test, users specify *what to test* by declaring all the operations to examine; the framework automatically handles the rest. As a result, tests written with `Lincheck` are concise and easy to understand. The framework automatically generates a set of concurrent scenarios, examines them using stress-testing or bounded model checking, and verifies that the results of each invocation are correct. Notably, if an error is detected via model checking, `Lincheck` provides an easy-to-follow trace to reproduce it, significantly simplifying the bug investigation.

To the best of our knowledge, `Lincheck` is the first production-ready tool on the JVM that offers such a simple way of writing concurrent tests, without requiring special skills or expertise. We successfully integrated `Lincheck` in the development process of several large projects, such as Kotlin Coroutines, and identified new bugs in popular concurrency libraries, such as a race in Java's standard `ConcurrentLinkedDeque` and a liveliness bug in Java's `AbstractQueuedSynchronizer` framework, which is used in most of the synchronization primitives. We believe that `Lincheck` can significantly improve the quality and productivity of concurrent algorithms research and development and become the state-of-the-art tool for checking their correctness.

1 Introduction

Concurrent programming is known to be notoriously hard and error-prone. Writing a good and robust test for a concurrent data structure may be even more challenging than implementing it. Programmers produce many such stress tests every day, but they often are nondeterministic, cover only specific cases, and do not catch all the bugs. Both the industry and academia need a tool that would simplify writing reliable tests for concurrent data structures.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 156–169, 2023.
https://doi.org/10.1007/978-3-031-37706-8_8

In this paper, we present Lincheck [1], a new practical framework for JVM-based languages (such as Java, Kotlin, and Scala), which simplifies writing reliable concurrent tests. While most existing tools require writing the algorithm in a special language [2], specifying all possible concurrent scenarios and their outcomes [3–6], or learning a large amount of theory [7,8], Lincheck provides a more pragmatic *declarative* approach. It requires users only to list the data structure operations, thus, specifying *what to test* instead of *how*. Taking these operations, Lincheck generates a set of concurrent scenarios and examines them via stress testing or model checking, verifying that the outcome results are correct. The default correctness property is linearizability [9], but various relaxations [10–12] are also supported. One may think of Lincheck as a mix of a fuzzer (that generates concurrent scenarios) and a model checker or stress runner (which examines these scenarios) equipped with an automatic outcome verifier.

Lincheck by Example. The "classic" way to write a concurrent test is to manually run parallel threads, invoking the data structure operations in them and checking that some sequential history can explain the produced results. Such tests typically contain hundreds of lines of boilerplate code and cover only easy-to-verify scenarios. Lincheck automates the machinery, making tests short and declarative. To illustrate that, we present a test for the ConcurrentLinkedDeque collection (double-ended queue, which supports insertions and removals at both ends) of the standard Java library in Listing 1.

The initial state of the testing data structure is specified in the constructor; here, we simply create a new empty deque at line 2. The following lines 4–9 declare the deque operations; they should be annotated with @Operation. Finally, we run the analysis by invoking ModelCheckingOptions.check(..) on the testing class at line 11. Replacing ModelCheckingOptions with StressOptions switches to stress testing, which essentially runs parallel threads.

```
1  class DequeTest {
2   val deque = ConcurrentLinkedDeque<Int>()
3
4   @Operation fun addFirst(e: Int) = deque.addFirst(e)
5   @Operation fun addLast(e: Int)  = deque.addLast(e)
6   @Operation fun pollFirst()      = deque.pollFirst()
7   @Operation fun pollLast()       = deque.pollLast()
8   @Operation fun peekFirst()      = deque.peekFirst()
9   @Operation fun peekLast()       = deque.peekLast()
10
11  @Test fun runTest() = ModelCheckingOptions()
12                          .check(this::class)
13 }
```

Listing 1. Concurrent test via Lincheck for Java's ConcurrentLinkedDeque. The code is written in Kotlin; import statements are omitted.

After executing the test, we get an error presented in Fig. 1. Surprisingly, this class from the standard Java library has a bug; the error was originally detected via Lincheck by the authors [13] (notably, there were several unsuc-

```
= Invalid execution results =                          Comment: this text is a Lincheck output,
| addLast(-6)      | addFirst(-8)   |                  while the scheme is drawn by the authors
| peekFirst(): -8  | pollLast(): -8 |

= The following interleaving leads to the error =
|                  | addFirst(-8)                                                      |
|                  | pollLast()                                                        |
|                  |  pollLast(): -8 at DequeTest.pollLast(DequeTest.kt:35)            |
|                  |   last(): Node@1 at CLD.pollLast(CLD.java:936)                    |
|                  |   item.READ: null at CLD.pollLast(CLD.java:938)                   |
|                  |   prev.READ: Node@2 at CLD.pollLast(CLD.java:946)                 |
|                  |   item.READ: -8 at CLD.pollLast(CLD.java:938)                     |
|                  |   next.READ: null at CLD.pollLast(CLD.java:940)                   |
|                  |   switch                                                          |
| addLast(-6)      |                                                                   |
| peekFirst(): -8  |                                                                   |
|                  |   item.CAS(-8,null): true at CLD.pollLast(CLD.java:941)           |
|                  |   unlink(Node@2) at CLD.pollLast(CLD.java:942)                    |
|                  |  result: -8                                                       |
```


Fig. 1. The incorrect execution of the Java's `ConcurrentLinkedDeque` identified by the `Lincheck` test from Listing 1 and illustrated by a pictured diagram. To narrow the test output, `ConcurrentLinkedDeque` is replaced with `CLD`.

cessful attempts to fix the incorrectness before that [14,15]). Obviously, the produced results are non-linearizable: for `pollLast()` in the second thread to return `-8`, it should be called before `addLast(-6)` in the first thread; however, that would require the following `peekFirst()` to return `-6` instead of `-8`. While `Lincheck` always prints a failing scenario with incorrect results (if found), the model checker also provides a detailed *interleaving trace* that reproduces the error.

Providing a detailed and informative trace is a game-changer. With it, we can easily understand why `ConcurrentLinkedDeque` is incorrect. The underlying data structure forms a doubly-linked list, with `head` and `tail` pointers approximating its first and last nodes. Initially, `head` and `tail` point to a logically removed `(Node.item == null)` node. After `addFirst(-8)` in the second thread is applied, a new node is added to the beginning; `head` and `tail` remain unchanged. Then, `pollLast()` starts; it finds the last non-empty node (the previously added one) and gets preempted before extracting the element. (The procedure linearizes on changing the `Node.item` value to `null` via atomic `Compare-and-Set (CAS)` instruction.) After invoking `addLast(-6)` in the first thread, a new node is added to the end of the list. The following `peekFirst()` does not change the data structure logically but advances the `head` pointer.

Finally, the execution switches back to the second thread. The `pollLast()` operation successfully removes the node containing `-8` (which is no longer the last element), extracting the item via `CAS` followed by unlinking the node physically. These twelve lines of straightforward code easily find a bug in the standard library of Java and provide a detailed trace that leads to the error, reducing the investigation time from hours to minutes. We also believe that with such an instrument as `Lincheck`, the bug would not have been released in the first place.

Practical-Oriented Design. `Lincheck` was designed as a tool for testing real-world concurrent code. The following its properties are crucial in practice:

- **Declarative testing.** `Lincheck` takes only a list of operations and optional configuration parameters (we discuss them further), which results in short and intuitive tests — no need to learn a new language or technology.
- **No implementation restrictions.** `Lincheck` can test any real-world implementations, including those that utilize low-level JVM constructs like `Unsafe` or `VarHandle`, without imposing any restrictions.
- **No false positives.** `Lincheck` reports only reproducible errors, which is vital for using the framework in continuous integration (CI/CD) and unit tests.
- **User-friendliness.** `Lincheck` streamlines bug investigation by providing a thorough trace of the discovered error, saving programmers countless hours.
- **Flexibility.** `Lincheck` supports popular constraints, such as the single-producer/consumer workload, as well as a range of linearizability relaxations, enabling custom scenario generation and verification when necessary.

Real-World Applications. We have successfully integrated `Lincheck` in the development processes of Kotlin Coroutines [16] and JCTools [17] libraries, enabling reliable testing of their core data structures, which are often complex and several thousand lines of code long. `Lincheck`'s support of popular workload constraints and linearizability relaxations and its ability to handle blocking operations, such as those of `Mutex` and `Channel`, were crucial for these tests. Furthermore, for over five years, we have successfully used `Lincheck` in our "Parallel Programming" course to automate the verification of more than 4K student solutions annually.

We have also detected several new bugs [18] in popular libraries, including the previously discussed race in Java's `ConcurrentLinkedDeque` [13], non-linearizabi-lity of `NonBlockingHashMapLong` from JCTools [19], and liveness bugs in Java's `AbstractQueuedSynchronizer` [18] and `Mutex` in Kotlin Coroutines [20].

In conclusion, `Lincheck` is a powerful and versatile tool for testing complex concurrent programs. It provides non-trivial features in terms of generality, ease of use, and performance. We provide a comprehensive overview of `Lincheck` in the rest of the paper and believe that it will greatly save time and (mental) energy tracking down concurrency bugs.

2 Lincheck Overview

We now dive into `Lincheck` internals, presenting its key features as we go along. The testing process can be broken down into three stages, as depicted in the diagram below. `Lincheck` generates a set of concurrent scenarios and examines them via either model checking or stress testing, verifying that each scenario invocation results satisfy the desirable correctness property (linearizability [9] by default). If the outcome is incorrect, the invocation hangs, or the code throws an unexpected exception, the test fails with an error similar to the one in Fig. 1.

Minimizing Failing Scenarios. When an error is detected, it is often possible to reproduce it with fewer threads and operations [21]. `Lincheck` automatically "minimizes" the failing scenario in a greedy way: it repeatedly removes an operation from the scenario until the test stops failing, thus finding a *minimal* failing scenario. While this approach is not theoretically-optimal, we found it working well in practice[1].

User Guide. This section focuses mainly on the technical aspects behind the `Lincheck` features. For those readers who are interested in using the framework in their project, we suggest taking a look at the official `Lincheck` guide [22].

2.1 Phase 1: Scenario Generation

`Lincheck` allows to tune the number of parallel threads, operations in them, and the number of scenarios to be generated when creating `ModelCheckingOptions` or `StressOptions`. The framework then generates a set of concurrent scenarios by filling threads with randomly picked operations (annotated with `@Operation`) and generating (by default random) arguments for these operations.

Operation Arguments. Consider testing a concurrent hash table. If it has a bug, it is more likely to be detected when accessing the same element concurrently. To increase the probability of such scenarios, users can narrow the range of possible elements passed to the operations; Listing 2 illustrates how to configure the test in a way so the generated elements are always between `1` and `3`.

```
1 @Param(name = "elem", gen = IntGen::class, conf = "1:3")
2 @OpGroupConfig(name="writer", nonParallel=true)
3 class SingleWriterHashSetTest {
4  val s = SingleWriterHashSet<Int>()
5
6  @Operation(group = "writer"
 ) // never executes concurrently
7  fun add(@Param(name = "elem") e: Int) = s.add(e)
8  @Operation
9  fun contains(@Param(name = "elem")
   e: Int) = s.contains(e)
```

[1] Finding the *minimum* failing scenario is a highly complex problem, as it could be not based on any of the generated scenarios.

```
10  @Operation(group = "writer"
  ) // never executes concurrently
11  fun remove(@Param(name = "elem")  e: Int) = s.remove(e)
12
13  @Test fun runTest() = ModelCheckingOptions()
14                          .check(this::class)
15 }
```

Listing 2. Testing single-writer set with custom argument generation (highlighted with yellow) and single-writer workload constraint (highlighted with red).

Workload Constraints. Some data structures may require a part of operations not to be executed concurrently, such as single-producer/consumer queues. `Lincheck` provides out-of-the-box support for such constraints, generating scenarios accordingly. The framework API requires grouping such operations and restricting their parallelism; Listing 2 illustrates how to test a single-writer set.

2.2 Phase 2: Scenario Running

`Lincheck` uses stress testing and model checking to examine generated scenarios. The stress testing mode was influenced by JCStress [3], but `Lincheck` automatically generates scenarios and verifies outcomes, while JCStress requires listing both scenarios and correct results manually. The main issue with stress testing is the complexity of analysing a bug after detecting it. To mitigate this, `Lincheck` supports bounded model checking, providing detailed traces that reproduce bugs, similar to the one in Fig. 1. The rest of the subsection focuses on the model-checking approach, discussing the most significant details.

Bounded Model Checker. The model-checking mode has drawn inspiration from the CHESS (also known as Line-Up) framework for C# [5]. It assumes the sequentially consistent memory model and evaluates all possible schedules with a limited number of context switches. Unlike CHESS, `Lincheck` bounds the number of schedules rather than context switches, which makes testing time independent of scenario size and algorithm complexity.

In some cases, the specified number of schedules may not be enough to explore all interleavings, so `Lincheck` studies them evenly, probing logically different scenarios first. For instance, imagine a case where Lincheck is analyzing interleavings with a single context switch and has previously explored only one interleaving, which originated from the first thread containing four atomic operations. Under these circumstances, Lincheck presumes that 25% of the interleavings have been explored when starting from the first thread, while the second thread remains unexplored. As a result, Lincheck becomes more inclined to select the second thread as the starting point for the next exploration.

Switch Points. To control the execution, `Lincheck` inserts internal method calls at shared memory accesses by on-the-fly byte-code transformation via ASM framework [23]. These internal methods serve as *switch points*, enabling manual context switching. Notably, `Lincheck` supports shared memory access through

AtomicFieldUpdater, VarHandle, and Unsafe and handles built-in synchronization via MONITORENTER/MONITOREXIT, park/unpark, and wait/notify. Internally, it replaces there synchronization primitives with custom implementations, thus, enabling full control of the execution.

Progress Guarantees. While exploring potential switch points, Lincheck can detect active synchronization, handling it similarly to locks. This capability to detect blocking code enables Lincheck to verify the testing algorithm for *obstruction-freedom*[2], the weakest non-blocking guarantee [10]. Although more popular lock- and wait-freedom are part of Lincheck's future plans, the majority of practical liveness bugs are caused by unexpected blocking code, making the obstruction-freedom check fairly useful for lock-free and wait-free algorithms.

Optimizations. Lincheck uses various heuristics to speed up the analysis and increase the coverage. The most impactful one excludes final field accesses from the analysis, as their values are unchanging. Our internal experiments indicate a reduction in the number of inserted switch points by over $\times 2$ in real-world code. Another important optimization tracks objects that are not shared with other threads, excluding accesses to them from the analysis. This heuristic eliminates an additional 10–15% of switch points in practice.

Happens-Before. When an operation starts, Lincheck collects which operations from other threads are already completed to establish the "happens-before" relation; this information is further passed to the results verifier.

Modular Testing. When constructing new algorithms, it is common to use existing non-trivial data structures as building blocks. Considering such underlying data structures to be correct and treating their operations as atomic may significantly reduce the number of possible interleavings and check only meaningful ones, thus increasing the testing quality. Lincheck makes it possible with the modular testing feature; please read the official guide for details [22].

Limitations. For the model checking mode, the testing data structure must be deterministic to ensure reproducible executions, which is a common requirement for bug reproducing tools [24]. For the algorithms that utilize randomization, Lincheck offers out-of-the-box support by fixing seeds for Random; thus, making the latter deterministic. To our experience, Random is the only source of non-determinism in practical concurrent algorithms.

Model Checking vs Stress Testing. The primary benefit of using model checking is obtaining a comprehensive trace reproducing the detected error, as demonstrated in Fig. 1. However, the current implementation assumes the sequentially consistent memory model, which can result in missed bugs caused by low-level effects, such as an omitted volatile modifier in Java. We are in the process of incorporating the GenMC algorithm [6,25] to support weak memory models and increase analysis coverage through the partial order reduction

[2] The *obstruction-freedom* property ensures that any operation completes within a limited number of steps if all other threads are stopped.

technique. In the meantime, we suggest using stress testing in addition to model checking.

2.3 Phase 3: Verification of Outcome Results

Once the scenario is executed, the operation results should be verified against the specified correctness property, which is linearizability [9] by default. In brief, `Lincheck` tries to match the operation results to a sequential history that preserves the order of operations in threads and does not violate the "happens-before" relation established during the execution.

LTS. Instead of generating all possible sequential executions, `Lincheck` lazily builds a *labeled transition system (LTS)* [26] and tries to explain the obtained results using it. Roughly, LTS is a directed graph, which nodes represent the data structure states, while edges specify the transitions and are labeled with operations and their results. Execution results are considered valid if there exists a finite path in the LTS (i.e., sequential history) that leads to the same results. `Lincheck` lazily builds LTS by invoking operations on the testing data structure in one thread. Thus, the sequential behavior is specified implicitly. Figure 2 illustrates an LTS lazily constructed by `Lincheck` for verifying incorrect results of `ConcurrentLinkedDeque` from Fig. 1.

Sequential Specification. By default, `Lincheck` sequentially manipulates the testing data structure to build an LTS. It is possible to specify the sequential behavior explicitly, providing a separate class with the same methods as those annotated with `@Operation`. It allows for a single `Lincheck` test instead of separate sequential and concurrent ones. For API details, please refer to the guide [22].

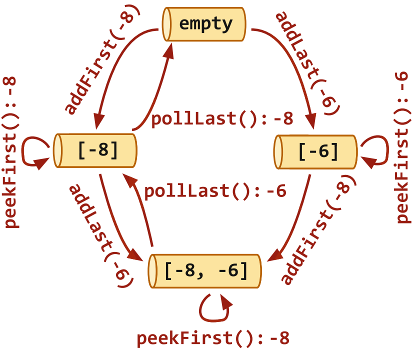


Fig. 2. An LTS constructed for verifying `ConcurrentLinkedDeque` results from Fig. 1.

Validation Functions. It is possible to validate the data structure invariants at the end of the test, adding the corresponding function and annotating it with `@Validate`. For example, we have uncovered a memory leak in the algorithm for removing nodes from a concurrent linked list in [27] by validating that logically removed nodes are unreachable at the end.

Linearizability Relaxations. Additionally to linearizability, `Lincheck` supports various relaxations, such as quiescent consistency [10], quantitative relaxation [11], and quasi-linearizability [12].

Blocking Operations. Some structures are blocking by design, such as the case of `Mutex` or `Channel`. Consider a *rendezvous channel*, also known as "synchronous

queue”, as an example: senders and receivers perform a rendezvous handshake as a part of their protocol (senders wait for receivers and vice versa). If we run `send(e)` and `receive()` in parallel, they both succeed. However, executing the operations sequentially will result in suspending the first one. To reason about correctness, the *dual data structures* formalism [28] is usually used. Essentially, it splits each operation into two parts at the point of suspension, linearizing these parts separately. We extend this formalism by allowing suspended requests to cancel and by making it more efficient for verification.

3 Evaluation

`Lincheck` has already gained adoption in Kotlin and Java communities, as well as by companies and universities. It has been integrated into the development processes of Kotlin Coroutines [16] and JCTools [17], enabling reliable testing of their core data structures, and was used to find several new bugs in popular concurrency libraries and algorithms published at top-tier conferences. Furthermore, for over five years, we have successfully used `Lincheck` in our “Parallel Programming” course to automate the verification of more than 4K student solutions per year. Notably, many users appear to especially appreciate `Lincheck`’s low entry threshold and its ability to “explain” errors with detailed traces.

Novel Bugs Discovered with **`Lincheck`*.*** We have uncovered multiple new concurrency bugs in popular libraries and authors’ implementations of algorithms published at top conferences. These bugs are listed in Table 1 and include some found in the standard Java library. `Lincheck` not only detects non-linearizability and unexpected exception bugs, but also liveliness issues. For example, it identified an obstruction-freedom violation in Java’s `AbstractQueuedSynchronizer` framework, which is a foundation for building most synchronization primitives in the standard Java library.

Notably, the tests that uncover the bugs listed in Table tab1 are **publicly available** [18], allowing readers to easily reproduce these bugs.

Running Time Analysis. We have designed `Lincheck` for daily use and expect it to be fast enough in interactive mode. Various factors, including the complexity of the testing algorithm and the number of threads, operations, and invocations, can impact its performance. We suggest using two configurations for the best user experience and robustness: a *fast* configuration for local builds to catch simple bugs quickly and a *long* configuration to perform a more thorough analysis on CI/CD (Continuous Integration) servers:

- **Fast:** 30 scenarios of 2 threads $\times$ 3 operations, 1000 invocations per each;
- **Long:** 100 scenarios of 3 threads $\times$ 4 operations, 10000 invocations per each.

We assess the performance and reliability of `Lincheck` with these *fast* and *long* configurations by measuring the testing times and showing whether the expected bugs were detected. We run the experiment on the buggy algorithms listed in Table 1, along with `ConcurrentHashMap` and `ConcurrentLinkedQueue` from

Table 1. Novel bugs discovered with `Lincheck`; tests are publicly available [18].

Source	**Data structure**	**Description**
Java	`ConcurrentLinkedDeque`	Non-linearizable [13]; see Fig. 1
Java	`AbstractQueuedSynchronizer`	Liveliness error
Kotlin Coroutines [16]	`Mutex`	Liveliness error [20]
JCTools [17]	`NonBlockingHashMapLong`	Non-linearizable [19]
Concurrent-Trees [29]	`ConcurrentRadixTree`	Non-linearizable [30]
PPoPP'10 [31]	`SnapTree`	Unexpected internal exception
PPoPP'14 [32]	`LogicalOrderingAVL`[a]	Deadlock
ISPDC'15 [33]	`CATree`	Deadlock
Euro-Par'17 [34]	`ConcurrencyOptimalTree`	Unexpected internal exception

[a] The deadlock in the `LogicalOrderingAVL` algorithm was originally found by Trevor Brown and later confirmed with `Lincheck`.

Table 2. Running times of `Lincheck` tests with *fast* and *long* configurations using both stress testing and model checking (MC) for the listed data structures. Failed tests, which detect bugs, are highlighted with **red**. Notably, finding a bug may take longer than testing a correct implementation due to scenario minimization.

Data Structure	Fast Configuration		Long Configuration	
	Stress	MC	Stress	MC
`ConcurrentHashMap` (Java)	0.3 s	2.7 s	38.1 s	1 m 44 s
`ConcurrentLinkedQueue` (Java)	0.4 s	1.7 s	1 m 26 s	1 m 41 s
`LockFreeTaskQueue` (Kotlin Coroutines)	1.1 s	1.4 s	39.6 s	54.8 s
`Semaphore` (Kotlin Coroutines)	2.1 s	3.6 s	22.3 s	1 m 44 s
`ConcurrentLinkedDeque` (Java)	0.4 s	**1.2 s**	**19.7 s**	**10.7 s**
`AbstractQueueSynchronizer` (Java)	1.6 s	**0.5 s**	18.2 s	**8.6s**
`Mutex`(Kotlin Coroutines)	0.9 s	**2.6 s**	23.6 s	**8.7 s**
`NonBlockingHashMapLong` (JCTools)	**0.6 s**	**1.3 s**	**4.4 s**	**7 s**
`ConcurrentRadixTree` ([29])	2.9 s	10.6 s	40.9 s	**2 m 30 s**
`SnapTree` [31]	1.7 s	5.8 s	38.4 s	**5 m 6 s**
`LogicalOrderingAVL` [32]	1.5 s	4.2 s	17.1 s	**36.9 s**
`CATree` [33]	**20.1 s**	**0.8 s**	**41.3 s**	**6.5 s**
`ConcurrencyOptimalTree` [34]	**0.4 s**	**1.5 s**	**3 s**	**7.3 s**

the Java standard library and a quasi-linearizable `LockFreeTaskQueue` with `Semaphore` from Kotlin Coroutines. The results are available in Table 2. The experiment was conducted on a Xiaomi Mi Notebook Pro 2019 with Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz and 32 Gb RAM. The results show that the *fast* configuration ensures short running times, being suitable for use as unit tests without slowing down the build and able to uncover some bugs. However,

some bugs are detected only with the *long* configuration, emphasizing the need for more operations and invocations to guarantee correctness. Despite this, the running time remains practical and acceptable.

4 Related Work

Several excellent tools for linearizability testing and model checking have been proposed, e.g. [4,5,35–41], and some even support relaxed memory models [6,25,42,43] and linearizability relaxations [36,44]. Due to space limitations, we focus our discussion on the works that shaped `Lincheck`.

Inspiration. `Lincheck` was originally inspired by the JCStress [3] tool for JVM, which is designed to test the memory model implementation. However, JCStress does not offer a declarative approach to writing tests. The bounded model checker in `Lincheck` was influenced by CHESS (Line-Up) [5] for C#, which is also non-declarative and does not support linearizability extensions. `Lincheck` offers several novel features and usability advantages compared to these inspirations, making it a versatile platform for research in testing and model checking. Although other tools such as GenMC [6,25,43] have superior features, `Lincheck` is designed to be extensible and can integrate new tools. In particular, we are working on incorporating the GenMC algorithm into `Lincheck` at the moment of writing this paper.

Lincheck Compared to Other Solutions. To the best of our knowledge, no other tool offers similar functionality. In particular, `Lincheck` allows certain operations to never execute in parallel (supporting single-producer/consumer constraints), detects obstruction-freedom violations (which is crucial for checking non-blocking algorithms), provides a way to specify sequential behavior explicitly (enabling oracle-based testing), and supports blocking operations for Kotlin Coroutines. Furthermore, `Lincheck` is a highly user-friendly framework, featuring a simple API and easy-to-understand output, which we have found users to highly appreciate.

5 Discussion

We introduced `Lincheck`, a versatile and expandable framework for testing concurrent data structures. As `Lincheck` is not just a tool but a platform for incorporating advancements in concurrency testing and model checking, we plan to integrate cutting-edge model checkers that support weak memory models. Written in Kotlin, `Lincheck` is also interoperable with native languages such as Swift or C/C++. Our goal is to extend `Lincheck` testing to these languages, making it the leading tool for checking correctness of concurrent algorithms. We believe that `Lincheck` has the potential to significantly improve the quality and efficiency of concurrent algorithms development, reducing time and effort to write reliable tests and investigate bugs.

References

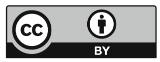
1. Lincheck - A framework for testing concurrent data structures on JVM. https://github.com/Kotlin/kotlinx-lincheck
2. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6
3. The Java Concurrency Stress tests. https://openjdk.java.net/projects/code-tools/jcstress
4. Lindstrom, G., Mehlitz, P.C., Visser, W.: Model checking real time java using java pathfinder. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 444–456. Springer, Heidelberg (2005). https://doi.org/10.1007/11562948_33
5. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. SIGPLAN Not. **42**(6), 446–455 (2007)
6. Kokologiannakis, M., Vafeiadis, V.: GenMC: a model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 427–440. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_20
7. Jung, R., et al.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: Conference Record of the Annual ACM Symposium on Principles of Programming Languages 2015, pp. 637–650 (2015)
8. Coq - a formal proof management system. https://coq.inria.fr
9. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(3), 463–492 (1990)
10. Herlihy, M., Shavit, N., Luchangco, V., Spear, M.: The art of multiprocessor programming. Newnes (2020)
11. Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In ACM SIGPLAN Notices, vol. 48, pp. 317–328. ACM (2013)
12. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: relaxed consistency for improved concurrency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 395–410. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17653-1_29
13. [JDK-8256833] ConcurrentLinkedDeque is non-linearizable. https://bugs.openjdk.java.net/browse/JDK-8256833
14. [JDK-8188900] ConcurrentLinkedDeque linearizability. https://bugs.openjdk.java.net/browse/JDK-8188900
15. [JDK-8189387] ConcurrentLinkedDeque linearizability continued. https://bugs.openjdk.java.net/browse/JDK-8189387
16. Kotlin Coroutines. https://github.com/Kotlin/kotlin-coroutines
17. JCTools - Java Concurrency Tools for the JVM. https://github.com/JCTools/JCTools
18. Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM. Zenodo (2023)
19. Race in NonBlockingHashMapLong in JCTools. https://github.com/JCTools/JCTools/issues/319
20. MutexLincheckTest.modelCheckingTest detects non lock-free execution path in Mutex #2590. https://github.com/Kotlin/kotlinx.coroutines/issues/2590
21. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 329–339 (2008)

22. Lincheck User Guide. https://kotlinlang.org/docs/lincheck-guide.html
23. ObjectWeb ASM. https://asm.ow2.io
24. Elmas, T., Burnim, J., Necula, G., Sen, K.: Concurrit: a domain specific language for reproducing concurrency bugs. ACM SIGPLAN Not. **48**, 06 (2013)
25. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 96–110. Association for Computing Machinery, New York (2019)
26. Tretmans, J.: Conformance testing with labelled transition systems: implementation relations and test generation. Comput. Netw. ISDN Syst. **29**(1), 49–79 (1996)
27. Koval, N., Alistarh, D., Elizarov, R.: Scalable fifo channels for programming via communicating sequential processes. In European Conference on Parallel Processing. Springer, Heidelberg (2019). http://pub.ist.ac.at/dalistar/Scalable_FIFO_Channels_EuroPar.pdf
28. Scherer III, W.N., Scott, M.L.: Nonblocking concurrent objects with condition synchronization. In: Proceedings of the 18th International Symposium on Distributed Computing, pp. 2121–2128 (2004)
29. Concurrent Radix and Suffix Trees for Java. https://github.com/npgall/concurrent-trees
30. Race in ConcurrentSuffixTree in Concurrent-Trees library. https://github.com/npgall/concurrent-trees/issues/33
31. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. SIGPLAN Not. **45**(5), 257–268 (2010)
32. Drachsler, D., Vechev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2014, pp. 343–356. Association for Computing Machinery, New York (2014)
33. Sagonas, K., Winblad, K.: Contention adapting search trees. In: 2015 14th International Symposium on Parallel and Distributed Computing, pp. 215–224 (2015)
34. Aksenov, V., Gramoli, V., Kuznetsov, P., Malova, A., Ravi, S.: A concurrency-optimal binary search tree, pp. 580–593 (2017)
35. Pradel, M., Gross, T.R.: Fully automatic and precise detection of thread safety violations. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 521–530 (2012)
36. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 330–340 (2010)
37. Li, G., Lu, S., Musuvathi, M., Nath, S., Padhye, R.: Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 162–180 (2019)
38. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2003, pp. 167–178. Association for Computing Machinery, New York (2003)
39. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, pp. 308–319. Association for Computing Machinery, New York (2006)
40. Sen, K.: Race directed random testing of concurrent programs. In PLDI 2008 (2008)

41. Huang, J., O'Neil Meredith, P., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. SIGPLAN Not. **49**(6), 337–348 (2014)
42. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Proceedings of the 25th International Conference on Computer Aided Verification, vol. 8044, pp. 141–157 (2013)
43. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for c/c++ concurrency. Proc. ACM Program. Lang. **2**(POPL) (2017)
44. Emmi, M., Enea, C.: Violat: generating tests of observational refinement for concurrent objects. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 534–546. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_30

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



nekton: A Linearizability Proof Checker

Roland Meyer[1], Anton Opaterny[1](✉), Thomas Wies[2], and Sebastian Wolff[2]

[1] TU Braunschweig, Braunschweig, Germany
{roland.meyer,anton.opaterny}@tu-bs.de
[2] New York University, New York, USA
{wies,sebastian.wolff}@cs.nyu.edu

Abstract. nekton is a new tool for checking linearizability proofs of highly complex concurrent search structures. The tool's unique features are its parametric heap abstraction based on separation logic and the flow framework, and its support for hindsight arguments about future-dependent linearization points. We describe the tool, present a case study, and discuss implementation details.

Keywords: separation logic · proof checker · linearizability · flow framework

1 Introduction

We present nekton, a mostly automated deductive program verifier based on separation logic (SL) [23,27]. The tool is designed to aid the construction of linearizability proofs for complex concurrent search structures. Similar to many other SL-based tools [2,8,14,22,33,33], nekton uses an SMT solver to automate basic SL reasoning. Similar to the original implementation of CIVL [7], it uses non-interference reasoning à la Owicki-Gries [25] to automate thread modularity. What makes nekton stand out among these relatives is its inbuilt support for expressing complex inductive heap invariants using the flow framework [12,13,20] and the ability to (partially) automate complex linearizability arguments that require hindsight reasoning [4,5,15,18,19,24]. Together, these features enable nekton to verify challenging concurrent data structures such as the FEMRS tree [4] with little user guidance.

nekton [17] is derived from the tool plankton [18,19], which shares the same overall goals and features as nekton but strives for full proof automation at the expense of generality. In terms of the trade-off between automation and expressivity, nekton aims to occupy a sweet spot between plankton and general purpose program verifiers. In the following, we discuss nekton's unique features in more detail and explain how it deviates from plankton's design.

The flow framework can be used to express global properties of graph structures in a node-local manner, aiding compositional verification of recursive data structures. The framework is parametric in a *flow domain* which determines what global information about the graph is provided at each node. Various flow domains have been proposed that have shown to be useful in concurrency proofs [11,26]. To simplify proof automation, plankton uses a fixed flow domain that is geared towards verifying functional correctness of search structures. In contrast, nekton is parametric in the flow domain. For instance, it supports custom domains for reasoning about overlayed

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 170–183, 2023.
https://doi.org/10.1007/978-3-031-37706-8_9

structures and other data-structure-specific invariants. This design choice significantly increases the expressivity of the tool at the cost of a mild increase in the annotation burden for the user. For instance, the FEMRS tree case study that we present in this paper relies on a flow domain that is beyond the scope of `plankton`. In fact, the flow domain is also beyond state-of-the-art abstract interpretation-based verification tools checking linearizability [1]. However, computing relative to a given flow domain is considerably more difficult than computing with a hard-coded one: it requires parametric versions for (1) computing post images, (2) checking entailment, and (3) checking non-interference. Yet, it allows for sufficient automation compared to general user-defined (recursive) predicates as accepted by, e.g., Viper [22] and VeriFast [9].

The second key feature of `nekton` is its support for *hindsight reasoning*. Intuitively, hindsight arguments rely on statements of the form "if q holds in the current state and p held in some past state, then r must have held in some intermediate state". Such arguments can greatly simplify the reasoning about complex concurrent algorithms that involve future-dependent linearization points. At a technical level, hindsight reasoning is realized by lifting a state-based separation logic to one defined over computation histories [18,19]. `nekton`'s support for this style of reasoning goes beyond the simple hindsight rule in [18] but does not yet implement the general *temporal interpolation* rule introduced more recently in [19], which is already supported by `plankton`.

These features set `nekton` apart from its competitors. First, it offers more expressivity compared to tools with a higher degree of automation like `plankton` [18,19], `Cave` [29–31], and `Poling` [34]. Second, it's proofs require less annotation effort than more flexible refinement-proofs for fine-grained concurrency, like those of `CIVL` [7,10] and `Armada` [16]. Last, it integrates techniques for proving linearizability, which are missing in industrial grade tools like `Anchor` [6].

In the remainder of this paper, we provide a high-level overview of the tool (Sect. 2), present a case study (Sect. 3), and discuss implementation details some of which also concern `plankton` and have not yet been reported on before (Sect. 4).

2 Input

`nekton` checks the correctness of proof outlines for the linearizability of concurrent data structures. Its distinguishing feature compared to its ancestor `plankton` is that the heap abstraction is not hard-coded inside the tool, but taken as an input parameter. That is, `nekton`'s input is a *heap abstraction* and a set of *proof outlines*, one for each function manipulating the data structure state. The heap abstraction defines how the data structure's heap representation is mapped onto a labeled graph that captures the properties of interest and that can then be reasoned about in separation logic. It also embeds the mechanism for checking linearizability.

`nekton` works with the recent flow graphs proposed by Krishna et al. [12,13], in their latest formulation due to [18]. Flow graphs augment heap graphs with ghost state. The ghost state can be understood as a certificate formulating global properties of heap graphs in a node-local manner. It takes the form of a so-called flow value that has been propagated through the heap graph and, therefore, brings global information with it. The propagation is like in static analysis, except that we work over heap graphs rather than control-flow graphs. To give an example, assume we want to express the global property

that the heap graph is a tree. A helpful certificate would be the path count, the number of paths from a distinguished root node to the node of interest. It allows us to formulate the tree property node-locally, by saying that the path count is always at most one.

Our first input is a flow domain (M, gen). The parameter $(M, +, 0)$ is a commutative monoid from which we draw the flow values. The propagation needs standard fixed point theory: the natural ordering $a \leq a + b$ for $a, b \in M$ on the monoid should form an ω-complete partial order. We expect the user to specify both $+$ and $\leq$ to avoid the quantifier over the offset in the definition of $\leq$. The parameter gen generates the transfer functions labeling the edges in the heap graph. Transfer functions transform flow values to record information about the global shape. The generator has the type

$$gen : \mathsf{PointerFld} \to (\mathsf{DataFld} \to Data) \to \mathsf{Mon}(M \to M) \,.$$

We assume flow graphs distinguish between pointer fields (PointerFld) and fields that hold data values (DataFld). Flow values are propagated along every pointer field, in a way that depends on the current data values but that does not depend on the target of the field. To see that the data values are important, imagine a node has already been deleted logically but not yet physically from a data structure, as is often the case in lock-free processing. Then the logical deletion would be indicated by a raised flag (a distinguished data field), and we would not forward the current path count. To reason about flow values with SMT solvers, we restrict the allowed types of flow values to

$$M \quad ::= \quad \mathbb{B} \mid \mathbb{N} \mid \mathbb{P}(\mathbb{B}) \mid \mathbb{P}(\mathbb{N}) \mid M \times M \,.$$

Flow values are (sets of) Booleans or integers, or products over these base types. When defining a product type, the user has to label each component with a selector allowing to project a tuple onto this component. Importantly, the user can define the addition operation $+$ for the flow monoid freely over the chosen type as long as the definition is expressible within the underlying SMT theory (e.g., for $\mathbb{N}$ one may choose as $+$ the usual addition or the maximum). The tool likewise inherits the assertion language for integers and Booleans that is supported by the SMT solver. There are two more user-defined inputs that are tightly linked to the heap representation.

Linearizability. We establish the linearizability of functions manipulting a data structure with the help of the keyset framework [11,28], which we encode using flows. A crucial problem when proving linearizability are membership queries: we have to determine whether a given key has been in the data structure at some point in time while the function was running. The keyset framework localizes these membership queries from the overall data structure to single nodes. It assigns to each node n a set of keys for which n is responsible, in the sense that n has to answer the membership queries for these keys. This set of keys is n's *keyset*. Imagine we have a singly linked list

$$\mathtt{Head} \xrightarrow{(-\infty,\infty)} (n_1, 5) \xrightarrow{[6,\infty)} (n_2, 7)^{\dagger} \xrightarrow{[6,\infty)} (n_3, 10) \xrightarrow{[11,\infty)} \bot \,.$$

The shared pointer `Head` propagates the keys in the interval $(-\infty, \infty)$ as a flow value to node n_1 holding key 5. This set is called n_1's *inset*. The inset of a node n contains all keys k for which a search will reach n. If $k > 5$, the search will proceed to n_2, otherwise it will stay at n_1. Thus, the keyset of n_1 is $(-\infty, 5]$. That is, if $k \in (-\infty, 5]$, the answer

to the membership query is determined by the test $k = 5$. Node n_1 forwards $[6, \infty)$ to the successor node n_2 with key 7. Since n_2 has been logically deleted, indicated by the tombstone †, it cannot answer membership queries: the keyset is empty. Instead, the node forwards its entire inset $[6, \infty)$ to node n_3, which is now responsible for the keyset $[6, 10]$. We speak of a framework because whether a given key k belongs to a node's keyset or whether it is propagated to one of the node's successors is specific to each data structure, but the way in which the linearizability argument for membership queries is localized to individual flow graph nodes is always the same.

In nekton, the user can define $\mathbb{P}(\mathbb{N})$ for sets of keys as (a component in) the flow domain of interest. With parameter gen, they can implement the propagation. We also provide flexibility in the definition of the keyset and membership queries in the form of two predicates rsp (responsible) resp. $cnts$ (contains). To give an example, we would define

$$rsp(x, k) \triangleq k \in x \rightarrow \texttt{flow.is} * k \leq x \rightarrow \texttt{key} * \neg x \rightarrow \texttt{marked} .$$

With $x \rightarrow \texttt{flow}$, we denote x's flow value. The flow domain is a product, and we refer to the component called is. With $x \rightarrow \texttt{key}$ and $x \rightarrow \texttt{marked}$ we denote the x's key and marked fields. Formally, the dereference notation is a naming convention for logical variables that refer to values of resources defined in the node-local invariant explained below. Reconsider the example and let $k = 6$. The key belongs to the inset $[6, \infty)$ that n_2 receives from n_1. We discussed that the node's keyset is empty, and indeed $rsp(n_2, 6)$ is false. For n_3, we have $rsp(n_3, 6)$ true. With the predicate rsp in place, we can also refer to n.keyset in assertions.

For verifying functions with non-fixed linearization points, nekton implements the hindsight principle [24]. Reasoning with that principle goes as follows. We record information about bygone states of the data structure in past predicates $\diamondminus a$. For example, $\diamondminus(k \in x \rightarrow \texttt{flow.is})$ says that the key of interest was in the node's inset at some point while the function was running. Moreover, the assertion about the current state may tell us that the key is smaller than the key held by the node and that the node is not marked now, $k \leq x \rightarrow \texttt{key} * \neg n \rightarrow \texttt{marked}$. Then the hindsight principle will guarantee that there has been a state in between the two moments where the node still had the key in its inset, the inequality held true, and the node was unmarked. This is $\diamondminus\, rsp(n, k)$ as defined above. To draw this conclusion, the hindsight principle inspects the interferences the data structure state may experience from concurrently executed functions. In the example, no interferene can unmark a node or change a key. So the predicates encountered in the current state must have held already in the past state when $k \in x \rightarrow \texttt{flow.is}$ was true. This form of hindsight reasoning is stronger than the one in [18] but not yet as elaborate as the one in [19]. From a program logic point of view, hindsight reasoning relies on a lifting of state-based to computation-based separation algebras [18].

Implications. Reasoning about automatically generated transfer functions is difficult, in particular when they relate different components in a product flow domain. Consider $\mathbb{N} \times \mathbb{P}(\mathbb{N})$ with the first component the path count at a node and the second component the keyset. The transfer functions will never forget to count a path, and so the following implication will be valid over all heap graphs:

$$(x \rightarrow \texttt{flow.pcount}) = 0 \quad \Longrightarrow \quad (x \rightarrow \texttt{flow.keyset}) = \varnothing . \qquad (1)$$

Despite the help of an SMT solver, `nekton` will fail to establish the validity of such an implication. Therefore, the user may input a set of such formulas that the tool will then take for being valid without further checks. Correctness of a proof is always relative to this set of implications.

2.1 Proof Outlines

A concurrent data structure consists of a set of structs defining the heap elements and a set of functions for manipulating the data structure state. `nekton` expects as input a proof outline for each such function. The program logic implemented by `nekton` is an Owicki-Gries system that, besides partial correctness, requires interference freedom of the given proof outlines. The user is expected to give the interferences as input.

The proof outlines accepted by `nekton` take the form $\{\, pre \,\}\ po\ \{\, post \,\}$ with

$$po \ ::= \ \texttt{com} \mid \{\, a \,\} \mid po \,;\, po \mid (po + po)\ \{\, a \,\} \mid \{\, a \,\}\ po^{*}\ \{\, a \,\} \mid \texttt{atomic}\ po\ .$$

The proof outlines are partial in that intermediary assertions, say in $\texttt{com}_1\ ;\ \texttt{com}_2$, may be omitted. `nekton` will automatically generate the missing information using strongest postconditions. What has to be given are loop invariants and unifying assertions for the different branches of if-then-else statements. Consecutive assertions $\{\, a \,\}\ ;\ \{\, b \,\}$ are interpreted as a weaking of a to b.

Programs are given in a dialect of `C`. Commands are assignments to/from variables and memory locations, allocations, assumptions, and acquires/releases of locks

$$\begin{aligned} \texttt{com} \ ::= \ & \texttt{p} := \texttt{q} \mid \texttt{p}\!\rightarrow\!\texttt{fld} := \texttt{q} \mid \texttt{p} := \texttt{q}\!\rightarrow\!\texttt{fld} \mid \texttt{p} := \texttt{malloc} \\ \mid\ & \texttt{assume(cond)} \mid \texttt{acquire(p}\!\rightarrow\!\texttt{fld)} \mid \texttt{release(p}\!\rightarrow\!\texttt{fld)}\ . \end{aligned}$$

Here, `p`, `q` are program variables, `fld` is a field name, and dereferences are denoted by an arrow. The language is strictly typed with base types `void`, `bool`, and `int`. The latter represents the mathematical integers, i.e., has an infinite domain. We admit the usual conditions over the base types. Using the `struct` keyword users can specify their own types. In addition, `nekton` supports syntactic sugar like if-then-else, (do-)while loops, non-recursive macros, break and return statements, assertions, simultaneous assignments, and compare-and-swaps. These can be expressed in terms of the core language in the expected way.

The assertion language is a standard separation logic defined over the base types, heap graphs, and the given flow domain. It has the separating conjunction and classical implication (no magic wand). Our heap model is divided into a local and a shared heap, and we use the box operator $\boxed{\text{a}}$ to indicate assertions over the shared state. The shared state is represented by an iterated separating conjunction. Since this conjunction refers to a set of nodes and we want to reason first-order, we handle it implicitly. We let each assertion a in a proof outline stand for $\exists \boldsymbol{x}.\ a * \mathop{\Huge *}_{n \in \mathbb{N} \setminus Nodes(a)} NInv(n)$. The iterated separating conjunction is over all nodes that do not occur in a, and asserts a node-local invariant for each of them. The existential quantifier is over all logical variables in the assertion. Keeping it implicit makes the assertions more concise and aids automation.

Node Invariants. `nekton` expects the node-local invariant $NInv(n)$ as another input. The role of this invariant is to make use of the flow framework and state global properties of the data structure in a local way. The invariant would say, for instance, that

sentinel nodes are never marked. Compared to the implication list, the node-local invariant has the advantage that its claims are actually checked. Technically, the node-local invariant is a separation logic formula that is only allowed to refer to the given node n and its fields. It will often define logical variables like $n{\rightarrow}\texttt{flow}$ that refer to the entry of the flow field and can be used outside the node-local invariant. These variables are quantified away by $\exists \boldsymbol{x}$ above.

Interferences. Interferences are RGSep actions [32] restricted to the format

$$NInv(x).\ \{\ a\ \} \rightsquigarrow [\texttt{fld}_1, \ldots, \texttt{fld}_n]\{\ b\ \}\ . \tag{2}$$

To give an example, we formulate that a concurrently executed function may mark a node using the action $NInv(x).\ \{\ \neg(x{\rightarrow}\texttt{marked})\ \} \rightsquigarrow [\texttt{marked}]\{\ x{\rightarrow}\texttt{marked}\ \}$. An action refers to a single node in the heap graph as described by the above node-local invariant. The action applies if the assertion a evaluates to true, and modifies the node in a way that satisfies b. Like the invariant, the assertions a and b have to be node-local and only refer to the values of x's fields. The assertions may introduce logical variables that are implicitly existentially quantified and whose scope extends over a and b. Such variables allow us to relate the pre- and post-state of the interference. The fields given in the brackets are the ones that may change under the action. If assertion b does not refer to the value of a field that is given in the list, the field may receive arbitrary values. If a field is not named, it is guaranteed to stay unchanged.

3 Case Study

We present a linearizability proof of the FEMRS tree [4] conducted with `nekton`. We omit the data structure's maintenance operation because it leads to flow updates that neither `nekton` nor another state-of-the-art technique aimed at automation can handle. Each node in the tree stores one key and points to up to two child nodes `left` and `right`, storing keys with lower and higher values, respectively. In addition, each node contains two Boolean fields `del` and `rem` for the removal of nodes. This is because the tree distinguishes the logical removal, indicated by the `del` flag, from the physical unlinking of a node, indicated by the `rem` flag. As long as a logically removed node has not been unlinked, it can become part of the tree again. The idea is to save the creation of new nodes for keys that are physically but no longer logically part of the tree. Lastly, every node can be locked.

Figure 1 depicts a possible state of the FEMRS tree. Each node is labeled with its key. Dashed nodes have been logically removed. To prove linearizability, we rely on the keyset framework. The inset flow is used to define the keysets, as explained earlier. The edges in the figure are labeled with the flow they propagate. The transfer functions leading to this propagation stem from the following generator *gen*:

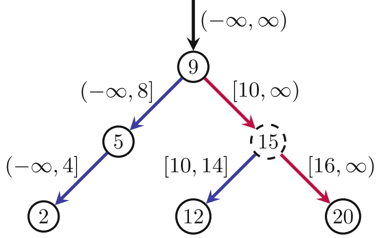


Fig. 1. A state of the FEMRS tree.

$$gen(\texttt{fld}) \ \triangleq\ \lambda f.\ x{\rightarrow}\texttt{del}\ ?\ f\ :\ f \setminus \big(\texttt{fld} = \texttt{left}\ ?\ [x{\rightarrow}\texttt{key}, \infty) : (-\infty, x{\rightarrow}\texttt{key}]\big)\ .$$

The predicates defining the keyset and membership are

$$\begin{aligned} rsp(x,k) \triangleq\ & k \in x{\to}\texttt{flow.is} * k = x{\to}\texttt{key} \\ & \vee\, k \in x{\to}\texttt{flow.is} * k < x{\to}\texttt{key} * x{\to}\texttt{left} = \texttt{nil} \\ & \vee\, k \in x{\to}\texttt{flow.is} * k > x{\to}\texttt{key} * x{\to}\texttt{right} = \texttt{nil} \\ cnts(x,k) \triangleq\ & k \in x{\to}\texttt{flow.is} * k = x{\to}\texttt{key} * \neg x{\to}\texttt{del}\ . \end{aligned}$$

In the example, $rsp(5,7)$, $rsp(15,15)$, $rsp(20,17)$, $cnts(12,12)$ and more hold.

The set of interferences expresses this: (I1) As long as the lock of the node is not held by the thread under consideration and as long as the node has not been marked unlinked, the child pointers and the (logical and physical) removal flags may change arbitrarily. The proof does not rely, e.g., on the fact that the `rem` flag is raised only once and only when the `del` flag is true. (I2) A lock that is not held by the thread may change arbitrarily. (I3) A node that is being physically unlinked ceases to receive flow. The following `nekton` actions formalize this:

$$NInv(x).\{x{\to}\texttt{lock}{\neq}\texttt{owned} * \neg x{\to}\texttt{rem}\} \rightsquigarrow [\texttt{left},\texttt{right},\texttt{del},\texttt{rem}]\{\, true \,\} \tag{I1}$$

$$NInv(x).\{x{\to}\texttt{lock}{\neq}\texttt{owned}\} \rightsquigarrow [\texttt{lock}]\{\, true \,\} \tag{I2}$$

$$NInv(x).\{x{\to}\texttt{lock}{\neq}\texttt{owned} * x{\to}\texttt{flow.is}{\neq}\varnothing * x{\to}\texttt{rem}\} \rightsquigarrow [\texttt{is}]\{x{\to}\texttt{flow.is}{=}\varnothing\}. \tag{I3}$$

We prove the linearizability of the functions $\texttt{contains}(k)$, $\texttt{insert}(k)$, and $\texttt{remove}(k)$. All of them call the auxiliary function $\texttt{locate}(k)$, which returns the last edge it traversed during a search for key k. Figure 2 gives the proof outline of `locate`. The proof for the full implementation can be found in [17].

We use a product flow domain $\mathbb{P}(\mathbb{N}) \times \mathbb{N}$. The first component is the inset flow with the generator function discussed above. The second component is the pathcount, whose $gen()$ simply yields the identity for all edges. The benefit of the product flow is that we can prove memory safety on the side, while conducting the linearizability proof.

In the node-local invariant, we introduce logical variables like $x{\to}\texttt{left}$ to make the proof more readable. We refer to these variables in the generator function. The invariant for the node pointed to by the shared `Root` differs from that of the remaining nodes:

$$\begin{aligned} NInv(x) \ \triangleq\ & x \mapsto \langle\, \texttt{flow} = (x{\to}\texttt{flow.is},\ x{\to}\texttt{flow.pcount}), \\ & \qquad \texttt{left} = x{\to}\texttt{left},\ \texttt{right} = x{\to}\texttt{right},\ \texttt{key} = x{\to}\texttt{key}, \\ & \qquad \texttt{lock} = x{\to}\texttt{lock},\ \texttt{del} = x{\to}\texttt{del},\ \texttt{rem} = x{\to}\texttt{rem}\,\rangle \\ & * \ NInv_{\texttt{all}}(x) * (x = \texttt{Root} \Rightarrow NInv_{\texttt{Root}}(x)) \\ NInv_{\texttt{Root}}(x) \ \triangleq\ & x{\to}\texttt{key} = -\infty * \neg x{\to}\texttt{del} * \neg x{\to}\texttt{rem} \\ & * \ x{\to}\texttt{flow.is} = (-\infty,\infty) \ * x{\to}\texttt{flow.pcount} = 1 \\ NInv_{\texttt{all}}(x) \ \triangleq\ & (\neg x{\to}\texttt{rem} \Rightarrow x{\to}\texttt{key} \in x{\to}\texttt{flow.is}) * x{\to}\texttt{flow.pcount} < 3 \\ & * \ (x{\to}\texttt{rem} \Rightarrow x{\to}\texttt{del}) * (x{\to}\texttt{left} = x{\to}\texttt{right} \Rightarrow x{\to}\texttt{left} = \texttt{nil})\ . \end{aligned}$$

The node-local invariant makes the expected claims. The root has key $-\infty$, is neither logically deleted nor unlinked, has as incoming keys $(-\infty,\infty)$ and the pathcount is 1.

```
1   { −∞ < k < ∞ ∗ NInv(Root) }
2   inline <Node*, Node*> locate(data_t k) {
3     Node* p, c; p = Root; c = Root;
4     { p = c = Root ∗ NInv(Root) ∗ ⟐[NInv(c) ∗ k ∈ c→flow.is] ∗ −∞ < k < ∞ }
5     do { NInv(p) ∗ (NInv(c) ∗ ⟐[NInv(c) ∗ k ∈ c→flow.is] ∨ ⟐[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
6     { { NInv(p) ∗ NInv(c) ∗ ⟐[NInv(c) ∗ k ∈ c→flow.is] ∗ c→key ≠ k }
7       p = c;
8       if (p→key < k) {
9         assert(p→right = nil || p→right ≠ nil);
10        c = p→right;
11        { NInv(p) ∗ (NInv(c) ∨ ⟐[NInv(p) ∗ p→right = nil] ∗ c = nil)
              ∗ ⟐[NInv(p) ∗ k ∈ p→flow.is] ∗ p→right = c ∗ p→key < k }
12        { NInv(p) ∗ (NInv(c) ∗ ⟐[NInv(c) ∗ k ∈ c→flow.is] ∨ ⟐[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
13      } else { /* symmetric to 'then' branch */ }
14      { NInv(p) ∗ (NInv(c) ∗ ⟐[NInv(c) ∗ k ∈ c→flow.is] ∨ ⟐[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
15    } while (c ≠ nil && c→key ≠ k);
16    { NInv(p) ∗ (NInv(c) ∗ ⟐[NInv(c) ∗ k ∈ c→flow.is] ∗ c→key = k
                                    ∨ ⟐[NInv(p) ∗ rsp(p, k)] ∗ c = nil) }
17    return <p, c>;
18  }
```

Fig. 2. Proof outline for `locate` as verified by `nekton`.

These flow values are established by the data structure's initialization function using an auxiliary edge with an appropriate generator. For all nodes, we have that their key is in the inflow, provided the node has not yet been unlinked, the path count is at most 3, a node has to be first logically deleted before it can be unlinked, and the only case in which the left and the right child can coincide is when they are both the null pointer. We treat `nil` as a node outside the set of nodes $\mathbb{N}$. This in particular means the node-local invariant does not apply to it. It will follow from the definition of the generator function that the keysets are disjoint. We do not need to state this in the invariant as it is only important when interpreting the verification results.

The assertion on line 9 helps our implication engine, which is designed for conjunctive assertions, deal with the disjunctions.

We explain the implication between Lines 11 and 12. It starts with the assertion $\{\ NInv(\mathtt{p}) * NInv(\mathtt{c}) * \diamondsuit[NInv(\mathtt{p}) * k \in \mathtt{p{\to}flow.is}] * \mathtt{p{\to}right} = \mathtt{c} * \mathtt{p{\to}key} < k\ \}$. To apply the hindsight principle, we derive the following guarantees from the set of interferences. A node's key is never changed. The only way a node's inset can shrink is by unlinking, after which its `left` and `right` pointers are no longer changed. The right child of `p` is not `nil` in the current state. From this information, the hindsight principle concludes $\{\ \diamondsuit[NInv(\mathtt{p}) * NInv(\mathtt{c}) * k \in \mathtt{p{\to}flow.is} * \mathtt{p{\to}key} < k * \mathtt{p{\to}right} = \mathtt{c}]\ \}$. Together with the definition of the transfer functions labeling the edges, this assertion yields $\{\ \diamondsuit[NInv(\mathtt{c}) * k \in c{\to}\mathtt{flow.is}]\ \}$. Another hindsight application starts with $\{\ NInv(\mathtt{p}) * \mathtt{c} = \mathtt{nil} * \diamondsuit[\ NInv(\mathtt{p}) * k \in \mathtt{p{\to}flow.is}\ ] * \mathtt{p{\to}right} = \mathtt{c} * \mathtt{p{\to}key} < k\ \}$

and moves the facts known in the current state into the past predicate. The definition of $rsp(x, k)$ then yields $\{\, \Diamond[NInv(\mathsf{p}) * rsp(\mathsf{p}, k)] \,\}$.

The full proof consists of 99 lines of code, 48 lines of assertions to prove them linearizable, and 56 lines of definitions for the flow domain, interferences, and invariants. nekton takes $45s$ to verify the proof's correctness on an Apple M1 Pro.

4 Correctness and Implementation

nekton checks that the verification conditions generated from the given proof outlines hold and that the assertions are interference-free. The program logic from [18, 19] then gives the following semantic guarantee: no matter how many client threads execute the data structure functions, partial correctness holds. That is, if a function is executed from a state satisfying the precondition and terminates, it must have reached a state in which the postcondition held true. Termination itself is not guaranteed. The postcondition will relate the function's return value to a statement about membership of the given key in the data structure, and the keyset framework will allow us to conclude linearizability from this relation. The verification conditions will in particular make sure the node invariant is maintained. We discuss the actual checks.

The first step is to derive and check verification conditions for all commands com. If the command is surrounded by assertions, $\{\, p \,\}$; com; $\{\, q \,\}$, the verification condition is $sp(p, \texttt{com}) \models q$, the strongest postcondition sp of p under com entails q. If the assertion $\{\, q \,\}$ is not given, nekton completes the given proof by using $q = sp(p, \texttt{com})$. The verification conditions for loops are similar. For two consecutive assertions $\{\, p \,\}$; $\{\, q \,\}$, as they occur for example at the end of a branch, the verification condition is $p \models q$.

The second step is to check that the assertions $\{\, p \,\}$ and $\{\, q \,\}$ in the proof are interference-free, i.e., cannot be invalidated by the actions of other threads.

Finally, nekton checks that the interferences given by the user cover the actual interferences of the program. We review the above steps in more detail.

Strongest Postconditions. The computation of the strongest postcondition follows the standard axioms for separation logic [23]. However, they do not deal with the flow which may not only be directly modified by com but also indirectly by an update elsewhere. To deal with such indirect updates, nekton computes a *footprint* fp: a subset of the heap locations that the standard axioms require plus those locations whose flow changes due to com. The footprint yields a decomposition $p = fp * f$ of predicate p, where f is a frame that is not affected by the update. From this decomposition, we compute the strongest postcondition as $sp(p, \texttt{com}) = sp(fp, \texttt{com}) * f$, using the frame rule. Actually, nekton also shows that the update maintains the node invariant, which only requires a check for $sp(fp, \texttt{com})$.

For fp to be a footprint wrt. com, all nodes outside fp should receive the same flow from $sp(fp, \texttt{com})$ as from fp. This holds if fp and $sp(fp, \texttt{com})$ induce the same flow transformer function [20]. To determine a footprint, nekton takes a strategy that is justified by lock-free programming [18]. Starting from the updated nodes, it gathers a small (fixed) set of locations that forms an acyclic subgraph. Acyclicity guarantees that fp and $sp(fp, \texttt{com})$ have the same transformer iff they agree on the transformation along

all paths: if n belongs to fp and $n{\to}\mathtt{fld}$ does not, then $n{\to}\mathtt{fld}$ must point to the same location and transform inflows to outflows in the same way in fp and in $sp(fp,\ \mathtt{com})$.

The strongest postcondition above is for state-based reasoning. For predicates over computations, which have state and past predicates, we use the following observation: past predicates are never invalidated by commands. This allows us to just copy them to the postcondition: $sp(p * \Diamond q,\ \mathtt{com}) = sp(p,\ \mathtt{com}) * \Diamond p * \Diamond q$. Note that we add the precondition as a new past predicate. Moreover, we may add *new* past predicates derived by hindsight arguments. As these derived past predicates are implied by the postcondition, they formally do not strengthen the assertion, but of course help the tool.

Hindsight Reasoning. Recall from Sect. 2 that hindsight reasoning draws conclusions of the form $\Diamond p * q \Rightarrow \Diamond r$: every computation from a p-state must inevitably transition through r in order to reach q. In nekton, p and q are restricted to node-local predicates in the sense defined above, and r is fixed to $p \wedge q$.

To prove the implication, assume it did not hold. Then there is a computation where p is invalidated before q is established. This is covered by the interference: there is an action act_p invalidating p and an action act_q establishing q. Let act_p and act_q be $NInv(n).\{\, o_p \,\} \rightsquigarrow [\ldots]\{\ldots\}$ resp. $NInv(n).\{\, o_q \,\} \rightsquigarrow [\ldots]\{\ldots\}$. There is (always) a decomposition $o_p = o_p^i * o_p^m$ such that o_p^i is immutable. Immutability holds if o_p^i is shared and interference-free. Consequently, o_p^i must still hold when q is established. Now, we check if o_p^i and o_q are contradictory, $o_p^i \wedge o_q \models \mathit{false}$. If so, act_q is not enabled after act_p. This, in turn, means q cannot be established after p is invalidated—the computation cannot exist. nekton draws the hindsight conclusion if it can prove the contradiction for all pairs act_p, act_q of interferences that invalidate p and establish q.

Entailment. Our assertions $p * \bigast_{i\in I} \Diamond p_i$ consist of a predicate p for the current state and a set of past predicates $\Diamond p_i$ tracking information about the computation. We have $p * \bigast_{i\in I} \Diamond p_i \models q * \bigast_{j\in J} \Diamond q_j$, if $p \models q$ and $\forall j\, \exists i.\ \Diamond p_i \models \Diamond q_j$. To show $\Diamond p_i \models \Diamond q_j$, we rely on the algorithm for state predicates and prove $p_i \models q_j$.

Entailment checks $p \models q$ between state predicates decompose into reasoning about resources and reasoning about logically pure facts. The latter degenerates to an implication in classical logic: nekton uses a straightforward encoding into SMT and discharges it with Z3 [21]. For reasoning about resources, nekton implements a custom matching procedure to correlate the resources in p and q. The procedure is guided by the program variables x: if the value of x is a in p and b in q, then a and b are matched, meaning b is renamed to a. The procedure then continues to match the fields of already matched addresses. Finally, nekton checks syntactically if all the resources in q occur in p.

If nekton fails to prove an implication, it consults the implication list. It takes the implications as they are, and does not try to embed them into a context as would be justified by congruence. nekton does not track the precise implications it has used.

Interference Freedom. A state predicate p is interference-free wrt. act of the form $NInv(n).\{\, r \,\} \rightsquigarrow [\mathtt{fld}_1, \ldots, \mathtt{fld}_n]\{\, o \,\}$, if the strongest postcondition of p under act entails p itself, $sp(p,\ act) \models p$. Towards $sp(p,\ act)$, let $p = NInv(x) * q$, meaning x is an accessible location. Applying act to x in p acts like an assignment to the fields such that their new values satisfy o. The strongest postcondition for this is standard [3]:

$$sp_x(p,\ act) \ \triangleq\ o[n \backslash x] * \exists y_1 \cdots y_n.\ (p * r[n \backslash x])[x{\to}\mathtt{fld}_1 \backslash y_1, \ldots, x{\to}\mathtt{fld}_n \backslash y_n]\ .$$

We strengthen p with the precondition r of act to make sure the action is enabled. We use $r[n\backslash x]$ for r with n replaced by x, meaning we instantiate r to location x. We replace the old values of the updated fields with fresh quantified variables and add the fields' new valuation $o[n\backslash x]$. Then, the strongest postcondition $sp(p,\ act)$ applies $sp_x(p,\ act)$ to all locations x in p.

Interference Coverage. Consider $act_1 = NInv(x).\{\, p \,\} \rightsquigarrow [\mathtt{fld}_1, \ldots, \mathtt{fld}_n]\{\, q \,\}$ and $act_2 = NInv(x).\{\, r \,\} \rightsquigarrow [\mathtt{fld}'_1, \ldots, \mathtt{fld}'_m]\{\, o \,\}$. We say that act_1 covers act_2 if act_1 can produce all updates induced by act_2. This is the case if $r \models p$, $o \models q$, and $\{\, \mathtt{fld}'_1, \ldots, \mathtt{fld}'_m \,\} \subseteq \{\, \mathtt{fld}_1, \ldots, \mathtt{fld}_n \,\}$. It remains to extract the actual interferences of the program and check if they are covered by the user-specified ones. The extraction is done while computing the strongest postcondition sp: the computed footprints fp and $sp(fp,\ \mathtt{com})$ from above reveal the updated fields as well as the pre- and post-states.

Flow Encoding. The flow monoid is not yet parsed from the user input but defined programmatically in `nekton`. The transfer function generator is parsed. `nekton` has five flow domains predefined, including path counting and keysets, which are easy to extend. `nekton` does not check whether the flow monoid is indeed a monoid and satisfies the requirements of an ω-cpo, nor whether $\leq$ coincides with the natural partial order.

The main task in dealing with a parametric rather than fixed flow domain is to encode predicates involving the flow into SMT formulas. This encoding is then used to implement the aforementioned components for strongest postconditions, hindsight, entailment, and interferences. Devising the encoding is challenging because it requires a representation of flow values that is sufficiently expressive to define relevant flow domains, yet sufficiently restricted to have efficient SMT solver support (we use `Z3` [21]). With the input format described in Sect. 2, we encode flows using the theory of integers and uninterpreted functions.

Limitations. For the future, we see several directions for extensions of our current implementation: (i) a parser for flow monoids rather than a programmatic interface, (ii) support for *partial* annotations that are automatically completed by `nekton`, (iii) the ability to prove atomic triples instead of just linearizability for sets, and (iv) more helpful error messages or counterexamples to guide the proof-writing user.

Acknowledgments. This work was funded in part by an Amazon Research Award. The work was also supported by the DFG project *EDS@SYN: Effective Denotational Semantics for Synthesis*. The fourth author is supported by a Junior Fellowship from the Simons Foundation (855328, SW).

Data Availability Statement. The `nekton` tool and case studies generated and/or analysed in the present paper are available in the Zenodo repository [17], https://doi.org/10.5281/zenodo.7931936.

References

1. Abdulla, P.A., Jonsson, B., Trinh, C.Q.: Fragment abstraction for concurrent shape analysis. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 442–471. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_16

2. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
3. Dijkstra, E.W., Scholten, C.S.: Predicate calculus and program semantics. In: Texts and Monographs in Computer Science. Springer, Heidelberg (1990). https://doi.org/10.1007/978-1-4612-3228-5
4. Feldman, Y.M.Y., Enea, C., Morrison, A., Rinetzky, N., Shoham, S.: Order out of chaos: proving linearizability using local views. In: DISC. LIPIcs, vol. 121, pp. 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.DISC.2018.23
5. Feldman, Y.M.Y., et al.: Proving highly-concurrent traversals correct. Proc. ACM Program. Lang. **4**(OOPSLA), 128:1–128:29 (2020). https://doi.org/10.1145/3428196
6. Flanagan, C., Freund, S.N.: The anchor verifier for blocking and non-blocking concurrent software. Proc. ACM Program. Lang. **4**(OOPSLA), 156:1–156:29 (2020). https://doi.org/10.1145/3428224
7. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 449–465. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_26
8. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
9. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_21
10. Kragl, B., Qadeer, S.: Layered concurrent programs. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 79–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_5
11. Krishna, S., Patel, N., Shasha, D.E., Wies, T.: Verifying concurrent search structure templates. In: PLDI, pp. 181–196. ACM (2020). https://doi.org/10.1145/3385412.3386029
12. Krishna, S., Shasha, D.E., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. Proc. ACM Program. Lang. **2**(POPL), 37:1–37:31 (2018). https://doi.org/10.1145/3158125
13. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP 2020. LNCS, vol. 12075, pp. 308–335. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44914-8_12
14. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007-2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_7
15. Lev-Ari, K., Chockler, G., Keidar, I.: A constructive approach for proving data structures' linearizability. In: Moses, Y. (ed.) DISC 2015. LNCS, vol. 9363, pp. 356–370. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48653-5_24
16. Lorch, J.R., et al.: Armada: low-effort verification of high-performance concurrent programs. In: PLDI, pp. 197–210. ACM (2020). https://doi.org/10.1145/3385412.3385971
17. Meyer, R., Opaterny, A., Wies, T., Wolff, S.: Artifact for "nekton: a linearizability proof checker" (2023). https://doi.org/10.5281/zenodo.7931936
18. Meyer, R., Wies, T., Wolff, S.: A concurrent program logic with a future and history. Proc. ACM Program. Lang. **6**(OOPSLA2), 1378–1407 (2022). https://doi.org/10.1145/3563337
19. Meyer, R., Wies, T., Wolff, S.: Embedding hindsight reasoning in separation logic. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591296

20. Meyer, R., Wies, T., Wolff, S.: Make flows small again: revisiting the flow framework. In: TACAS (1). Lecture Notes in Computer Science, vol. 13993, pp. 628–646. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_32
21. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
22. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
23. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
24. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC, pp. 85–94. ACM (2010). https://doi.org/10.1145/1835698.1835722
25. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134
26. Patel, N., Krishna, S., Shasha, D.E., Wies, T.: Verifying concurrent multicopy search structures. Proc. ACM Program. Lang. **5**(OOPSLA), 1–32 (2021). https://doi.org/10.1145/3485490
27. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
28. Shasha, D.E., Goodman, N.: Concurrent search structure algorithms. ACM Trans. Database Syst. **13**(1), 53–90 (1988). https://doi.org/10.1145/42201.42204
29. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-93900-9_27
30. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40
31. Vafeiadis, V.: RGSep action inference. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 345–361. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11319-2_25
32. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_18
33. Wolf, F.A., Schwerhoff, M., Müller, P.: Concise outlines for a complex logic: a proof outline checker for TaDA. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 407–426. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_22
34. Zhu, H., Petri, G., Jagannathan, S.: POLING: SMT aided linearizability proofs. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 3–19. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_1

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Overcoming Memory Weakness with Unified Fairness

Systematic Verification of Liveness in Weak Memory Models

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Adwait Godbole[2](✉), Shankaranarayanan Krishna[3], and Mihir Vahanwala[4]

[1] Uppsala University, Uppsala, Sweden
{parosh,mohamed_faouzi.atig}@it.uu.se
[2] University of California Berkeley, Berkeley, USA
adwait@berkeley.edu
[3] IIT Bombay, Mumbai, India
krishnas@cse.iitb.ac.in
[4] MPI-SWS, Saarbrücken, Germany
mvahanwa@mpi-sws.org

Abstract. We consider the verification of liveness properties for concurrent programs running on weak memory models. To that end, we identify notions of fairness that preclude demonic non-determinism, are motivated by practical observations, and are amenable to algorithmic techniques. We provide both logical and stochastic definitions of our fairness notions, and prove that they are equivalent in the context of liveness verification. In particular, we show that our fairness allows us to reduce the liveness problem (repeated control state reachability) to the problem of simple control state reachability. We show that this is a general phenomenon by developing a uniform framework which serves as the formal foundation of our fairness definition, and can be instantiated to a wide landscape of memory models. These models include SC, TSO, PSO, (Strong/Weak) Release-Acquire, Strong Coherence, FIFO-consistency, and RMO.

1 Introduction

Safety and liveness properties are the cornerstones of concurrent program verification. While safety and liveness are complementary, verification methodologies for the latter tend to be more complicated for two reasons. First, checking safety properties, in many cases, can be reduced to the (simple) reachability problem, while checking liveness properties usually amounts to checking repeated reachability of states [47]. Second, concurrency comes with an inherent *scheduling non-determinism*, i.e., at each step, the scheduler may non-deterministically select the next process to run. Therefore, liveness properties need to be accompanied by appropriate fairness conditions on the scheduling policies to prohibit trivial blocking behaviors [42]. In the example of two processes trying to acquire a lock, demonic non-determinism [20] may always favour one process over the other, leading to starvation.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 184–205, 2023.
https://doi.org/10.1007/978-3-031-37706-8_10

Despite the gap in complexity, the verification of liveness properties has attracted much research in the context of programs running under the classical Sequential Consistency (SC) [40]. An execution of a program under SC is a non-deterministically chosen interleaving of its processes' atomic operations. A write by any given process is immediately visible to all other processes, and reads are made from the most recent write to the memory location in question. SC is (relatively) simple since the only non-determinism comes from interleaving.

Weak memory models forego the fundamental SC guarantee of immediate visibility of writes to optimize for performance. More precisely, a write operation by a process may asynchronously be propagated to the other processes. The delay could be owed to physical buffers or caches, or could simply be a virtual one thanks to instruction reorderings allowed by the semantics of the programming language. Hence we have to contend with a (potentially unbounded) number of write operations that are "in transit", i.e., they have been issued by a process but they have yet to reach the other processes. In this manner, weak memory introduces a second source of non-determinism, namely *memory non-determinism*, reflecting the fact that write operations are non-deterministically (asynchronously) propagated to the different processes. Formal models for weak memory, ranging from declarative models [8,21,35,39,41] to operational ones [15,30,43,46] make copious use of non-determinism (non-determinism over entire executions in the case of declarative models and non-deterministic transitions in the case of operational models). While we have seen extensive work on verifying safety properties for program running under weak memory models, the literature on liveness for programs running under weak memory models is relatively sparse, and it is only recently we have seen efforts in that direction [5,36].

As mentioned earlier, we need fairness conditions to exclude demonic behaviors when verifying liveness properties. A critical issue here is to come up with an *appropriate* fairness condition, i.e., a condition that (i) is sufficiently strong to eliminate demonic non-determinism and (ii) is sufficiently weak to allow all "good" program behaviors. To illustrate the idea, let us go back to the case of SC. Here, traditional fairness conditions on processes, such as *strong fairness* [31], are too weak if interpreted naively, e.g. "along any program run, each process is scheduled infinitely often". The problem is that even though a strongly fair scheduler may pick a process infinitely often, it may choose to do so only in configurations where the process cannot progress since its guards are not satisfied. Such guards may, for instance, be conditions on the values of the shared variables. For example, executions of the program in Fig. 1 may not terminate under SC, since the second process may only get scheduled when the value of `x` is 2, thereby looping infinitely around the `do-while` loop.

Stronger fairness conditions such as transition fairness, and probabilistic fairness [11,27] can help avoid this problem. They imply that any *transition* enabled infinitely often is also taken infinitely often (with probability one in the case of probabilistic fairness). Transition fairness eliminates demonic scheduler non-determinism, and hence it is an appropriate notion of fairness in the case o SC.

```
r = 0;                                 ||  do { s = x; } until (s == 1)
while (r != 1)  {                      ||  y = 1;
    x = 1; x = 2; r = y;               ||
}                                      ||
```

Fig. 1. Does this program always terminate? Only if we can guarantee that the process to the right will eventually be scheduled to read when $\mathsf{x} = 1$.

However, it is unable to eliminate demonic memory non-determinism. The reason is that transition fairness allows runs of the programs where write events occur at a higher frequency than the frequency in which they are propagated to the processes. This means that, in the long run, a process may only see its own writes, potentially preventing its progress and, therefore, the system's progress as a whole. This scenario is illustrated in Fig. 2.

```
do { x = 1; }                          ||  do { x = 2; }
    until (x = 2 or y = 1);            ||      until (x = 1 or y = 1);
y = 1;                                 ||  y = 1;
```

Fig. 2. This program is guaranteed to terminate under any model only if pending propagation is guaranteed to not accumulate unboundedly: e.g. in TSO, each process may never see the other's writes due to an overflowing buffer.

To deal with memory non-determinism, we exploit the fact that the sizes of physical buffers or caches are bounded, and instruction reorderings are bounded in scope. Therefore, in any practical setting, the number of writes in transit at a given moment cannot be unbounded indefinitely. This is what we seek to capture in our formalism. Based on this observation, we propose three new notions of fairness that (surprisingly) all turn out to be equivalent in the context of liveness. First, we introduce *boundedness fairness* which only considers runs of the system for which there is a bound b on the number of events in transit, in each configuration of the run. Note that the value of b is arbitrary (but fixed for a given run). Bounded fairness is apposite: (i) it is sufficiently strong to eliminate demonic memory non-determinism, and (ii) it is sufficiently weak to allow all reasonable behaviors (as mentioned above, practical systems will bound the number of transient messages). Since we do not fix the value of the bound, this allows parameterized reasoning, e.g., about buffers of any size: our framework does not depend on the actual value of the bound, only on its mere existence. Furthermore, we define two additional related notions of fairness for memory non-determinism. The two new notions rely on *plain configurations*: configurations in which there are no transient operations (all the writes operations have reached all the processes). First, we consider *plain fairness*: along each infinite run, the set of plain configurations is visited infinitely often, and then define the *probabilistic* version: each run almost surely visits the set of plain configurations.

We show that the three notions of fairness are equivalent (in Sect. 4, we make precise the notion of equivalence we use).

After we have defined our fairness conditions, we turn our attention to the verification problem. We show that verifying the repeated reachability under the three fairness conditions, for a given memory model m, is reducible to the simple reachability under m. Since our framework does not perform program transformations we can prove liveness properties for program P through proving simple reachability on the same program P. As a result we obtain two important sets of corollaries: if the simple reachability problem is decidable for m, then the repeated reachability problem under the three fairness conditions are also decidable. This is the case when the memory model m is TSO, PSO, SRA, etc. Even when the simple reachability problem is not decidable for m, e.g., when m is RA, RMO, we have still succeeded to reduce the verification of liveness properties under fairness conditions to the verification of simple probability. This allows leveraging proof methodologies developed for the verification of safety properties under these weak memory models (e.g., [22,29]).

Having identified the fairness conditions and the verification problem, there are two potential approaches, each with its advantages and disadvantages. We either instantiate a framework for individual memory models one after one or define a general framework in which we can specify multiple memory models and apply the framework "once for all". The first approach has the benefit of making each instantiation more straightforward, however, we always need to translate our notion of fairness into the specific formulation. In the second approach, although we incur the cost of heavier machinery, we can subsequently take for granted the fact that the notion of fairness is uniform across all models, and coincides with our intuition. This allows us to be more systematic in our quest to verify liveness. In this paper, we have thus chosen to adopt the second approach. We define a general model of weak memory models in which we represent write events as sequences of messages ordered per variable and process. We augment the message set with additional conditions describing which messages have reached which processes. We use this data structure to specify our fairness conditions and solve our verification problems. We instantiate our framework to apply our results to a wide variety of memory models, such as RMO [12], FIFO consistency, RA, SRA, WRA [34,35], TSO [13], PSO, StrongCOH (the relaxed fragment of RC11) [30], and SC [40].

In summary, we make the following contributions

- Define new fairness conditions that eliminate demonic memory non-determinism.
- Reduce checking the repeated reachability problem under these fairness conditions to the simple reachability problem.
- Introduce a general formalism for weak memory models that allows applying our results uniformly to a broad class of memory models.

– Prove the decidability of liveness properties for models such as TSO, PSO, SRA, WRA, StrongCOH, and opening the door for leveraging existing proof frameworks for simple reachability for other models such as RA.

We give an overview of a wide landscape of memory models in Sect. 3.3, and provide a high level explanation of the versatility of our framework.

Structure of the Paper. We begin by casting concurrent programs as transition systems in Sect. 2. In Sect. 3, we develop our framework for the memory such that the desired fairness properties can be meaningfully defined across several models. In Sect. 4, we define useful fairness notions and prove their equivalence. Finally, in Sect. 5 we show how the liveness problems of repeated control state reachability reduce to the safety problem of control state reachability, and obtain decidability results. A full version of this paper is available at [6].

2 Modelling Concurrent Programs

We consider concurrent programs as systems where a set of processes run in parallel, computing on a set of process-local variables termed as *registers* and communicating through a set of *shared variables*. This inter-process communication, which consists of reads from, writes to, and atomic compare-and-swap operations on shared variables, is mediated by the *memory subsystem*. The overall system can be visualized as a composition of the process and memory subsystems working in tandem. In this section we explain how concurrent programs naturally induce *labelled transition systems*.

2.1 Labelled Transition Systems

A labelled transition system is a tuple $\mathcal{T} = \langle \Gamma, \rightarrow, \Lambda \rangle$ where Γ is a (possibly-infinite) set of configurations, $\rightarrow \subseteq \Gamma \times \Lambda \times \Gamma$ is a transition relation, and Λ is the set of labels that annotate transitions. We also refer to them as annotations to disambiguate from instruction labels. We write $\gamma \xrightarrow{l} \gamma'$ to denote that $(\gamma, l, \gamma') \in \rightarrow$, in words that there is a transition from γ to γ' with label l. We denote the transitive closure of $\rightarrow$ by $\xrightarrow{*}$, and the k-fold self-composition (for $k \in \mathbb{N}$) as $\xrightarrow{k}$.

Runs and Paths. A (possibly infinite) sequence of valid transitions $\rho = \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \cdots$ is called a run. We say that a run is a γ-run if the initial configuration in the run is γ, and denote the set of γ-runs as $\mathsf{Runs}(\gamma)$. We call a (finite) prefix of a run as a *path*. In some cases transition systems are *initialized*, i.e. an initial set $\Gamma_{\mathsf{init}} \subseteq \Gamma$ is specified. In such cases, we call runs starting from some initial configuration ($\gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \ldots$ with $\gamma_1 \in \Gamma_{\mathsf{init}}$) as initialized runs.

2.2 Concurrent Programs

The sequence of instructions executed by each process is dictated by a concurrent program, which induces a *process subsystem*. We begin by formulating the notion of a program. We assume a finite set $\mathbb{P}$ of processes that operate over a (finite) set $\mathbb{X}$ of shared variables. Figure 3 gives the grammar for a small but general assembly-like language that we use for defining the syntax of concurrent programs. A program instance, prog is described by a set of shared variables, $\mathtt{var}^*$, followed by the codes of the processes, $(\mathtt{proc}\ \mathtt{reg}^*\ \mathtt{instr}^*)^*$. Each process $p \in \mathbb{P}$ has a finite set $\mathsf{Regs}(p)$ of (local) *registers*. We assume w.l.o.g. that the sets of registers of the different processes are disjoint, and define $\mathsf{Regs}(\mathsf{prog}) := \cup_{p\in\mathbb{P}}\mathsf{Regs}(p)$. We assume that the data domain of both the shared variables and registers is a finite set $\mathbb{D}$, with a special element $\mathsf{0} \in \mathbb{D}$. The code of a process starts by declaring its set of registers, $\mathtt{reg}^*$, followed by a sequence of instructions.

```
prog  ::= var* (proc reg* instr*)*
instr ::= lbl : stmt
stmt  ::= var:=reg | reg:=var | reg:=CAS(var,reg,reg) |
reg:=expr | if reg then lbl | term
```

Fig. 3. A simple programming language.

An instruction i is of the form l : `stmt` where l is a unique (across all processes) instruction label that identifies the instruction, and `stmt` is a statement. The labels comprise the set of values the program counters of the processes may take. The problems of (repeated) instruction label reachability, which ask whether a section of code is accessed (infinitely often), are of importance to us.

Read (`reg := var`) and write (`var : = reg`) statements read the value of a shared variable into a register, and write the value of a register to a shared variable respectively. The CAS statement is the *compare-and-swap* operation which atomically executes a read followed by a write. We consider a non-blocking version of the CAS operation which returns a boolean indicating whether the operation was successful (the expected value was read and atomically updated to the new value). The write is performed only if the read matches the expected value.

We assume a set `expr` of expressions containing a set of operators applied to constants and registers without referring to the shared variables. The `reg := expr` statement updates the value of register `reg` by evaluating expression `expr`. We exact set of expressions is orthogonal to our treatment, and hence left uninterpreted. The `if`-statement has its usual interpretation, and control flow commands such as `while`, `for`, and `goto`-statements, can be encoded with branching and `if`-statements as usual.

2.3 Concurrent Programs as Labelled Transition Systems

We briefly explain the abstraction of a concurrent program as a labelled transition system. The details for the process component, i.e. evolution of the program counter and register contents, follow naturally. The key utility of this approach lies in the modelling of the memory subsystem, which we devote §3 to.

Configurations. A configuration γ is expressed as a tuple $\langle (L, R), \gamma_{\mathtt{m}} \rangle$, where L maps processes to their current program counter values, R maps registers to their current values, and $\gamma_{\mathtt{m}}$ captures the current state of the (weak) memory.

Transitions. In our model, a step in our system is either: (a) a silent memory update, or (b) a process executing its current instruction. In case (a), only the memory component $\gamma_{\mathtt{m}}$ of γ changes. The relation is governed by the definition of the memory subsystem. In case (b), if the instruction is the terminal one, or assigns an expression to a register, or a conditional, then only the process component (L, R) of γ changes. Here, the relation is obvious. Otherwise, the two components interact via a read, write or CAS, and both undergo changes. Here again, the relation is governed by what the memory subsystem permits.

Annotations. Silent memory update steps are annotated with $\mathtt{m} : \mathtt{Upd}$. Transitions involving process p executing an instruction that does not involve memory are annotated with $p : \bot$. On the other hand, $p : \mathtt{R}(\mathsf{x}, \mathsf{d})$, $p : \mathtt{W}(\mathsf{x}, \mathsf{d})$, $p : \mathtt{CAS}(\mathsf{x}, \mathsf{d}, \mathsf{d}', \mathsf{b})$ represent reads, writes and CAS operations by p respectively. The annotations indicate the variable and the associated values.

To study this transition system, one must understand which transitions, annotated thus, are enabled. For this, it is clear that we must delve into the details of the memory subsystem.

3 A Unified Framework for Weak Memory Models

In this section, we present our unified framework for representing weak memory models. We begin by describing the modelling aspects of our framework at a high level.

We use a message-based framework, where each write event generates a *message*. A process can use a write event to justify its read only if the corresponding message has been *propagated* to it by the memory subsystem. The total chronological order in which a process p writes to variable x is given by poloc (per-location program order). We work with models where the order of propagation is consistent with poloc. This holds for several models of varying strengths. This requirement allows us to organise messages into per-variable, per-process *channels*. We discuss these aspects of the framework in Sect. 3.1. Weak memory models define additional causal dependencies over poloc. Reading a message may cause other messages it is dependent on to become illegible. We discuss our mechanism to capture these dependencies in Sect. 3.2. The strength of the constraints levied by causal dependencies varies according to memory model.

In Sect. 3.3, we briefly explain how our framework allows us to express causality constraints of varying strength, by considering a wide landscape of weak memory models. We refer the reader to [6] for the technical details of the instantiations.

3.1 Message Structures

Message. A write by a process to a variable leads to the formation of a *message*, which, first and foremost records the value being written. In order to ensure atomicity, a message also records a boolean denoting whether the message can be used to justify the read of an *atomic* read-write operation, i.e. CAS. Finally, to help with the tracking of causal dependencies generated by read events, a message records a set of processes $\mathsf{seen} \subseteq \mathbb{P}$ that have sourced a read from it. Thus, a message is a triple and we define the set of messages as: $\mathsf{Msgs} = \mathbb{D} \times \mathbb{B} \times 2^{\mathbb{P}}$.

Channels. A *channel* $\mathsf{e}(\mathsf{x}, p)$ is the sequence of messages corresponding to writes to x by process p. The total poloc order of these writes naturally induces the channel order. By design, we will ensure that the configuration holds finitely many messages in each channels. We model each channel as a word over the message set: $\mathsf{e}(\mathsf{x}, p) \in \mathsf{Msgs}^*$. A message structure is a collection of these channels: $\mathsf{e} : \mathbb{X} \times \mathbb{P} \to \mathsf{Msgs}^*$.

3.2 Ensuring Consistency of Executions

Memory models impose constraints restricting the set of message that can be read by a process. The framework uses state elements $\mathsf{frontier}$, source, $\mathsf{constraint}$ that help enforce these constraints. These elements reference positions within each channel which is something that we now discuss.

Channel Positions. The channel order provides the order of propagation of write messages to any process (which in turn is aligned with poloc). Thus, for any process p', channel $\mathsf{e}(\mathsf{x}, p)$ is partitioned into a prefix of messages that are outdated, a null or singleton set of messages that can be used to justify a read, and a suffix of messages that are yet to be propagated. In order to express these partitions, we need to identify not only nodal positions, but also to *internodes* (space between nodes). To this end, we index channels using the set $\mathbb{W} = \mathbb{N} \cup \mathbb{N}^+$. Positions indexed by $\mathbb{N}$ denote nodal positions (with a message), while positions indexed with $\mathbb{N}^+$ denote internodes. For a channel of length n, the positions are ordered as: $\top = 0^+ < 1 < 1^+ < 2 \cdots < n < n^+ = \bot$. A process can read from the message located at $\mathsf{e}(\cdot, \cdot)[i]$ for $i \in \mathbb{N}$.

Frontier. With respect to a given process, a message can either have been propagated but not readable, propagated and readable, or none. Since the propagation order of messages follows channel order, the propagated set of messages forms a prefix of the channel. This prefix-partitioning is achieved by a map

$\mathsf{frontier} : \mathbb{P} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W}$. If $\mathsf{frontier}(p, \cdot, \cdot)$ is an internode (of form i^+) then the message $\mathsf{v} = \mathsf{e}[i]$ has been propagated to p, but cannot be read because it is outdated. On the other hand, if $\mathsf{frontier}(p, \cdot, \cdot) = i \in \mathbb{N}$, then the message $\mathsf{e}[i]$ can be read by the process. In Fig. 4, $\mathsf{frontier}(p_1, \mathsf{x}, p_1/p_2/p_3)$ equal $\mathsf{v}_1^+/\mathsf{v}_2/\mathsf{v}_3$ respectively (the colored nodes). Consequently, the message at index v (or the ones before it) are unreadable (denoted by the pattern). On the other hand the messages at $\mathsf{v}_2, \mathsf{v}_3$ are readable.

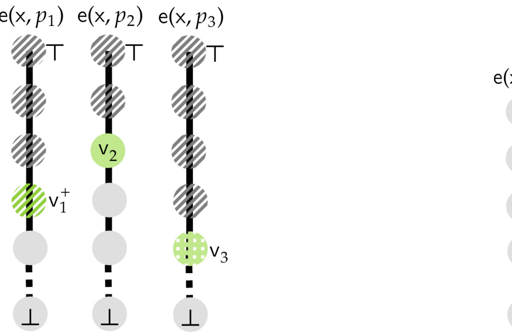


Fig. 4. Frontier and source.

Fig. 5. Constraint.

Source. Given process p and variable x, the process potentially can source the read from any of the $|\mathbb{P}|$ channels on x. The second state element, $\mathsf{source} : \mathbb{P} \times \mathbb{X} \to \mathbb{P}$ performs arbitration over this choice of read sources: p can read v only if $\mathsf{v} = \mathsf{frontier}(p, \mathsf{x}, \mathsf{source}(p, \mathsf{x}))$. In Fig. 4, while both nodes $\mathsf{v}_2, \mathsf{v}_3$ are not outdated, $\mathsf{source}(p_1, \mathsf{x}) = p_3$, making v_3 the (checkered) node which p_1 reads from.

Constraint. The $\mathsf{constraint}$ element tracks causal dependencies between messages. For each message m, and channel, it identifies the last message on the channel that is a causal predecessor of m. It is defined as a map $\mathsf{constraint} : \mathbb{N} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W}$. Figure 5 illustrates possible $\mathsf{constraint}(\mathsf{v}_3, \cdot, \cdot)$ pointers for message node v_3 in the context of the channel configuration in Fig. 4.

Constraint. The $\mathsf{frontier}$ state marks the last messages in each channel that can be read by a process. Messages that are earlier than the $\mathsf{frontier}$ of all processes can be effectively eliminated from the system since they are illegible. We call this garbage collection (denoted as GC).

The overall memory configuration,

$$\gamma_{\mathtt{m}} = \langle \mathsf{e}, \underbrace{(\mathbb{P} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W})}_{\mathsf{frontier}}, \underbrace{(\mathbb{P} \times \mathbb{X} \to \mathbb{P})}_{\mathsf{source}}, \underbrace{(\mathbb{V} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W})}_{\mathsf{constraint}} \rangle$$

consists of the message structure along with the consistency enforcing state.

Read Transition. Our framework allows a unified read transition relation which is independent of the memory model that we work with. We now discuss this transition rule which is given in Fig. 6. Suppose process p is reading from variable x. First, we identify the arbitrated process p_s which is read from using the source state. Then we pick the message on the (x, p_s) channel which the $\mathsf{frontier}$ of p points to. Note that this must be a node of form $\mathbb{N}$. The read value is the value in this message. Finally, we update the $\mathsf{frontier}(p, \cdot, \cdot)$ to reflect the fact that all messages in the causal prefix of the read message have propagated to p.

$$\frac{\begin{array}{c} p_s = \gamma.\mathsf{source}(p, \mathsf{x}) \qquad \mathsf{v} = \gamma.\mathsf{frontier}(p, \mathsf{x}, p_s) \qquad \mathsf{v.value} = \mathsf{d} \\ \gamma_1 = \gamma_1[\mathsf{v.seen} \leftarrow \mathsf{v.seen} \cup \{p\}] \\ \gamma_2 = \mathsf{GC}(\gamma_1[\lambda \mathsf{y}.\lambda p'.\ \mathsf{frontier}(p, \mathsf{y}, p') \leftarrow \max(\mathsf{frontier}(p, \mathsf{y}, p'), \mathsf{constraint}(\mathsf{v}, \mathsf{y}, p'))]) \end{array}}{\gamma \xrightarrow{p:\mathtt{R}(\mathsf{x},\mathsf{d})}_{\mathtt{m}} \gamma_2}$$

Fig. 6. The **read** transition, common to all models across the framework. For $\gamma \in \Gamma_{\mathtt{m}}$, $\gamma.\mathsf{frontier}, \gamma.\mathsf{source}, \gamma.\mathsf{constraint}$ represent the respective components of γ. For a node $\mathsf{v} \in \mathsf{Msgs}$, $\mathsf{v.value} \in \mathbb{D}$ represents the written value in the message node v.

Example 1 (Store Buffer (SB)). Fig. 7 shows the Store Buffer (SB) litmus test. The annotated outcome of store buffering is possible in all WRA/RA/SRA/TSO models. Right after p_1 (resp. p_2) has performed both its writes to x (resp. y), we have $\mathsf{e}(\mathsf{y}, p_2) = \top \mathsf{v}_\mathsf{y}^0 \mathsf{v}_\mathsf{y}^1 \bot$, and $\mathsf{e}(\mathsf{x}, p_1) = \top \mathsf{v}_\mathsf{x}^0 \mathsf{v}_\mathsf{x}^1 \bot$.

p_1	p_2
`x = 0 // ` v_x^0	`y = 0 // ` v_y^0
`x = 1 // ` v_x^1	`y = 1 // ` v_y^1
`r = y // 0`	`s = x // 0`

Fig. 7. SB

This example illustrates how weak memory models allow non-deterministic delays the propagation of messages. In this example, $\mathsf{frontier}(p_2, \mathsf{x}, p_1) = \mathsf{v}_\mathsf{x}^0$, and $\mathsf{frontier}(p_1, \mathsf{y}, p_2) = \mathsf{v}_\mathsf{y}^0$, both processes see non-recent messages. On the other hand, the annotated outcomes are observed if $\mathsf{source}(p_1, \mathsf{y}) = p_2$ and $\mathsf{source}(p_2, \mathsf{x}) = p_1$.

We now turn to a toy example (Fig. 8) to illustrate the dependency enforcing and book-keeping mechanisms we have introduced.

Example 2. Consider an program with two shared variables, x, y, and two processes, p_1, p_2. We omit the channel $\mathsf{e}(p_2, \mathsf{y})$ for space. Process p_1's frontiers are shown in violet, p_2's frontiers are shown in orange. We begin with the first memory configuration. The arrow depicts $\mathsf{constraint}(\mathsf{v}_1, \mathsf{y}, p_1) = \mathsf{v}_2$. This situation can arise in a causally consistent model where the writer of v_1 was aware of v_2 before writing v_1. The first transition shows p_2 updating and moving its $\mathsf{frontier}$ (to v_1). This results in a redundant node (v_3 in hashed texture) since the $\mathsf{frontier}$ of both p_1 and p_2 has crossed it. This is cleaned up by GC. Now, p_2 begins its read from v_1. Reading v_1, albeit on x, makes all writes by p_1 to y prior to v_2 redundant. When p_2 reads v_1, its frontier on $\mathsf{e}(\mathsf{y}, p_1)$ advances as prescribed by

constraint$(\mathsf{v}_1, \mathsf{y}, p_1)$, as shown in the fourth memory configuration. Note that this makes another message (v_4) redundant: all frontiers are past it. Once again, GC discards the message obtaining the last configuration.

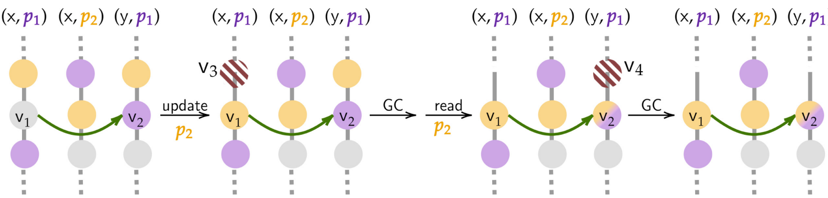


Fig. 8. Update, constraint in action during a read, and garbage collection

3.3 Instantiating the Framework

Versatility of the Framework. The framework we introduce can be instantiated to RMO [12], FIFO consistency, RA, SRA, WRA [34,35], TSO [13], PSO, StrongCOH (the relaxed fragment of RC11) [30], and SC [40].

This claim is established by constructing semantics for each of these models using the components that we have discussed. We provide a summary of the insights, and defer the technical details to the full version [6].

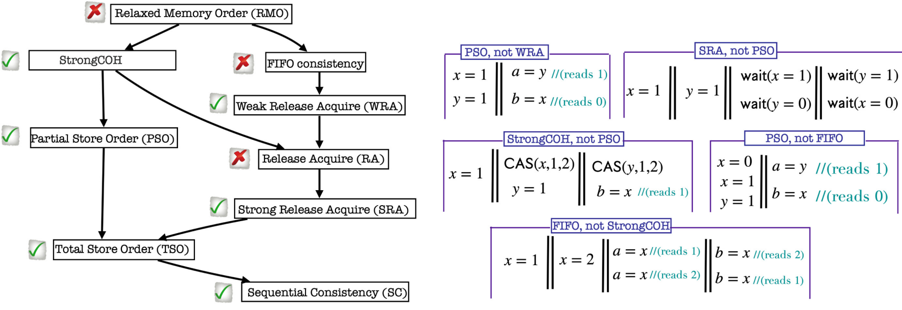


Fig. 9. Memory models, arranged by their strength. An arrow from A to B denotes that B is strictly more restrictive than A. A green check (resp. red cross) denotes the control state reachability is decidable (resp. undecidable). (Color figure online)

We briefly explain how our framework accounts for the increasing restrictive strength of these memory models. The weakest of these is RMO, which only enforces poloc. There are no other causal dependencies, and thus for any message the constraint on other channels is $\top$. RMO can be strengthened in two ways:

StrongCOH does it by requiring a total order on writes to the same variable, i.e. mo_x. Here the constraint is nontrivial only on channels of the same variable. On the other hand, FIFO enforces consistency with respect to the program order. Here, the constraint is nontrivial only on channels of the same process. WRA strengthens FIFO by enabling reads to enforce causal dependencies between write messages. This is captured by the non-trivial constraint, and we note that seen (the set of processes to have sourced a read from a message) plays a crucial role here. RA enforces the mo_x of StrongCOH as well as the causal dependencies of WRA. PSO strengthens StrongCOH by requiring a stronger precondition on the execution of an atomic read-write. More precisely, in any given configuration, for every variable, there is at most one write message that can be used to source a CAS operation, i.e. with the CAS flag set to true. SRA and TSO respectively strengthen RA and PSO by doing away with write races. Here, the Boolean CAS flag in messages is all-important as an enforcer of atomicity. TSO strengthens SRA in the same way as PSO strengthens StrongCOH. Finally, when we get to SC, the model is so strong that all messages are instantly propagated. Here, for any message, the pointer on other channels is $\perp$.

4 Fairness Properties

Towards the goal of verifying liveness, we use the framework we developed to introduce fairness properties in the the classical and probabilistic settings in Sect. 4.1 and Sect. 4.2 respectively. Our approach thus has the advantage of generalising over weak memory. In Sect. 4.3 we relate these fairness properties in the context of repeated control state reachability: a key liveness problem.

4.1 Transition and Memory Fairness

In this section, we consider fairness in the classical (non-probabilistic) case. We begin by defining transition fairness, which [11] is a standard notion of fairness that disallows executions which neglect certain transitions while only taking others. For utility in weak memory liveness, we then augment transition fairness to meet practical assumptions on the memory subsystem. Transition fairness and probabilistic fairness are intrinsically linked [27, Section 11]. Our augmentations are designed to carry over to the probabilistic domain in the same spirit.

Definition 1 (Transition fairness, [11]). *We say that a program execution is transition fair if for every configuration that is reached infinitely often, each transition enabled from it is also taken infinitely often.*

We argued the necessity of transition fairness in the introduction; however, it is vacuously satisfied by an execution that visits any configuration only finitely often. This could certainly be the case in weak memory, where there are infinitely many configurations. To make a case for the implausibility of this scenario, we begin by characterising classes of weak memory configurations.

Definition 2 (Configuration size). *Let γ be a program configuration with memory component* $(\mathsf{e}, \mathsf{frontier}, \mathsf{source}, \mathsf{constraint})$. *We denote the configuration size by* $\mathsf{size}(\gamma)$ *and it is defined as* $\sum_{\mathsf{x}} \sum_{p} \mathsf{len}(\mathsf{e}(\mathsf{x}, p))$, *i.e. the total number of messages in the message structure.*

Intuitively, the size of the configuration is the number of messages "in transit", and hence a measure of the weakness of the behaviour of the execution. We note that overly weak behaviour is rarely observed in practice [23,45]. For instance, instruction reorderings that could be observed as weak behaviour are limited in scope. Another source of weak behaviour is the actual reading of stale values at runtime. However, the hardware (i.e. caches, buffers, etc.) that stores these values is finite, and is typically flushed regularly. Informally, the finite footprint of the system architecture (eg. micro-architecture) implies a bound, albeit hard to compute, on the size of the memory subsystem. Thus, we use the notion of configuration size to define:

Definition 3 (Size Bounded Executions). *An execution $\gamma_0, \gamma_1, \ldots$ is said to be size bounded, if there exists an N such that for all $n \in \mathbb{N}$,* $\mathsf{size}(\gamma_n) \leq N$. *If this N is specified, we refer to the execution as N-bounded.*

Already, the requirement of size-boundedness enables our system to refine our practical heuristics. However, if the bound N is unknown, it isn't immediately clear how this translates into a technique for liveness verification. We will now use the same rationale to motivate and develop an alternate augmentation which lends itself more naturally to algorithmic techniques. Recall that we intuitively relate the size of the configuration to the extent of weak behaviour. Now, consider Sequential Consistency, the strongest of the models. All messages are propagated immediately, and hence, the configuration has minimal size throughout. We call minimally sized configurations *plain*, and they are of particular interest to us:

Definition 4 (Plain message structure). *A message structure* $(\mathsf{V}, \mathsf{msgmap}, \mathsf{e})$ *is called plain, if for each variable* x, $\sum_{p} \mathsf{len}(\mathsf{e}(\mathsf{x}, p)) = 1$.

Drawing a parallel with SC, one could reason that the recurrence of plain configurations is a hallmark of a system that doesn't exhibit overly weak behaviour. This is captured with the following fairness condition.

Definition 5 (Repeatedly Plain Executions). *An execution $\gamma_0, \gamma_1, \ldots$ is said to be repeatedly plain, if γ_i is a plain configuration for infinitely many i.*

Following the memory transition system introduced in Sect. 2 and Sect. 3, we observe that every configuration has a (finite) path to some plain configuration (by performing a sequence of update steps). Hence, if a configuration is visited infinitely often in a fair execution, a plain configuration will also be visited infinitely often. Consequently, size bounded transition fair runs are also repeatedly plain transition fair.

4.2 Probabilistic Memory Fairness

Problems considered in a purely logical setting ask whether *all* executions satisfying a fairness condition fulfill a liveness requirement. However, if the answer is negative, one might be interested in quantifying the fair executions which do not repeatedly reach the control state. We perform this quantification by considering the probabilistic variant of the model proposed earlier, and defining fairness analogously as a property of Markov Chains.

Markov chains A Markov chain is a pair $\mathcal{C} = \langle \Gamma, \mathtt{M} \rangle$ where Γ is a (possibly-infinite) set of configurations and $\mathtt{M}$ is the transition matrix which assigns to each possible transition, a *transition probability*: $\mathtt{M} : \Gamma \times \Gamma \to [0, 1]$. Indeed, this matrix needs to be stochastic, i.e., $\sum_{\gamma' \in \Gamma} \mathtt{M}(\gamma, \gamma') = 1$ should hold for all configurations.

We can convert our concurrent program transition (Sect. 2) into a Markov chain $\mathtt{M}$ by adding probabilities to the transitions. We assign $\mathtt{M}(\gamma, \gamma')$ to a nonzero value if and only if the transition $\gamma \to \gamma'$ is allowed in the underlying transition system. Markov Chain executions are, by construction, transition fair with probability 1. We now present the analog of the repeatedly plain condition.[1]

Definition 6 (Probabilistic Memory Fairness. *A Markov chain is considered to satisfy probabilistic memory fairness if a plain configuration is reached infinitely often with probability one.*

This parallel has immense utility because verifying liveness properties for a class of Markov Chains called Decisive Markov Chains is well studied. [7] establishes that the existence of a *finite attractor*, i.e. a finite set of states F that is repeatedly reached with probability 1, is sufficient for decisiveness. The above definition asserts that the set of plain configurations is a finite attractor.

4.3 Relating Fairness Notions

Although repeatedly plain transition fairness is weaker than size bounded transition fairness and probabilistic memory fairness, these three notions are equivalent with respect to canonical liveness problems, i.e. repeated control state reachability and termination. The proof we present for repeated reachability can be adapted for termination.

Theorem 1. *There exists $N_0 \in \mathbb{N}$ such that for all $N \geq N_0$, the following are equivalent for any control state (program counters and register values) c:*

1. *All repeatedly plain transition fair runs visit c infinitely often.*
2. *All N-bounded transition fair runs visit c infinitely often.*
3. *c is visited infinitely often under probabilistic memory fairness with probability 1.*

[1] A concrete Markov Chain satisfying the declarative definition may be adapted from the one described in [5] in a similar setting.

Proof. For each $N \in \mathbb{N}$, we construct a connectivity graph $\mathcal{G}_N$. The vertices are the finitely many plain configurations γ, along with the finitely many control states c. We draw a directed edge from γ_i to γ_j, if γ_j is reachable from γ_i via configurations of size at most N. We additionally draw an edge from a plain configuration γ to control state c iff c is reachable from γ via configurations of size at most N. We similarly construct a connectivity graph $\mathcal{G}$ without bounds on intermediate configuration sizes. We note:

1. There are only finitely many possibilities for $\mathcal{G}_N$
2. As N increases, edges can only be added to $\mathcal{G}_N$. This guarantees saturation.
3. Any witness of reachability is necessarily finite, hence the saturated graph is the same as $\mathcal{G}$, i.e. for all sufficiently large N, $\mathcal{G}_N = \mathcal{G}$

Since plain configurations are attractors, the graph $\mathcal{G}$ is instrumental in deciding repeated control state reachability. Consider the restriction of $\mathcal{G}$ to plain configurations, i.e. $\mathcal{G}_\Gamma$. Transition fairness (resp. Markov fairness) implies that γ is visited infinitely often (resp. with probability 1) only if it is in a bottom strongly connected component (scc). In turn any control state c will be guaranteed to be reached infinitely often if and only if it is reachable from every bottom scc of $\mathcal{G}_\Gamma$. The if direction follows using the transition fairness and attractor property, while the converse follows by simply identifying a finite path to a bottom scc from which c isn't reachable. The equivalence follows because the underlying graph is canonical for all three notions of fairness.

5 Applying Fairness Properties to Decision Problems

In this section, we show how to decide liveness as a corollary of the proof of Theorem 1. We begin by noting that techniques for termination are subsumed by those for repeated control state reachability. This is because termination is not guaranteed iff one can reach a plain configuration from which a terminal control state is inaccessible. Hence, in the sequel, we focus on repeated control state reachability.

5.1 Deciding Repeated Control State Reachability

We observe that under the fairness conditions we defined, liveness, i.e. repeated control state reachability reduces to a *safety* query.

Problem 1 (Repeated control state reachability). Given a control state (program counters and register values) c, do all infinite executions (in the probabilistic case, a set of measure 1) satisfying fairness condition $\mathcal{A}$ reach c infinitely often?

Problem 2 (Control state reachability). Given a control state c and a configuration γ, is $(c, \gamma_{\mathtt{m}})$ reachable from γ for some $\gamma_{\mathtt{m}}$?

Theorem 2. *Problem 1 for repeatedly fair transition fairness and probabilistic memory fairness reduces to Problem 2. Moreover, the reduction can compute the N_0 from Theorem 1 such that it further applies to size bounded transition fairness.*

Proof. This follows by using Problem 2 to compute the underlying connectivity graph $\mathcal{G}$ from the proof of Theorem 1. A small technical hurdle is that plain configuration reachability is not the same as control state reachability. However, the key to encode this problem as a control state query is to use the following property: for a configuration γ and a message m ($\in \mathsf{e}(\mathsf{x}, p)$), if for every process p', m is not redundant (formally, $\mathsf{frontier}(p', \mathsf{x}, p) \leq m$), then there exists a plain configuration γ' containing m such that γ' is reachable from γ via a sequence of update steps. The plan, therefore, is to read and verify whether the messages desired in the plain configuration are, and remain accessible to all processes. Finally, the computation of N_0 follows by enumerating $\mathcal{G}_N$.

5.2 Quantitative Control State Repeated Reachability

We set the context of a Markov chain $\mathcal{C} = \langle \Gamma, \mathtt{M} \rangle$ that refines the underlying the transition system induced by the program. We consider is the quantitative variant of repeated reachability, where instead of just knowing whether the probability is one or not, we are interested in computing it.

Problem 3 (Quantitative control state repeated reachability). Given a control state c and an error margin $\epsilon \in \mathbb{R}$, find a δ such that for Markov chain $\mathcal{C}$, $|\text{Prob}(\gamma_{\mathsf{init}} \models \Box\Diamond c) - \delta| \leq \epsilon$

We refer the reader to [6] for details on the standard reduction, from which the following result follows:

Theorem 3. *If Problem 2 is decidable for a memory model, then Problem 3 is computable for Markov chains that satsify probabilistic memory fairness.*

5.3 Adapting Subroutines to Our Memory Framework

We now briefly sketch how to adapt known solutions to Problem 2 for PSO, TSO, StrongCOH, WRA and SRA to our framework.

PSO and TSO. Reachability between plain configurations (a special case of Problem 2) under these models has already been proven decidable [12]. The store buffer framework is similar to the one we describe, and hence the results go through. Moreover, [5, Lemmas 3, 4] shows the decidability of our Problem 2 for TSO. The same construction, which uses an augmented program to reduce to ex-plain configuration reachability, works for PSO as well.

StrongCOH. Decidability of reachability under StrongCOH is shown in [1]. The framework used, although quite different in notation, is roughly isomorphic to the one we propose. The relaxed semantics of StrongCOH allow the framework to be set up as a WSTS [2,26], which supports backward reachability analysis, yielding decidability. Backward reachability gives an upward closed set of states that can reach a target label. Checking whether an arbitrary state is in this upward closed set requires a comparison with only the finitely many elements in the basis. This solves Problems 2.

WRA and SRA. Decidability of reachability under WRA and SRA has recently been shown in [34]. The proof follows the WSTS approach, however, the model used in the proof has different syntax and semantics from the one we present here. However, a reconciliation is possible, and we briefly sketch it here. A state in the proof model is a map from processes to *potentials*. A potential is a finite set of finite traces that a process may execute. These proof-model states are well-quasi-ordered and operating on them sets up a WSTS. Backward reachability gives us a set of maps from processes to *potentials* that allow us to reach the target label. The key is to view a process-potential map as a requirement on our message based configuration. Higher a map in the wqo, stronger the requirement it enforces. In this sense, the basis of states returned by backward reachability constitute the minimal requirements our configuration may meet in order for the target label to be reachable. Formally, let γ be a configuration of our framework. The target label is reachable from γ if and only if: there exists a process-potential map $\mathcal{B}$ is in the backward reachable set, such that every trace in every process' potential in $\mathcal{B}$ is enabled in γ. It suffices to check the existence of $\mathcal{B}$ over the finite basis of the backward reachable set. Note that γ is completely arbitrary: this solves our Problem 2.

6 Related Work

Fairness. Only recently has fairness for weak memory started receiving increasing attention. The work closest to ours is by [4], who formulate a probabilistic extension for the Total Store Order (TSO) memory model and show decidability results for associated verification problems. Our treatment of fairness is richer, as we relate same probabilistic fairness with two alternate logical fairness definitions. Similar proof techniques notwithstanding, our verification results are also more general, thanks to the development of a uniform framework that applies to a landscape of models. [37] develop a novel formulation of fairness as a declarative property of event structures. This notion informally translates to "Each message is eventually propagated." We forego axiomatic elegance to motivate and develop stronger practical notions of fairness in our quest to verify liveness.

Probabilistic Verification. There are several works on verification of *finite-state Markov chains* (e.g. [14,33]). However, since the messages in our memory

systems are unbounded, these techniques do not apply. There is also substantive literature on the verification of infinite state probabilistic system, which have often been modelled as infinite Markov chains [17–19, 24, 25]. However their results cannot be directly leveraged to imply ours. The machinery we use for showing decidability is relies on decisive Markov chains, a concept formulated in [7] and used in [4].

Framework. On the modelling front, the ability to specify memory model semantics as first-order constraints over the program-order (po), reads-from (rf), and modification-order (mo) have led to elegant declarative frameworks based on event structures [9, 10, 21, 28]. There are also approaches that, instead of natively characterizing semantics, prescribe constraints on their ISA-level behaviours in terms of program transformations [38]. On the operational front, there have been works that model individual memory models [43, 46] and clusters of similar model [30, 35], however we are not aware of any operational modelling framework that encompasses as wide a range of models as we do. The operationalization in [16] uses write buffers which resemble our channels, however, their operationalization too focuses on a specific semantics.

7 Conclusion, Future Work, and Perspective

Conclusion. The ideas developed in Sect. 4 lie at the heart of our contribution: we motivate and define transition fairness augmented with memory size boundedness or the recurrence of plain configurations, as well as the analogous probabilistic memory fairness. These are equivalent for the purpose of verifying repeated control state reachability, i.e. liveness, and lie at the core of the techniques we discuss in Sect. 5. These techniques owe their generality to the versatile framework we describe in Sect. 3.

Future Work. There are several interesting directions for future work. We believe that our framework can be extended to handle weak memory models that allow speculation, such as ARM and POWER. In such a case, we would need to extend our fairness conditions to limit the amount of allowed speculation. It is also interesting to mix transition fairness with probabilistic fairness, i.e., use the former to solve scheduler non-determinism and the latter to resolve memory non-determinism, leading to (infinite-state) Markov Decision Process model. Along these lines, we can also consider synthesis problems based on $2\frac{1}{2}$-games. To solve such game problems, we could extend the framework of Decisive Markov chains that have been developed for probabilistic and game theoretic problems over infinite-state systems [7] A natural next step is developing efficient algorithms for proving liveness properties for programs running on weak memory models. In particular, since we reduce the verification of liveness properties to simple reachability, there is high hope one can develop CEGAR frameworks relying both on over-approximations, such as predicate abstraction, and under-approximations such as bounded context-switching [44] and stateless model checking [3, 32].

Perspective. Leveraging techniques developed over the years by the program verification community, and using them to solve research problems in programming languages, architectures, databases, etc., has substantial potential added value. Although it requires a deep understanding of program behaviors running on such platforms, we believe it is about finding the right concepts, combining them correctly, and then applying the existing rich set of program verification techniques, albeit in a non-trivial manner. The current paper is a case in point. Here, we have used a combination of techniques developed for reactive systems [31], methods for the analysis of infinite-state systems [7], and semantical models developed for weak memory models [12,30,34,35] to obtain, for the first time, a framework for the systematic analysis of liveness properties under weak memory models.

Acknowledgements. This project was partially supported by the Swedish Research Council and SERB MATRICS grant MTR/2019/000095. Adwait Godbole was supported in part by Intel under the Scalable Assurance program, DARPA contract FA8750-20-C0156 and National Science Foundation (NSF) grant 1837132.

References

1. Abdulla, P.A., Atig, M.F., Godbole, A., Krishna, S., Vafeiadis, V.: The decidability of verification under PS 2.0. In: ESOP 2021. LNCS, vol. 12648, pp. 1–29. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_1
2. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bull. Symb. Log. **16**(4), 457–515 (2010). https://doi.org/10.2178/bsl/1294171129
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: a foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017)
4. Abdulla, P.A., Atig, M.F., Agarwal, R.A., Godbole, A., S., K.: Probabilistic total store ordering. In: ESOP 2022. LNCS, vol. 13240, pp. 317–345. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99336-8_12
5. Abdulla, P.A., Atig, M.F., Agarwal, R.A., Godbole, A., Krishna, S.: Probabilistic total store ordering (2022). https://arxiv.org/abs/2201.10213
6. Abdulla, P.A., Atig, M.F., Godbole, A., Krishna, S., Vahanwala, M.: Overcoming memory weakness with unified fairness (2023). https://arxiv.org/abs/2305.17605
7. Abdulla, P.A., Henda, N.B., Mayr, R.: Decisive markov chains. LMCS **3**(4) (2007)
8. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: a tutorial. IEEE Comput. **29**(12), 66–76 (1996)
9. Alglave, J.: A formal hierarchy of weak memory models. Formal Methods Syst. Des. **41**, 178–210 (2012)
10. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data-mining for weak memory. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (2014)
11. Aminof, B., Ball, T., Kupferman, O.: Reasoning about systems with transition fairness. In: Baader, F., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3452, pp. 194–208. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32275-7_14

12. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 7–18 (2010). https://doi.org/10.1145/1706299.1706303
13. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_9
14. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press, Cambridge (2008)
15. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing c++ concurrency. In: ACM-SIGACT Symposium on Principles of Programming Languages (2011)
16. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 392–403. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1480881.1480930
17. Brázdil, T., Chatterjee, K., Kučera, A., Novotný, P., Velan, D.: Deciding fast termination for probabilistic VASS with nondeterminism. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 462–478. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_27
18. Brázdil, T., Kiefer, S., Kucera, A., Novotný, P., Katoen, J.: Zero-reachability in probabilistic multi-counter automata. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014, Vienna, Austria, 14–18 July 2014, pp. 22:1–22:10. ACM (2014). https://doi.org/10.1145/2603088.2603161
19. Brázdil, T., Kiefer, S., Kucera, A., Vareková, I.H.: Runtime analysis of probabilistic programs with unbounded recursion. J. Comput. Syst. Sci. **81**(1), 288–310 (2015). https://doi.org/10.1016/j.jcss.2014.06.005
20. Broy, M., Wirsing, M.: On the algebraic specification of nondeterministic programming languages. In: CAAP (1981)
21. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290383
22. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: Hollingsworth, J.K., Keidar, I. (eds.) Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, 16–20 February 2019, pp. 355–365. ACM (2019). https://doi.org/10.1145/3293883.3295702
23. Elver, M., Nagarajan, V.: Tso-cc: consistency directed cache coherence for tso. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 165–176 (2014). https://doi.org/10.1109/HPCA.2014.6835927
24. Esparza, J., Kucera, A., Mayr, R.: Model checking probabilistic pushdown automata. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), Turku, Finland, 14–17 July 2004, Proceedings, pp. 12–21. IEEE Computer Society (2004). https://doi.org/10.1109/LICS.2004.1319596
25. Etessami, K., Yannakakis, M.: Recursive markov decision processes and recursive stochastic games. J. ACM **62**(2), 11:1–11:69 (2015). https://doi.org/10.1145/2699431

26. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. **256**(1–2), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X
27. van Glabbeek, R.J., Höfner, P.: Progress, justness, and fairness. ACM Comput. Surv. (CSUR) **52**, 1–38 (2019)
28. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory, pp. 759–767 (2016). https://doi.org/10.1145/2933575.2934536
29. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: reasoning about release-acquire consistency in iris. In: Proceedings of the 31st European Conference on Object-Oriented Programming, ECOOP 2017, pp. 17:1–17:29 (2017)
30. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 175–189. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3009837.3009850
31. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model checking with strong fairness. Formal Methods Syst. Des. **28**(1), 57–84 (2006). https://doi.org/10.1007/s10703-006-4342-y
32. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**, 17:1–17:32 (2018)
33. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
34. Lahav, O., Boker, U.: What's decidable about causally consistent shared memory? ACM Trans. Program. Lang. Syst. **44**(2) (2022). https://doi.org/10.1145/3505273
35. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 649–662. ACM (2016)
36. Lahav, O., Namakonov, E., Oberhauser, J., Podkopaev, A., Vafeiadis, V.: Making weak memory models fair. ArXiv arXiv:abs/2012.01067 (2020)
37. Lahav, O., Namakonov, E., Oberhauser, J., Podkopaev, A., Vafeiadis, V.: Making weak memory models fair (2020). https://doi.org/10.48550/ARXIV.2012.01067
38. Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: FM (2016)
39. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 618–632. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3062341.3062352
40. Lamport, L.: How to make a multiprocessor that correctly executes multiprocess programs. IEEE Trans. Comput. C **28**, 690–691 (1979)
41. Mador-Haim, S., et al.: An axiomatic memory model for POWER multiprocessors. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 495–512. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_36
42. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, Heidelberg (1992). https://doi.org/10.1007/978-1-4612-0931-7

43. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for c/c++11 concurrency. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pp. 111–128. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2983990.2983997
44. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_7
45. Ros, A., Kaxiras, S.: Callback: efficient synchronization without invalidation with a directory just for spin-waiting. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA 2015, pp. 427–438. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2749469.2750405
46. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)
47. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: POPL, pp. 184–193. ACM Press (1986). https://doi.org/10.1145/512644.512661

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Rely-Guarantee Reasoning for Causally Consistent Shared Memory

Ori Lahav[1], Brijesh Dongol[2(✉)], and Heike Wehrheim[3]

[1] Tel Aviv University, Tel Aviv, Israel
orilahav@tau.ac.il
[2] University of Surrey, Guildford, UK
b.dongol@surrey.ac.uk
[3] University of Oldenburg, Oldenburg, Germany
heike.wehrheim@uni-oldenburg.de

Abstract. Rely-guarantee (RG) is a highly influential compositional proof technique for concurrent programs, which was originally developed assuming a sequentially consistent shared memory. In this paper, we first generalize RG to make it parametric with respect to the underlying memory model by introducing an RG framework that is applicable to any model axiomatically characterized by Hoare triples. Second, we instantiate this framework for reasoning about concurrent programs under *causally consistent memory*, which is formulated using a recently proposed *potential-based* operational semantics, thereby providing the first reasoning technique for such semantics. The proposed program logic, which we call Piccolo, employs a novel assertion language allowing one to specify ordered sequences of states that each thread may reach. We employ Piccolo for multiple litmus tests, as well as for an adaptation of Peterson's algorithm for mutual exclusion to causally consistent memory.

1 Introduction

Rely-guarantee (RG) is a fundamental compositional proof technique for concurrent programs [21,48]. Each program component P is specified using *rely* and *guarantee* conditions, which means that P can tolerate any environment interference that follows its rely condition, and generate only interference included in its guarantee condition. Two components can be composed in parallel provided that the rely of each component agrees with the guarantee of the other.

The original RG framework and its soundness proof have assumed a sequentially consistent (SC) memory [33], which is unrealistic in modern processor architectures and programming languages. Nevertheless, the main principles behind RG are not at all specific for SC. Accordingly, our first main contribution,

Lahav is supported by the Israel Science Foundation (grants 1566/18 and 814/22) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811). Dongol is supported by EPSRC grants EP/X015149/1, EP/V038915/1, EP/R025134/2, VeTSS, and ARC Discovery Grant DP190102142. Wehrheim is supported by the German Research Council DFG (project no. 467386514).

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 206–229, 2023.
https://doi.org/10.1007/978-3-031-37706-8_11

is to formally decouple the underlying memory model from the RG proof principles, by proposing a generic RG framework parametric in the input memory model. To do so, we assume that the underlying memory model is axiomatized by Hoare triples specifying pre- and postconditions on memory states for each primitive operation (e.g., loads and stores). This enables the formal development of RG-based logics for different shared memory models as instances of one framework, where all build on a uniform soundness infrastructure of the RG rules (e.g., for sequential and parallel composition), but employ different specialized assertions to describe the possible memory states, where specific soundness arguments are only needed for primitive memory operations.

The second contribution of this paper is an instance of the general RG framework for *causally consistent shared memory*. The latter stands for a family of wide-spread and well-studied memory models weaker than SC, which are sufficiently strong for implementing a variety of synchronization idioms [6,12,26]. Intuitively, unlike SC, causal consistency allows different threads to observe writes to memory in different orders, as long as they agree on the order of writes that are causally related. This concept can be formalized in multiple ways, and here we target a strong form of causal consistency, called *strong release-acquire* (SRA) [28,31] (and equivalent to "causal convergence" from [12]), which is a slight strengthening of the well-known release-acquire (RA) model (used by C/C++11). (The variants of causal consistency only differ for programs with write/write races [10,28], which are rather rare in practice.)

Our starting point for axiomatizing SRA as Hoare triples is the *potential-based* operational semantics of SRA, which was recently introduced with the goal of establishing the decidability of control state reachability under this model [27,28] (in contrast to undecidability under RA [1]). Unlike more standard presentations of weak memory models whose states record information about the *past* (e.g., in the form of store buffers containing executed writes before they are globally visible [36], partially ordered execution graphs [8,20,31], or collections of timestamped messages and thread views [11,16,17,23,25,47]), the states of the potential-based model track possible *futures* ascribing what sequences of observations each thread can perform. We find this approach to be a particularly appealing candidate for Hoare-style reasoning which would naturally generalize SC-based reasoning. Intuitively, while an assertion in SC specifies possible observations at a given program point, an assertion in a potential-based model should specify possible *sequences* of observations.

To pursue this direction, we introduce a novel assertion language, resembling temporal logics, which allows one to express properties of sequences of states. For instance, our assertions can express that a certain thread may currently read $\mathtt{x} = 0$, but it will have to read $\mathtt{x} = 1$ once it reads $\mathtt{y} = 1$. Then, we provide Hoare triples for SRA in this assertion language, and incorporate them in the general RG framework. The resulting program logic, which we call Piccolo, provides a novel approach to reason on concurrent programs under causal consistency, which allows for simple and direct proofs, and, we believe, may constitute a basis for automation in the future.

$\{\mathtt{y} \neq 1\}$

Thread $\mathtt{T}_1$	**Thread** $\mathtt{T}_2$
$\{\mathit{True}\}$	$\{\mathtt{y} = 1 \Rightarrow \mathtt{x} = 1\}$
$1 : \mathtt{STORE}(\mathtt{x}, 1);$	$3 : \mathtt{a} := \mathtt{LOAD}(\mathtt{y});$
$\{\mathtt{x} = 1\}$	$\{\mathtt{a} = 1 \Rightarrow \mathtt{x} = 1\}$
$2 : \mathtt{STORE}(\mathtt{y}, 1)$	$4 : \mathtt{b} := \mathtt{LOAD}(\mathtt{x})$
$\{\mathit{True}\}$	$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$

$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$

Fig. 1. Message passing in SC

$\{\mathtt{T}_0 \ltimes [\mathtt{y} \neq 1]\}$

Thread $\mathtt{T}_1$	**Thread** $\mathtt{T}_2$
$\{\mathit{True}\}$	$\{\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] ; [\mathtt{x} = 1]\}$
$1 : \mathtt{STORE}(\mathtt{x}, 1);$	$3 : \mathtt{a} := \mathtt{LOAD}(\mathtt{y});$
$\{\mathtt{T}_1 \ltimes [\mathtt{x} = 1]\}$	$\{\mathtt{a} = 1 \Rightarrow \mathtt{T}_2 \ltimes [\mathtt{x} = 1]\}$
$2 : \mathtt{STORE}(\mathtt{y}, 1)$	$4 : \mathtt{b} := \mathtt{LOAD}(\mathtt{x})$
$\{\mathit{True}\}$	$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$

$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$

Fig. 2. Message passing in SRA

2 Motivating Example

To make our discussion concrete, consider the message passing program (MP) in Figs. 1 and 2, comprising shared variables x and y and local registers a and b. The proof outline in Fig. 1 assumes SC, whereas Fig. 2 assumes SRA. In both cases, at the end of the execution, we show that if a is 1, then b must also be 1. We use these examples to explain the two main concepts introduced in this paper: (*i*) a generic RG framework and (*ii*) its instantiation with a potential-focused assertion system that enables reasoning under SRA.

Rely-Guarantee. The proof outline in Fig. 1 can be read as an RG derivation:

1. Thread $\mathtt{T}_1$ locally establishes its postcondition when starting from any state that satisfies its precondition. This is trivial since its postcondition is *True*.
2. Thread $\mathtt{T}_1$ relies on the fact that its used assertions are *stable* w.r.t. interference from its environment. We formally capture this condition by a rely set $\mathcal{R}_1 \triangleq \{\mathit{True}, \mathtt{x} = 1\}$.
3. Thread $\mathtt{T}_1$ guarantees to its concurrent environment that its only interferences are $\mathtt{STORE}(\mathtt{x}, 1)$ and $\mathtt{STORE}(\mathtt{y}, 1)$, and furthermore that $\mathtt{STORE}(\mathtt{y}, 1)$ is only performed when $\mathtt{x} = 1$ holds. We formally capture this condition by a guarantee set $\mathcal{G}_1 \triangleq \{\{\mathit{True}\}\ \mathtt{T}_1 \mapsto \mathtt{STORE}(\mathtt{x}, 1), \{\mathtt{x} = 1\}\ \mathtt{T}_1 \mapsto \mathtt{STORE}(\mathtt{y}, 1)\}$, where each element is a command guarded by a precondition.
4. Thread $\mathtt{T}_2$ locally establishes its postcondition when starting from any state that satisfies its precondition. This is straightforward using standard Hoare rules for assignment and sequential composition.
5. Thread $\mathtt{T}_2$'s rely set is again obtained by collecting all the assertions used in its proof: $\mathcal{R}_2 \triangleq \{\mathtt{y} = 1 \Rightarrow \mathtt{x} = 1, \mathtt{a} = 1 \Rightarrow \mathtt{x} = 1, \mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$. Indeed, the local reasoning for $\mathtt{T}_2$ needs all these assertions to be stable under the environment interference.
6. Thread $\mathtt{T}_2$'s guarantee set is given by:

   $$\mathcal{G}_2 \triangleq \{\{\mathtt{y} = 1 \Rightarrow \mathtt{x} = 1\}\ \mathtt{T}_2 \mapsto \mathtt{a} := \mathtt{LOAD}(\mathtt{y}), \{\mathtt{a} = 1 \Rightarrow \mathtt{x} = 1\}\ \mathtt{T}_2 \mapsto \mathtt{b} := \mathtt{LOAD}(\mathtt{x})\}$$

7. To perform the parallel composition, $\langle \mathcal{R}_1, \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2, \mathcal{G}_2 \rangle$ should be *non-interfering*. This involves showing that each $R \in \mathcal{R}_i$ is *stable* under each $G \in \mathcal{G}_j$ for $i \neq j$. That is, if $G = \{P\}\ \tau \mapsto c$, we require the Hoare triple $\{P \cap R\}\ \tau \mapsto c\ \{R\}$ to hold. In this case, these proof obligations are straightforward

to discharge using Hoare's assignment axiom (and is trivial for $i = 1$ and $j = 2$ since load instructions leave the memory intact).

Remark 1. Classical treatments of RG involve two related ideas [21]: (1) specifying a component by rely and guarantee conditions (together with standard pre- and postconditions); and (2) taking the relies and guarantees to be binary relations over states. Our approach adopts (1) but not (2). Thus, it can be seen as an RG presentation of the Owicki-Gries method [37], as was previously done in [32]. We have not observed an advantage for using binary relations in our examples, but the framework can be straightforwardly modified to do so.

Now, observe that substantial aspects of the above reasoning are *not* directly tied with SC. This includes the Hoare rules for compound commands (such as sequential composition above), the idea of specifying a thread using collections of stable rely assertions and guaranteed guarded primitive commands, and the non-interference condition for parallel composition. To carry out this generalization, we assume that we are provided an assertion language whose assertions are interpreted as *sets of memory states* (which can be much more involved than simple mappings of variables to values), and a set of valid Hoare triples for the primitive instructions. The latter is used for checking validity of primitive triples, (e.g., $\{P\}\ \mathtt{T}_1 \mapsto \mathtt{STORE}(\mathtt{x}, 1)\ \{Q\}$), as well as non-interference conditions (e.g., $\{P \cap R\}\ \mathtt{T}_1 \mapsto \mathtt{STORE}(\mathtt{x}, 1)\ \{R\}$). In Sect. 4, we present this generalization, and establish the soundness of RG principles independently of the memory model.

Potential-Based Reasoning. The second contribution of our work is an application of the above to develop a logic for a potential-based operational semantics that captures SRA. In this semantics every memory state records sequences of store mappings (from shared variables to values) that each thread may observe. For example, assuming all variables are initialized to 0, if $\mathtt{T}_1$ executed its code until completion before $\mathtt{T}_2$ even started (so under SC the memory state is the store $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\}$), we may reach the SRA state in which $\mathtt{T}_1$'s potential consists of one store $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\}$, and $\mathtt{T}_2$'s potential is the sequence of stores:

$$\langle \{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0\}, \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 0\}, \{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1\} \rangle,$$

which captures the stores that $\mathtt{T}_2$ may observe in the order it may observe them. Naturally, potentials are *lossy* allowing threads to non-deterministically lose a subsequence of the current store sequence, so they can progress in their sequences. Thus, $\mathtt{T}_2$ can read 1 from $\mathtt{y}$ only after it loses the first two stores in its potential, and from this point on it can only read 1 from $\mathtt{x}$. Now, one can see that *all* potentials of $\mathtt{T}_2$ at its initial program point are, in fact, subsequences of the above sequence (regardless of where $\mathtt{T}_1$ is), and conclude that $\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1$ holds when $\mathtt{T}_2$ terminates.

To capture the above informal reasoning in a Hoare logic, we designed a new form of assertions capturing possible locally observable sequences of stores, rather than one global store, which can be seen as a restricted fragment of linear temporal logic. The proof outline using these assertions is given in Fig. 2. In particular, $[\mathtt{x} = 1]$ is satisfied by all store sequences in which every store maps $\mathtt{x}$

to 1, whereas $[\mathtt{y} \neq 1] ; [\mathtt{x} = 1]$ is satisfied by all store sequences that can be split into a (possibly empty) prefix whose value for y is not 1 followed by a (possibly empty) suffix whose value for x is 1. Assertions of the form $\tau \ltimes I$ state that the potential of thread τ includes only store sequences that satisfy I.

The first assertion of $\mathtt{T}_2$ is implied by the initial condition, $\mathtt{T}_0 \ltimes [\mathtt{y} \neq 1]$, since the potential of the parent thread $\mathtt{T}_0$ is inherited by the forked child threads and $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1]$ implies $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] ; I$ for any I. Moreover, $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] ; [\mathtt{x} = 1]$ is preserved by (i) line 1 because writing 1 to x leaves $[\mathtt{y} \neq 1]$ unchanged and re-establishes $[\mathtt{x} = 1]$; and (ii) line 2 because the semantics for SRA ensures that after reading 1 from y by $\mathtt{T}_2$, the thread $\mathtt{T}_2$ is confined by $\mathtt{T}_1$'s potential just before it wrote 1 to y, which has to satisfy the precondition $\mathtt{T}_1 \ltimes [\mathtt{x} = 1]$. (SRA allows to update the other threads' potential only when the suffix of the potential after the update is observable by the writer thread.)

In Sect. 6 we formalize these arguments as Hoare rules for the primitive instructions, whose soundness is checked using the potential-based operational semantics and the interpretation of the assertion language. Finally, Piccolo is obtained by incorporating these Hoare rules in the general RG framework.

Remark 2. Our presentation of the potential-based semantics for SRA (fully presented in Sect. 5) deviates from the original one in [28], where it was called loSRA. The most crucial difference is that while loSRA's potentials consist of lists of per-location read options, our potentials consist of lists of *stores* assigning a value to every variable. (This is similar in spirit to the adaptation of load buffers for TSO [4,5] to snapshot buffers in [2]). Additionally, unlike loSRA, we disallow empty potential lists, require that the potentials of the different threads agree on the very last value to each location, and handle read-modify-write (RMW) instructions differently. We employed these modifications to loSRA as we observed that direct reasoning on loSRA states is rather unnatural and counterintuitive, as loSRA allows traces that *block* a thread from reading any value from certain locations (which cannot happen in the version we formulate). For example, a direct interpretation of our assertions over loSRA states would allow states in which $\tau \ltimes [x = v]$ and $\tau \ltimes [x \neq v]$ both hold (when τ does not have any option to read from x), while these assertions are naturally contradictory when interpreted on top of our modified SRA semantics. To establish confidence in the new potential-based semantics we have proved *in Coq* its equivalence to the standard execution-graph based semantics of SRA (over 5K lines of Coq proofs) [29].

3 Preliminaries: Syntax and Semantics

In this section we describe the underlying program language, leaving the shared-memory semantics parametric.

Syntax. The syntax of programs, given in Fig. 3, is mostly standard, comprising primitive (atomic) commands c and compound commands C. The non-standard

$$\begin{array}{rlrl} \textit{values} & v \in \mathsf{Val} = \{0, 1, \ldots\} & \textit{shared variables} & x, y \in \mathsf{Loc} = \{\mathtt{x}, \mathtt{y}, \ldots\} \\ \textit{local registers} & r \in \mathsf{Reg} = \{\mathsf{a}, \mathsf{b}, \ldots\} & \textit{thread identifiers} & \tau, \pi \in \mathsf{Tid} = \{\mathtt{T}_0, \mathtt{T}_1, \ldots\} \end{array}$$

$$\begin{array}{l} e ::= r \mid v \mid e + e \mid e = e \mid \neg e \mid e \wedge e \mid e \vee e \mid \ldots \\ c ::= r := e \mid \mathtt{STORE}(x, e) \mid r := \mathtt{LOAD}(x) \mid \mathtt{SWAP}(x, e) \qquad \tilde{c} ::= \langle c, \vec{r} := \vec{e} \rangle \\ C ::= c \mid \tilde{c} \mid \mathtt{skip} \mid C \mathbin{;} C \mid \mathtt{if}\ e\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}\ e\ \mathtt{do}\ C \mid C\ {}^{\tau}\|^{\tau}\ C \end{array}$$

Fig. 3. Program syntax

$$\frac{\gamma' = \gamma[r \mapsto \gamma(e)]}{r := e \gg \gamma \xrightarrow{\varepsilon} \gamma'} \qquad \frac{l = \mathtt{W}(x, \gamma(e))}{\mathtt{STORE}(x, e) \gg \gamma \xrightarrow{l} \gamma} \qquad \frac{l = \mathtt{R}(x, v) \qquad \gamma' = \gamma[r \mapsto v]}{r := \mathtt{LOAD}(x) \gg \gamma \xrightarrow{l} \gamma'}$$

$$\frac{l = \mathtt{RMW}(x, v, \gamma(e))}{\mathtt{SWAP}(x, e) \gg \gamma \xrightarrow{l} \gamma} \qquad \frac{\begin{array}{c} c \gg \gamma \xrightarrow{l_\varepsilon} \gamma_0 \\ r_1 := e_1 \gg \gamma_0 \xrightarrow{\varepsilon} \gamma_1 \quad \ldots \quad r_n := e_n \gg \gamma_{n-1} \xrightarrow{\varepsilon} \gamma_n \end{array}}{\langle c, \langle r_1, \ldots, r_n \rangle := \langle e_1, \ldots, e_n \rangle \rangle \gg \gamma \xrightarrow{l_\varepsilon} \gamma_n}$$

Fig. 4. Small-step semantics of (instrumented) primitive commands ($\tilde{c} \gg \gamma \xrightarrow{l_\varepsilon} \gamma'$)

components are instrumented commands $\tilde{c}$, which are meant to atomically execute a primitive command c and a (multiple) assignment $\boldsymbol{r} := \boldsymbol{e}$. Such instructions are needed to support auxiliary (a.k.a. ghost) variables in RG proofs. In addition, SWAP (a.k.a. atomic exchange) is an example of an RMW instruction. For brevity, other standard RMW instructions, such as FADD and CAS, are omitted.

Unlike many weak memory models that only support top-level parallelism, we include dynamic thread creation via commands of the form $C_1\ {}^{\tau_1}\|^{\tau_2}\ C_2$ that forks two threads named τ_1 and τ_2 that execute the commands C_1 and C_2, respectively. Each C_i may itself comprise further parallel compositions. Since thread identifiers are explicit, we require commands to be *well formed*. Let $\mathsf{Tid}(C)$ be the set of all thread identifiers that appear in C. A command C is *well formed*, denoted $\mathsf{wf}(C)$, if parallel compositions inside employ disjoint sets of thread identifiers. This notion is formally defined by induction on the structure of commands, with the only interesting case being $\mathsf{wf}(C_1\ {}^{\tau_1}\|^{\tau_2}\ C_2)$ if $\mathsf{wf}(C_1) \wedge \mathsf{wf}(C_2) \wedge \tau_1 \neq \tau_2 \wedge \mathsf{Tid}(C_1) \cap \mathsf{Tid}(C_2) = \emptyset$.

Program Semantics. We provide small-step operational semantics to commands independently of the memory system. To connect this semantics to a given memory system, its steps are instrumented with labels, as defined next.

Definition 1. A *label* l takes one of the following forms: a read $\mathtt{R}(x, v_{\mathtt{R}})$, a write $\mathtt{W}(x, v_{\mathtt{W}})$, a read-modify-write $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$, a fork $\mathtt{FORK}(\tau_1, \tau_2)$, or a join $\mathtt{JOIN}(\tau_1, \tau_2)$, where $x \in \mathsf{Loc}$, $v_{\mathtt{R}}, v_{\mathtt{W}} \in \mathsf{Val}$, and $\tau_1, \tau_2 \in \mathsf{Tid}$. We denote by Lab the set of all labels.

$$\frac{\tilde{c} \gg \gamma \xrightarrow{l_\varepsilon} \gamma'}{\langle \tilde{c}, \gamma \rangle \xrightarrow{l_\varepsilon} \langle \texttt{skip}, \gamma' \rangle} \qquad \frac{\langle C_1, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C_1', \gamma' \rangle}{\langle C_1 \,;\, C_2, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C_1' \,;\, C_2, \gamma' \rangle} \qquad \frac{}{\langle \texttt{skip} \,;\, C_2, \gamma \rangle \xrightarrow{\varepsilon} \langle C_2, \gamma \rangle}$$

$$\frac{\begin{array}{c}\gamma(e) = \mathit{true} \Rightarrow i = 1 \\ \gamma(e) \neq \mathit{true} \Rightarrow i = 2\end{array}}{\langle \texttt{if}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2, \gamma \rangle \xrightarrow{\varepsilon} \langle C_i, \gamma \rangle} \qquad \frac{C' = \texttt{if}\ e\ \texttt{then}\ (C \,;\, \texttt{while}\ e\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}}{\langle \texttt{while}\ e\ \texttt{do}\ C, \gamma \rangle \xrightarrow{\varepsilon} \langle C', \gamma \rangle}$$

Fig. 5. Small-step semantics of commands ($\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle$)

$$\frac{\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle}{\langle \mathcal{C}_0 \uplus \{\tau \mapsto C\}, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle \mathcal{C}_0 \uplus \{\tau \mapsto C'\}, \gamma \rangle} \qquad \frac{\begin{array}{c}\mathcal{C}(\tau) = C_1 \ {}^{\tau_1}\|^{\tau_2}\ C_2 \\ \tau_1 \notin \mathit{dom}(\mathcal{C}) \qquad \tau_2 \notin \mathit{dom}(\mathcal{C}) \\ l = \texttt{FORK}(\tau_1, \tau_2) \\ \mathcal{C}' = \{\tau_1 \mapsto C_1, \tau_2 \mapsto C_2\}\end{array}}{\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l} \langle \mathcal{C} \uplus \mathcal{C}', \gamma \rangle} \qquad \frac{\begin{array}{c}\mathcal{C} = \left\{\begin{array}{l}\tau \mapsto C_1 \ {}^{\tau_1}\|^{\tau_2}\ C_2, \\ \tau_1 \mapsto \texttt{skip}, \tau_2 \mapsto \texttt{skip}\end{array}\right\} \\ l = \texttt{JOIN}(\tau_1, \tau_2) \\ \mathcal{C}' = \{\tau \mapsto \texttt{skip}\}\end{array}}{\langle \mathcal{C}_0 \uplus \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l} \langle \mathcal{C}_0 \uplus \mathcal{C}', \gamma \rangle}$$

Fig. 6. Small-step semantics of command pools ($\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle \mathcal{C}', \gamma' \rangle$)

Definition 2. A *register store* is a mapping $\gamma : \mathsf{Reg} \to \mathsf{Val}$. Register stores are extended to expressions as expected. We denote by Γ the set of all register stores.

The semantics of (instrumented) primitive commands is given in Fig. 4. Using this definition, the semantics of commands is given in Fig. 5. Its steps are of the form $\langle C, \gamma \rangle \xrightarrow{l_\varepsilon} \langle C', \gamma' \rangle$ where C and C' are commands, γ and γ' are register stores, and $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$ (ε denotes a thread internal step). We lift this semantics to *command pools* as follows.

Definition 3. A *command pool* is a non-empty partial function $\mathcal{C}$ from thread identifiers to commands, such that the following hold:

1. $\mathsf{Tid}(\mathcal{C}(\tau_1)) \cap \mathsf{Tid}(\mathcal{C}(\tau_2)) = \emptyset$ for every $\tau_1 \neq \tau_2$ in $\mathit{dom}(\mathcal{C})$.
2. $\tau \notin \mathsf{Tid}(\mathcal{C}(\tau))$ for every $\tau \in \mathit{dom}(\mathcal{C})$.

We write command pools as sets of the form $\{\tau_1 \mapsto C_1, \ldots, \tau_n \mapsto C_n\}$.

Steps for command pools are given in Fig. 6. They take the form $\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau, l_\varepsilon} \langle \mathcal{C}', \gamma' \rangle$, where $\mathcal{C}$ and $\mathcal{C}'$ are command pools, γ and γ' are register stores, and $\langle \tau : l_\varepsilon \rangle$ (with $\tau \in \mathsf{Tid}$ and $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$) is a *command transition label.*

Memory Semantics. To give semantics to programs under a memory model, we synchronize the transitions of a command C with a memory system. We leave the memory system parametric, and assume that it is represented by a labeled transition system (LTS) $\mathcal{M}$ with set of states denoted by $\mathcal{M}.\mathtt{Q}$, and steps denoted by $\rightarrow_{\mathcal{M}}$. The transition labels of general memory system $\mathcal{M}$ consist of non-silent program transition labels (elements of $\mathsf{Tid} \times \mathsf{Lab}$) and a (disjoint) set $\mathcal{M}.\boldsymbol{\Theta}$ of internal memory actions, which is again left parametric (used, e.g., for memory-internal propagation of values).

Example 1. The simple memory system that guarantees sequential consistency is denoted here by SC. This memory system tracks the most recent value written to each variable and has no internal transitions ($\mathsf{SC}.\boldsymbol{\Theta} = \emptyset$). Formally, it is defined by $\mathsf{SC}.\mathtt{Q} \triangleq \mathsf{Loc} \to \mathsf{Val}$ and $\to_{\mathsf{SC}}$ is given by:

$$\frac{l = \mathtt{R}(x, v_{\mathtt{R}}) \quad m(x) = v_{\mathtt{R}}}{m \xrightarrow{\tau,l}_{\mathsf{SC}} m} \qquad \frac{l = \mathtt{W}(x, v_{\mathtt{W}}) \quad m' = m[x \mapsto v_{\mathtt{W}}]}{m \xrightarrow{\tau,l}_{\mathsf{SC}} m'} \qquad \frac{l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \quad m(x) = v_{\mathtt{R}} \quad m' = m[x \mapsto v_{\mathtt{W}}]}{m \xrightarrow{\tau,l}_{\mathsf{SC}} m'} \qquad \frac{l \in \{\mathtt{FORK}(_,_), \mathtt{JOIN}(_,_)\}}{m \xrightarrow{\tau,l}_{\mathsf{SC}} m}$$

The composition of a program with a general memory system is defined next.

Definition 4. The *concurrent system* induced by a memory system $\mathcal{M}$, denoted by $\overline{\mathcal{M}}$, is the LTS whose transition labels are the elements of $(\mathsf{Tid} \times (\mathsf{Lab} \cup \{\varepsilon\})) \uplus \mathcal{M}.\boldsymbol{\Theta}$; states are triples of the form $\langle \mathcal{C}, \gamma, m \rangle$ where $\mathcal{C}$ is a command pool, γ is a register store, and $m \in \mathcal{M}.\mathtt{Q}$; and the transitions are "synchronized transitions" of the program and the memory system, using labels to decide what to synchronize on, formally given by:

$$\frac{\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau,l} \langle \mathcal{C}', \gamma' \rangle \quad l \in \mathsf{Lab} \quad m \xrightarrow{\tau,l}_{\mathcal{M}} m'}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\tau,l}_{\overline{\mathcal{M}}} \langle \mathcal{C}', \gamma', m' \rangle} \qquad \frac{\langle \mathcal{C}, \gamma \rangle \xrightarrow{\tau,\varepsilon} \langle \mathcal{C}', \gamma' \rangle}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\tau,\varepsilon}_{\overline{\mathcal{M}}} \langle \mathcal{C}', \gamma', m \rangle} \qquad \frac{\theta \in \mathcal{M}.\boldsymbol{\Theta} \quad m \xrightarrow{\theta}_{\mathcal{M}} m'}{\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\theta}_{\overline{\mathcal{M}}} \langle \mathcal{C}, \gamma, m' \rangle}$$

4 Generic Rely-Guarantee Reasoning

In this section we present our generic RG framework. Rather than committing to a specific assertion language, our reasoning principles apply on the *semantic level*, using sets of states instead of syntactic assertions. The structure of proofs still follows program structure, thereby retaining RG's compositionality. By doing so, we decouple the semantic insights of RG reasoning from a concrete syntax. Next, we present proof rules serving as blueprints for memory model specific proof systems. An instantiation of this blueprint requires lifting the semantic principles to syntactic ones. More specifically, it requires

1. a language with (a) concrete assertions for specifying sets of states and (b) operators that match operations on sets of states (like $\wedge$ matches $\cap$); and
2. sound Hoare triples for primitive commands.

Thus, each instance of the framework (for a specific memory system) is left with the task of identifying useful abstractions on states, as well as a suitable formalism, for making the generic semantic framework into a proof system.

RG Judgments. We let $\mathcal{M}$ be an arbitrary memory system and $\Sigma_{\mathcal{M}} \triangleq \Gamma \times \mathcal{M}.\mathtt{Q}$. Properties of programs $\mathcal{C}$ are stated via *RG judgments*:

$$\mathcal{C} \ \underline{sat}_{\mathcal{M}} \ (P, \mathcal{R}, \mathcal{G}, Q)$$

where $P, Q \subseteq \Sigma_{\mathcal{M}}$, $\mathcal{R} \subseteq \mathcal{P}(\Sigma_{\mathcal{M}})$, and $\mathcal{G}$ is a set of *guarded commands*, each of which takes the form $\{G\}\ \tau \mapsto \alpha$, where $G \subseteq \Sigma_{\mathcal{M}}$ and α is either an (instrumented) primitive command $\tilde{c}$ or a fork/join label (of the form $\texttt{FORK}(\tau_1, \tau_2)$ or $\texttt{JOIN}(\tau_1, \tau_2)$). The latter is needed for considering the effect of forks and joins on the memory state.

Interpretation of RG Judgments. RG judgments $\mathcal{C}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q)$ state that a terminating run of $\mathcal{C}$ starting from a state in P, under any concurrent context whose transitions preserve each of the sets of states in $\mathcal{R}$, will end in a state in Q and perform only transitions contained in $\mathcal{G}$. To formally define this statement, following the standard model for RG, these judgments are interpreted on *computations* of programs. Computations arise from runs of the concurrent system (see Definition 4) by abstracting away from concrete transition labels and including arbitrary "environment transitions" representing steps of the concurrent context. We have:

- *Component* transitions of the form $\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\texttt{cmp}} \langle \mathcal{C}', \gamma', m' \rangle$.
- *Memory* transitions, which correspond to internal memory steps (labeled with $\theta \in \mathcal{M}.\mathbf{\Theta}$), of the form $\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\texttt{mem}} \langle \mathcal{C}, \gamma, m' \rangle$.
- *Environment* transitions of the form $\langle \mathcal{C}, \gamma, m \rangle \xrightarrow{\texttt{env}} \langle \mathcal{C}, \gamma', m' \rangle$.

Note that memory transitions do not occur in the classical RG presentation (since SC does not have internal memory actions).

A *computation* is a (potentially infinite) sequence

$$\xi = \langle \mathcal{C}_0, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \mathcal{C}_1, \gamma_1, m_1 \rangle \xrightarrow{a_2} \ldots$$

with $a_i \in \{\texttt{cmp}, \texttt{env}, \texttt{mem}\}$. We let $\langle \mathcal{C}_{\mathsf{last}(\xi)}, \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle$ denotes its last element, when ξ is finite. We say that ξ is a computation *of a command pool* $\mathcal{C}$ when $\mathcal{C}_0 = \mathcal{C}$ and for every $i \geq 0$:

- If $a_i = \texttt{cmp}$, then $\langle \mathcal{C}_i, \gamma_i, m_i \rangle \xrightarrow{\tau, l_\varepsilon}_{\mathcal{M}} \langle \mathcal{C}_{i+1}, \gamma_{i+1}, m_{i+1} \rangle$ for some $\tau \in \mathsf{Tid}$ and $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$.
- If $a_i = \texttt{mem}$, then $\langle \mathcal{C}_i, \gamma_i, m_i \rangle \xrightarrow{\theta}_{\mathcal{M}} \langle \mathcal{C}_{i+1}, \gamma_{i+1}, m_{i+1} \rangle$ for some $\theta \in \mathcal{M}.\mathbf{\Theta}$.

We denote by $\mathit{Comp}(\mathcal{C})$ the set of all computations of a command pool $\mathcal{C}$.

To define validity of RG judgments, we use the following definition.

Definition 5. Let $\xi = \langle \mathcal{C}_0, \gamma_0, m_0 \rangle \xrightarrow{a_1} \langle \mathcal{C}_1, \gamma_1, m_1 \rangle \xrightarrow{a_2} \ldots$ be a computation, and $\mathcal{C}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q)$ an RG-judgment.

- ξ admits P if $\langle \gamma_0, m_0 \rangle \in P$.
- ξ admits $\mathcal{R}$ if $\langle \gamma_i, m_i \rangle \in R \Rightarrow \langle \gamma_{i+1}, m_{i+1} \rangle \in R$ for every $R \in \mathcal{R}$ and $i \geq 0$ with $a_{i+1} = \texttt{env}$.
- ξ admits $\mathcal{G}$ if for every $i \geq 0$ with $a_{i+1} = \texttt{cmp}$ and $\langle \gamma_i, m_i \rangle \neq \langle \gamma_{i+1}, m_{i+1} \rangle$ there exists $\{P\}\ \tau \mapsto \alpha \in \mathcal{G}$ such that $\langle \gamma_i, m_i \rangle \in P$ and
  - if $\alpha = \tilde{c}$ is an instrumented primitive command, then for some $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$, we have $\langle \{\tau \mapsto \tilde{c}\}, \gamma_i, m_i \rangle \xrightarrow{\tau, l_\varepsilon}_{\mathcal{M}} \langle \{\tau \mapsto \texttt{skip}\}, \gamma_{i+1}, m_{i+1} \rangle$

$$\text{SKIP}\ \frac{}{\{\tau \mapsto \texttt{skip}\}\ \underline{sat}_{\mathcal{M}}\ (P, \{P\}, \emptyset, P)} \qquad \text{COM}\ \frac{\mathcal{M} \vDash \{P\}\ \tau \mapsto \tilde{c}\ \{Q\}}{\{\tau \mapsto \tilde{c}\}\ \underline{sat}_{\mathcal{M}}\ (P, \{P, Q\}, \{\{P\}\ \tau \mapsto \tilde{c}\}, Q)}$$

$$\text{SEQ}\ \frac{\{\tau \mapsto C_1\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}_1, \mathcal{G}_1, R) \qquad \{\tau \mapsto C_2\}\ \underline{sat}_{\mathcal{M}}\ (R, \mathcal{R}_2, \mathcal{G}_2, Q)}{\{\tau \mapsto C_1\ ;\ C_2\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{G}_1 \cup \mathcal{G}_2, Q)}$$

$$\text{IF}\ \frac{\{\tau \mapsto C_1\}\ \underline{sat}_{\mathcal{M}}\ (P \cap [\![e]\!], \mathcal{R}_1, \mathcal{G}_1, Q) \qquad \{\tau \mapsto C_2\}\ \underline{sat}_{\mathcal{M}}\ (P \setminus [\![e]\!], \mathcal{R}_2, \mathcal{G}_2, Q)}{\{\tau \mapsto \texttt{if}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P\}, \mathcal{G}_1 \cup \mathcal{G}_2, Q)}$$

$$\text{WHILE}\ \frac{P \setminus [\![e]\!] \subseteq Q \qquad \{\tau \mapsto C\}\ \underline{sat}_{\mathcal{M}}\ (P \cap [\![e]\!], \mathcal{R}, \mathcal{G}, P)}{\{\tau \mapsto \texttt{while}\ e\ \texttt{do}\ C\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R} \cup \{P, Q\}, \mathcal{G}, Q)}$$

$$\text{PAR}\ \frac{\begin{array}{c}\{\tau_1 \mapsto C_1\}\ \underline{sat}_{\mathcal{M}}\ (P_1, \mathcal{R}_1, \mathcal{G}_1, Q_1) \qquad \{\tau_2 \mapsto C_2\}\ \underline{sat}_{\mathcal{M}}\ (P_2, \mathcal{R}_2, \mathcal{G}_2, Q_2) \\ P \subseteq P_1 \cap P_2 \qquad Q_1 \cap Q_2 \subseteq Q \qquad \langle \mathcal{R}_1, \mathcal{G}_1 \rangle \text{ and } \langle \mathcal{R}_2, \mathcal{G}_2 \rangle \text{ are non-interfering}\end{array}}{\{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{P, Q\}, \mathcal{G}_1 \cup \mathcal{G}_2, Q)}$$

$$\text{FORK-JOIN}\ \frac{\begin{array}{c}\mathcal{M} \vDash \{P\}\ \tau \mapsto \texttt{FORK}(\tau_1, \tau_2)\ \{P'\} \qquad \mathcal{M} \vDash \{Q'\}\ \tau \mapsto \texttt{JOIN}(\tau_1, \tau_2)\ \{Q\} \\ \{\tau_1 \mapsto C_1\} \uplus \{\tau_2 \mapsto C_2\}\ \underline{sat}_{\mathcal{M}}\ (P', \mathcal{R}, \mathcal{G}, Q') \\ \mathcal{G}' = \mathcal{G} \cup \{\{P\}\ \tau \mapsto \texttt{FORK}(\tau_1, \tau_2), \{Q'\}\ \tau \mapsto \texttt{JOIN}(\tau_1, \tau_2)\}\end{array}}{\{\tau \mapsto C_1\ {}^{\tau_1}\|^{\tau_2}\ C_2\}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R} \cup \{P, Q\}, \mathcal{G}', Q)}$$

Fig. 7. Generic sequential RG proof rules (letting $[\![e]\!] = \{\langle \gamma, m \rangle \mid \gamma(e) = \mathit{true}\}$)

  - if $\alpha \in \{\texttt{FORK}(\tau_1, \tau_2), \texttt{JOIN}(\tau_1, \tau_2)\}$, then $m_i \xrightarrow{\tau,\alpha}_{\mathcal{M}} m_{i+1}$ and $\gamma_i = \gamma_{i+1}$.
- ξ admits Q if $\langle \gamma_{\mathsf{last}(\xi)}, m_{\mathsf{last}(\xi)} \rangle \in Q$ whenever ξ is finite and $\mathcal{C}_{\mathsf{last}(\xi)}(\tau) = \texttt{skip}$ for every $\tau \in \mathit{dom}(\mathcal{C}_{\mathsf{last}(\xi)})$.

We denote by $\mathsf{Assume}(P, \mathcal{R})$ the set of all computations that admit P and $\mathcal{R}$, and by $\mathsf{Commit}(\mathcal{G}, Q)$ the set of all computations that admit $\mathcal{G}$ and Q.

Then, *validity* of a judgment if defined as

$$\vDash \mathcal{C}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q) \overset{\triangle}{\Leftrightarrow} \mathit{Comp}(\mathcal{C}) \cap \mathsf{Assume}(P, \mathcal{R}) \subseteq \mathsf{Commit}(\mathcal{G}, Q)$$

Memory Triples. Our proof rules build on *memory triples*, which specify pre- and postconditions for primitive commands for a memory system $\mathcal{M}$.

Definition 6. A *memory triple for a memory system* $\mathcal{M}$ is a tuple of the form $\{P\}\ \tau \mapsto \alpha\ \{Q\}$, where $P, Q \subseteq \Sigma_{\mathcal{M}}$, $\tau \in \mathsf{Tid}$, and α is either an instrumented primitive command, a fork label, or a join label. A memory triple for $\mathcal{M}$ is *valid*, denoted by $\mathcal{M} \vDash \{P\}\ \tau \mapsto \alpha\ \{Q\}$, if the following hold for every $\langle \gamma, m \rangle \in P$, $\gamma' \in \Gamma$ and $m' \in \mathcal{M}.\mathsf{Q}$:

- if α is an instrumented primitive command and $\langle \{\tau \mapsto \alpha\}, \gamma, m \rangle \xrightarrow{\tau, l_\varepsilon}_{\mathcal{M}} \langle \{\tau \mapsto \texttt{skip}\}, \gamma', m' \rangle$ for some $l_\varepsilon \in \mathsf{Lab} \cup \{\varepsilon\}$, then $\langle \gamma', m' \rangle \in Q$.

– If $\alpha \in \{\texttt{FORK}(\tau_1, \tau_2), \texttt{JOIN}(\tau_1, \tau_2)\}$ and $m \xrightarrow{\tau,\alpha}_{\mathcal{M}} m'$, then $\langle \gamma, m' \rangle \in Q$.

Example 2. For the memory system SC introduced in Example 1, we have, e.g., memory triples of the form $\mathsf{SC} \vDash \{e(r := x)\}\ \tau \mapsto r := \texttt{LOAD}(x)\ \{e\}$ (where $e(r := x)$ is the expression e with all occurrences of r replaced by x).

RG Proof Rules. We aim at proof rules deriving valid RG judgments. Figure 7 lists (semantic) proof rules based on externally provided memory triples. These rules basically follows RG reasoning for sequential consistency. For example, rule SEQ states that RG judgments of commands C_1 and C_2 can be combined when the postcondition of C_1 and the precondition of C_2 agree, thereby uniting their relies and guarantees. Rule COM builds on memory triples. The rule PAR for parallel composition combines judgments for two components when their relies and guarantees are *non-interfering*. Intuitively speaking, this means that each of the assertions that each thread relied on for establishing its proof is preserved when applying any of the assignments collected in the guarantee set of the other thread. An example of non-interfering rely-guarantee pairs is given in step 7 in Sect. 2. Formally, non-interference is defined as follows:

Definition 7. Rely-guarantee pairs $\langle \mathcal{R}_1, \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2, \mathcal{G}_2 \rangle$ are *non-interfering* if $\mathcal{M} \vDash \{R \cap P\}\ \tau \mapsto \alpha\ \{R\}$ holds for every $R \in \mathcal{R}_1$ and $\{P\}\ \tau \mapsto \alpha \in \mathcal{G}_2$, and similarly for every $R \in \mathcal{R}_2$ and $\{P\}\ \tau \mapsto \alpha \in \mathcal{G}_1$.

In turn, FORK-JOIN combines the proof of a parallel composition with proofs of fork and join steps (which may also affect the memory state). Note that the guarantees also involve guarded commands with FORK and JOIN labels.

Additional rules for consequence and introduction of auxiliary variables are elided here (they are similar to their SC counterparts), and provided in the extended version of this paper [30].

Soundness. To establish soundness of the above system we need an additional requirement regarding the internal memory transitions (for SC this closure vacuously holds as there are no such transitions). We require all relies in $\mathcal{R}$ to be *stable under internal memory transitions*, i.e. for $R \in \mathcal{R}$ we require

$$\forall \gamma, m, m', \theta \in \mathcal{M}.\boldsymbol{\Theta}.\ m \xrightarrow{\theta}_{\mathcal{M}} m' \Rightarrow (\langle \gamma, m \rangle \in R \Rightarrow \langle \gamma, m' \rangle \in R) \qquad (\mathsf{mem})$$

This condition is needed since the memory system can non-deterministically take its internal steps, and the component's proof has to be stable under such steps.

Theorem 1 (Soundness). $\vdash C\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q) \Longrightarrow \vDash C\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q)$.

With this requirement, we are able to establish soundness. The proof, which generally follows [48] is given in the extended version of this paper [30]. We write $\vdash \mathcal{C}\ \underline{sat}_{\mathcal{M}}\ (P, \mathcal{R}, \mathcal{G}, Q)$ for provability of a judgment using the semantic rules presented above.

5 Potential-Based Memory System for SRA

In this section we present the potential-based semantics for Strong Release-Acquire (SRA), for which we develop a novel RG logic. Our semantics is based on the one in [27,28], with certain adaptations to make it better suited for Hoare-style reasoning (see Remark 2).

In weak memory models, threads typically have different views of the shared memory. In SRA, we refer to a memory snapshot that a thread may observe as a *potential store*:

Definition 8. A *potential store* is a function $\delta : \mathsf{Loc} \to \mathsf{Val} \times \{\mathtt{R}, \mathtt{RMW}\} \times \mathsf{Tid}$. We write $\mathtt{val}(\delta(x))$, $\mathtt{rmw}(\delta(x))$, and $\mathtt{tid}(\delta(x))$ to retrieve the different components of $\delta(x)$. We denote by Δ the set of all potential stores.

Having $\delta(x) = \langle v, \mathtt{R}, \tau \rangle$ allows to read the value v from x (and further ascribes that this read reads from a write performed by thread τ, which is technically needed to properly characterize the SRA model). In turn, having $\delta(x) = \langle v, \mathtt{RMW}, \tau \rangle$ further allows to perform an RMW instruction that atomically reads and modifies x.

Potential stores are collected in *potential store lists* describing the values which can (potentially) be read and in what order.

Notation 9. Lists over an alphabet A are written as $L = a_1 \cdot \ldots \cdot a_n$ where $a_1, \ldots, a_n \in A$. We also use $\cdot$ to concatenate lists, and write $L[i]$ for the i'th element of L and $|L|$ for the length of L.

A *(potential) store list* is a finite sequence of potential stores ascribing a possible sequence of stores that a thread can observe, in the order it will observe them. The RMW-flags in these lists have to satisfy certain conditions: once the flag for a location is set, it remains set in the rest of the list; and the flag must be set at the end of the list. Formally, store lists are defined as follows.

Definition 10. A *store list* $L \in \mathcal{L}$ is a non-empty finite sequence of potential stores with *monotone RMW-flags* ending with an RMW, that is: for all $x \in \mathsf{Loc}$,

1. if $\mathtt{rmw}(L[i](x)) = \mathtt{RMW}$, then $\mathtt{rmw}(L[j](x)) = \mathtt{RMW}$ for every $i < j \leq |L|$, and
2. $\mathtt{rmw}(L[|L|](x)) = \mathtt{RMW}$.

Now, SRA states ($\mathsf{SRA}.\mathtt{Q}$) consist of *potential mappings* that assign potentials to threads as defined next.

Definition 11. A *potential* D is a non-empty set of potential store lists. A *potential mapping* is a function $\mathcal{D} : \mathsf{Tid} \rightharpoonup \mathcal{P}(\mathcal{L}) \setminus \{\emptyset\}$ that maps thread identifiers to potentials such that all lists agree on the very final potential store (that is: $L_1[|L_1|] = L_2[|L_2|]$ whenever $L_1 \in \mathcal{D}(\tau_1)$ and $L_2 \in \mathcal{D}(\tau_2)$).

These potential mappings are "lossy" meaning that potential stores can be arbitrarily dropped. In particular, dropping the first store in a list enables reading from the second. This is formally done by transitioning from a state $\mathcal{D}$ to a "smaller" state $\mathcal{D}'$ as defined next.

$$\textsc{Write}\ \frac{\begin{array}{c}\forall L' \in \mathcal{D}'(\tau).\, \exists L \in \mathcal{D}(\tau).\, L' = L[x \mapsto \langle v_{\mathtt{W}}, \mathtt{RMW}, \tau\rangle] \\ \forall \pi \in \mathit{dom}(\mathcal{D}) \setminus \{\tau\}, L' \in \mathcal{D}'(\pi).\, \exists L_0, L_1. \\ L_0 \cdot L_1 \in \mathcal{D}(\pi) \land L_1 \in \mathcal{D}(\tau) \land \\ L' = L_0[x \mapsto \mathtt{R}] \cdot L_1[x \mapsto \langle v_{\mathtt{W}}, \mathtt{RMW}, \tau\rangle]\end{array}}{\mathcal{D} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{SRA}} \mathcal{D}'} \qquad \textsc{Lose}\ \frac{\mathcal{D}' \sqsubseteq \mathcal{D}}{\mathcal{D} \xrightarrow{\varepsilon}_{\mathsf{SRA}} \mathcal{D}'} \qquad \textsc{Dup}\ \frac{\mathcal{D} \preceq \mathcal{D}'}{\mathcal{D} \xrightarrow{\varepsilon}_{\mathsf{SRA}} \mathcal{D}'}$$

$$\textsc{Read}\ \frac{\begin{array}{c}\exists \pi.\, \forall L \in \mathcal{D}(\tau).\, \mathtt{val}(L[1](x)) = v_{\mathtt{R}} \land \\ \mathtt{tid}(L[1](x)) = \pi\end{array}}{\mathcal{D} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\mathsf{SRA}} \mathcal{D}} \qquad \textsc{RMW}\ \frac{\begin{array}{c}\forall L \in \mathcal{D}(\tau).\, \mathtt{rmw}(L[1](x)) = \mathtt{RMW} \\ \mathcal{D} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\mathsf{SRA}} \mathcal{D} \qquad \mathcal{D} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{SRA}} \mathcal{D}'\end{array}}{\mathcal{D} \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})}_{\mathsf{SRA}} \mathcal{D}'}$$

$$\textsc{Fork}\ \frac{\begin{array}{c}\mathcal{D}_{\text{new}} = \{\tau_1 \mapsto \mathcal{D}(\tau), \tau_2 \mapsto \mathcal{D}(\tau)\} \\ \mathcal{D}' = \mathcal{D}|_{\mathit{dom}(\mathcal{D}) \setminus \{\tau\}} \uplus \mathcal{D}_{\text{new}}\end{array}}{\mathcal{D} \xrightarrow{\tau, \mathtt{FORK}(\tau_1, \tau_2)}_{\mathsf{SRA}} \mathcal{D}'} \qquad \textsc{Join}\ \frac{\begin{array}{c}\mathcal{D}_{\text{new}} = \{\tau \mapsto \mathcal{D}(\tau_1) \cap \mathcal{D}(\tau_2)\} \\ \mathcal{D}' = \mathcal{D}|_{\mathit{dom}(\mathcal{D}) \setminus \{\tau_1, \tau_2\}} \uplus \mathcal{D}_{\text{new}}\end{array}}{\mathcal{D} \xrightarrow{\tau, \mathtt{JOIN}(\tau_1, \tau_2)}_{\mathsf{SRA}} \mathcal{D}'}$$

Fig. 8. Steps of SRA (defining $\delta[x \mapsto \langle v, u, \tau\rangle](y) = \langle v, u, \tau\rangle$ if $y = x$ and $\delta(y)$ else, and $\delta[x \mapsto \mathtt{R}]$ to set all RMW-flags for x to $\mathtt{R}$; both pointwise lifted to lists)

Definition 12. The (overloaded) partial order $\sqsubseteq$ is defined as follows:

1. on potential store lists: $L' \sqsubseteq L$ if L' is a nonempty subsequence of L;
2. on potentials: $D' \sqsubseteq D$ if $\forall L' \in D'.\, \exists L \in D.\, L' \sqsubseteq L$;
3. on potential mappings: $\mathcal{D}' \sqsubseteq \mathcal{D}$ if $\mathcal{D}'(\tau) \sqsubseteq \mathcal{D}(\tau)$ for every $\tau \in \mathit{dom}(\mathcal{D})$.

We also define $L \preceq L'$ if L' is obtained from L by duplication of some stores (e.g., $\delta_1 \cdot \delta_2 \cdot \delta_3 \preceq \delta_1 \cdot \delta_2 \cdot \delta_2 \cdot \delta_3$). This is lifted to potential mappings as expected.

Figure 8 defines the transitions of SRA. The LOSE and DUP steps account for losing and duplication in potentials. Note that these are both internal memory transitions (required to preserve relies as of (mem)). The FORK and JOIN steps distribute potentials on forked threads and join them at the end. The READ step obtains its value from the first store in the lists of the potential of the reader, provided that all these lists agree on that value and the writer thread identifier. RMW steps atomically perform a read and a write step where the read is restricted to an RMW-marked entry.

Most of the complexity is left for the WRITE step. It updates to the new written value for the writer thread τ. For every other thread, it updates a *suffix* (L_1) of the store list with the new value. For guaranteeing causal consistency this updated suffix cannot be arbitrary: it has to be in the potential of the writer thread ($L_1 \in \mathcal{D}(\tau)$). This is the key to achieving the "shared-memory causality principle" of [28], which ensures causal consistency.

Example 3. Consider again the MP program from Fig. 2. After the initial fork step, threads $\mathtt{T}_1$ and $\mathtt{T}_2$ may have the following store list in their potentials:

$$\mathtt{L} = \begin{bmatrix}\mathtt{x} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0\rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0\rangle\end{bmatrix} \cdot \begin{bmatrix}\mathtt{x} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0\rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0\rangle\end{bmatrix} \cdot \begin{bmatrix}\mathtt{x} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0\rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0\rangle\end{bmatrix}.$$

Then, $\mathtt{STORE}(\mathtt{x}, 1)$ by $\mathtt{T}_1$ can generate the following store list for $\mathtt{T}_2$:

$$\mathtt{L}_2 = \begin{bmatrix} \mathtt{x} \mapsto \langle 0, \mathtt{R}, \mathtt{T}_0 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0 \rangle \end{bmatrix}.$$

Thus $\mathtt{T}_2$ keeps the possibility of reading the "old" value of $\mathtt{x}$. For $\mathtt{T}_1$ this is different: the model allows the writing thread to only see its new value of $\mathtt{x}$ and all entries for $\mathtt{x}$ in the store list are updated. Thus, for $\mathtt{T}_1$ we obtain store list

$$\mathtt{L}_1 = \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{RMW}, \mathtt{T}_0 \rangle \end{bmatrix}.$$

Next, when $\mathtt{T}_1$ executes $\mathtt{STORE}(\mathtt{y}, 1)$, again, the value for $\mathtt{y}$ has to be updated to 1 in $\mathtt{T}_1$ yielding

$$\mathtt{L}_1' = \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_0 \rangle \\ \mathtt{y} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \end{bmatrix}.$$

For $\mathtt{T}_2$ the write step may change $\mathtt{L}_2$ to

$$\mathtt{L}_2' = \begin{bmatrix} \mathtt{x} \mapsto \langle 0, \mathtt{R}, \mathtt{T}_0 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{R}, \mathtt{T}_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 0, \mathtt{R}, \mathtt{T}_0 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \end{bmatrix}.$$

Thus, thread $\mathtt{T}_2$ can still see the old values, or lose the prefix of its list and see the new values. Importantly, it cannot read 1 from $\mathtt{y}$ *and then* 0 from $\mathtt{x}$. Note that $\mathtt{STORE}(\mathtt{y}, 1)$ by $\mathtt{T}_1$ *cannot* modify $\mathtt{L}_2$ to the list

$$\mathtt{L}_2'' = \begin{bmatrix} \mathtt{x} \mapsto \langle 0, \mathtt{R}, \mathtt{T}_0 \rangle \\ \mathtt{y} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \end{bmatrix} \cdot \begin{bmatrix} \mathtt{x} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \\ \mathtt{y} \mapsto \langle 1, \mathtt{RMW}, \mathtt{T}_1 \rangle \end{bmatrix},$$

as it requires $\mathtt{T}_1$ to have $\mathtt{L}_2$ in its *own potential*. This models the intended semantics of message passing under causal consistency.

The next theorem establishes the equivalence of SRA as defined above and opSRA from [28], which is an (operational version of) the standard strong release-acquire declarative semantics [26,31]. (As a corollary, we obtain the equivalence between the potential-based system from [28] and the variant we define in this paper.)

Our notion of equivalence employed in the theorem is *trace equivalence*. We let a trace of a memory system be a sequence of transition labels, ignoring ε transitions, and consider traces of SRA starting from an initial state $\lambda\tau \in \{\mathtt{T}_1, \ldots, \mathtt{T}_N\}.\, \{\langle \lambda x.\, \langle 0, \mathtt{RMW}, \mathtt{T}_0 \rangle \rangle\}$ and traces of opSRA starting from the initial execution graph that consists of a write event to every location writing 0 by a distinguished initialization thread $\mathtt{T}_0$.

Theorem 2. *A trace is generated by* SRA *iff it is generated by opSRA.*

The proof is of this theorem is by simulation arguments (forward simulation in one direction and backward for the converse). It is mechanized in Coq [29]. The mechanized proof does not consider fork and join steps, but they can be straightforwardly added.

$$
\begin{array}{lll}
\textit{extended expressions} & E & ::= e \mid x \mid \mathtt{R}(x) \mid E + E \mid \neg E \mid E \wedge E \mid \ldots \\
\textit{interval assertions} & I & ::= [E] \mid I\,;I \mid I \wedge I \mid I \vee I \\
\textit{assertions} & \varphi, \psi & ::= \tau \ltimes I \mid e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi
\end{array}
$$

Fig. 9. Assertions of Piccolo

6 Program Logic

For the instantiation of our RG framework to SRA, we next (1) introduce the assertions of the logic Piccolo and (2) specify memory triples for Piccolo. Our logic is inspired by *interval logics* like Moszkowski's ITL [35] or duration calculus [13].

Syntax and Semantics. Figure 9 gives the grammar of Piccolo. We base it on *extended expressions* which—besides registers—can also involve locations as well as expressions of the form $\mathtt{R}(x)$ (to indicate RMW-flag R). Extended expressions E can hold on entire *intervals* of a store list (denoted $[E]$). Store lists can be split into intervals satisfying different interval expressions (I_1 ; ... ; I_n) using the ";" operator (called "chop"). In turn, $\tau \ltimes I$ means that all store lists in τ's potential satisfy I. For an assertion φ, we let $fv(\varphi) \subseteq \mathsf{Reg} \cup \mathsf{Loc} \cup \mathsf{Tid}$ be the set of registers, locations and thread identifiers occurring in φ, and write $\mathtt{R}(x) \in \varphi$ to indicate that the term $\mathtt{R}(x)$ occurs in φ.

As an example consider again MP (Fig. 2). We would like to express that $\mathtt{T}_2$ upon seeing y to be 1 cannot see the old value 0 of x anymore. In Piccolo this is expressed as $\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1]\,;[\mathtt{x} = 1]$: the store lists of $\mathtt{T}_2$ can be split into two intervals (one possibly empty), the first satisfying $\mathtt{y} \neq 1$ and the second $\mathtt{x} = 1$.

Formally, an assertion φ describes register stores coupled with SRA states:

Definition 13. Let γ be a register store, δ a potential store, L a store list, and $\mathcal{D}$ a potential mapping. We let $[\![e]\!]_{\langle\gamma,\delta\rangle} = \gamma(e)$, $[\![x]\!]_{\langle\gamma,\delta\rangle} = \delta(x)$, and $[\![\mathtt{R}(x)]\!]_{\langle\gamma,\delta\rangle} =$ if $\mathtt{rmw}(\delta(x)) = \mathtt{R}$ then *true* else *false*. The extension of this notation to any extended expression E is standard. The validity of assertions in $\langle\gamma, \mathcal{D}\rangle$, denoted by $\langle\gamma, \mathcal{D}\rangle \models \varphi$, is defined as follows:

1. $\langle\gamma, L\rangle \models [E]$ if $[\![E]\!]_{\langle\gamma,\delta\rangle} = true$ for every $\delta \in L$.
2. $\langle\gamma, L\rangle \models I_1\,;I_2$ if $\langle\gamma, L_1\rangle \models I_1$ and $\langle\gamma, L_2\rangle \models I_2$ for some (possibly empty) L_1 and L_2 such that $L = L_1 \cdot L_2$.
3. $\langle\gamma, L\rangle \models I_1 \wedge I_2$ if $\langle\gamma, L\rangle \models I_1$ and $\langle\gamma, L\rangle \models I_2$ (similarly for $\vee$).
4. $\langle\gamma, \mathcal{D}\rangle \models \tau \ltimes I$ if $\langle\gamma, L\rangle \models I$ for every $L \in \mathcal{D}(\tau)$.
5. $\langle\gamma, \mathcal{D}\rangle \models e$ if $\gamma(e) = true$.
6. $\langle\gamma, \mathcal{D}\rangle \models \varphi_1 \wedge \varphi_2$ if $\langle\gamma, \mathcal{D}\rangle \models \varphi_1$ and $\langle\gamma, \mathcal{D}\rangle \models \varphi_2$ (similarly for $\vee$).

Note that with $\wedge$ and $\vee$ as well as negation on expressions,[1] the logic provides the operators on sets of states necessary for an instantiation of our RG framework. Further, the requirements from SRA states guarantee certain properties:

[1] Negation just occurs on the level of simple expressions e which is sufficient for calculating $P \setminus [\![e]\!]$ required in rules IF and WHILE.

Assumption	Pre	Command	Post	Reference
	$\{\varphi(r := e)\}$	$\tau \mapsto r := e$	$\{\varphi\}$	SUBST-ASGN
$x \notin fv(\varphi)$	$\{\varphi\}$	$\tau \mapsto \texttt{WRITE}(x, e)$	$\{\varphi\}$	STABLE-WR
$r \notin fv(\varphi)$	$\{\varphi\}$	$\tau \mapsto r := \texttt{LOAD}(x)$	$\{\varphi\}$	STABLE-LD
$\tau \notin fv(\varphi)$	$\{\varphi\}$	$\tau \mapsto \texttt{FORK}(\tau_1, \tau_2)$	$\{\varphi\}$	STABLE-FORK
$\tau \notin fv(\varphi)$	$\{\varphi\}$	$\tau \mapsto \texttt{JOIN}(\tau_1, \tau_2)$	$\{\varphi\}$	STABLE-JOIN
	$\{e \wedge \tau \ltimes I\}$	$\tau \mapsto \texttt{FORK}(\tau_1, \tau_2)$	$\{e \wedge \tau_1 \ltimes I \wedge \tau_2 \ltimes I\}$	FORK
	$\{e \wedge \tau_1 \ltimes I \wedge \tau_2 \ltimes I\}$	$\tau \mapsto \texttt{JOIN}(\tau_1, \tau_2)$	$\{e \wedge \tau \ltimes I\}$	JOIN
	$\{True\}$	$\tau \mapsto \texttt{WRITE}(x, e)$	$\{\tau \ltimes [x = e]\}$	WR-OWN
$\texttt{R}(x) \notin I$	$\{\pi \ltimes I\}$	$\tau \mapsto \texttt{WRITE}(x, e)$	$\{\pi \ltimes (I \wedge [\texttt{R}(x)]) ; [x = e]\}$	WR-OTHER-1
$x \notin fv(I_\tau)$, $\texttt{R}(x) \notin I$	$\{\tau \ltimes I_\tau \wedge \pi \ltimes I ; I_\tau\}$	$\tau \mapsto \texttt{WRITE}(x, e)$	$\{\pi \ltimes I ; I_\tau\}$	WR-OTHER-2
$x \notin fv(I_\tau)$	$\{\tau \ltimes I_\tau\}$	$\tau \mapsto \texttt{WRITE}(x, e)$	$\{\pi \ltimes [\texttt{R}(x)] ; I_\tau\}$	WR-OTHER-3
$x \notin fv(I)$	$\{\tau \ltimes [\texttt{R}(x)] ; I\}$	$\tau \mapsto \texttt{SWAP}(x, e)$	$\{\tau \ltimes I\}$	SWAP-SKIP

Fig. 10. Memory triples for Piccolo using $\texttt{WRITE} \in \{\texttt{SWAP}, \texttt{STORE}\}$ and assuming $\tau \neq \pi$

- For $\varphi_1 = \tau \ltimes [E_1^\tau] ; ... ; [E_n^\tau]$ and $\varphi_2 = \pi \ltimes [E_1^\pi] ; ... ; [E_m^\pi]$: if $E_i^\tau \wedge E_j^\pi \Rightarrow False$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$, then $\varphi_1 \wedge \varphi_2 \Rightarrow False$ (follows from the fact that all lists in potentials are non-empty and agree on the last store).
- If $\langle \gamma, \mathcal{D} \rangle \models \tau \ltimes [\texttt{R}(x)] ; [E]$, then every list $L \in \mathcal{D}(\tau)$ contains a non-empty suffix satisfying E (since all lists have to end with RMW-flags set on).

All assertions are preserved by steps LOSE and DUP. This stability is required by our RG framework (Condition (mem))[2]. Stability is achieved here because negations occur on the level of (simple) expressions only (e.g., we cannot have $\neg(\tau \ltimes [x = v])$, meaning that τ must have a store in its potential whose value for x is not v, which would not be stable under LOSE).

Proposition 1. *If* $\langle \gamma, \mathcal{D} \rangle \models \varphi$ *and* $\mathcal{D} \xrightarrow{\varepsilon}_{\mathsf{SRA}} \mathcal{D}'$, *then* $\langle \gamma, \mathcal{D}' \rangle \models \varphi$.

Memory Triples. Assertions in Piccolo describe sets of states, thus can be used to formulate memory triples. Figure 10 gives the base triples for the different primitive instructions.

We see the standard SC rule of assignment (SUBST-ASGN) for registers followed by a number of stability rules detailing when assertions are not affected by instructions. Axioms FORK and JOIN describe the transfer of properties from forking thread to forked threads and back.

The next four axioms in the table concern write instructions (either SWAP or STORE). They reflect the semantics of writing in SRA: (1) In the writer thread τ all stores in all lists get updated (axiom WR-OWN). Other threads π will have (2) their lists being split into "old" values for x with R flag and the new value for x (WR-OTHER-1), (3) properties (expressed as I_τ) of suffixes of lists being preserved when the writing thread satisfies the same properties (WR-OTHER-2) and (4) their lists consisting of R-accesses to x followed by properties of the

[2] Such stability requirements are also common to other reasoning techniques for weak memory models, e.g., [19].

writer (WR-OTHER-3). The last axiom concerns SWAP only: as it can only read from store entries marked as RMW it discards intervals satisfying $[\mathtt{R}(x)]$.

Example 4. We employ the axioms for showing one proof step for MP, namely one pair in the non-interference check of the rely $\mathcal{R}_2$ of $\mathtt{T}_2$ with respect to the guarantees $\mathcal{G}_1$ of $\mathtt{T}_1$:

$$\{\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] ; [\mathtt{x} = 1] \wedge \mathtt{T}_1 \ltimes [\mathtt{x} = 1]\}\ \mathtt{T}_1 \mapsto \mathtt{STORE}(\mathtt{x}, 1)\ \{\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] ; [\mathtt{x} = 1]\}$$

By taking I_τ to be $[\mathtt{x} = 1]$, this is an instance of WR-OTHER-2.

In addition to the axioms above, we use a *shift* rule for load instructions:

$$\text{LD-SHIFT}\ \frac{\{\tau \ltimes I\}\ \tau \mapsto r := \mathtt{LOAD}(x)\ \{\psi\}}{\{\tau \ltimes [(e \wedge E)(r := x)] ; I\}\ \tau \mapsto r := \mathtt{LOAD}(x)\ \{(e \wedge \tau \ltimes [E]; I) \vee \psi\}}$$

A load instruction reads from the first store in the lists, however, if the list satisfying $[(e \wedge E)(r := x)]$ in $[(e \wedge E)(r := x)] ; I$ is empty, it reads from a list satisfying I. The shift rule for LOAD puts this shifting to next stores into a proof rule. Like the standard Hoare rule SUBST-ASGN, LD-SHIFT employs backward substitution.

Example 5. We exemplify rule LD-SHIFT on another proof step of example MP, one for local correctness of $\mathtt{T}_2$:

$$\{\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] ; [\mathtt{x} = 1]\}\ \mathtt{T}_2 \mapsto \mathsf{a} := \mathtt{LOAD}(\mathtt{y})\ \{\mathsf{a} = 1 \Rightarrow \mathtt{T}_2 \ltimes [\mathtt{x} = 1]\}$$

From axiom STABLE-LD we get $\{\mathtt{T}_2 \ltimes [\mathtt{x} = 1]\}\ \mathtt{T}_2 \mapsto \mathsf{a} := \mathtt{LOAD}(\mathtt{y})\ \{\mathtt{T}_2 \ltimes [\mathtt{x} = 1]\}$. We obtain $\{\mathtt{T}_2 \ltimes [\mathtt{y} \neq 1] ; [\mathtt{x} = 1]\}\ \mathtt{T}_2 \mapsto \mathsf{a} := \mathtt{LOAD}(\mathtt{y})\ \{\mathsf{a} \neq 1 \vee \mathtt{T}_2 \ltimes [\mathtt{x} = 1]\}$ using the former as premise forLD-SHIFT.

In addition, we include the standard conjunction, disjunction and consequence rules of Hoare logic. For instrumented primitive commands we employ the following rule:

$$\text{INSTR}\ \frac{\{\psi_0\}\ \tau \mapsto c\ \{\psi_1\}\{\psi_1\}\ \tau \mapsto r_1 := e_1\ \{\psi_2\} \ldots \{\psi_{n-1}\}\ \tau \mapsto r_n := e_n\ \{\psi_n\}}{\{\psi_0\}\ \tau \mapsto \langle c, \langle r_1, \ldots, r_n\rangle := \langle e_1, \ldots, e_n\rangle\rangle\ \{\psi_n\}}$$

Finally, it can be shown that all triples derivable from axioms and rules are valid memory triples.

Lemma 1. *If a* Piccolo *memory triple is derivable,* $\vdash_{\mathsf{Piccolo}} \{\varphi\}\ \tau \mapsto \alpha\ \{\psi\}$*, then* $\mathsf{SRA} \models \{\{\langle\gamma, \mathcal{D}\rangle \mid \langle\gamma, \mathcal{D}\rangle \models \varphi\}\}\ \tau \mapsto \alpha\ \{\{\langle\gamma, \mathcal{D}\rangle \mid \langle\gamma, \mathcal{D}\rangle \models \psi\}\}$.

$\{\mathtt{T}_0 \ltimes [\mathtt{x} \neq 2]\}$

Thread $\mathtt{T}_1$	**Thread** $\mathtt{T}_2$
$\{\mathtt{T}_1 \ltimes I_0^{\mathtt{x}}\}$	$\{\mathtt{T}_2 \ltimes I_{012}^{\mathtt{x}}\}$
$1 : \mathtt{STORE}(\mathtt{x}, 1);$	$3 : \mathtt{a} := \mathtt{LOAD}(\mathtt{x});$
$\{\mathtt{T}_1 \ltimes I_1^{\mathtt{x}}\}$	$\{\mathtt{a} = 2 \Rightarrow \mathtt{T}_2 \ltimes I_2^{\mathtt{x}}\}$
$2 : \mathtt{STORE}(\mathtt{x}, 2)$	$4 : \mathtt{b} := \mathtt{LOAD}(\mathtt{x})$
$\{\mathit{True}\}$	$\{\mathtt{a} = 2 \Rightarrow \mathtt{b} = 2\}$

$\{\mathtt{a} = 2 \Rightarrow \mathtt{b} \neq 1\}$

Fig. 11. RRC for two threads (a.k.a. CoRR0)

$\{\mathtt{T}_0 \ltimes I_0^{\mathtt{x}}\}$

Thread $\mathtt{T}_1$	**Thread** $\mathtt{T}_2$	**Thread** $\mathtt{T}_3$	**Thread** $\mathtt{T}_4$
$\{\mathtt{T}_1 \ltimes I_{02}^{\mathtt{x}}\}$	$\{\mathtt{T}_1 \ltimes I_{01}^{\mathtt{x}}\}$	$\{\mathtt{T}_3 \ltimes (I_{012}^{\mathtt{x}} \vee I_{021}^{\mathtt{x}})\}$	$\{\mathtt{T}_4 \ltimes (I_{012}^{\mathtt{x}} \vee I_{021}^{\mathtt{x}})\}$
$1 : \mathtt{STORE}(\mathtt{x}, 1)$	$2 : \mathtt{STORE}(\mathtt{x}, 2)$	$3 : \mathtt{a} := \mathtt{LOAD}(\mathtt{x});$	$5 : \mathtt{c} := \mathtt{LOAD}(\mathtt{x});$
$\{\mathit{True}\}$	$\{\mathit{True}\}$	$\{\mathtt{a} = 2 \Rightarrow \mathtt{T}_3 \ltimes I_{21}^{\mathtt{x}}\}$	$\{\mathtt{c} = 1 \Rightarrow \mathtt{T}_4 \ltimes I_{12}^{\mathtt{x}}\}$
		$4 : \mathtt{b} := \mathtt{LOAD}(\mathtt{x})$	$6 : \mathtt{d} := \mathtt{LOAD}(\mathtt{x})$
		$\{\langle \mathtt{a}, \mathtt{b}\rangle = \langle 2, 1\rangle \Rightarrow \mathtt{T}_3 \ltimes I_1^{\mathtt{x}}\}$	$\{\langle \mathtt{c}, \mathtt{d}\rangle = \langle 1, 2\rangle \Rightarrow \mathtt{T}_4 \ltimes I_2^{\mathtt{x}}\}$

$\{\langle \mathtt{a}, \mathtt{b}\rangle = \langle 2, 1\rangle \Rightarrow \langle \mathtt{c}, \mathtt{d}\rangle \neq \langle 1, 2\rangle\}$

Fig. 12. RRC for four threads (a.k.a. CoRR2)

7 Examples

We discuss examples verified in Piccolo. Additional examples can be found in the extended version of this paper [30].

Coherence. We provide two coherence examples in Figs. 11 and 12, using the notation $I^x_{v_1 v_2 \ldots v_n} = [x = v_1] \mathbin{;} [x = v_2] \mathbin{;} \ldots \mathbin{;} [x = v_n]$. Figure 11 enforces an ordering on writes to the shared location $\mathtt{x}$ on thread $\mathtt{T}_1$. The postcondition guarantees that after reading the second write, thread $\mathtt{T}_2$ cannot read from the first. Figure 12 is similar, but the writes to $\mathtt{x}$ occur on two different threads. The postcondition of the program guarantees that the two different threads agree on the order of the writes. In particular if one reading thread (here $\mathtt{T}_3$) sees the value 2 then 1, it is impossible for the other reading thread (here $\mathtt{T}_4$) to see 1 then 2.

Potential assertions provide a compact and intuitive mechanism for reasoning, e.g., in Fig. 11, the precondition of line 3 precisely expresses the order of values available to thread $\mathtt{T}_2$. This presents an improvement over view-based assertions [16], which required a separate set of assertions to encode write order.

Peterson's Algorithm. Figure 13 shows Peterson's algorithm for implementing mutual exclusion for two threads [38] together with Piccolo assertions. We depict only the code of thread $\mathtt{T}_1$. Thread $\mathtt{T}_2$ is symmetric. A third thread $\mathtt{T}_3$ is assumed stopping the other two threads at an arbitrary point in time. We

Thread $\mathtt{T}_1$

$\{\neg\mathtt{a}_1 \wedge \neg\mathtt{a}_2 \wedge \mathtt{mx}_1 = 0\}$

$\mathtt{while}\ \neg\mathtt{stop}\ \mathtt{do}\ \ \{\neg\mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee \mathtt{T}_1 \ltimes [\mathtt{R}(\mathtt{turn})] \mathbin{;} [\mathtt{flag}_2])\}$

$1: \quad \mathtt{STORE}(\mathtt{flag}_1, \mathit{true}); \{\neg\mathtt{a}_1 \wedge \mathtt{T}_1 \ltimes [\mathtt{flag}_1] \wedge (\neg\mathtt{a}_2 \vee \mathtt{T}_1 \ltimes [\mathtt{R}(\mathtt{turn})] \mathbin{;} [\mathtt{flag}_2])\}$

$2: \quad \langle \mathtt{SWAP}(\mathtt{turn}, 2); \mathtt{a}_1 := \mathit{true} \rangle;$

$3: \quad \mathtt{do}\ \{\mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee \mathtt{T}_1 \ltimes [\mathtt{flag}_2 \wedge \mathtt{turn} \neq 1] \vee P)\}$

$4: \qquad \mathtt{fl}_1 := \mathtt{LOAD}(\mathtt{flag}_2); \{\mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee (\mathtt{fl}_1 \wedge \mathtt{T}_1 \ltimes [\mathtt{flag}_2 \wedge \mathtt{turn} \neq 1]) \vee P)\}$

$5: \qquad \mathtt{tu}_1 := \mathtt{LOAD}(\mathtt{turn}); \{\mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee (\mathtt{fl}_1 \wedge \mathtt{tu}_1 \neq 1 \wedge \mathtt{T}_1 \ltimes [\mathtt{flag}_2 \wedge \mathtt{turn} \neq 1]) \vee P)\}$

$6: \quad \mathtt{until}\ \neg\mathtt{fl}_1 \vee (\mathtt{tu}_1 = 1); \{\mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee P)\}$

$7: \quad \mathtt{STORE}(\mathtt{cs}, \bot); \{\mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee P)\}$

$8: \quad \mathtt{STORE}(\mathtt{cs}, 0); \{\mathtt{T}_1 \ltimes [\mathtt{cs} = 0] \wedge \mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee P)\}$

$9: \quad \mathtt{mx}_1 := \mathtt{LOAD}(\mathtt{cs}); \{\mathtt{mx}_1 = 0 \wedge \mathtt{a}_1 \wedge (\neg\mathtt{a}_2 \vee P)\}$

$10: \langle \mathtt{STORE}(\mathtt{flag}_1, 0); \mathtt{a}_1 := \mathit{false} \rangle$

$\{\mathtt{mx}_1 = 0\}$

Fig. 13. Peterson's algorithm, where $P = \mathtt{T}_1 \ltimes [\mathtt{R}(\mathtt{turn})] \mathbin{;} [\mathtt{flag}_2 \wedge \mathtt{turn} = 1]$. Thread $\mathtt{T}_2$ is symmetric and we assume a stopper thread $\mathtt{T}_3$ that sets `stop` to *true*.

use $\mathtt{do}\ C\ \mathtt{until}\ e$ as a shorthand for $C \mathbin{;} \mathtt{while}\ e\ \mathtt{do}\ C$. For correctness under SRA, all accesses to the shared variable `turn` are via a `SWAP`, which ensures that `turn` behaves like an SC variable.

Correctness is encoded via registers $\mathtt{mx}_1$ and $\mathtt{mx}_2$ into which the contents of shared variable `cs` is loaded. Mutual exclusion should guarantee both registers to be 0. Thus neither threads should ever be able to read `cs` to be $\bot$ (as stored in line 7). The proof (like the associated SC proof in [9]) introduces auxiliary variables $\mathtt{a}_1$ and $\mathtt{a}_2$. Variable $\mathtt{a}_i$ is initially *false*, set to *true* when a thread $\mathtt{T}_i$ has performed its swap, and back to *false* when $\mathtt{T}_i$ completes.

Once again potentials provide convenient mechanisms for reasoning about the interactions between the two threads. For example, the assertion $\mathtt{T}_1 \ltimes [\mathtt{R}(\mathtt{turn})] \mathbin{;} [\mathtt{flag}_2]$ in the precondition of line 2 encapsulates the idea that an RMW on `turn` (via $\mathtt{SWAP}(\mathtt{turn}, 2)$) must read from a state in which $\mathtt{flag}_2$ holds, allowing us to establish $\mathtt{T}_1 \ltimes [\mathtt{flag}_2]$ as a postcondition (using the axiom Swap-skip). We obtain disjunct $\mathtt{T}_1 \ltimes [\mathtt{flag}_2 \wedge \mathtt{turn} \neq 1]$ after additionally applying Wr-own.

8 Discussion, Related and Future Work

Previous RG-like logics provided ad-hoc solutions for other concrete memory models such as x86-TSO and C/C++11 [11,16,17,32,39,40,47]. These approaches established soundness of the proposed logic with an ad-hoc proof that couples together memory and thread transitions. We believe that these logics can be formulated in our proposed general RG framework (which will require extensions to other memory operations such as fences).

Moreover, Owicki-Gries logics for different fragments of the C11 memory model [16,17,47] used specialized assertions over the underlying view-based semantics. These include *conditional-view assertion* (enabling reasoning about

MP), and *value-order* (enabling reasoning about coherence). Both types of assertions are special cases of the potential-based assertions of Piccolo.

Ridge [40] presents an RG reasoning technique tailored to x86-TSO, treating the write buffers in TSO architectures as threads whose steps have to preserve relies. This is similar to our notion of stability of relies under internal memory transitions. Ridge moreover allows to have memory-model specific assertions (e.g., on the contents of write buffers).

The OGRA logic [32] for Release-Acquire (which is slightly weaker form of causal consistency compared to SRA studied in this paper) takes a different approach, which cannot be directly handled in our framework. It employs simple SC-like assertions at the price of having a non-standard non-interference condition which require a stronger form of stability.

Coughlin et al. [14,15] provide an RG reasoning technique for weak memory models with a semantics defined in terms of *reordering relations* (on instructions). They study both multicopy and non-multicopy atomic architectures, but in all models, the rely-guarantee assertions are interpreted over SC.

Schellhorn et al. [41] develop a framework that extends ITL with a compositional interleaving operator, enabling proof decomposition using RG rules. Each interval represents a sequence of states, strictly alternating between program and environment actions (which may be a skip action). This work is radically different from ours since (1) their states are interpreted using a standard SC semantics, and (2) their intervals represent an *entire execution* of a command as well the interference from the environment while executing that command.

Under SC, rely-guarantee was combined with separation logic [44,46], which allows the powerful synergy of reasoning using stable invariants (as in rely-guarantee) and ownership transfer (as in concurrent separation logic). It is interesting to study a combination of our RG framework with concurrent separation logics for weak memory models, such as [43,45].

Other works have studied the decidability of verification for causal consistency models. In work preceding the potential-based SRA model [28], Abdulla et al. [1] show that verification under RA is undecidable. In other work, Abdulla et al. [3] show that the reachability problem under TSO remains decidable for systems with dynamic thread creation. Investigating this question under SRA is an interesting topic for future work.

Finally, the spirit of our generic approach is similar to Iris [22], Views [18], Ogre and Pythia [7], the work of Ponce de León et al. [34], and recent axiomatic characterizations of weak memory reasoning [19], which all aim to provide a *generic* framework that can be instantiated to underlying semantics.

In the future we are interested in automating the reasoning in Piccolo, starting from automatically checking for validity of program derivations (using, e.g., SMT solvers for specialised theories of sequences or strings [24,42]), and, including, more ambitiously, synthesizing appropriate Piccolo invariants.

References

1. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.N.: Verification of programs under the release-acquire semantics. In: PLDI, pp. 1117–1132. ACM (2019). https://doi.org/10.1145/3314221.3314649
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Kumar, K.N., Saivasan, P.: Deciding reachability under persistent x86-TSO. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434337
3. Abdulla, P.A., Atig, M.F., Bouajjani, A., Narayan Kumar, K., Saivasan, P.: Verifying reachability for TSO programs with dynamic thread creation. In: Koulali, M.A., Mezini, M. (eds.) Networked Systems, NETYS 2022. LNCS, vol. 13464. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17436-0_19
4. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: The benefits of duality in verifying concurrent programs under TSO. In: CONCUR. LIPIcs, vol. 59, pp. 5:1–5:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.5
5. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: A load-buffer semantics for total store ordering. Log. Methods Comput. Sci. **14**(1) (2018). https://doi.org/10.23638/LMCS-14(1:9)2018
6. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distrib. Comput. **9**(1), 37–49 (1995). https://doi.org/10.1007/BF01784241
7. Alglave, J., Cousot, P.: Ogre and Pythia: an invariance proof method for weak consistency models. In: Castagna, G., Gordon, A.D. (eds.) POPL, pp. 3–18. ACM (2017). https://doi.org/10.1145/3009837.3009883
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014). https://doi.org/10.1145/2627752
9. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science. Springer, Heidelberg (2009). https://doi.org/10.1007/978-1-84882-745-5
10. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. Log. Meth. Comput. Sci. **17**(1) (2021). http://lmcs.episciences.org/7149
11. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based Owicki–Gries reasoning for persistent x86-TSO. In: ESOP 2022. LNCS, vol. 13240, pp. 234–261. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99336-8_9
12. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: POPL, pp. 626–638. ACM (2017). https://doi.org/10.1145/3009837.3009888
13. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991). https://doi.org/10.1016/0020-0190(91)90122-X
14. Coughlin, N., Winter, K., Smith, G.: Rely/guarantee reasoning for multicopy atomic weak memory models. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 292–310. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_16
15. Coughlin, N., Winter, K., Smith, G.: Compositional reasoning for non-multicopy atomic architectures. Form. Asp. Comput. (2022). https://doi.org/10.1145/3574137
16. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries reasoning for C11 RAR. In: ECOOP. LIPIcs, vol. 166, pp. 11:1–11:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.11

17. Dalvandi, S., Dongol, B., Doherty, S., Wehrheim, H.: Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. J. Autom. Reason. **66**(1), 141–171 (2022). https://doi.org/10.1007/s10817-021-09610-2
18. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL, pp. 287–300. ACM (2013). https://doi.org/10.1145/2429069.2429104
19. Doherty, S., Dalvandi, S., Dongol, B., Wehrheim, H.: Unifying operational weak memory verification: an axiomatic approach. ACM Trans. Comput. Log. **23**(4), 27:1–27:39 (2022). https://doi.org/10.1145/3545117
20. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: PPoPP, pp. 355–365. ACM (2019). https://doi.org/10.1145/3293883.3295702
21. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577
22. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
23. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: reasoning about release-acquire consistency in Iris. In: ECOOP. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.17
24. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: CertiStr: a certified string solver. In: CPP, pp. 210–224. ACM (2022). https://doi.org/10.1145/3497775.3503691
25. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL, pp. 175–189. ACM (2017). https://doi.org/10.1145/3009837.3009850
26. Lahav, O.: Verification under causally consistent shared memory. ACM SIGLOG News **6**(2), 43–56 (2019). https://doi.org/10.1145/3326938.3326942
27. Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: PLDI, pp. 211–226. ACM (2020). https://doi.org/10.1145/3385412.3385966
28. Lahav, O., Boker, U.: What's decidable about causally consistent shared memory? ACM Trans. Program. Lang. Syst. **44**(2), 8:1–8:55 (2022). https://doi.org/10.1145/3505273
29. Lahav, O., Dongol, B., Wehrheim, H.: Artifact: rely-guarantee reasoning for causally consistent shared memory. Zenodo (2023). https://doi.org/10.5281/zenodo.7929646
30. Lahav, O., Dongol, B., Wehrheim, H.: Rely-guarantee reasoning for causally consistent shared memory (extended version) (2023). https://doi.org/10.48550/arXiv.2305.08486
31. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL, pp. 649–662. ACM (2016). https://doi.org/10.1145/2837614.2837643
32. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
33. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439

34. de León, H.P., Furbach, F., Heljanko, K., Meyer, R.: BMC with memory models as modules. In: FMCAD, pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603021
35. Moszkowski, B.C.: A complete axiom system for propositional interval temporal logic with infinite time. Log. Meth. Comput. Sci. **8**(3) (2012). https://doi.org/10.2168/LMCS-8(3:10)2012
36. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
37. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134
38. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
39. Raad, A., Lahav, O., Vafeiadis, V.: Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA), 151:1–151:28 (2020). https://doi.org/10.1145/3428219
40. Ridge, T.: A rely-guarantee proof system for x86-TSO. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 55–70. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_4
41. Schellhorn, G., Tofan, B., Ernst, G., Pfähler, J., Reif, W.: RGITL: a temporal logic framework for compositional reasoning about interleaved programs. Ann. Math. Artif. Intell. **71**(1–3), 131–174 (2014). https://doi.org/10.1007/s10472-013-9389-z
42. Sheng, Y., et al.: Reasoning about vectors using an SMT theory of sequences. In: Blanchette, J., Kováics, L., Pattinson, D. (eds.) Automated Reasoning, IJCAR 2022. LNCS, vol. 13385, pp. pp. 125–143. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_9
43. Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., Vafeiadis, V.: A separation logic for a promising semantics. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 357–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_13
44. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge, UK (2008). https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221
45. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA, pp. 867–884. ACM (2013). https://doi.org/10.1145/2509136.2509532
46. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_18
47. Wright, D., Batty, M., Dongol, B.: Owicki-Gries reasoning for C11 programs with relaxed dependencies. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 237–254. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_13
48. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2), 149–174 (1997). https://doi.org/10.1007/BF01211617

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Unblocking Dynamic Partial Order Reduction

Michalis Kokologiannakis, Iason Marmanis(✉), and Viktor Vafeiadis

MPI-SWS, Kaiserslautern, Saarbrücken, Germany
{michalis,imarmanis,viktor}@mpi-sws.org

Abstract. Existing dynamic partial order reduction (DPOR) algorithms scale poorly on concurrent data structure benchmarks because they visit a huge number of blocked executions due to spinloops.

In response, we develop AWAMOCHE, a sound, complete, and strongly optimal DPOR algorithm that avoids exploring any useless blocked executions in programs with await and confirmation-CAS loops. Consequently, it outperforms the state-of-the-art, often by an exponential factor.

1 Introduction

Dynamic partial order reduction (DPOR) [13] has been promoted as an effective verification technique for concurrent programs: starting from a single execution of the program under test, DPOR repeatedly reverses the order of conflicting accesses in order to generate all (meaningfully) different program executions.

Applying DPOR in practice, however, reveals a major performance and scalability bottleneck: it explores a huge number of *blocked executions*, often outnumbering the complete program executions by an exponential factor. Blocked executions most commonly occur in programs with *spinloops*, i.e., loops that do not make progress unless some condition holds. Such loops are usually transformed into *assume statements* [14,18], effectively requiring that the loop exits at its first iteration (and blocking otherwise).

We distinguish three classes of such blocked executions.

The first class occurs in programs with non-terminating spinloops, such as a program awaiting for $x > 42$ in a context where $x = 0$. For this program, modeled as the statement $\texttt{assume}(x > 42)$, DPOR obviously explores a blocked execution as the only existing value for x violates the assume condition. Such blocked executions *should* be explored because they indicate program errors.

The second class occurs in programs with await loops. To see how such loops lead to blocked executions, consider the following program under *sequential consistency* (SC) [23] (initially $x = y = 0$),

$$\begin{array}{l} x := 2 \\ \texttt{assume}(y \leq 1) \end{array} \quad \Bigg\| \quad \begin{array}{l} y := 2 \\ \texttt{assume}(x \leq 1) \\ y := 1 \end{array}$$

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 230–250, 2023.
https://doi.org/10.1007/978-3-031-37706-8_12

where each `assume` models an await loop, e.g., `do` $a := y$ `while` $(a > 1)$ for the `assume` of the first thread. Suppose that DPOR executes this program in a left-to-right manner, thereby generating the interleaving $x := 2, \texttt{assume}(y \leq 1), y := 2$. At this point, $\texttt{assume}(x \leq 1)$ cannot be executed, since x would read 2. Yet, DPOR cannot simply abort the exploration. To generate the interleaving where the first thread reads $y = 1$, DPOR must consider the case where the read of x is executed before the $x := 2$ assignment. In other words, DPOR has to explore blocked executions in order to generate non-blocked ones.

The third class occurs in programs with confirmation-CAS loops such as:

```
do
  a := x
  b := f(a)
while (¬CAS(x, a, b))
```

which is modeled as:

```
a := x
b := f(a)
assume(CAS(x, a, b))
```

Consider a program comprising two threads running the code above, with a and b being local variables. Suppose that DPOR first obtains the (blocked) trace where both threads concurrently try to perform their CAS: $a_1 := x$, $a_2 := x$, $\texttt{CAS}(x, a_1, b_1)$, $\texttt{CAS}(x, a_2, b_2)$. Trying to satisfy the blocked assume of thread 2 by reversing the CAS instructions is fruitless because then thread 1 will be blocked.

In this paper, we show that exploring blocked executions of the second and third classes is unnecessary.

We develop Awamoche, a sound, complete, and optimal DPOR algorithm that avoids generating any blocked executions for programs with await and confirmation-CAS loops. Our algorithm is *strongly optimal* in that no exploration is wasted: it either yields a complete execution or a termination violation. Awamoche extends TruSt [15], an optimal DPOR algorithm that supports weak memory models and has polynomial space requirements, with three new ideas:

1. Awamoche identifies certain reads as *stale*, meaning that they will never be affected by a race reversal due to TruSt's maximality condition on reversals, and avoids exploring any executions that block on stale-read values.
2. To deal with await loops, since it cannot completely avoid generating executions with blocking reads, Awamoche *revisits* such executions *in place* if a same-location write is later encountered. If no such write is found, then the blocked execution witnesses a program termination bug [21,25].
3. To effectively deal with confirmation-CAS loops, Awamoche only considers executions where the confirmation succeeds, by reversing not only races between conflicting instructions, but also speculatively revisiting traces with two reads reading from the same write event to enable a later in-place revisit.

As we shall see in Sect. 5, supporting these DPOR modifications is by no means trivial when it comes to proving correctness and (strong) optimality. Indeed, TruSt's correctness proof proceeds in a backward manner, assuming a way to determine the last event that was added to a given trace. The presence of in-place and speculative revisits, however, makes this impossible.

We therefore develop a completely different proof that works in a forward manner: from each configuration that is a prefix of a complete trace, we construct a sequence of steps that will lead to a larger configuration that is also a prefix of the trace. Our proof assumes that same-location writes are causally ordered, which invariably holds in correct data structure benchmarks, but is otherwise more general than TruSt's assuming less about the underlying memory model.

Our contributions can be summarized as follows:

Section 2 We describe how and why DPOR encounters blocked executions.
Section 3 We intuitively present AWAMOCHE's three novel key ideas: stale reads, in-place revisits, and speculative revisits.
Section 4 We describe our algorithm in detail in a memory-model-agnostic framework.
Section 5 We generalize TruSt's proof and prove AWAMOCHE sound, complete, and strongly optimal.
Section 6 We evaluate AWAMOCHE, and demonstrate that it outperforms the state-of-the-art, often by an exponential factor.

2 DPOR and Blocked Executions

Before presenting AWAMOCHE, we recall the fundamentals of DPOR (Sect. 2.1), and explain why spinloops lead to blocked explorations (Sect. 2.2).

2.1 Dynamic Partial Order Reduction

DPOR algorithms verify a concurrent program by enumerating a representative subset of its interleavings. Specifically, they partition the interleavings into equivalence classes (two interleavings are equivalent if one can be obtained from the other by reordering independent instructions), and strive to explore one interleaving per equivalence class. Optimal algorithms [2,15] achieve this goal.

DPOR algorithms explore interleavings dynamically. After running the program and obtaining an initial interleaving, they detect *racy* instructions (i.e., instructions accessing the same variable with at least one of them being a write), and proceed to explore an interleaving where the race is reversed.

Let us clarify the exploration procedure with the following example, where both variables x and y are initialized to zero.

$$\begin{array}{c||c} \texttt{if}\ (x = 0) & x := 1 \\ y := 1 & x := 2 \end{array} \qquad \text{(RW+WW)}$$

The RW+WW program has 5 interleavings that can be partitioned into 3 equivalence classes. Intuitively, the $y := 1$ is irrelevant because the program contains no other access to y; all that matters is the ordering among the x accesses.

The exploration steps for RW+WW can be seen in Fig. 1[1]. DPOR obtains a full trace of the program, while also recording the transitions that it took at each

[1] The exploration procedure has been simplified for presentational purposes. For a full treatment, please refer to [2,15].

init ⓪ ⇝ ⋯ ⇝

init ①
↓ $\{r_x\}$
(r_x) `if` (...)
↓ $\{w_y\}$
(w_y) $y := 1$

⇝ ⋯ ⇝

init ②
↓ $\{r_x, w_1\}$
(r_x) `if` (...)
↓ $\{w_y\}$
(w_y) $y := 1$
↓ $\{w_1\}$
(w_1) $x := 1$
↓ $\{w_2\}$
(w_2) $x := 2$

$\overset{bt}{⇝}$

init ③
↓ $\{w_1, r_x\}$
(w_1) $x := 1$

⇝ ⋯ ⇝

init ④
↓ $\{w_1, r_x\}$
(w_1) $x := 1$
↓ $\{r_x, w_2\}$
(r_x) `if` (...)
↓ $\{w_2\}$
(w_2) $x := 2$

$\overset{bt}{⇝}$

init ⑤
↓ $\{r_x, w_1\}$
(w_1) $x := 1$
↓ $\{r_x, w_2\}$
(w_2) $x := 2$

⇝

init ⑥
↓ $\{r_x, w_1\}$
(w_1) $x := 1$
↓ $\{r_x, w_2\}$
(w_2) $x := 2$
↓ $\{r_x\}$
(r_x) `if` (...)

Fig. 1. A DPOR exploration of RW+WW

step at the respective transition's backtrack set (traces ⓪ to ②). After obtaining a full trace, it initiates a race-detection phase. During this phase, DPOR detects the races between r_x and the two writes w_1 and w_2. (While w_1 and w_2 also write the same variable, they do not constitute a race, as they are causally related.) For the first race, DPOR adds w_1 in the backtrack set of the first transition, so that it can subsequently execute w_1 instead of r_x. For the second one, while w_2 is not in the backtrack set of the first transition, w_2 cannot be directly executed as the first transition without its causal predecessors (i.e., w_1) having already executed. Since w_1 is already in the backtrack set of the first transition, DPOR cannot do anything else, and the race-detection phase is over.

After the race-detection phase is complete, the exploration proceeds in an analogous manner: DPOR backtracks to the first transition, fires w_1 instead of r_x (trace ③), re-runs the program to obtain a full trace (trace ④), and initiates another race-detection phase. During the latter, a race between r_x and w_2 is detected, and w_2 is inserted in the backtrack set of the second transition.

Finally, DPOR backtracks to the second transition, executes w_2 instead of r_x (trace ⑤), and eventually obtains the full trace ⑥. During the last race-detection phase of the exploration, DPOR detects the races between r_x and the two writes w_1 and w_2. As r_x is already in the backtrack set of the first two transitions, DPOR has nothing else to do, and thus concludes the exploration.

Observe that DPOR explored one representative trace from each equivalence class (traces ②, ④, and ⑥). To avoid generating multiple equivalent interleavings, optimal DPOR algorithms extend the description above by restricting when a race reversal is considered. In particular, the TruSt algorithm [15] imposes a maximality condition on the part of the trace that is affected by the reversal.

2.2 Assume Statements and DPOR

```
while (x=0) {}  ||  x := 1          (RW+WW-L)
y := 1          ||  x := 2

assume(x≠0)     ||  x := 1          (RW+WW-A)
y := 1          ||  x := 2
```

Fig. 2. A variation of RW+WW with an await loop (left) and an `assume` (right)

To see how `assume` statements arise in concurrent programs, suppose that we replace the `if`-statement of RW+WW with an await loop (Fig. 2). Although the change does not really affect the possible outcomes for x, it makes DPOR diverge: DPOR examines executions where the loop terminates in 1, 2, 3, ... steps. Since, however, the loop has no side-effects, we can actually transform it into an `assume`(x) statement, effectively modeling a loop bound of one.

Doing so guarantees DPOR's termination but not its good performance. The reason is ascribed to the very nature of DPOR. Indeed, suppose that DPOR executes the first instruction of the left thread and then blocks due to `assume` statement. At this point, DPOR cannot simply stop the exploration due to the `assume` statement not being satisfied; it has to explore the rest of the program, so that the race reversals make the `assume` succeed. All in all, DPOR explores 2 complete and 1 blocked traces for RW+WW-A.

In general, DPOR cannot know whether some future reversal will ever make an `assume` succeed. Worse yet, it might be the case that there is an exponential number of traces to be explored (due to the other program threads), until DPOR is certain that the `assume` statement cannot be unblocked.

To see this, consider the following program where RW+WW-A runs in parallel with some threads accessing z:

$$RW + WW - A \;\|\; z := 1 \;\|\; a_1 := z \;\|\; ... \;\|\; a_N := z \qquad \text{(RW+WW-A-PAR)}$$

For the trace of RW+WW-A where the `assume` fails, DPOR fruitlessly explores 2^N traces in the hope that an access to x is found that will unblock the `assume` statement.

Given that executing an `assume` statement that fails leads to blocked executions, one might be tempted to consider a solution where `assume` statements are only scheduled if they succeed. Even though such a solution would eliminate blocking for RW+WW-A, it is not a panacea. To see why, consider a variation of RW+WW-A where the first thread executes `assume`$(x = 0)$ instead of `assume`$(x \neq 0)$. In such a case, the `assume` can be scheduled first (as it succeeds), but reversing the races among the x accesses will lead to blocked executions. It becomes evident that a more sophisticated solution is required.

3 Key Ideas

Awamoche, our optimal DPOR algorithm, extends TruSt [15] with three novel key ideas: stale-read annotations (Sect. 3.1), in-place revisits (Sect. 3.2) and speculative revisits (Sect. 3.3). As we will shortly see, these ideas guarantee that

AWAMOCHE is *strongly optimal*: it never initiates fruitless explorations, and all explorations lead to executions that are either complete or denote termination violations. In the rest of the paper, we call such executions *useful*.

3.1 Avoiding Blocking Due to Stale Reads

Race reversals are at the heart of any DPOR algorithm. TruSt distinguishes two categories of race reversals: (1) write-read and write-write reversals, (2) read-write reversals. While the former category can be performed by modifying the trace directly in place (called a "forward revisit"), the latter may require removing events from the trace (called a "backward revisit"). To ensure optimality for backward revisits, TruSt checks a certain maximality condition for the events affected by them, namely the read, which will be reading from a different write, and all events to be deleted.

An immediate consequence is that any read events not satisfying TruSt's maximality condition, which we call *stale reads*, will never be affected by a subsequent revisit. As an example, consider the following program with a read that blocks if it reads 0:

$$x := 1 \;\Big\|\; \texttt{assume}(x = 1) \qquad \text{(W+R)}$$

After obtaining the trace $x := 1; \texttt{assume}(x = 1)$, TruSt forward-revisits the read in-place, and makes it read 0. At this point, we know that (1) the assume will fail, and (2) that both the read and the events added before it cannot be backward-revisited, due to the read reading non-maximally (which violates TruSt's maximality condition). As such, no useful execution is ever going to be reached, and there is no point in continuing the exploration.

Leveraging the above insight, we make AWAMOCHE immediately drop traces where some assume is not satisfied due to a stale read. To do this, AWAMOCHE automatically annotates reads followed by assume statements with the condition required to satisfy the assume, and discards all forward revisits that do not satisfy the annotation.

Even though stale-read annotations are greatly beneficial in reducing blocking, they are merely a remedy, not a cure. As already mentioned, they are only leveraged in write-read reversals, and are thus sensitive to DPOR's exploration order. To completely eliminate blocking, AWAMOCHE performs in-place and speculative revisits, described in the next sections.

3.2 Handling Await Loops with In-Place Revisits

AWAMOCHE's solution to eliminate blocking is to not blindly reverse all races whenever a trace is blocked, but rather to only try and reverse those that might unblock the exploration.

As an example, consider Rw+ww-A-PAR (Fig. 3). After AWAMOCHE obtains the first full trace, it detects the races among the z accesses, as well as the $\langle r_x, w_1 \rangle$

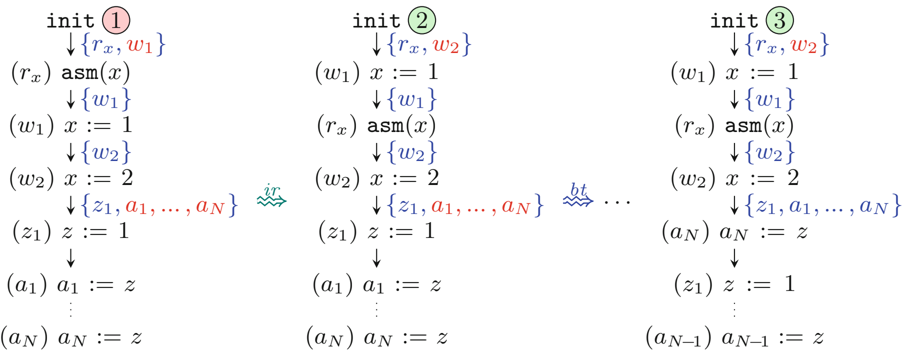


Fig. 3. Key steps in AWAMOCHE's exploration of RW+WW-A-PAR

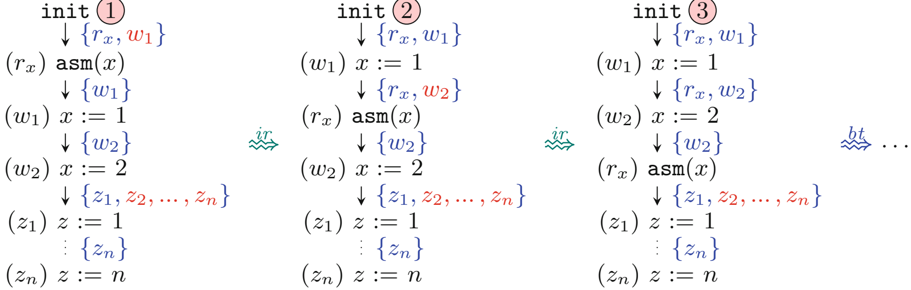


Fig. 4. An AWAMOCHE exploration of RW+WW

race. (Recall that AWAMOCHE is based on TruSt and therefore does not consider the $\langle r_x, w_2\rangle$ race in this trace.) At this point, a standard DPOR would start reversing the races among the z accesses. Doing so, however, is wasteful, since reversing races after the blockage will lead to the exploration of more blocked executions.

Instead, AWAMOCHE chooses to reverse the $\langle r_x, w_1\rangle$ race (as this might make the `assume` succeed), and completely drops the races among the z accesses. We call this procedure *in-place revisiting* (denoted by $\overset{ir}{\rightsquigarrow}$ in Fig. 3). Intuitively, ignoring the z races is safe to do as they will have the chance to manifest in the trace where the $\langle r_x, w_1\rangle$ race has been reversed.

Indeed, reversing the $\langle r_x, w_1\rangle$ does make the `assume` succeed, at which point the exploration proceeds in the standard DPOR way. AWAMOCHE explores 2^N traces where the read of x reads 1, and another 2^N where it reads 2. Note that, even though in this example AWAMOCHE explores 2/3 of the traces that standard DPOR explores, as we show in Sect. 6 the difference can be exponential.

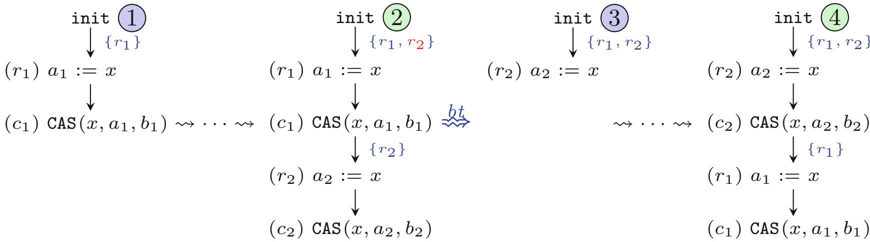


Fig. 5. An AWAMOCHE exploration of the confirmation-CAS example.

Suppose now that we change the `assume`(x) in Rw+ww-A-PAR to `assume`(x = 42) so that there is no trace where the assume is satisfied. The key steps of AWAMOCHE's exploration can be seen in Fig. 4. Upon obtaining a full trace, all races to z are ignored and AWAMOCHE revisits r_x in place. Subsequently, as the assume is still not satisfied, AWAMOCHE again revisits r_x in place (trace (2)). At this point, since there are no other races on x it can reverse, AWAMOCHE reverses all the races on z, and finishes the exploration.

In total, AWAMOCHE explores 2^N blocked executions for the updated example, which are all useful. As r_x is reading from the latest write to x in all these executions and the assume statement (corresponding to an await loop) still blocks, each of these executions constitutes a distinct liveness violation.

3.3 Handling Confirmation CASes with Speculative Revisits

In-place revisiting alone suffices to eliminate useless blocking in programs whose assume statements arise only due to await loops. It does not, however, eliminate blocking in *confirmation-CAS loops*. Confirmation-CAS loops consist of a speculative read of some shared variable, followed by a (possibly empty) sequence of local accesses and other reads, and a confirmation CAS that only succeeds if it reads from the same write as the speculative read.

As an example, consider the confirmation-CAS example from Sect. 1 and a trace where both reads read the initial value, the CAS of the first thread succeeds, and the CAS of the second thread reads the result of the CAS of the first. Although this trace is blocked and explored by DPOR (since the CAS read of the second thread is reading from the latest, same-location write), it does not constitute an actual liveness violation. In fact, even though the CAS read that blocks does read from the latest, same-location write, the $r := x$ read in the same loop iteration does not. In order for a blocked trace (involving a loop) to be an actual liveness violation, all reads corresponding to a given iteration need to be reading the latest value, and not just one.

To avoid exploring blocked traces altogether for cases likes this, we equip AWAMOCHE with some builtin knowledge about confirmation-CAS loops and treat them specially when reversing races. To see how this is done, we present a run of AWAMOCHE on the confirmation-CAS example of Sect. 1 (see Fig. 5).

While building the first full trace (trace (1)), another big difference between Awamoche and standard DPOR algorithms is visible: Awamoche does not maintain backtrack sets for confirmation CASes. Indeed, there is no point in reversing a race involving a confirmation CAS, as such a reversal will make the CAS read from a different write than the speculative read, and hence lead to an `assume` failure.

After obtaining the first full trace (trace (2)), Awamoche initiates a race-detection phase. At this point, the final big difference between Awamoche and previous DPORs is revealed. Awamoche will not reverse races between reads and CASes, but rather between *speculative reads*. (While speculative reads are not technically conflicting events, they conflict with the later confirmation-CASes.) As can be seen in trace (3), Awamoche schedules the speculative read of the second thread before that of the first thread so that it explores the scenario where the confirmation of the second thread succeeds before the one of the first.

Finally, simply by adding the remaining events of the second thread before the ones of the first thread, Awamoche explores the second and final trace of the example (trace (4)), while avoiding having blocked traces altogether.

4 Await-Aware Model Checking Algorithm

Awamoche is based on TruSt [15], a state-of-the-art stateless model checking algorithm that explores *execution graphs* [9], and thus seamlessly supports weak memory models. In what follows, we formally define execution graphs (Sect. 4.1), and then present Awamoche (Sect. 4.2).

4.1 Execution Graphs

An execution graph G consists of a set of events (nodes), representing instructions of the program, and a few relations of these events (edges), representing interactions among the instructions.

Definition 1. *An* event, $e \in \mathsf{Event}$, *is either the initialization event* `init`, *or a thread event* $\langle t, i, lab \rangle$ *where* $t \in \mathsf{Tid}$ *is a thread identifier,* $i \in \mathsf{Idx} \triangleq \mathbb{N}$ *is a serial number inside each thread, and* $lab \in \mathsf{Lab}$ *is a label that takes one of the following forms:*

- *Block label:* `B` *representing the blockage of a thread (e.g., due to the condition of an "*`assume`*" statement failing).*
- *Error label:* `error` *representing the violation of some program assertion.*
- *Write label:* $\mathtt{W}^{k_w}(l, v)$ *where* $k_w \subseteq \mathsf{Wattr} \triangleq \{\mathsf{excl}\}$ *denotes special attributes the write may have (i.e., exclusive),* $l \in \mathsf{Loc}$ *is the location accessed, and* $v \in \mathsf{Val}$ *the value written.*
- *Read label:* $\mathtt{R}^{k_r}(l)$ *where* $k_r \subseteq \mathsf{Rattr} \triangleq \{\mathsf{awt}, \mathsf{spec}, \mathsf{excl}\}$ *denotes special attributes the read may have (i.e., await, speculative, exclusive), and* $l \in \mathsf{Loc}$ *is the location accessed. We note that if a read has the* awt *or the* spec *attribute, then it cannot have any other attribute.*

We omit the $\emptyset$ *for read/write labels with no attributes. The functions* $\mathtt{tid}$, $\mathtt{idx}$, $\mathtt{loc}$, *and* $\mathtt{val}$, *respectively return the thread identifier, serial number, location, and value of an event, when applicable. We use* $\mathtt{R}$, $\mathtt{W}$, $\mathtt{B}$, *and* $\mathtt{error}$ *to denote the set of all read, write, block, and error events, respectively, and assume that* $\mathtt{init} \in \mathtt{W}$. *We use superscript and subscripts to further restrict those sets (e.g.,* $\mathtt{W}_l \triangleq \{\mathtt{init}\} \cup \{w \in \mathtt{W} \mid \mathtt{loc}(w) = l\}$*).*

In the definition above, read and write events come with various attributes. Specifically, we encode successful CAS operations and other similar atomic operations, such as fetch-and-add, as two events: an exclusive read followed by an exclusive write (both denoted by the excl attribute). Moreover, we have a spec attribute for speculative reads, and write $\mathtt{R}^{\mathsf{conf}}$ for the corresponding confirmation reads (i.e., the first exclusive, same-location read that is $\mathtt{po}$-after a given $r \in \mathtt{R}^{\mathsf{spec}}$). Finally, we have the $\mathtt{awt}$ attribute for reads the outcome of which is tied with an $\mathtt{assume}$ statement, and write $\mathtt{R}^{\mathsf{blk}}$ for the subset of $\mathtt{R}^{\mathsf{awt}}$ that are reading a value that makes the assume fail (see below).

Definition 2. *An* execution graph G *consists of:*

1. *a set* $G.\mathtt{E}$ *of events that includes* $\mathtt{init}$ *and does not contain multiple events with the same thread identifier and serial number.*
2. *a total order* $\leq_G$ *on* $G.\mathtt{E}$, *representing the order in which events were incrementally added to the graph,*
3. *a function* $G.\mathtt{rf} : G.\mathtt{R} \to G.\mathtt{W}$, *called the* reads-from *function, that maps each read event to a same-location write from where it gets its value, and*
4. *a strict partial order* $G.\mathtt{co} \subseteq \bigcup_{l \in \mathsf{Loc}} G.\mathtt{W}_l \times G.\mathtt{W}_l$, *called the* coherence order, *which is total on* $G.\mathtt{W}_l$ *for every location* $l \in \mathsf{Loc}$.

We write $G.\mathtt{R}$ *for the set* $G.\mathtt{E} \cap \mathtt{R}$ *and similarly for other sets. Given two events* $e_1, e_2 \in G.\mathtt{E}$, *we write* $e_1 <_G e_2$ *if* $e_1 \leq_G e_2$ *and* $e_1 \neq e_2$. *We write* $G|_E$ *for the restriction of an execution graph* G *to a set of events* E, *and* $G \setminus E$ *for the graph obtained by removing a set of events* E.

Based on the above graph representation, we define $G.\mathtt{po}$, which orders events in the same thread according to their i component, and $\mathtt{porf}$, which is the causal order among the graph events, as follows:

$$G.\mathtt{po} \triangleq \{\langle \mathtt{init}, e\rangle \mid e \in G.\mathtt{E} \setminus \{\mathtt{init}\}\} \cup \{\langle e, e'\rangle \in G.\mathtt{E} \times G.\mathtt{E} \mid \mathtt{tid}(e) = \mathtt{tid}(e') \wedge \mathtt{idx}(e) < \mathtt{idx}(e')\}$$

$$G.\mathtt{porf} \triangleq (G.\mathtt{po} \cup G.\mathtt{rf})^+$$

The semantics of a program P under a memory model m is the set of execution graphs corresponding to the program that satisfy the consistency predicate of m. Consistency predicates generally constrain the possible choices of $\mathtt{co}$ and $\mathtt{rf}$, thereby indirectly constraining the possible final values of memory locations and the values that reads can return.

TruSt (and by extension, Awamoche), assumes some properties on the memory model [15]: $\mathtt{porf}$ acyclicity, $\mathtt{porf}$-prefix-closedness, $\mathtt{co}$-maximal-extensibility. Intuitively, extensibility captures the idea that executing a program should never get stuck if a thread has more statements to execute.

Algorithm 1. AWAMOCHE's exploration algorithm

```
1: procedure VERIFY(P)
2:     VISIT_P(G_∅)

3: procedure VISIT_P(G)
4:     if ¬consistent_m(G) ∨ ∃b ∈ G.R^blk. ¬maximal(G, b) then return
5:     switch a ← next_P(G) do
6:         G ← G ++ a
7:         case a = ⊥
8:             return "Visited full execution graph G"
9:         case a ∈ error
10:            exit("error")
11:
12:
13:
14:        case a ∈ R \R^conf
15:            for w ∈ G.W_loc(a) do
16:
17:
18:
19:                VISIT_P(SetRF(G, a, w))
20:        case a ∈ W
21:            if WWRace(G) then exit("Write-write race")
22:            VISIT_P( IPR (G, a))
23:            Revs ← G.R_loc(a) \ dom(G.porf; [a])
24:            MAYBEBACKWARDREVISIT_P(G, Revs, a)
25:        case _
26:            VISIT_P(G)
```

4.2 Awamoche

Similarly to TruSt, AWAMOCHE verifies a concurrent program P by enumerating all of its consistent execution graphs (see Algorithm 1). In contrast to TruSt, however, AWAMOCHE is *strongly optimal*: it never explores an execution G where there exists some blocked read $r \in G.\mathtt{R}^{\mathsf{blk}}$ that is reading from a non-co-maximal write. In other words, AWAMOCHE only visits graphs that lead to useful executions[2]. In order to be able to do so, AWAMOCHE makes stronger assumptions on the underlying memory model m, namely that there are no write-write races, and that m does not allow porf to contradict co (i.e., that $\mathtt{co} \subseteq \mathtt{porf}$).

Next, we first describe how TruSt works, and then proceed with AWAMOCHE's modifications .

Given a program P, VERIFY visits all consistent execution graphs of P by calling VISIT on the execution graph $G_\emptyset$ containing only the initialization event.

[2] Recall that blocked reads that read from maximal writes *are* useful, as they denote liveness violations.

At each step (Line 4), as long as the current graph remains consistent under the specified memory model m, VISIT obtains a new event a via $\mathsf{next}_P(G)$ (Line 5), and extends the current graph G with a (Line 6). We assume that $G{+\!\!+}a$ adds a to $G.\mathtt{E}$, and also to $G.\mathtt{co}$, in case a is a write. (Recall that $\mathtt{co} \subseteq \mathtt{porf}$ and so a's $\mathtt{co}$-placing is unique.)

If there are no more events to add to the graph, then G is complete, and VISIT returns (Line 7). If a denotes an error, then it is reported to the user and verification terminates (Line 9).

If a is a read, VISIT needs to examine all possible places where a could read from. To that end, for each same-location write w in G (Line 15), VISIT recursively explores the possibility that a reads from w (Line 19). Formally, $\mathsf{SetRF}(G, r, w)$ returns a graph G' that is identical to G except for its $\mathtt{rf}$ component:

$$G'.\mathtt{rf} = G.\mathtt{rf} \setminus (G.\mathtt{E} \times \{r\}) \cup \{\langle w, r\rangle\}$$

If a is a write, VISIT examines both the case when a is simply added to G (Line 22) and the "*backward-revisit*" cases for each existing same-location read in G that could read from a (Line 5). When a backward-revisits a read r, the resulting graph G' only contains the events that were added before r, or are $\mathtt{porf}$-before a, and updates r to read from a. Since, however, there might be many backward revisits that lead to the exact same graph G', to ensure optimality, G' is visited only when the current graph G forms a *maximal extension* of G'. We do not provide TruSt's definition of maximal extensions here, as AWAMOCHE modifies it to achieve strong optimality.

Let us now move to the parts of Algorithm 1 that are AWAMOCHE-specific.

First, AWAMOCHE discards all graphs where some blocked read is reading non-maximally (Line 4). As explained in Sect. 3.2, such reads cannot be revisited and will thus only lead to blocked executions. In addition, to guarantee correctness, AWAMOCHE raises an error if it detects unordered writes (Line 21).

Second, whenever a write event a is added, AWAMOCHE revisits all same-location blocked reads *in place* making them read from a (Line 22) and excluding them from the normal backward-revisit procedure (Line refvisitsp-sipr:visitspsrevs). Formally, we define $\mathsf{IPR}(G, a)$ to return a graph G' that is identical to G apart from its $\mathtt{rf}$ component:

$$G'.\mathtt{rf} = G.\mathtt{rf} \setminus (G.\mathtt{E} \times G.\mathtt{R}^{\mathsf{blk}}_{\mathtt{loc}(a)}) \cup (\{a\} \times G.\mathtt{R}^{\mathsf{blk}}_{\mathtt{loc}(a)})$$

Third, whenever a confirmation read a is added (Line 11), i.e., an exclusive read that succeeds an unmatched speculative read e, AWAMOCHE only explores the execution where a reads from the same write as e (Line 13): any other write would make the confirmation CAS fail.

Fourth, whenever a speculative read a is added to read from a candidate write w and there is another speculative read b reading from the same write w (Line 16), AWAMOCHE backward-revisits b to read from a. Note that, due to the atomicity of the confirming CASes, there can be at most one other speculative read b reading from w, and so AWAMOCHE revisits it to read from a, making it blocked, so that it get revisited in place when the confirming CAS of a is added to the graph. (To ensure graph well-formedness, we assume that $\mathsf{IPR}(G, b)$ does

Algorithm 2. AWAMOCHE's backward-revisit algorithm

1: **procedure** $\textsc{MaybeBackwardRevisit}_P(G, Revs, a)$
2: **for** $r \in Revs$ **do**
3: $[d_1, \dots, d_n] \leftarrow \mathsf{sort}_{G_<}(\{e \in G.\mathtt{E} \mid r < e \wedge \langle e, a\rangle \notin G.\mathtt{porf}\})$
4: **if** $\exists G', G''$ such that $G' \overset{r}{\rightsquigarrow} G'' \overset{d_1}{\rightsquigarrow} \cdots \overset{d_n}{\rightsquigarrow} G|_{G.\mathtt{E}\setminus\{a\}}$ and $r \notin G''.\mathtt{R}^{\mathsf{blk}}$ **then**
5: $\textsc{Visit}_P(\mathsf{IPR}(\mathsf{SetRF}(G' +\!\!+ [r, a], r, a), a))$

not modify G when called with a read argument b, and that $\mathsf{SetRF}(G, b, _)$ makes b read from $\bot$, which IPR also considers.)

Finally, similarly to TruSt, AWAMOCHE only performs a backward revisit if G forms a maximal extension, though AWAMOCHE employs a slightly different definition of maximal extensions. AWAMOCHE's backward-revisit algorithm can be seen in Algorithm 2.

Roughly, AWAMOCHE performs a backward revisit from a to r that leads to a graph $\mathsf{IPR}(G_r, a)$ if, starting from G_r without r and a, and adding r and all the deleted events in a $\mathtt{co}$-maximal way (and performing in-place revisits along the way), leads to G. Formally, we write $G_1 \overset{e}{\rightsquigarrow} G_2$ if there exists G_1' such that $G_2 = \mathsf{IPR}(G_1', e)$, $G_1' = G_1 +\!\!+ e$ and:

$$
\begin{array}{lll}
G_1'.\mathtt{rf} = G_1.\mathtt{rf} \cup \{\langle \mathsf{max}_{G.\mathtt{co}_e}, e\rangle\} & G_1'.\mathtt{co} = G_1.\mathtt{co} & \text{if } e \in \mathtt{R} \\
G_1'.\mathtt{rf} = G_1.\mathtt{rf} & G_1'.\mathtt{co} = G_1.\mathtt{co} \cup \{\langle w, e\rangle \mid w \in G.\mathtt{W}\} & \text{if } e \in \mathtt{W} \\
G_1'.\mathtt{rf} = G_1.\mathtt{rf} & G_1'.\mathtt{co} = G_1.\mathtt{co} & \text{otherwise}
\end{array}
$$

We note that, for the special case where $e \in \mathtt{R}^{\mathsf{spec}}$ and there is $e' \in G.\mathtt{R}^{\mathsf{spec}}_{\mathtt{loc}(e)}$ such that e' is not followed by the matching confirmation CAS, we consider $\bot$ as the $\max_{G.\mathtt{co}_e}$. As a final remark, note that, AWAMOCHE modifies $\mathsf{next}_P(G)$ so that (a) after scheduling a speculative read, it keeps scheduling events in the same threads until the respective confirming CAS is added, and (b) it does not schedule events from a thread whose last (speculative) read reads $\bot$. These modifications ensure that the confirmation patterns are added one at a time, and that in-place revisits take place among confirming CASes and speculative reads.

5 Correctness and Optimality

Proving AWAMOCHE correct is non-trivial, as we had to develop a novel proof strategy. In what follows, we first review TruSt's proof argument, show why it is inapplicable for AWAMOCHE. Then, we explain our proof strategy (Sect. 5.1) and state our completeness and optimality results (Sect. 5.2).

5.1 Approaches to Correctness

TruSt. The proof of TruSt proceeds in a backward manner. Specifically, TruSt's proof is based on a procedure PREV that, given an execution G, recovers the

$\texttt{assume}(x \neq 0)$ $y := 1$ $\parallel$ $a := y$ $\parallel$ $x := 1$

Fig. 6. TruSt: In-place revisits make it impossible to determine the last step taken

unique "previous" execution G_p that the algorithm must reach in order to visit G. To do so, assuming a left-to-right addition order of events, PREV(G) finds the rightmost porf-maximal event e of G, and decides whether e was added in a non-revisit step, or e is a read that was just revisited by a write event located to its right. If e was added in a non-revisit step, then G_p is simply G without e. Otherwise, PREV obtains G_p from G in the following way: it removes e along with the write w that e reads from, and then iteratively adds the leftmost available event to G in a co-maximal way, until w is about to be added.

TruSt's completeness and optimality are proved using PREV. For the former, one can show that each consistent final execution can reach the initial empty execution through a series of PREV steps, and each of these steps is matched by a forward step of TruSt. For the latter, one can show that each step of TruSt is matched by the (unique) PREV step.

To see why we cannot follow a similar approach for AWAMOCHE, consider the program of Fig. 6, along with one of its executions. We will show that in-place revisits make it impossible to trace the algorithm's last step merely by inspecting the execution. Assuming a left-to-right addition order, AWAMOCHE will reach this execution as follows: it first adds $\mathtt{R}(x)$, $\mathtt{R}(y)$ and $\mathtt{W}(x, 1)$ (notice that at this point the first read is blocked), then in-place revisit $\mathtt{R}(x)$, and finally add $\mathtt{W}(y, 1)$ and backward-revisit $\mathtt{R}(y)$. This last revisit, however, creates a problem: TruSt's proof assumes that a backward revisit $\langle r, w\rangle$ implies that w is located at the right of r, which is clearly not the case here. The fact that in AWAMOCHE backward revisits can happen in both directions, makes it impossible to trace the algorithm's last step simply by inspecting an execution.

Awamoche. In contrast to TruSt, AWAMOCHE's proof proceeds in a forward fashion. For each consistent final execution G_f we show 1. which steps are taken by the algorithm in order to reach G_f, and 2. that these are the only possible ones that lead to G_f. To do so, we first define a notion of a *prefix*: we say that an execution G is a prefix of G' (written $G \sqsubseteq G'$), if G' can be reached from G with a series of *operational steps*. In turn, we define an operational step to be a step that the algorithm may take in the non-revisit case (without demanding it is the one actually taken by the algorithm), that may perform in-place revisits as well.

Using this notion of prefixes, our proof defines a procedure SUCCS that, given a consistent execution G_f and an execution G produced by the algorithm such that $G \sqsubseteq G_f$, SUCCS returns the minimal sequence of algorithm steps that reach some execution G' for which it is $G \sqsubseteq G' \sqsubseteq G_f$. Concretely, if $\mathsf{next}_P(G)$ can

be added to G such that the resulting execution G' is a prefix of G_f, SUCCS returns this addition step. Otherwise, $\mathsf{next}_P(G)$ is a read event r that must be first revisited by an event e in order to reach an execution that is a prefix of G_f. SUCCS then returns the sequence of algorithm steps that reach the execution resulting from extending G with the porf-prefix of e and setting r to read from e (or from $\bot$, if e is a speculative read). Both completeness and optimality follow from SUCCS's properties, as well as from the observation that every consistent final execution can be reached by a series of operational steps.

5.2 Awamoche: Completeness, Optimality, and Strong Optimality

Before stating our results, we first formally define useful executions. Recall that these are executions where all blocking reads corresponding to await loops are reading maximally (such executions denote liveness violations), and no confirmation CAS fails.

Definition 3. *A consistent execution G is* useful *if every read in $G.\mathtt{R}^{\mathsf{blk}}$ reads from a $G.\mathtt{co}$-maximal write and no confirmation CAS fails.*

Next, we define the class of input programs that satisfy our assumptions.

Definition 4. *A program P is* well-formed *if every speculative read is followed by a confirmation CAS with no write in-between, and all writes to locations accessed by speculative reads write distinct values.*

Completeness and Optimality. Completeness guarantees that every useful final execution is explored. AWAMOCHE is complete for well-formed programs that do not exhibit write-write races.

Theorem 1 (Completeness). *Given a well-formed program P,* VERIFY(P) *either detects a write-write race and exits, or visits every useful final execution of P.*

Optimality states that (1) no equivalent final executions are explored, (2) there are no *fruitless* explorations that never lead to a consistent final execution.

Definition 5. *We call an execution G visited by* AWAMOCHE fruitless *if it does not recursively lead to any* VISIT(P, G_f) *call, for any consistent final execution G_f.*

AWAMOCHE is optimal for well-formed programs.

Theorem 2 (Optimality). *Given a well-formed program P (1)* VERIFY(P) *never visits two equivalent final executions, and (2) if* VISIT(P, G) *directly leads to a call to* VISIT(P, G') *with G being fruitless, then* VISIT(P, G') *will not initiate any other* VISIT *calls.*

Observe that in the optimality theorem above, fruitless exploration can lead to an extra VISIT step. The reason for that is the treatment of CASes: the read part of a CAS c can be added so that it reads from the same write as a different (successful) CAS. In such a case, there is no way to consistently add the pending write of c without revisiting, which in turn may not be able to happen due to AWAMOCHE's maximality condition.

Strong Optimality. Strong optimality states that, apart from being optimal, only useful executions are visited. AWAMOCHE is strongly-optimal for well-formed programs.

Theorem 3 (Strong Optimality). *Given a well-formed program P,* VERIFY(P, G) *only visits useful executions.*

6 Evaluation

We implemented AWAMOCHE as a tool that verifies C/C++ programs under the RC11 memory model [22]. Similarly to other stateless model checkers, AWAMOCHE works at the level of the LLVM Intermediate Representation (LLVM-IR).

In what follows, we evaluate the effectiveness of AWAMOCHE's key ideas (namely, stale-read annotations, in-place revisiting and speculative revisiting) both individually, and as a whole. To that end, we evaluate AWAMOCHE on a set of benchmarks that both amplify the weaknesses of standard DPOR, as well as demonstrate the applicability of our approach in realistic workloads. In all our tests, we compare AWAMOCHE against a vanilla version of TruSt, a version of TruSt that employs stale-read annotations (TruSt$_{\text{STALE}}$), and a version of TruSt that employs both stale-read annotation and in-place revisiting (TruSt$_{\text{IPR}}$).

Even though there are other stateless model checking tools that can be used to verify C/C++ programs (namely, GENMC [19] and NIDHUGG [1]), we do not compare against them here, as we care about AWAMOCHE's performance compared to TruSt. We only mention in passing that we expect GENMC's performance to be similar to that of TruSt$_{\text{STALE}}$ (as its implementation incorporates various optimizations for `assume` statements), and NIDHUGG's similar to TruSt$_{\text{IPR}}$ (as it employs an optimization with a similar effect to in-place revisiting [14]). We also note that comparing with NIDHUGG is difficult since it operates under a different memory model, and does not transform the same types of loops to `assume` statements as AWAMOCHE (also see Sect. 7).

We draw two major conclusions from our evaluation. First, AWAMOCHE's optimization yields exponential performance benefits compared to standard DPOR approaches. Second, these benefits do not only apply to small synthetic benchmarks, but also extend to realistic concurrent data structures.

Experimental Setup. We conducted all experiments on a Dell PowerEdge M620 blade system, running a custom Debian-based distribution, with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz), and 256 GB of RAM. We used LLVM 11.0.1 for AWAMOCHE. Unless explicitly noted otherwise, all reported times are in seconds. We set a timeout limit of 30 min.

Table 1. Synthetic benchmarks

	Executions	TruSt		$\text{TruSt}_{\text{STALE}}$		$\text{TruSt}_{\text{IPR}}$		AWAMOCHE	
		Blocked	*Time*	*Blocked*	*Time*	*Blocked*	*Time*	*Blocked*	*Time*
`orch-run(4)`	1	15	.01	0	.01	0	.01	0	.01
`orch-run(5)`	1	31	.01	0	.01	0	.01	0	.01
`orch-run(6)`	1	63	.01	0	.01	0	.01	0	.01
`wait-workers(4)`	24	96	.03	96	.02	0	.01	0	.01
`wait-workers(5)`	120	600	.09	600	.09	0	.03	0	.03
`wait-workers(6)`	720	4320	.56	4320	.56	0	.14	0	.14
`nr+nw(3,2)`	0	27	.01	10	.03	1	.01	1	.01
`nr+nw(5,4)`	0	3125	.1	126	.03	1	.01	1	.01
`nr+nw(6,5)`	0	46656	1.32	462	.06	1	.01	1	.01
`conf-loop(4)`	24	256	.04	176	.03	124	.03	0	0.01
`conf-loop(5)`	120	3905	.09	2010	.10	1185	.06	0	0.02
`conf-loop(6)`	720	75156	1.40	26916	.96	13086	.54	0	0.08

`orch-run`: N threads are spawned and wait to be signaled before they start performing thread-local computations.
`wait-workers`: A worker thread waits for N workers to publish their results before it starts running.
`nr+nw`: A synthetic benchmark where K reader threads wait until a variable written L times by a writer thread satisfies some condition (which cannot be satisfied).
`conf-loop`: N threads perform a confirmation-CAS loop similar to the one of Sect. 1.

6.1 Results

Let us first focus on some benchmarks that help us better understand where each of AWAMOCHE's components can be applied (Table 1). Starting with `orch-run`, we see that even though blocked executions greatly outnumber complete executions, stale-reads annotations alone suffice to bring the number of blocked executions down to zero. This, however, is partly due to luck: in `orch-run`, `main()` spawns a number of workers that do not execute until they are signaled by `main()` using a special variable. In turn, because $\text{TruSt}_{\text{STALE}}$ follows a left-to-right scheduling, when DPOR encounters the worker threads, the scenario where they are not signaled is not considered, since it implies reading a stale value.

By contrast, in `wait-workers` and `nr+nw`, stale-reads annotations are insufficient to eliminate blocking. In these benchmarks, some designated threads wait for the rest of the workers to perform some tasks before proceeding. However, it is not guaranteed that these designated threads are going to be always processed after the rest of the threads by DPOR, and thus stale-reads annotations have little to no effect. Employing in-place revisiting, on the other hand, leads to a dramatic performance improvement: the number of blocked executions is effectively eliminated (the single blocked execution in `nr+nw` is a liveness violation).

Table 2. Real-world benchmarks

	Executions	TruSt		$\text{TruSt}_{\text{STALE}}$		$\text{TruSt}_{\text{IPR}}$		AWAMOCHE	
		Blocked	*Time*	*Blocked*	*Time*	*Blocked*	*Time*	*Blocked*	*Time*
`mpmc-enq(4)`	576	1084	.25	710	.22	532	.17	0	.12
`mpmc-enq(5)`	7200	31325	4.12	16382	3.27	12205	2.72	0	1.48
`mpmc-enq(6)`	86400	730626	82.28	303362	51.29	227766	42.14	0	19.71
`treiber-push(4)`	24	256	.07	176	.04	124	.04	0	.04
`treiber-push(5)`	120	3905	.41	2010	.29	1185	.19	0	.05
`treiber-push(6)`	720	75156	7.49	26916	3.61	13086	1.85	0	.23
`m-enq(4)`	24	124	0.05	124	0.04	124	0.04	0	0.02
`m-enq(5)`	120	1185	0.11	1185	0.14	1185	0.13	0	0.04
`m-enq(6)`	720	13086	1.04	13086	1.05	13086	1.18	0	0.24

`mpmc-enq`: N threads enqueue an item in a multiple-producer multiple-consumer queue.
`treiber-push`: A lock-free stack implementation. N threads are pushing an item.
`m-enq`: A modification of the Michael-Scott queue without the tail pointer. N threads are enqueueing an item.

Analogously to `wait-workers` and `nr+nw`, `conf-loop` demonstrates why in-place revisiting is insufficient when the success of an `assume` does not depend on a single load, but rather on a sequence of actions (as is the case in confirmation loops). As it can be seen, $\text{TruSt}_{\text{IPR}}$ still explores blocked executions, which AWAMOCHE manages to eliminate thanks to speculative revisits.

Moving to the final part of our evaluation, Table 2 demonstrates that the benefits of AWAMOCHE extend to realistic workloads as well. As can be seen from Table 1, none of AWAMOCHE's optimizations is redundant, as they are often all required to eliminate the exploration of blocked executions. Observe, however, that our benchmarks only exercise push or enqueue operations. This is because the respective pop or dequeue operations contain `assume` statements in their confirmation-CAS loops, and therefore cannot be optimized by AWAMOCHE.

7 Related Work

The seminal work of Flanagan and Godefroid [13] has spawned a number of papers on DPOR. Among these, OPTIMAL-DPOR [2] and TruSt [15] stand out, as they provide the first optimal DPOR algorithm, and the first optimal DPOR algorithm with polynomial memory consumption, respectively. TruSt is based on [17] and thus has the extra advantage of being parametric in the choice of the underlying weak memory model.

A lot of works improve on DPOR one way or another. Many techniques introduce coarser equivalence partitionings to combat the state-space explosion problem (e.g., [3,6–8,10–12]). Other works focus on extending it to weak memory models [1,4,5,17,20,24], while others try to leverage particular programming patterns [14,16,18]. Kokologiannakis, Ren, and Vafeiadis [18] in particular, deal with transforming spinloops into `assume` statements, the handling of which we optimize in this paper.

Among those, the work that is closest to ours is GODOT [14]. GODOT is an extension to DPOR that has a similar effect to in-place revisiting in the sense that it only explores executions that are either complete, or denote program termination errors. That said, GODOT only works under SC, and cannot handle stale-read annotations or confirmation loops (which are instrumental in scaling the verification of concurrent data structures, as we saw in Sect. 6). In addition, GODOT's loop transformation is static (in contrast to AWAMOCHE's, which is dynamic), making it easy to construct examples where GODOT's transformation does not work. Finally, even though GODOT does not impose a "no write-write race" restriction on the input programs, this restriction is trivially satisfied for models like SC or TSO [26]: in such models, it is sound to transform writes to atomic exchange statements that write the value they read, thereby ordering all writes to each location.

8 Conclusion

We presented AWAMOCHE, the first memory-model-agnostic DPOR algorithm that is sound, complete, and strongly optimal for programs with await and confirmation-CAS loops. AWAMOCHE avoids blocked executions that arise due to await loops by revisiting blocking reads in-place, and deals with confirmation-CAS loops by also considering revisits whenever two speculative reads read from the same write.

As our theoretical and experimental results demonstrate, AWAMOCHE yields exponential benefits over the current state-of-the-art. Yet, it does not support certain more advanced patterns commonly appearing in concurrent programs, the handling of which we leave as future work. Examples of such patterns include confirmation-CAS loops with `assume` statements between the speculative and the confirmation reads (such statements may arise due to `break`/`continue` instructions), elimination backoff data structures, and await loops that use CASes instead of plain reads. We also believe that our key ideas for achieving strong optimality in these cases should be applicable in other scenarios as well, such as in programs with mutual exclusion locks or transactions.

Acknowledgments. We thank the anonymous reviewers for their feedback. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

References

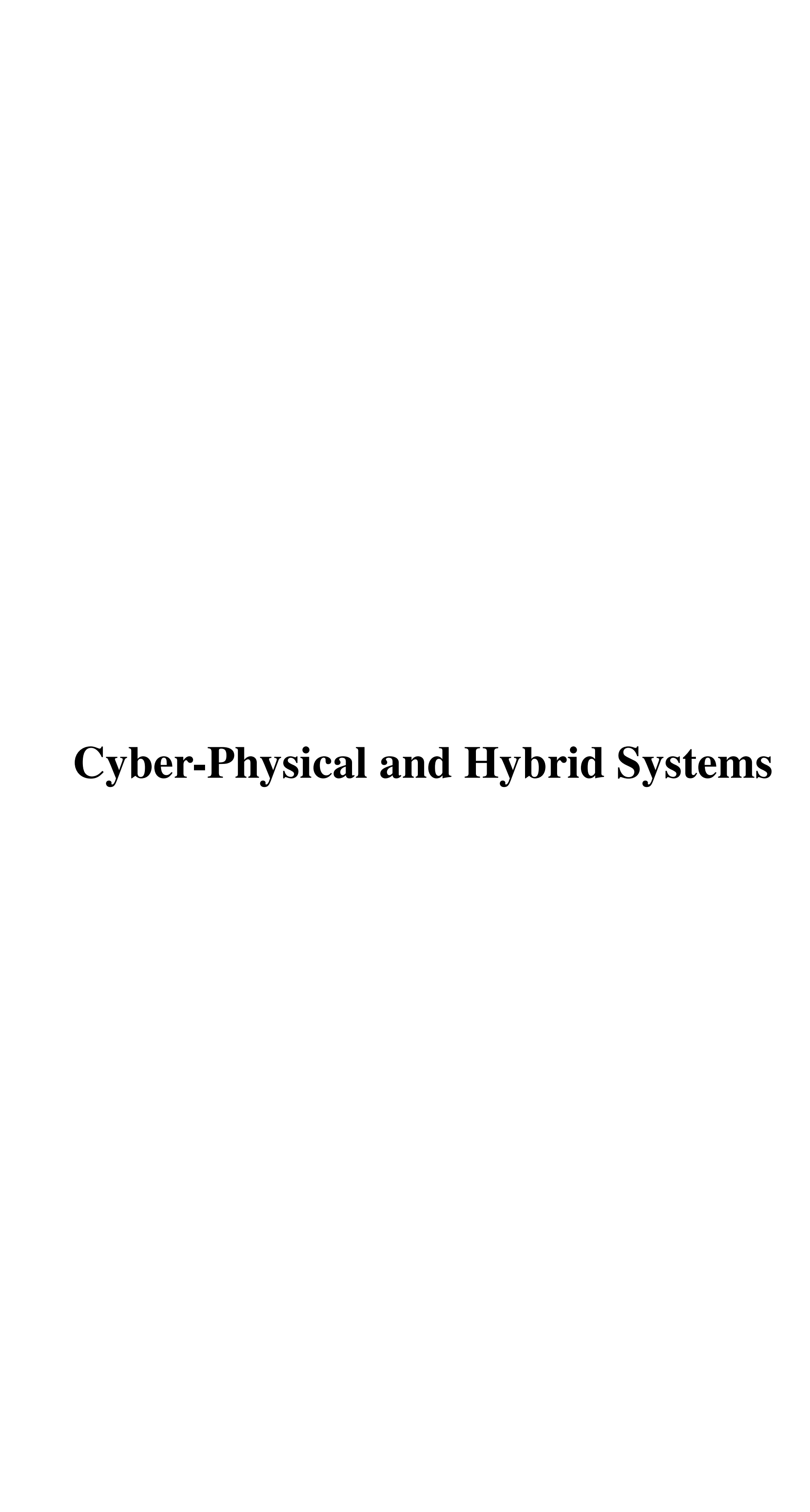
1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28

2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM, New York (2014). https://doi.org/10.1145/2535838.2535845
3. Abdulla, P.A., Atig, M.F., Jonsson, B., Låang, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**, 150:1–150:29 (2019). https://doi.org/10.1145/3360576
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 134–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_8
5. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA), 135:1–135:29 (2018). https://doi.org/10.1145/3276505
6. Agarwal, P., Chatterjee, K., Pathak, S., Pavlogiannis, A., Toman, V.: Stateless model checking under a reads-value-from equivalence. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 341–366. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_16
7. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 526–543. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_26
8. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 392–410. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_24
9. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014). https://doi.org/10.1145/2627752
10. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 229–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_14
11. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Datacentric dynamic partial order reduction. Proc. ACM Program. Lang. **2**(POPL), 31:1–31:30 (2017). https://doi.org/10.1145/3158119
12. Chatterjee, K., Pavlogiannis, A., Toman, V.: Value-centric dynamic partial order reduction. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360550
13. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121. ACM, New York (2005). https://doi.org/10.1145/1040305.1040315
14. Jonsson, B., Lång, M., Sagonas, K.: Awaiting for Godot: stateless model checking that avoids executions where nothing happens. In: FMCAD 2022, pp. 163–172. TU Wien Academic Press (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_35
15. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498711
16. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360599

17. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: PLDI 2019. ACM, New York (2019). https://doi.org/10.1145/3314221.3314609
18. Kokologiannakis, M., Ren, X., Vafeiadis, V.: Dynamic partial order reductions for spinloops. In: FMCAD 2021, pp. 163–172. IEEE (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_25
19. Kokologiannakis, M., Vafeiadis, V.: GenMC: a model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 427–440. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_20
20. Kokologiannakis, M., Vafeiadis, V.: HMC: model checking for hardware memory models. In: ASPLOS 2020, pp. 1157–1171. ACM, Lausanne, Switzerland (2020). https://doi.org/10.1145/3373376.3378480
21. Lahav, O., Namakonov, E., Oberhauser, J., Podkopaev, A., Vafeiadis, V.: Making weak memory models fair. Proc. ACM Program. Lang. **5**(OOPSLA) (2021). https://doi.org/10.1145/3485475
22. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632. ACM, Barcelona, Spain (2017). https://doi.org/10.1145/3062341.3062352
23. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
24. Norris, B., Demsky, B.: CDSChecker: checking concurrent data structures written with C/C++ atomics. In: OOPSLA 2013, pp. 131–150. ACM (2013). https://doi.org/10.1145/2509136.2509514
25. Oberhauser, J., et al.: VSync: push-button verification and optimization for synchronization primitives on weak memory models. In: ASPLOS 2021 - Virtual, pp. 530–545. ACM, USA (2021). https://doi.org/10.1145/3445814.3446748
26. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Cyber-Physical and Hybrid Systems

3D Environment Modeling for Falsification and Beyond with Scenic 3.0

Eric Vin[1(✉)], Shun Kashiwa[1], Matthew Rhea[3], Daniel J. Fremont[1], Edward Kim[2], Tommaso Dreossi[4], Shromona Ghosh[5], Xiangyu Yue[6], Alberto L. Sangiovanni-Vincentelli[2], and Sanjit A. Seshia[2]

[1] University of California, Santa Cruz, USA
{evin,shkashiw,dfremont}@ucsc.edu
[2] University of California, Berkeley, USA
[3] SentinelOne, Mountain View, USA
[4] insitro, San Francisco, USA
[5] Waymo LLC, Mountain View, USA
[6] The Chinese University of Hong Kong, Hong Kong, China

Abstract. We present a major new version of Scenic, a probabilistic programming language for writing formal models of the environments of cyber-physical systems. Scenic has been successfully used for the design and analysis of CPS in a variety of domains, but earlier versions are limited to environments that are essentially two-dimensional. In this paper, we extend Scenic with native support for 3D geometry, introducing new syntax that provides expressive ways to describe 3D configurations while preserving the simplicity and readability of the language. We replace Scenic's simplistic representation of objects as boxes with precise modeling of complex shapes, including a ray tracing-based visibility system that accounts for object occlusion. We also extend the language to support arbitrary temporal requirements expressed in LTL, and build an extensible Scenic parser generated from a formal grammar of the language. Finally, we illustrate the new application domains these features enable with case studies that would have been impossible to accurately model in Scenic 2.

Keywords: Scenario description language · Synthetic data · Probabilistic programming · Automatic test generation · Simulation

1 Introduction

A major challenge in the design of cyber-physical systems (CPS) like autonomous vehicles is the heterogeneity and complexity of their environments. Increasingly, problems of perception, planning, and control in such environments have been tackled using machine learning (ML) algorithms whose behavior is not well-understood. This trend calls for verification techniques for ML-based CPS; however, a significant barrier has been the difficulty of constructing *formal models*

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 253–265, 2023.
https://doi.org/10.1007/978-3-031-37706-8_13

that capture the diversity of these systems' environments [25]. Indeed, building such models is a prerequisite not only for verification but any formal analysis.

Scenic [10,12] is a probabilistic programming language that addresses this challenge by providing a precise yet readable formalism for modeling the environments of CPS. A Scenic program defines a *scenario* describing physical objects in a world, placing a probability distribution on their positions and other properties; a single program can generate many different concrete *scenes* by sampling from this distribution. Scenic also allows defining a stochastic policy describing how agents behave over time, and implementing the resulting dynamic scenarios in a variety of external simulators. Environment models defined in Scenic can be used for many tasks: falsification, as in the VerifAI toolkit [5], but also debugging, training data generation, and real-world experiment design [13]. These tasks have been successfully demonstrated in a variety of domains including autonomous driving [29], aviation [9], and reinforcement learning agents [1].

Despite Scenic's successes, it has several limitations that prevent its use in a number of applications of interest. First, the original language models the world as being *two-dimensional*, since this enables a substantial simplification in the language's syntax (e.g., orientations being a single angle) as well as optimizations in its implementation. The 2D assumption is reasonable for domains such as driving but leaves Scenic unable to properly model environments for aerial and underwater vehicles, for example. There can be problems even for ground vehicles: Scenic could not generate a scene where a robot vacuum is underneath a table, as their 2D bounding boxes would overlap and Scenic would treat them as colliding. The use of bounding boxes rather than precise shapes also leads Scenic to use a simplistic visibility model that ignores occlusion, making it possible for Scenic to claim objects are visible when they are not and vice versa: a serious problem when generating training data for a perception system.

Fundamentally, verification of AI-based autonomous systems requires reasoning about perception and physics in a 3D world. To support such reasoning, a formal environment modeling language must provide faithful representations of 3D geometry. Towards this end, we present Scenic 3.0[1], a largely backwards-compatible major release featuring:

- **Native 3D Syntax**: We update Scenic's existing syntax to support 3D geometry, and add new syntax making it possible to define complex 3D scenarios simply. For example, an object's orientation can be specified as being tangent to a surface and facing another object as much as possible.
- **Precise 3D Shapes**: The shapes of objects (as well as surfaces and volumes) can be given by arbitrary 3D meshes, with Scenic performing precise reasoning about collisions, containment, tangency, etc.
- **Precise Visibility**: We use ray tracing for precise visibility checks that take occlusion into account.
- **Temporal Requirements**: We support arbitrary Linear Temporal Logic [21] properties to constrain dynamic scenarios (vs. only $\mathbf{G}p$ and $\mathbf{F}p$ in Scenic 2).

[1] Available at: https://github.com/BerkeleyLearnVerify/Scenic/.

- **Rewritten Parser**: We give a Parsing Expression Grammar [8] for Scenic, using it to generate a parser with more precise error messages and better support for new syntax and optimization passes.

We first define the new features in Scenic 3 in detail in Sect. 2, working through several toy examples. Then, in Sect. 3, we describe two case studies using Scenic with scenarios that could not be accurately modeled without the new features: falsifying a specification for a robot vacuum and generating training data constrained by an LTL formula for a self-driving car's perception system.

Related Work. There are many tools for test and data generation [3]. Some approaches learn from examples [7,26] and so do not provide specific control over scenarios as Scenic does. Approaches based on rules or grammars [17,20,26] provide some control but have difficulty enforcing requirements over the generated data as a whole. Several probabilistic programming languages have been used for generation of objects and scenes [15,22,23], but none of them provide specialized syntax to lay out geometric scenarios, nor for describing dynamic behaviors. Finally, there has been work on synthetic data generation of 3D scenes and objects using ML techniques such as GANs (e.g., [7,14,30]), but these lack the specificity and controllability provided by a programming language like Scenic.

2 New Features

2.1 3D Geometry

The primary new feature in Scenic 3 is the generalization of the language to 3 dimensions. Some changes, like changing the type system so that vectors have length 3, are obvious: here we focus on cases where the existing syntax of Scenic does not easily generalize, using simple scenarios to motivate our design choices.

The first challenge when moving to 3D is the representation of an object's orientation in space: Scenic's existing `heading` property, providing a single angle, is no longer sufficient. Instead, we introduce `yaw`, `pitch`, and `roll` angles, using the common convention for aircraft that these represent *intrinsic* rotations (i.e., `yaw` is applied first, then `pitch` is applied to the resulting orientation, etc.). Using intrinsic angles makes it easy to compose rotations: for example if we point an airplane towards a landing strip with `yaw` and `pitch` (either manually or using Scenic's `facing toward` specifier — more on this below), we can add an additional `roll` by adding to that property. To further simplify composition, we add a `parentOrientation` property which specifies the local coordinate system in which the 3 angles above should be interpreted (by default, the global coordinate system). This allows the user to specify an orientation with respect to a previously-computed orientation, for instance that of a tilted surface.

Scenic provides a flexible system of natural language *specifiers* which can be combined to define properties of objects. Consider the following Scenic 3 code:

```
1  objectA = new Object at (1, 2, 3), facing (45 deg, 0, 90 deg)
2  objectB = new Object left of objectA by 1
3  objectC = new Object above objectB by 1,
4      facing (Range(0,30) deg, Range(0,30) deg, 0)
```

Here, we use the `at` specifier to define a specific `position` for object A; the `facing` specifier defines the object's `orientation` using explicit yaw, pitch, and roll angles. We then place object B left of A by 1 unit with the `left of` specifier: this specifier now not only sets the `position` property, but also sets the `parentOrientation` property to the orientation of object A (unless explicitly overridden). Thus object B will be oriented the same way as A. Similarly, object C is positioned relative to B and so inherits its orientation as its `parentOrientation`. However, this time we use the `facing` specifier to define random `yaw` and `pitch` angles, so object C will face up to 30° off of B.

Another way to specify an object's orientation is the `facing toward` specifier. This is a case where the 2D semantics become ambiguous in 3D. Consider a scenario where the user wants an airplane to be "facing toward" a runway: the plane's body should be oriented toward the runway (giving its yaw), but it is not clear whether in addition the plane should be pitched downward so that its nose points directly toward the runway. To allow for both interpretations, Scenic 3 has `facing toward` only specify `yaw`, while the new `facing directly toward` specifier also specifies `pitch`. This is illustrated in Fig. 1.

Another common practice in 3D space is to place one object *on* another. For example, we may want to place a chair on a floor, or a painting on a wall. Scenic's existing `on` specifier, which sets the `position` of an object to be a uniformly random point in a given region, does not suffice for such cases because it would cause the chair to intersect the floor or the painting to penetrate the wall (or both). To fix this issue, we allow each object to define a *base* point, which `on` positions instead of the object's center. The default base point is the bottom center of the object's bounding box, suitable for cars and chairs for example; a `Painting` class could override this to be the back center. Finally, to enable placing objects on each other, objects can provide a `topSurface` property specifying the surface which is considered the "top" for the purposes of the `on` specifier. As before, there is a reasonable default (the upward-pointing faces of the object's mesh) that can be overridden. This syntax is illustrated in Fig. 2.

A final 3D complication arises when positioning objects on irregular surfaces. Consider a pair of cars driving up an uneven mountain road, with one 10 m behind the other. We can use the `ahead of` specifier to place one car 10 m ahead of the other, but then the car will penetrate the road due to its upward slope. Alternatively, the `on` specifier can correctly place the car so it is tangent to the road, but then we cannot directly specify the distance between the cars. The natural semantics here would be to combine the constraints from *both* specifiers, but this is illegal in Scenic 2 where a given property (such as `position`) can only be specified by a single specifier at a time. We enable this usage in Scenic 3 by introducing the concept of a *modifying specifier* that modifies the value of a property already defined by another specifier. Specifically, if an object's

```
1 ego = new Ball at (0,0, 1.25)
2 new Plane at (2,0,0), facing toward ego
3 new Plane at (-2,0,0), facing directly toward ego
```

Fig. 1. Line-of-sight-based orientations in Scenic. The ego ball (highlighted green) is placed above the origin, as seen by the RGB global coordinate axes, with one plane facing towards the ego and another facing directly toward the ego. (Color figure online)

```
1 floor = Object with width 5, with length 5, with height 0.1
2 ego = new Chair on floor
```

Fig. 2. A Scenic program placing a chair on a floor. The Z-axis of the global coordinate axes protrudes from the floor, indicating which direction is up.

`position` is already specified, the `on` specifier will *project* that position down onto the given surface. This is illustrated by the green chair in Fig. 3.

Note that the green chair is correctly upright on the floor even though it was positioned relative to the cube, and so should inherit `parentOrientation` from the cube as discussed above. In this situation, the user has provided no explicit orientation for the chair, and both `below` and `on` can provide one. To resolve this ambiguity, we introduce a *specifier priority* system, where specifiers have different priorities for the properties they specify (generalizing Scenic's existing system where a specifier could specify a property *optionally*). In our example, `below` specifies `position` with priority 1 and `parentOrientation` with priority 3, while `on` specifies these with priorities 1 and 2 respectively. So both specifiers determine `position` (with `on` modifying the value from `below` as explained above), but `on` takes precedence over `below` when specifying `parentOrientation`. This yields the expected behavior while still allowing `below` to determine the orientation when used in combination with other specifiers than `on`.

```
1  floor = new Object with width 5, with length 5, with height 0.1
2  air_cube = new Object at (Range(-5,5), Range(-5,5), 3),
3          facing (Range(0,360 deg), Range(0,30 deg), 0)
4  new Chair below air_cube, with color (0,0,200)    # blue chair
5  ego = new Chair below air_cube, on floor    # green chair
```

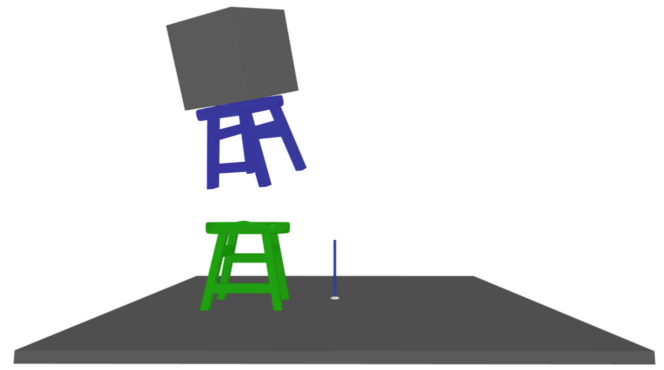


Fig. 3. A Scenic program placing a green chair on the floor under a rotated cube in midair. A blue chair is placed directly under the cube for clarity. (Color figure online)

2.2 Mesh Shapes and Regions

Scenic 2's approximation of objects by their bounding boxes was adequate for 2D driving scenarios, for example, but is wholly inadequate in 3D, where objects are commonly far from box-shaped. For example, consider placing a chair tucked in under a table. Since the bounding boxes of these two objects intersect, Scenic 2 would always reject this situation as a collision and try to generate a new scene, even if the chair and table are entirely separate. In Scenic 3, each object has a precise shape given by its `shape` property, which is set to an instance of the class `Shape`. The most general `Shape` class is `MeshShape`, which represents an arbitrary 3D mesh and can be loaded from standard formats; classes for primitive shapes like spheres are provided for convenience. These shapes are used to perform precise collision and containment checks between objects and regions.

Scenic also supports mesh regions, which can either represent surfaces or volumes in 3D space. For example, given a mesh representing an ocean we might want to sample on the surface for a boat or in the volume for a submarine.

All meshes in Scenic are handled using Trimesh [4], a Python library for triangular meshes, which internally calls out to the tools Blender [27] and OpenSCAD [28] for several operations. These operations tend to be expensive, so Scenic uses several heuristics to cheaply determine simple cases; these can give between a 10x–1000x speedup when sampling scenes.

2.3 Precise Visibility Model

Scenic 2's visibility system simply checks if the bounding box corners of objects are contained in the view cone of the viewing object, which is no longer adequate for 3D scenarios with complex shapes. Visibility checks are now done using ray tracing, and account for objects being able to occlude visibility. In addition to standard pyramidal view cones used for cameras, Scenic correctly handles wraparound view regions such as those of common LiDAR sensors. Visibility checks use a configurable density of rays, and are optimized to only send rays in areas where they could feasibly hit the object.

2.4 Temporal Requirements

A key feature of Scenic is the ability to declaratively impose constraints on generated scenes using `require` statements. However, Scenic 2 only provides limited support for *temporal* requirements constraining how a dynamic scenario evolves over time, with the `require always` and `require eventually` statements. Slightly more complex examples, like "cars A and B enter the intersection after car C", require the user to explicitly encode them as monitors, which is error-prone and yields verbose hard-to-read imperative code: this property requires an 8-line monitor in [12].

Scenic 3 extends `require` to arbitrary properties in Linear Temporal Logic [21], allowing natural properties like this to be concisely expressed:

```
1 require (carA not in intersection and carB not in intersection
2     until carC in intersection)
```

The semantics of the operators `always`, `eventually`, `next`, and `until` are taken from RV-LTL [2] to properly model the finite length of Scenic simulations.

2.5 Rewritten Parser

For interoperability with Python libraries, Scenic is compiled to Python, and the original Scenic parser was implemented on top of the Python parser. This approach imposed serious restrictions on the language design (e.g., forcing nonintuitive operator precedences), made extending the parser difficult, and led to misleading error messages which pointed to the wrong part of the program.

Scenic 3 uses a parser automatically generated from a Parsing Expression Grammar (PEG) [8] for the language. The parser is based on Pegen [24], the parser generator developed for CPython, and the grammar itself was obtained by extending the Python PEG. The new parser outputs an abstract syntax tree representing the structure of the original Scenic code (unlike the old parser), ensuring that syntax errors are correctly localized and simplifying the task of writing analysis and optimization passes for Scenic.

This new parser gives us flexibility in designing and implementing the language. For example, we carefully assigned precedence to the four new temporal operators so that users can naturally express temporal requirements without

unnecessary parentheses. There are additional benefits from having a precise machine-readable grammar for Scenic: for instance, as we wrote the grammar, we discovered ambiguities that had previously been unnoticed and made minor changes to the language to eliminate them. The grammar could also be be used to fuzz test the compiler and other tools operating on Scenic programs.

3 Case Studies

In this section, we discuss two case studies in the robotics simulator Webots [19]. The code for both case studies is available in the Scenic GitHub repository [11]. The first case study, performing falsification of a robot vacuum, illustrates a domain that could not be modeled in Scenic 2 due to the lack of 3D support. The second case study, generating data constrained by an LTL formula for testing or training the perception system of an autonomous vehicle, is an example of how the new features in Scenic 3 can significantly improve effectiveness even in one of Scenic's original target domains.

3.1 Falsification of a Robot Vacuum

In this example we evaluate the iRobot Create [16], a robot vacuum, on its ability to effectively clean a room filled with objects. We use a specification stating that the robot must clean at least a third of the room within 5 min: in Signal Temporal Logic [18], the formula $\varphi = F_{[0,300]}(\mathit{coverage} > 1/3)$. We use Scenic to generate a complete room and export it to Webots for simulation. The room is surrounded by four walls and contains two main sections: in the dining room section, we place a table of varied width and length randomly on the floor, with 3 chairs tucked in around it and another chair fallen over. In the living room section, we place a couch with a coffee table in front of it, both leaving randomly-sized spaces roughly the diameter of the robot vacuum. We then add a variable number of toys, modeled as small boxes, cylinders, cones, and spheres, placed randomly around the room; for a taller obstacle, we place a stack of 3 box toys somewhere in the room. Finally, we place the vacuum randomly on the floor, and use Scenic's `mutate` statement to add noise to the positions and yaw of the furniture. Several scenes sampled from this scenario are shown in Fig. 4.

We tested the default controller for the vacuum against 0, 1, 2, 4, 8, and 16-toy variants of our Scenic scenario, running 25 simulations for each variant. For each simulation, we computed the robustness value [6] of our spec φ. The average values are plotted in Fig. 5, showing a clear decline as the number of toys increases. Many of the runs actually falsified φ: up to 44% with 16 toys.

There are several aspects of this example that would not be possible in Scenic 2. First, the new syntax in Scenic 3 allows for convenient placement of objects, specifically the use of `on` in combination with `left of` and `right of`, to place the chairs on the appropriate side of the dining table but on the floor. Many of the objects are also above others and have overlapping bounding boxes, but because Scenic now models shapes precisely, it is able to properly register these

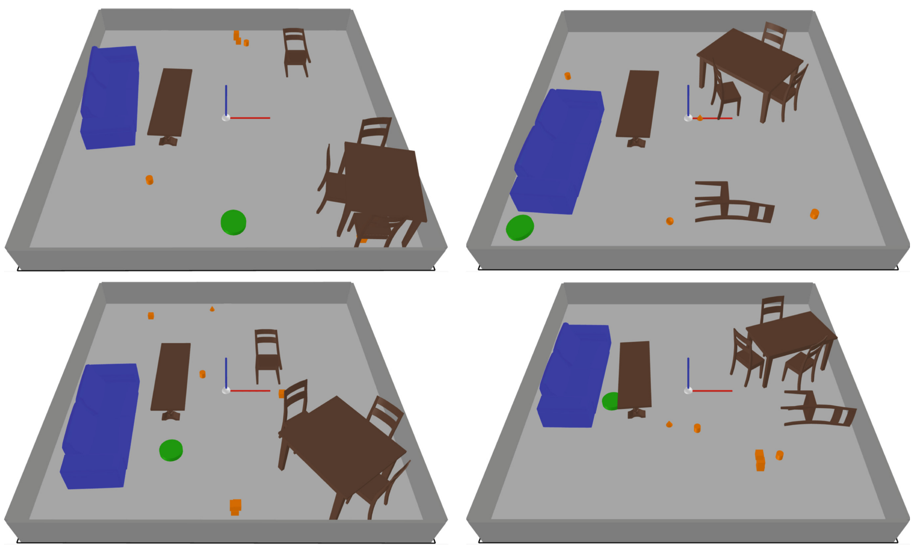


Fig. 4. Several sampled scenes from the robot vacuum scenario.

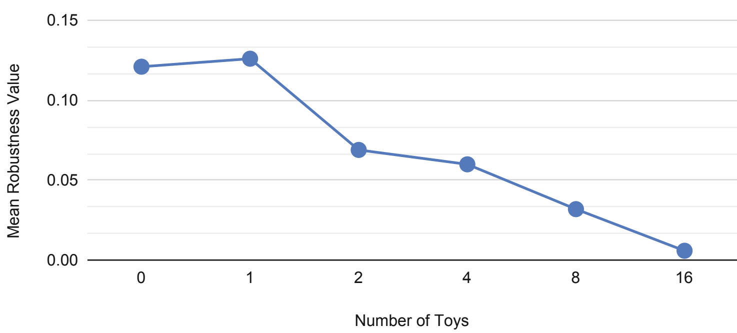


Fig. 5. Spec. robustness value vs. number of toys, averaged over 25 simulations.

objects as non-intersecting and place them in truly feasible locations (e.g., in Fig. 4, the toy under the dining table in the top left scene and the robot under the coffee table in the bottom right scene).

3.2 Constrained Data Generation for an Autonomous Vehicle

In this example we generate instances of a potentially-unsafe driving scenario for use in training or testing the perception system of an AV. Consider a car passing in front of the AV in an intersection where the AV must yield, and so needs to detect the other car before it becomes too late to brake and avoid a collision. We want to generate time series of images labeled with whether or not the crossing car is visible, for a variety of different scenes with different city

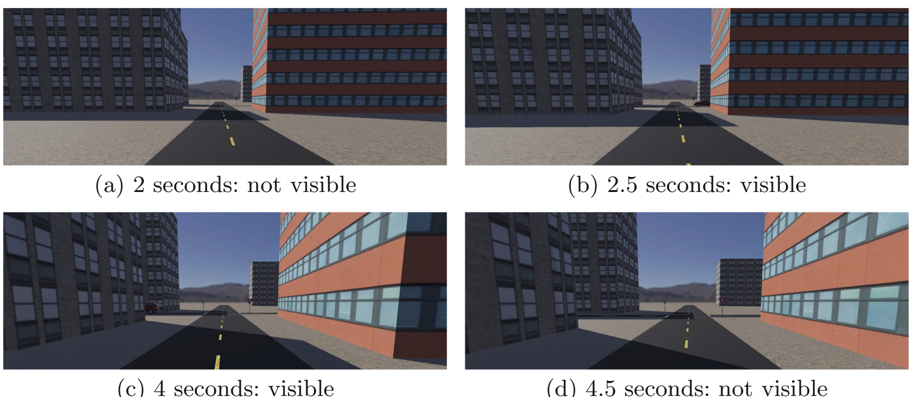

(a) 2 seconds: not visible (b) 2.5 seconds: visible (c) 4 seconds: visible (d) 4.5 seconds: not visible

Fig. 6. Intersection simulation images, with visibility label for the crossing car.

layouts to provide various openings and backdrops. Our scenario places both the ego car (the AV) and the crossing car randomly on the appropriate road ahead of the intersection. We place several buildings along the crossing road that block visibility, allowing some randomness in their position and yaw values. We also place several buildings completely randomly behind the crossing road to provide a diverse backdrop of buildings in the images. Finally, we want to constrain data generation to instances of this scenario where the crossing car is not visible until it is close to the AV, as these will be the most challenging for the perception system. Using the new LTL syntax, we simply write:

```
1  require (not ego can see car) until distance to car < 75
```

Figure 6 shows a simulation sampled from this scenario. In Scenic 2, the crossing car would be wrongly labeled as visible in image (a), since the occluding buildings would not be taken into account. This would introduce significant error into the generated training set, which in previous uses of Scenic had to be addressed by manually filtering out spurious images; this is avoided with the new system.

4 Conclusion

In this paper we presented Scenic 3, a major new version of the Scenic programming language that provides full native support for 3D geometry, a precise occlusion-aware visibility system, support for more expressive temporal operators, and a rewritten extensible parser. These new features extend Scenic's use cases for developing, testing, debugging, and verifying cyber-physical systems to a broader range of application domains that could not be accurately modeled in Scenic 2. Our case study in Sect. 3.1 demonstrated how Scenic 3 makes it easier to perform falsification for CPS with complex 3D environments. Our case study in Sect. 3.2 further showed that even in domains that could already be modeled in

Scenic 2, like autonomous driving, Scenic 3 allows for significantly more precise specifications due to its ability to reason accurately about 3D orientations, collisions, visibility, etc.; these concepts are often relevant to the properties we seek to prove about a system or an environment we want to specify. We expect the improvements to Scenic we describe in this paper will impact the formal methods community both by extending Scenic's proven use cases in simulation-based verification and analysis to a much wider range of application domains, and by providing a 3D environment specification language which is general enough to allow a variety of new CPS verification tools to be built on top of it.

In future work, we plan to develop 3D scenario optimization techniques (complementing the 2D methods Scenic already uses) and explore additional 3D application domains such as drones. We also plan to leverage the new parser to allow users to define their own custom specifiers and pruning techniques.

Acknowledgements. The authors thank Ellen Kalvan for helping debug and write tests for the prototype, and several anonymous reviewers for their helpful comments. This work was supported in part by DARPA contracts FA8750-16-C0043 (Assured Autonomy) and FA8750-20-C-0156 (Symbiotic Design of Cyber-Physical Systems), by Berkeley Deep Drive, by Toyota through the iCyPhy center, and NSF grants 1545126 (VeHICaL) and 1837132.

References

1. Azad, A.S., et al.: Programmatic modeling and generation of real-time strategic soccer environments for reinforcement learning. Proc. AAAI Conf. Artif. Intell. **36**, 6028–6036 (2022)
2. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. **20**(3), 651–674 (2010). https://doi.org/10.1093/logcom/exn075
3. Broy et al.: Model-based testing of reactive systems. Lecture Notes in Computer Science (2005). https://doi.org/10.1007/b137241
4. Dawson-Haggerty, M., et al.: Trimesh. https://trimsh.org
5. Dreossi, T., et al.: VerifAI: a toolkit for the formal design and analysis of artificial intelligence-based systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 432–442. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_25
6. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification, pp. 178–192. Springer, Berlin Heidelberg, Berlin, Heidelberg (2006)
7. Fisher, M., Ritchie, D., Savva, M., Funkhouser, T., Hanrahan, P.: Example-based synthesis of 3d object arrangements. ACM Trans. Graph. **31**(6), 1–11 (2012). https://doi.org/10.1145/2366145.2366154
8. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–122 (2004)

9. Fremont, D.J., Chiu, J., Margineantu, D.D., Osipychev, D., Seshia, S.A.: Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI. Computer Aided Verification, pp. 122–134 (2020)
10. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: A language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (2019)
11. Fremont, D.J., et al.: Scenic Repository. https://github.com/BerkeleyLearnVerify/Scenic
12. Fremont, D.J., et al.: Scenic: a language for scenario specification and data generation. Mach. Learn. (2022). https://doi.org/10.1007/s10994-021-06120-5
13. Fremont, D.J., et al.: Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In: 2020 IEEE Intelligent Transportation Systems Conference, ITSC 2020, pp. 913–920. IEEE (2020). https://arxiv.org/abs/2003.07739
14. Fu, R., Zhan, X., Chen, Y., Ritchie, D., Sridhar, S.: Shapecrafter: A recursive text-conditioned 3d shape generation model. CoRR abs/2207.09446 (2022). https://doi.org/10.48550/arXiv.2207.09446
15. Goodman, N.D., Stuhlmüller, A.: The Design and Implementation of Probabilistic Programming Languages. http://dippl.org (2014) Accessed 28 Jan 2023
16. iRobot: Create Educational Robot. https://edu.irobot.com/what-we-offer/create3
17. Jiang, C., et al.: Configurable 3D scene synthesis and 2D image rendering with per-pixel ground truth using stochastic grammars. Int. J. Comput. Vision **126**(9), 920–941 (2018). https://doi.org/10.1007/s11263-018-1103-5
18. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Proc. FORMATS (2004)
19. Michel, O.: Webots: Professional mobile robot simulation. J. Adv. Robot. Syst. **1**(1), 39–42 (2004). http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf
20. Müller, P., Wonka, P., Haegler, S., Ulmer, A., Van Gool, L.: Procedural modeling of buildings. ACM Trans. Graph. **25**(3) (2006). https://doi.org/10.1145/1141911.1141931
21. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
22. Ritchie, D.: Quicksand: A Lightweight Implementation of Probabilistic Programming for Procedural Modeling and Design. In: 3rd NIPS Workshop on Probabilistic Programming (2014). https://dritchie.github.io/pdf/qs.pdf
23. Ritchie, D.: Probabilistic programming for procedural modeling and design. Ph.D. thesis, Stanford (2016). https://purl.stanford.edu/vh730bw6700
24. Salgado, P.G., van Rossum, G., Nikolaou, L.: Pegen. https://we-like-parsers.github.io/pegen/
25. Seshia, S.A., Sadigh, D., Sastry, S.S.: Towards Verified Artificial Intelligence. ArXiv e-prints (July 2016)
26. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute force vulnerability discovery. Addison-Wesley (2007)
27. The Blender Community: Blender. http://www.blender.org
28. The OpenSCAD Community: OpenSCAD. https://openscad.org
29. Viswanadha, K., et al.: Addressing the IEEE AV test challenge with Scenic and VerifAI. In: 2021 IEEE International Conference on Artificial Intelligence Testing (AITest) (2021)

30. Wang, K., Savva, M., Chang, A.X., Ritchie, D.: Deep convolutional priors for indoor scene synthesis. ACM Trans. Graph. **37**(4), 70 (2018). https://doi.org/10.1145/3197517.3201362

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

A Unified Model for Real-Time Systems: Symbolic Techniques and Implementation

S. Akshay[1], Paul Gastin[2,4], R. Govind[1](✉), Aniruddha R. Joshi[1], and B. Srivathsan[3,4]

[1] Department of CSE, Indian Institute of Technology Bombay, Mumbai, India
{akshayss,govindr,aniruddhajoshi}@cse.iitb.ac.in
[2] Université Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, 91190 Gif-sur-Yvette, France
paul.gastin@ens-paris-saclay.fr
[3] Chennai Mathematical Institute, Chennai, India
sri@cmi.ac.in
[4] CNRS, ReLaX, IRL 2000, Siruseri, India

Abstract. In this paper, we consider a model of *generalized timed automata* (GTA) with two kinds of clocks, *history* and *future*, that can express many timed features succinctly, including timed automata, event-clock automata with and without diagonal constraints, and automata with timers.

Our main contribution is a new simulation-based zone algorithm for checking reachability in this unified model. While such algorithms are known to exist for timed automata, and have recently been shown for event-clock automata without diagonal constraints, this is the first result that can handle event-clock automata with diagonal constraints and automata with timers. We also provide a prototype implementation for our model and show experimental results on several benchmarks. To the best of our knowledge, this is the first effective implementation not just for our unified model, but even just for automata with timers or for event-clock automata (with predicting clocks) without going through a costly translation via timed automata. Last but not least, beyond being interesting in their own right, generalized timed automata can be used for model-checking event-clock specifications over timed automata models.

Keywords: Real-time systems · Timed automata · Event-clock automata · Clocks · Timers · Verification · Zones · Simulations · Reachability

This work was supported by UMI ReLaX, IRL 2000 and DST/CEFIPRA/INRIA Project EQuaVE. S Akshay was supported in part by DST/SERB Matrics Grant MTR/2018/000744. Paul Gastin was partially supported by ANR project Ticktac (ANR-18-CE40-0015).

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 266–288, 2023.
https://doi.org/10.1007/978-3-031-37706-8_14

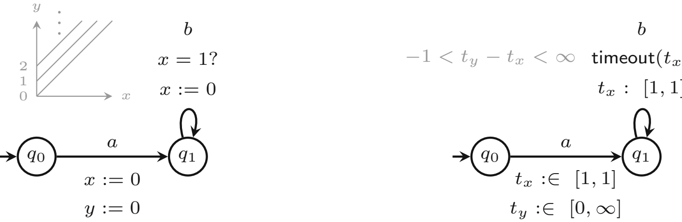


Fig. 1. An automaton with clocks on left, and timers on right for same constraints.

1 Introduction

The idea of adding real-time dynamics to formal verification models started as a hot topic of research in the 1980 s [6,11]. Over the years, timed automata [8,9] has emerged as a leading model for finite-state concurrent systems with real-time constraints. Timed automata make use of *clocks*, real-valued variables which increase along with time. Constraints over clock values can be used as guards for transitions, and clocks can be reset to 0 along transitions. It is notable that the early works in this area made use of *timers* to deal with real-time [13,22,32]. Timers are started by setting them to some initial value within a given interval. Their values decrease with time, and an *timeout* event can be used in transitions to detect the instant when the timers become 0. Quoting from [6], the shift from timers to clocks in timed automata, as we know them today, is attributed to the fact that: "*apart from some technical conveniences in developing the emptiness algorithm and proving its correctness, the reformulation allows a simple syntactic characterization of determinism for timed automata*". Over the last thirty years, the study of timed automata has led to the development of rich theory and industry-strength verification tools. The use of clocks has also allowed for the extension of the model to more complex constraints and assignments to clocks in transitions [14,17]. Furthermore, considering more sophisticated rates of evolution for clocks gives the yet another well-established model of hybrid automata [7].

When it comes to the reachability problem, timers do have some nice properties. Let us explain with an example. Figure 1 shows a timed automaton on the left, and an automaton with timers on the right, for the set of words ab^* such that the time between every consecutive letters is 1. The timed automaton sets clock x to 0 and checks for the guard $x = 1?$ to enforce the timing constraint. The automaton with timers, on the right, sets a timer t_x to 1, and asks for its expiry in the immediate next action. Clock y and timer t_y are not necessary for the required timing property, but we add them to illustrate a different aspect that we will describe now. To solve the reachability problem, a symbolic enumeration of the state space is performed. In the timed automaton, at state q_1, the enumeration gives constraints $y - x = n$ for every $n \geq 0$. Starting from $y - x = n$ and executing b gives $y - x = n + 1$, due to the combination of guard $x = 1?$ and reset $x := 0$. This shows that a naïve symbolic enumeration is not bound

to terminate. The question of developing finite abstractions for timed automata has been a central problem of study which started in the late 90s and continues till date (see recent surveys [18,38]). Such an issue does not occur with timers. In the automaton with timers on the right, t_x is set to 1 and t_y is set to some arbitrary value in the transition to q_1. This gives $-1 \leq t_y - t_x \leq \infty$ for the set of all possible timer values. When t_x times out, the value of t_y could still be any value from 0 to ∞. When t_x is set to 1 again, the set of possible timer values still satisfies the same constraint $-1 \leq t_y - t_x \leq \infty$ leading to a fixed point with a finite reachable state space. The fact that symbolic enumeration terminates on an automaton with timers was already observed in [22]. To our knowledge, later works on timed automata reachability never went back to timers, and there is no tool support that we know of to deal with models with timers directly. We find this surprising given that timers occur naturally while modeling real-time systems and moreover they enjoy this finiteness property.

In addition to clocks and timers, *event-clocks* are another special type of clock variables that are used to deal with timing constraints [10], which are attached to events. An event-recording clock for event a maintains the time since the previous occurrence of a, whereas an event-predicting clock for a gives the time to the next occurrence of a. Event-clocks have been used in the model of event-clock automata (ECA), and also in the logic of event-clocks [36]. These works argue that event-clocks can express typical real-time requirements. Theoretically, ECA can be determinized, and hence complemented. Therefore, model-checking an event-clock (logic or automaton) specification φ over a timed automaton $\mathcal{A}$ can be reduced to reachability on the product of $\mathcal{A}$ and the ECA for $\neg\varphi$. This makes event-clocks a convenient feature in specifications.

Recently, a symbolic enumeration algorithm for ECA was proposed [3]. It was noticed that when restricted to event-predicting clocks, the symbolic enumeration terminates without any additional checks (similar to the case of timers), whereas for the combination involving event-recording clocks, one needs simulation techniques from the timed automata literature. The same work showed how to adapt the best known simulation technique from timed automata into the setting of ECA. However, as discussed above, for model-checking we need a model containing both conventional clocks, timers and event-clocks. To our knowledge, no tool can directly work on such models.

Our goal in this work is to provide a one stop solution to real-time verification, be it reachability analysis or model-checking (over event-clock specifications), be it using models with clocks, or models with timers. We consider a unified model of a timed automaton over variables that can simulate normal clocks, timers and event-clocks. Here are our key contributions:

1. We define a new model of generalized timed automata (GTA) which have two types of variables, called *history* clocks and *future* clocks. History clocks generalize normal clocks as well as event-recording clocks, while future clocks generalize event-predicting clocks and timers. However, unlike event-clocks, clocks in GTA are not necessarily associated with events. We also consider a generic syntax that allows for diagonal constraints between variables.

2. We show undecidability of reachability for GTA, and study a *safe subclass* that makes the model decidable. Safe GTA already subsume timed automata, event-clock automata (with diagonal constraints) and automata with timers.
3. We adapt state-of-the-art symbolic enumeration techniques from timed automata literature to safe GTA. While we make use of ideas presented in [22] and [3], these works do not contain diagonal constraints between variables. Our main technical and theoretical innovation lies in a new termination analysis of the symbolic enumeration in the presence of diagonal constraints. Surprisingly, we show that the enumeration terminates as long as the diagonal constraints are restricted to usual clocks and event-clocks, but not timers.
4. We develop a prototype implementation of our model and algorithm in TCHECKER, an open-source platform for timed automata analysis, and show promising results on several existing and new benchmarks. To the best of our knowledge, our tool is the first that can handle event-clock automata, a model that till date has been the subject of many theoretical results.

Related Works. In the work that first introduced ECA, a translation from ECA to a timed automaton was also proposed. However, this translation is not efficient: in the worst case, this translation incurs a blowup in the number of clocks and states. In [27,28], an extrapolation approach using maximal constants has been studied for ECA. However, it has been observed that simulation-based techniques are both more effective [14,16] and efficient [5,24–26] than extrapolation for checking reachability. Recently, [3] proposed a zone-based reachability algorithm for diagonal-free ECA, using simulations for finiteness, but there was no accompanying implementation. Diagonal constraints have long been known to allow succinct modeling [15] for the class of timed-automata, but only recently a zone-based algorithm that directly works on such automata, was proposed. ECA with diagonals are more expressive than ECA [19]. In this work, we propose a zone-based algorithm for a unified model that subsumes ECA with diagonals.

The use of history clocks and prophecy clocks in ECAs is in the same spirit as past and future modalities in temporal logics - this makes ECAs an attractive model for writing timed specifications. Indeed, this has also led to a development of various temporal logics with event-clocks [1,23,36]. ECA with diagonal constraints have been well-studied, such as in the context of timeline based planning [19,20]. Finally, while there has been substantial advances in the theory of ECA, to the best of our knowledge, the only tool that handles ECA is TEMPO [37], and even this tool is restricted to just history clocks.

Structure of the Paper. In Sect. 2 we start by defining the generalized model. Section 3 examines its expressiveness, while Sect. 4 deals with the reachability problem and the safe subclass. Section 5 develops the symbolic enumeration technique, while Sect. 6 explains how distance graphs can be extended to this setting. Section 7 is dedicated to finiteness. Finally, we provide our experimental results in Sect. 8 and conclude with Sect. 9. All the missing proofs can be found in the full version of the paper [2].

2 Generalized timed automata

In this section we introduce the unified model. While we build on classical ideas from timed automata, almost every aspect is extended and below we highlight these changes. We define $X = X_H \uplus X_F$ to be a finite set of real-valued variables called *clocks*, where X_H is the set of *history clocks*, and X_F is the set of *future clocks*. History clocks always have a non-negative value and can increase arbitrarily along with time. Future clocks always have a non-positive value and can only increase until their values hit 0. History clocks simulate the usual clocks in timed automata and recording clocks of event-clock automata (ECA), and future clocks simulate timers and prophecy clocks of ECA. Both these clocks can take a special "undefined value" which marks that they are inactive. To deal with this naturally, we consider an extension of the reals with $+\infty$ and $-\infty$ as in [3]. The difference here is that we also have the so-called diagonal constraints.

Extending Clock Constraints. Let $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ denote the set of all real numbers along with $-\infty$ and $+\infty$. The usual $<$ order on reals is extended to deal with $\{-\infty, +\infty\}$ as: $-\infty < c < +\infty$ for all $c \in \mathbb{R}$ and $-\infty < \infty$. Similarly, $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$ denotes the set of all integers along with $-\infty$ and $+\infty$. Let $\mathbb{R}_{\geq 0}$ (resp. $\mathbb{R}_{\leq 0}$) be the set of non-negative (resp. non-positive) reals. Let $\mathcal{C} = \{(\triangleleft, c) \mid c \in \overline{\mathbb{R}} \text{ and } \triangleleft \in \{\leq, <\}\}$, called the set of weights.

Let $X \cup \{0\}$ be the set obtained by extending the clocks of GTA with the special constant clock 0. Note that this clock will always have the value 0. Let $\Phi(X)$ denote a set of clock constraints generated by the following grammar: $\varphi ::= x - y \triangleleft c \mid \varphi \wedge \varphi$ where $x, y \in X \cup \{0\}$, $(\triangleleft, c) \in \mathcal{C}$ and $c \in \overline{\mathbb{Z}}$. The introduction of the special constant clock 0 allows us to treat constraints with just a single clock as special cases: the constraint $x \triangleleft c$ is equivalent to $x - 0 \triangleleft c$ and the constraint $c \triangleleft x$ is equivalent to $0 - x \triangleleft -c$. We often write $x = c$ as a shorthand for $x \leq c \wedge c \leq x$. Constraints of the form $x - y \triangleleft c$ will be called *atomic constraints*. A constraint of the form $x - y \triangleleft c$ is a *diagonal* (resp. *non-diagonal*) constraint if $x, y \neq 0$ (resp. $x = 0$ or $y = 0$).

To evaluate the constraints allowed by $\Phi(X)$, we extend addition on real numbers with the convention that $(+\infty) + \alpha = \alpha + (+\infty) = +\infty$ for all $\alpha \in \overline{\mathbb{R}}$ and $(-\infty) + \beta = \beta + (-\infty) = -\infty$, as long as $\beta \neq +\infty$. We also extend the unary minus operation from real numbers to $\overline{\mathbb{R}}$ by setting $-(+\infty) = -\infty$ and $-(-\infty) = +\infty$. Abusing notation, we write $\beta - \alpha$ for $\beta + (-\alpha)$. Notice that with this extended addition, the minus operation does not distribute over addition[1].

Extending Valuations. A valuation of clocks is a function $v \colon X \cup \{0\} \mapsto \overline{\mathbb{R}}$ which maps the special clock 0 to 0, history clocks to $\mathbb{R}_{\geq 0} \cup \{+\infty\}$ and future clocks to $\mathbb{R}_{\leq 0} \cup \{-\infty\}$. We denote by $\mathbb{V}(X)$ or simply by $\mathbb{V}$ the set of valuations over X. We say that clock x is *defined* (resp. *undefined*) in v when $v(x) \in \mathbb{R}$ (resp. $v(x) \in \{-\infty, +\infty\}$). Let $x, y \in X \cup \{0\}$ be clocks (including 0) and let $(\triangleleft, c)$ be a weight. For valuations $v \in \mathbb{V}$, define $v \models y - x \triangleleft c$ as $v(y) - v(x) \triangleleft c$.

[1] Notice that $-(a + b) = (-a) + (-b)$ when a or b is finite or when $a = b$. But, when $a = +\infty$ and $b = -\infty$ then $-(a + b) = -\infty$ whereas $(-a) + (-b) = +\infty$.

We say that a valuation v satisfies a constraint φ in $\Phi(X)$, denoted as $v \models \varphi$, when v satisfies all atomic constraints in φ.

By definition, we easily check that the constraint $y - x \triangleleft c$ is equivalent to *true* (resp. *false*) when $(\triangleleft, c) = (\leq, +\infty)$ (resp. $(\triangleleft, c) = (<, -\infty)$). Constraints that are equivalent to *true* or *false* will be called trivial, whereas all others are non-trivial constraints. If $(\triangleleft, c) \neq (\leq, +\infty)$ then $v \models y - x \triangleleft c$ never holds when $v(x) = -\infty$. Also, if $v(x) = v(y) \in \{-\infty, +\infty\}$ then $v \models y - x \triangleleft c$ only holds for $(\triangleleft, c) = (\leq, +\infty)$. For a non-trivial constraint $y - x \triangleleft c$, we have

- $v \models y - x \triangleleft c$ iff $v(y) < +\infty = v(x)$ or ($v(x)$ is finite and $v(y) \triangleleft v(x) + c$).
- $v \models y - x \leq -\infty$ iff $v(y) < +\infty = v(x)$ or $v(y) = -\infty < v(x)$.
- $v \models y - x < +\infty$ iff $v(x) \neq -\infty$ and $v(y) \neq +\infty$.

We abuse notation and for $Y \subseteq X$, we define $Y \triangleleft c$ as $\bigwedge_{y \in Y} y \triangleleft c$, and $Y = c$ as $\bigwedge_{y \in Y} y = c$. We denote by $v + \delta$ the *valuation* obtained from valuation v by increasing by $\delta \in \mathbb{R}_{\geq 0}$ the value of all clocks in X. Note that, from a given valuation, not all time elapse result in valuations since future clocks need to stay at most 0. For example, from a valuation with $v(x) = -3$ and $v(y) = -2$, where x, y are future clocks, one can elapse at most 2 time units.

Extending Resets. For history clocks, the reset operation sets the clock to 0. For future clocks, the reset operation says that all constraints on the clock must be discarded, i.e., the clock is *released*. Given that the set of clocks is partitioned into history clocks and future clocks, we use the same notation $[R]v$ to talk about the change of clocks in R, whether it be reset/release. Formally, given a set of clocks $R \subseteq X$, we define $[R]v$ as $\{v' \in \mathbb{V} \mid v'(x) = 0 \ \forall\, x \in R \cap X_H \text{ and } v'(x) = v(x) \ \forall\, x \notin R\}$. Observe that *the release operation* is implicit: each future clock in R could take any value (not necessarily the same) from $[-\infty, 0]$ in $[R]v$. Note that $[R]v$ is a singleton when R contains only history clocks - this corresponds exactly to the reset operation in timed automata. Then, we simply write $v' = [R]v$ instead of $\{v'\} = [R]v$. When R contains only future clocks, $[R]v$ is the set of valuations obtained by releasing each clock in R while keeping the value of all other clocks unchanged. For $W \subseteq \mathbb{V}$, we let $[R]W = \bigcup_{v \in W} [R]v$. We have $[R' \cup R'']W = [R']([R'']W)$.

Extending Guards and Transitions. Before we define GTA, let us focus on the language to specify transitions. In normal timed automata, as shown in Fig. 2, a transition reads a letter, checks a guard $g \in \Phi(X_H)$ and then resets a subset R of (history) clocks. But in any one transition only a pair of guard, reset is performed and one cannot interleave them.

Fig. 2. A transition of TA (left) and of a GTA (right)

We generalize this to our setting with history and future clocks but also to allow arbitrary interleaving of guards and changes (to model this with a TA one may use a sequence of multiple transitions without delays in-between.) Formally, an *instantaneous timed program* is generated by the following grammar:

$$\text{prog} := \text{guard} \mid \text{change} \mid \text{prog}; \text{prog}$$

where guard $= g \in \Phi(X)$ and change $= [R]$ for some $R \subseteq X$. While guard and change are atomic programs, prog; prog refers to sequential composition. The set of all programs generated by the above grammar will be denoted Programs. Then on a transition, we simply have a pair of letter label and an instantaneous timed program, e.g., (a, prog) in Fig. 2 (right).

The semantics for programs on a transition must generalize semantics for guards (defined using satisfaction relation $\models$ above) and resets/release (defined using $[R]$ above). But there is an obvious difference between these two: a guard may be crossed only if the valuation before the guard satisfies it, whereas a *change* (reset or release) defines a relation between the valuations before and after the change. To capture both in a uniform way, we define the semantics of programs as relations on pairs of valuations. Formally, for $v, v' \in \mathbb{V}$, prog $\in$ Programs we define $(v, v') \models \text{prog}$, more conveniently written as $v \xrightarrow{\text{prog}} v'$, inductively:

- $v \xrightarrow{g} v'$ if $v \models g$ and $v' = v$,
- $v \xrightarrow{[R]} v'$ if $v' \in [R]v$,
- $v \xrightarrow{\text{prog}_1;\text{prog}_2} v'$ if $\exists v'' \in \mathbb{V}$ such that $v \xrightarrow{\text{prog}_1} v''$ and $v'' \xrightarrow{\text{prog}_2} v'$.

Now, we have all the pieces necessary to define our generalized model.

Definition 1 (Generalized timed automata). *A generalized timed automata $\mathcal{A}$ is given by a tuple $(Q, \Sigma, X, \Delta, (q_0, g_0), (Q_f, g_f))$, where Q is a finite set of states, Σ is a finite alphabet of actions, $X = X_F \uplus X_H$ is a set of clocks partitioned into future and history clocks, the initialization condition is a pair comprising of an initial state $q_0 \in Q$ and an initial guard $g_0 \in \Phi(X)$ which should be satisfied by initial valuations, similarly, the final condition is a pair comprising of a set of final states $Q_f \subseteq Q$ along with a final guard g_f that must be satisfied by final valuations, and $\Delta \subseteq (Q \times \Sigma \times Programs \times Q)$ is a finite set of transitions. Δ contains transitions of the form $(q, a, prog, q')$, where q is the source state, q' is the target state, a is the action triggering the transition, and prog is the instantaneous timed program that is executed in sequence (from left to right) while firing the transition.*

The semantics of a GTA $\mathcal{A} = (Q, \Sigma, X, \Delta, (q_0, g_0), (Q_f, g_f))$ is given by a transition system $\mathbb{TS}_{\mathcal{A}}$ whose states are *configurations* (q, v) of $\mathcal{A}$, where $q \in Q$ and $v \in \mathbb{V}$ is a valuation. A configuration (q, v) is initial if $q = q_0$ and $v \models g_0$. A configuration (q, v) is accepting if $q \in Q_f$ and $v \models g_f$. Transitions of $\mathbb{TS}_{\mathcal{A}}$ are of two forms: (1) *delay transition*: $(q, v) \xrightarrow{\delta} (q, v + \delta)$ if $(v + \delta) \models X_F \leq 0$, and

(2) *discrete transition*: $(q, v) \xrightarrow{t} (q', v')$ if $t = (q, a, \text{prog}, q') \in \Delta$ and $v \xrightarrow{\text{prog}} v'$. Thus, a discrete transition $t = (q, a, \text{prog}, q')$, where $\text{prog} = \text{prog}_1; \ldots; \text{prog}_n$ can be taken from (q, v) if there are valuations $v_1, \ldots, v_n$ such that $v \xrightarrow{\text{prog}_1} v_1 \xrightarrow{\text{prog}_2} \cdots \xrightarrow{\text{prog}_n} v_n = v'$. A *run* of a GTA is a finite sequence of transitions from an initial configuration of $\mathbb{TS}_{\mathcal{A}}$. A run is said to be *accepting* if its last configuration is accepting.

3 Expressivity of GTA and Examples

The GTA model defined above is rather expressive. Figure 3 illustrates an example which accepts words of the form $a^n b^m$ with $m \leq n$, where each a occurs at time 0, after which b's are seen one by one, with distance 1 between them. The history clock x is used to ensure the timing constraint. For every a that is read, the future clocks y, z decrease by 1. Hence the future clocks y, z maintain the opposite of the number of a's seen. When the automaton starts reading b, the future clocks also start elapsing time and since they cannot go above 0, the number of b's is at most the number of a's. Such a language cannot be accepted by timed automata since the untimed language obtained by removing the time stamps needs to be regular in the case of timed automata. The GTA model is not only expressive, it is also convenient for use. To see this we now show that three classical models of timed systems can be easily captured using GTA. We also illustrate the modeling convenience provided by GTA in Sect. 8 based on experiments.

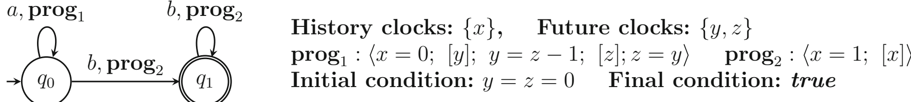


Fig. 3. Example of a GTA

Timed automata. Timed automata (TA) of Alur-Dill [9] can be modeled as a GTA as follows: (1) The set of states of the GTA is the same as the set of states of the TA. (2) There are no future clocks in the GTA and its history clocks are the clocks of the TA. (3) Each transition of the form $q \xrightarrow{a,g,R} q'$ in a TA , where g is a guard, a a letter and R a subset of clocks to be reset, is replaced by a transition $q \xrightarrow{a,\text{prog}} q'$ where $\text{prog} = \langle g; [R] \rangle$. (4) Initially, all clocks must be 0, captured by setting $g_0 = (X_H = 0)$. (5) The final guard is empty: $g_f = \texttt{True}$.

Event-clock Automata. Event-clock automata (ECA) of [10] can be modeled as a GTA as follows: (1) The set of states of the GTA is the same as the set of states of the ECA. (2) For each $a \in \Sigma$, the GTA has a history clock $\overleftarrow{a}$ and a future clock $\overrightarrow{a}$. (3) Each transition of the form $q \xrightarrow{a,g} q'$ in a ECA, where

g is a guard of the ECA, a a letter, is replaced by a transition $q \xrightarrow{a,\text{prog}} q'$ where prog $:= \langle(\overrightarrow{a} = 0); [\overrightarrow{a}]; g; [\overleftarrow{a}]\rangle$. (4) At initialization, history clocks must be undefined (set to ∞), captured by $g_0 = (X_H = \infty)$. (5) At acceptance, all future clocks must be undefined, i.e., $g_f = (X_F = -\infty)$.

Automata with Timers. The third model we consider is that of automata with timers. Timers are timing constructs that are started/initialized with a certain time value at some point/event and *count down* to 0. They measure the time from when they were started till the timer hits 0, where the event of hitting 0 is called *timeout*. However, they can be stopped using a *stop* event at any intermediate point instead and in which case the timer must be freed for reuse later. Timers are a common construct in protocol specification, e.g., the ITU standard which uses timers rather than clocks [30] and Mealy machines with timers [31].

In our setting, a timer can be seen as a specific instance of a future clock. More precisely Automata with timers ($A_{⌛}$) can be modeled as GTA as follows: (1) The set of states of the GTA is the same as the set of states of $A_{⌛}$. (2) The future clocks of GTA are the timers of $A_{⌛}$ and there are no history clocks. Initially, the timers are undefined, captured by $g_0 = (X_F = -\infty)$ and $g_f = \texttt{True}$. (4) A transition of $A_{⌛}$ with action a from q to q' is encoded as $q \xrightarrow{a,\text{prog}} q'$ with:

- if the transition starts timer x with value $c \in \mathbb{R}_{\geq 0}$, then prog $= \langle x = -\infty; [x]; x = -c\rangle$.
- if the transition is guarded by $\texttt{timeout}(x)$, then prog $= \langle x = 0; [x]; x = -\infty\rangle$.
- if the transition stops timer x, then prog $= \langle [x]; x = -\infty\rangle$.

We note that the timer above differs from a prophecy-event-clock (of ECA) though both are future clocks. Prophecy-clocks are released only when the event is seen, so at that point the value of the prophecy-clock must be 0. On the other hand timers can be stopped and released even when their value is not 0. This subtle difference has a surprising impact when we allow diagonal guards.

4 The Reachability Problem for GTA

We are interested in the *reachability problem* for GTA: given a GTA $\mathcal{A}$, does it have an accepting run? For normal TA, the reachability problem is decidable and PSPACE complete as shown in [9]. This was shown using the so-called region abstraction, by proving the existence of a finite time-abstract bisimulation. However, this is not the case for GTA. As explained in the previous subsection, GTA capture ECA, and as shown in [27,28], there exists ECA for which there is no finite time-abstract bisimulation. However, reachability is still decidable in the specific case of ECA, as again shown in [10]. We note that for ECA model of [27,28] there are no diagonal constraints. In this case they show decidability via zone-extrapolation. In [3], another approach for decidability via zone simulations is shown. But again even in this model diagonal constraints are disallowed. Even more critically in GTA, we can capture timers and a priori

we can have diagonal constraints even among timers. So, the question we ask is whether reachability is still decidable for GTA. Surprisingly, the answer is no. The intuition is that with future clocks and diagonal constraints, we get the ability to count (cf. Fig. 3).

Theorem 2. *Reachability for GTA is undecidable.*

Proof. We reduce from counter machines. Given a counter machine, we will build a GTA with one future clock y_C for each counter C and one extra future clock z. The reduction uses diagonal constraints between z and the future clocks y_C.

Initially and after each transition, the value of the future clock z will be 0. Since a future clock has to be non-positive, time elapse is impossible. As an invariant, the value of the future clock y_C is the opposite of the value of counter C. The operations on counter C are encoded with the following programs: (1) $\mathtt{zero}_C = \langle y_C = 0\rangle$ (2) $\mathtt{inc}_C = \langle [z]; z = y_C - 1; [y_C]; y_C = z; [z]; z = 0\rangle$ (3) $\mathtt{dec}_C = \langle y_C \leq -1; [z]; z = y_C + 1; [y_C]; y_C = z; [z]; z = 0\rangle$. In programs $\mathtt{inc}_C$ and $\mathtt{dec}_C$, each release of a future clock is followed by a constraint which restricts the value non-deterministically chosen during the release. For instance, $[z]; z = y_C - 1$ is equivalent to $z := y_C - 1$. Hence, the overall effect of $\mathtt{inc}_C$ is $y_C := y_C - 1$, maintaining all other clocks unchanged, including the invariant $z = 0$. □

Given this negative result, what can we do? A careful observation of the proof tells us that it is the interplay between diagonal constraints and arbitrary releases of future clocks that leads to undecidability. More precisely, the encoding depends on the fact that clocks z and y_C which are used in diagonal constraints ($z = y_C - 1$, $z = y_C + 1$ and $y_C = z$) may have arbitrary values when they are released. This suggests a restricted subclass that we formalize next.

Definition 3 (Safe GTA). *Let $X_D \subseteq X_F$ be a subset of future clocks.*

A program $prog = \langle g_1; [R_1]; g_2; [R_2]; \ldots; g_k; [R_k]; g_{k+1}\rangle$ is X_D-safe if

- *diagonal constraints between future clocks are restricted to clocks in X_D: if $x - y \triangleleft c$ with $x, y \in X_F$ occurs in some g_i then $x, y \in X_D$;*
- *clocks in X_D should be 0 or $-\infty$ before being released: if $x \in X_D \cap R_i$ then $x = 0$ or $x = -\infty$ occurs in g_i.*

A GTA $\mathcal{A}$ is X_D-safe if it only uses X_D-safe programs on its transitions and the initial guard g_0 sets each history clock to either 0 or ∞.

Observe that the three examples discussed in Sect. 3 are safe. Timed automata do not have future clocks so the condition is vacuously true. In ECA, event-predicting clocks are always checked for 0 before being released, hence they are safe as well with $X_D = X_F$. Automata with timers without diagonal constraints are also trivially safe with $X_D = \emptyset$. The importance of safety is the following theorem which is the center-piece of this article.

Theorem 4. *Reachability for X_D-safe GTA is decidable.*

We will establish this theorem by showing a finite, sound and complete zone based reachability algorithm for X_D-safe GTA. If the given GTA is not X_D-safe, then we lose proof of termination (unsurprisingly, since the problem is undecidable), but we still maintain soundness. Thus, even for such GTA when our algorithm does terminate it will give the correct answer.

5 Symbolic Enumeration

We adapt the $\mathcal{G}$-simulation framework presented in [26] for timed automata with diagonal constraints to GTA. Diagonal constraints offer succinct modeling [15], but are quite challenging to handle efficiently in zone-based algorithms, and have led to pitfalls in the past: [14] showed that the erstwhile algorithm based on zone-extrapolations that was implemented in tools is incorrect for models with diagonal constraints; moreover no extrapolation based method can work for automata with diagonal constraints. The simulation framework by-passes this impossibility result and is the state-of-the-art for timed automata with diagonal constraints. The framework was extended to event-clock automata without diagonal constraints in [3]. We show that the ideas from [26] and [3] can be suitably combined to give an effective procedure for safe GTAs. This extension to GTAs enables us to understand the mechanics of diagonal constraints in future clocks.

The algorithm based on the $\mathcal{G}$-simulation framework involves:

1. computation of a set of constraints at every state of the automaton by a *static analysis* of the model,
2. a symbolic enumeration using *zones* to compute the *zone graph*,
3. a *simulation relation* between zones to ensure termination of the enumeration.

We will next adapt the static analysis to the GTA setting. The algorithm for the zone graph computation and the implementation of the simulation relation over zones is taken off-the-shelf from [26] and [3], except for a minor adaptation to include diagonal constraints involving future clocks. What is absent, and requires a non-trivial analysis, is the proof of termination. Therefore, we will mainly focus on this aspect and devote Sect. 7 for the termination argument.

$\mathcal{G}$-Simulation and the Static Analysis for GTA. We fix a GTA $\mathcal{A} = (Q, \Sigma, X, T, (q_0, g_0), (Q_f, g_f))$ for this section. Our goal is to define a simulation relation on the semantics of $\mathcal{A}$, i.e., on $\mathbb{TS}(\mathcal{A})$. In the subsequent sections we will lift this to zones and show its finiteness. A simulation relation on $\mathbb{TS}(\mathcal{A})$ is a reflexive, transitive relation $(q, v) \preceq (q, v')$ relating configurations with the same control state and (1) for every $(q, v) \xrightarrow{\delta} (q, v + \delta)$, we have $(q, v') \xrightarrow{\delta} (q, v' + \delta)$ and $(q, v + \delta) \preceq (q, v' + \delta)$, (2) for every transition t, if $(q, v) \xrightarrow{t} (q_1, v_1)$ for some valuation v_1, then $(q, v') \xrightarrow{t} (q_1, v_1')$ for some valuation v_1' with $(q_1, v_1) \preceq (q_1, v_1')$.

For any set G of atomic constraints, we define a *preorder* $\preceq_G$ on valuations:

$$v \preceq_G v' \qquad \text{if } \forall \varphi \in G,\ \forall \delta \geq 0, \qquad v + \delta \models \varphi \implies v' + \delta \models \varphi\,.$$

Notice that in the definition above, we *do not* restrict δ to those such that $v+\delta$ is a valuation: we may have $v(x)+\delta > 0$ for some $x \in X_F$. In usual timed automata, this question does not arise, as elapsing any δ from any given valuation always results in a valuation. But this is crucial for the proof of Theorem 5 below.

Intuitively, the preorder above is a simulation wrt the constraints in G even after time elapse. But we need this to also be a simulation wrt discrete transitions. To achieve this, the set of constraints G should depend on the available

discrete transitions. In fact, we define a map $\mathcal{G}$ from states to set of constraints, in such a way that it captures the simulation wrt the discrete actions. In other words, our focus will be to choose state-dependent sets of constraints (given by the map $\mathcal{G}$) depending on $\mathcal{A}$ such that the resulting preorder induces a simulation on $\mathbb{TS}(\mathcal{A})$.

As a first step towards this, we define, for any set G of constraints and any program prog, a set of constraints $G' = \mathsf{pre}(\text{prog}, G)$ such that, if $v \preceq_{G'} v'$ and $v \xrightarrow{\text{prog}} v_1$ then there exists $v' \xrightarrow{\text{prog}} v_1'$ such that $v_1 \preceq_G v_1'$. This set is defined inductively as follows (G is a set of atomic constraints, R is a set of clocks, g is an *arbitrary* constraint, $y - x \triangleleft c$ is an *atomic* constraint):

$$\begin{aligned}\mathsf{pre}(\text{prog}_1;\text{prog}_2, G) &= \mathsf{pre}(\text{prog}_1, \mathsf{pre}(\text{prog}_2, G))\\ \mathsf{pre}(g, G) &= \mathsf{split}(g) \cup G\\ \mathsf{pre}([R], G) &= \bigcup_{\varphi \in G} \mathsf{pre}([R], \{\varphi\})\end{aligned} \qquad \mathsf{pre}([R], \{y - x \triangleleft c\}) = \begin{cases}\{y - x \triangleleft c\} & \text{if } x, y \notin R\\ \{y \triangleleft c\} & \text{if } x \in R, y \notin R\\ \{-x \triangleleft c\} & \text{if } x \notin R, y \in R\\ \emptyset & \text{if } x, y \in R\end{cases}$$

where $\mathsf{split}(g)$ is the set of atomic constraints occurring in g.

Now, the choice of suitable G will be obtained by static analysis, on the lines of what was done for timed automata with diagonals [24–26], but adapted to our more powerful model. More precisely, we define the map $\mathcal{G}$ from Q to sets of atomic constraints as the least fixpoint of the set of equations:

$$\mathcal{G}(q) = \{x \leq 0 \mid x \in X_F\} \cup \bigcup_{q \xrightarrow{a,\text{prog}} q'} \mathsf{pre}(\text{prog}, \mathcal{G}(q')) \tag{1}$$

Finally, based on $\preceq_G$ and the $\mathcal{G}(q)$ computation, we can define a preorder $\preceq_{\mathcal{A}}$ between configurations of $\mathbb{TS}(\mathcal{A})$ as $(q, v) \preceq_{\mathcal{A}} (q', v')$ if $q = q'$ and $v \preceq_{\mathcal{G}(q)} v'$. We then show that $\preceq_{\mathcal{A}}$ defined above is indeed a simulation relation.

Theorem 5. *The relation $\preceq_{\mathcal{A}}$ is a simulation on the transition system $\mathbb{TS}_{\mathcal{A}}$.*

Zones for GTA and the Zone Graph Computation. Roughly, *zones* [12] are sets of valuations that can be represented efficiently using constraints between differences of clocks. In this section, we introduce an analogous notion for generalized timed automata. We consider *GTA zones*, or simply *zones*, which are special sets of valuations of GTA. A GTA zone is a set of valuations satisfying a conjunction of constraints of the form $y - x \triangleleft c$, where $x, y \in X \cup \{0\}$, $c \in \overline{\mathbb{Z}}$ and $\triangleleft \in \{\leq, <\}$. Thus zones are an abstract representation of sets of valuations. Then, an abstract configuration, also called a *node*, is a pair consisting of a state and a zone. Firing a transition $t := (q, a, \text{prog}, q')$ in a GTA $\mathcal{A}$ from node (q, Z) will result in another node following a sequence of operations that we now define.

GTA Zone Operations. Let g be a guard, $R \subseteq X$ a set of clocks and Z a GTA zone.

- Guard intersection: $Z \cap g := \{v \mid v \in Z \text{ and } v \models g\}$
- Release/Reset: $[R]Z = \bigcup_{v \in Z} [R]v$ (as defined in Sect. 2)
- Time elapse: $\overrightarrow{Z} = \{v + \delta \mid v \in Z, \delta \in \mathbb{R}_{\geq 0} \text{ s.t. } v + \delta \models (X_F \leq 0)\}$

Successor Computation. We can show that starting from a zone Z, the successors after the above operations are also zones (see Theorem 29 in [2]). A guard g can be seen as yet another zone and hence guard intersection is just an intersection operation between two zones. Similarly, the change operation preserves zones. Finally, as is usual with timed automata, zones are closed under the time elapse operation.

Thus, for a transition $t := (q, a, \text{prog}, q')$ and a node (q, Z), we can define the successor node (q', Z'), and we write $(q, Z) \xrightarrow{t} (q', Z')$, where Z' is the zone computed by the following sequence of operations: Let $\text{prog} = \text{prog}_1; \dots; \text{prog}_n$, where each prog_i is an atomic program, i.e., a guard or a change. Then we define zones $Z_1, \dots, Z_{n+1}$ where, $Z_1 = Z$, $Z' = \overrightarrow{Z_{n+1}}$, and for each $1 \leq i \leq n$, $Z_{i+1} = Z_i \cap g_i$ if prog_i is a guard g_i, and $Z_{i+1} = [R_i]Z_i$ if prog_i is a change $[R_i]$.

Now, we can lift zone graphs, simulations from TA to GTA and obtain a symbolic reachability algorithm for GTA.

Definition 6 (GTA zone graph). *Given a GTA $\mathcal{A}$, its* GTA zone graph, denoted $\mathsf{GZG}(\mathcal{A})$*, is defined as follows: Nodes are of the form (q, Z) where q is a state and Z is a GTA zone. The initial node is $(q_0, \overrightarrow{Z_0})$ where q_0 is the initial state and Z_0 is the set of all valuations which satisfy the* initial constraint g_0*: Z_0 is given by $g_0 \wedge \big(X_F \leq 0\big) \wedge \big(X_H \geq 0\big)$. For every node (q, Z) and every transition $t := (q, a, prog, q')$ of $\mathcal{A}$, there is a transition $(q, Z) \xrightarrow{t} (q', Z')$ in the GTA zone graph. A node (q, Z) is accepting if $q \in Q_f$ and $Z \cap g_f$ is non-empty, i.e., there exists a valuation in Z satisfying the final constraint.*

Similar to the case of zone graphs for timed automata and event zone graphs for ECA, the GTA zone graph can be used to decide reachability for generalized timed automata. A node (q, Z) is said to be reachable (in $\mathcal{A}$) if there is a path from the initial node $(q_0, \overrightarrow{Z_0})$ to (q, Z) in $\mathsf{GZG}(A)$. Thus, reachability of a final state in $\mathcal{A}$ reduces to checking reachability of an accepting node in $\mathsf{GZG}(A)$. However, as in the case of zone graphs for timed automata, $\mathsf{GZG}(A)$ is also not guaranteed to be finite. Hence, we need to compute a finite truncation of the GTA zone graph, which is still sound and complete for reachability.

Definition 7 (Simulation on GTA zones and finiteness). *Let $\preceq$ be a simulation relation on $\mathbb{TS}(\mathcal{A})$. For two GTA zones Z, Z', we say $(q, Z) \preceq (q, Z')$ if for every $v \in Z$ there exists $v' \in Z'$ such that $(q, v) \preceq (q, v')$. The simulation $\preceq$ is said to be finite if for every sequence $(q, Z_1), (q, Z_2), \dots$ of* reachable *nodes, there exists $j > i$ such that $(q, Z_j) \preceq (q, Z_i)$.*

Now, the reachability algorithm, as in TA, enumerates the nodes of the GTA zone graph and uses the simulation $\preceq_{\mathcal{A}}$ from Theorem 5 to truncate nodes that are smaller with respect to the simulation. In Sect. 7, we will show that $\preceq_{\mathcal{A}}$ is finite when $\mathcal{A}$ is safe, which implies that the reachability algorithm terminates. But before that we discuss the issue of implementability.

6 Computing with GTA Zones Using Distance Graphs

To implement the reachability algorithm described above, we will view zones as *distance graphs*, as is usually done in the literature [12].

Recall the notion of weights $\mathcal{C} = \{(\triangleleft, c) \mid c \in \overline{\mathbb{R}} \text{ and } \triangleleft \in \{\leq, <\}$. An order relation $<$ between weights is defined as $(\triangleleft, c) < (\triangleleft', c')$ when either (1) $c < c'$, or (2) $c = c'$ and $\triangleleft$ is $<$ while $\triangleleft'$ is $\leq$. Note that since $(<, -\infty) < (\leq, -\infty) < (\triangleleft, c) < (<, \infty) < (\leq, \infty)$ for all $c \in \mathbb{R}$, this relation is a total order and therefore min of a finite set of weights is well defined. We also use the commutative and associative sum operation on weights defined in [4]. If $c, c' \in \mathbb{R}$ are finite, the definition is as usual: $(\triangleleft, c) + (\triangleleft', c') = (\triangleleft'', c + c')$ where $\triangleleft'' = \leq$ if $\triangleleft = \triangleleft' = \leq$ and $\triangleleft'' = <$ otherwise. Infinite weights α, β from the list $(<, +\infty), (\leq, -\infty), (\leq, +\infty), (<, -\infty)$ are all 'absorbants' wrt. weaker weights: $\alpha + \beta = \beta + \alpha = \alpha$ if α is stronger than β (i.e., α is listed after β). Also, $\alpha + (\triangleleft, c) = \alpha$ if $c \in \mathbb{R}$ is finite.

A distance graph $\mathbb{G}$ is a weighted directed graph without self-loops, with vertex set $X \cup \{0\} = X_F \cup X_H \cup \{0\}$, and edges labeled with weights from $\mathcal{C} \setminus \{(<, -\infty)\}$. We define its semantics $[\![\mathbb{G}]\!] := \{v \in \mathbb{V} \mid v \models y - x \triangleleft c$ for all edges $x \xrightarrow{\triangleleft c} y$ in $\mathbb{G}\}$. The weight of edge $x \to y$ is denoted $\mathbb{G}_{xy}$ and we set $\mathbb{G}_{xy} = (\leq, \infty)$ if there is no edge $x \to y$. The weight of a path is the sum of the weights of its edges. A cycle in $\mathbb{G}$ is said to be negative if its weight is strictly less than $(\leq, 0)$.

In classical timed automata, the significance of distance graphs stems from the observation that a distance graph has no negative cycles iff its semantics is non-empty. This property does not immediately hold for distance graphs over the extended algebra [4, Section 4.2] However, we can convert a distance graph $\mathbb{G}$ (in time polynomial in number of clocks) into a *standard form* where this characterization continues to hold. First, we set $\mathbb{G}'_{0x} = \min(\mathbb{G}_{0x}, (\leq, 0))$ for $x \in X_F$ and $\mathbb{G}'_{x0} = \min(\mathbb{G}_{x0}, (\leq, 0))$ for $x \in X_H$. Moreover, if $x \in X_F$ then we set $\mathbb{G}'_{x0} = \min(\mathbb{G}_{x0}, (<, \infty))$ if $\mathbb{G}_{xy} \neq (\leq, \infty)$ for some $y \neq x$, otherwise we keep $\mathbb{G}'_{x0} = \mathbb{G}_{x0}$. Similarly, if $y \in X_H$ then we set $\mathbb{G}'_{0y} = \min(\mathbb{G}_{0y}, (<, \infty))$ if $\mathbb{G}_{xy} \neq (\leq, \infty)$ for some $x \neq y$, otherwise we keep $\mathbb{G}'_{0y} = \mathbb{G}_{0y}$. Finally, for $x, y \in X$ with $x \neq y$ we set $\mathbb{G}'_{xy} = \mathbb{G}_{xy}$. The graph $\mathbb{G}'$ constructed above is called the standardization of $\mathbb{G}$, it is equivalent to $\mathbb{G}$ (i.e., $[\![\mathbb{G}']\!] = [\![\mathbb{G}]\!]$) and it has a negative cycle iff its semantics $[\![\mathbb{G}']\!]$ is empty [4].

Now, suppose $\mathbb{G}'$ (in standard form) has no negative cycles, then we construct $\mathbb{G}''$ by replacing the weight of an edge $x \to y$ by the minimum of the weights of the paths from x to y in $\mathbb{G}'$. Such a $\mathbb{G}''$ is called the *normalization* of $\mathbb{G}'$ and has several useful properties.

Let Z be a nonempty zone. Writing the constraints in Z as a distance graph, followed by standardizing and normalizing it, results in *its canonical distance graph* $\mathbb{G}(Z)$: $[\![\mathbb{G}(Z)]\!] = Z$ and $\mathbb{G}(Z)$ is minimal among the standard graphs G with $[\![G]\!] = Z$. We denote by Z_{xy} the weight of the edge $x \to y$ in $\mathbb{G}(Z)$.

[3] contains the algorithms for the zone operations when there are no diagonal constraints. Successor computation can be done in $\mathcal{O}(|X|^2 \cdot |g|)$ and the simulation in $\mathcal{O}(|X|^2)$. Incorporating intersection with diagonal constraints requires an additional standardization step since diagonal constraints may break this property. A detailed explanation of the successor computation of zones is provided in

[2]. For the simulation, the algorithm from [26] is used. However, in the presence of diagonal constraints, the simulation check becomes NP-complete in general, and makes use of heuristics that allows for a faster check in practice. What remains is to show that $\preceq_{\mathcal{A}}$ is a finite simulation for X_D-safe GTA.

7 Finiteness of the Simulation Relation

In this section, we show that the simulation relation $\preceq_{\mathcal{A}}$ proposed in Sect. 5 is finite for safe GTA, which proves termination of the symbolic enumeration-based reachability algorithm. We do this in two parts: first, we show that the zones that are reached during the enumeration satisfy some invariants, in particular, only finitely many values occur in constraints among future clocks. This is however not necessarily true for history clocks. There the simulation comes into play. In the second part of the proof, we combine the invariants with an equivalence relation to show finiteness of the simulation. Below, we sketch these arguments and provide intuition leaving formal details to [2] due to lack of space.

Throughout this section, we fix an X_D-safe GTA $\mathcal{A}$. Let $M = \max\{|c| \mid c \in \mathbb{Z}$ is used in some constraint of $\mathcal{A}\}$, called the maximal constant of $\mathcal{A}$. We say that a zone Z is reachable if there is some reachable node (q, Z) in $\mathsf{GZG}(\mathcal{A})$.

Part 1: Invariants on zones. We start by showing an important property of reachable zones: closure under valuations that agree on the value of history clocks, and satisfy the same set of safe constraints involving non-history clocks.

We say that a constraint $x - y \triangleleft c$ is *M-bounded* if either $c \in \mathbb{R}$ is such that $|c| \leq M$ or $c \in \{-\infty; +\infty\}$. It is X_D-safe if $x, y \in X_F$ implies $x, y \in X_D$. We say that it is (X_D, M)-safe if it is both M-bounded and X_D-safe.

Lemma 8. *Let $v, v' \in \mathbb{V}$ be such that $v'{\downarrow_{X_H}} = v{\downarrow_{X_H}}$ and, for all (X_D, M)-safe constraints $y - x \triangleleft c$ with $x, y \in X_F \cup \{0\}$, we have $v' \models y - x \triangleleft c$ if and only if $v \models y - x \triangleleft c$. Let Z be a* reachable *zone. Then, $v \in Z$ if and only if $v' \in Z$.*

The proof (given in [2]) works by establishing that the property is true in the initial zone, and showing that it is invariant under the zone operations used to compute $\mathsf{GZG}(\mathcal{A})$. This proof crucially uses the fact that $\mathcal{A}$ is X_D-safe. For the case of releasing a clock $x \in X_F \setminus X_D$, we use the fact that a diagonal constraint involving x may not use another future clock. For the case of releasing a clock $x \in X_D$, we use the fact that the value of the clock must be 0 or $-\infty$ just before the release. As a non-example, consider Fig. 3. Here, $X_D = \{y, z\}$ and $M = 1$. After two iterations of a, the zone Z_2 reached is $x = 0 \wedge y = z = -2$. Pick $v : x = 0, y = z = -2$ and $v' : x = 0, y = z = -3$. Notice that both of them satisfy the same set of (X_D, M)-safe constraints, but $v \in Z_2$, $v' \notin Z_2$. Indeed, the automaton is not X_D-safe since y and z are released arbitrarily.

From Lemma 8, we get the following corollary (with a more precise statement and proof in [2]). Namely, if a reachable zone Z contains a valuation v in which the difference between two future clocks x, y (including the zero clock) is finite and large enough, then Z contains valuations where the difference between x and y is any finite and large enough value.

Corollary 9. *Let Z be a reachable zone and let $v \in Z$. Let $n = \max(1, |X_D|)$. For all $x, y \in X_F \cup \{0\}$, if $-\infty < v(x) - v(y) < -nM$ then, for every α with $-\infty < \alpha < -nM$, we have a valuation $v' \in Z$ with $v'(x) - v'(y) = \alpha$.*

Notice that the property above does not hold if we simply take $n = 1$. For instance, if we have two clocks $x, z \in X_D$ then, applying the (X_D, M)-safe program $\langle [x, z]; z = -M \wedge x - z = -M \rangle$ from $\mathbb{V}$ results in a zone Z where all valuations v satisfy $v(x) = -2M$. So the property fails with $n = 1$, x and $y = 0$. This is a noteworthy difference between models with and without diagonals.

Using Corollary 9, we can prove the main invariants satisfied by the zones obtained during the enumeration. Essentially, the weights of edges involving non-history clocks come from a finite set which depends on the number of future clocks in X_D and the maximum constant M of the automaton. This also induces an invariant on the constraint between a history clock and a future clock.

Before stating the result, we first give two technical lemmas from [4] that we use extensively in the proof.

Lemma 10 ([4]).

1. *Let $(\triangleleft, c)$ be a weight and $\alpha \in \overline{\mathbb{R}}$. Then,*
   - *$\alpha \triangleleft c$ iff $(\leq, \alpha) \leq (\triangleleft, c)$ iff $(\leq, 0) \leq (\leq, -\alpha) + (\triangleleft, c)$,*
   - *$\alpha \not\triangleleft c$ iff $(\triangleleft, c) < (\leq, \alpha)$ iff $(\leq, -\alpha) + (\triangleleft, c) < (\leq, 0)$ iff $(\leq, -\alpha) + (\triangleleft, c) \leq (<, 0)$.*
2. *Let $(\triangleleft, c), (\triangleleft', c'), (\triangleleft'', c'')$ be weights with $(\leq, 0) \leq (\triangleleft, c) + (\triangleleft', c')$. Then, there exists $\alpha \in \overline{\mathbb{R}}$ such that $\alpha \triangleleft c$ and $-\alpha \triangleleft' c'$. If in addition we have $(\triangleleft'', c'') < (\triangleleft, c)$ then there exists such an α with $\alpha \not\triangleleft'' c''$.*

Lemma 11 ([4]). *Let $\mathbb{G} = \mathbb{G}(Z)$ for a non-empty GTA zone Z, and let $x, y \in X \cup \{0\}$ be a pair of distinct nodes and $\alpha \in \overline{\mathbb{R}}$. There is a valuation $v \in [\![\mathbb{G}]\!]$ with $v(y) - v(x) = \alpha$ if and only if*

1. *$(\leq, \alpha) \leq \mathbb{G}_{xy}$ and $(\leq, -\alpha) \leq \mathbb{G}_{yx}$, and*
2. *if $x, y \in X$ and $\alpha \in \mathbb{R}$ is finite then the weights $\mathbb{G}_{x0}, \mathbb{G}_{0x}, \mathbb{G}_{y0}, \mathbb{G}_{0y}$ are all different from $(\leq, -\infty)$, and*
3. *if $x, y \in X$ and $\alpha = -\infty$ then $\mathbb{G}_{0x} \neq (\leq, -\infty) \neq \mathbb{G}_{y0}$.*

Lemma 12. *Let Z be a nonempty reachable zone. Let $n = \max(1, |X_D|)$. Then, the normalized distance graph $\mathbb{G}(Z)$ satisfies the following $(\dagger)$ conditions:*

- $\dagger_1$ *For all $x \in X_F$, $y \in X_H \cup \{0\}$, if Z_{xy} is finite, then $(\leq, 0) \leq Z_{x0} \leq (\leq, nM)$.*
- $\dagger_2$ *For all $x \in X_F$, if Z_{0x} is finite, then $(<, -nM) \leq Z_{0x} \leq (\leq, 0)$.*
- $\dagger_3$ *For all $x \in X_H$, $y \in X_F$, if Z_{0y} is finite, then $Z_{x0} + (<, -nM) \leq Z_{xy}$.*
- $\dagger_4$ *For $x, y \in X_F$, if Z_{xy} is finite, then $(<, -nM) \leq Z_{xy} \leq (\leq, nM)$.*

Proof. We focus on $\dagger_1, \dagger_2$, leaving the more complicated cases to [2].

- $\dagger_1$ First, we consider the case where $y = 0$. So we assume that $(\leq, 0) \leq Z_{x0} < (<, \infty)$ is finite. Towards a contradiction, suppose that $(\leq, nM) < Z_{x0} < (<, \infty)$. Since Z is non-empty, we know that $(\leq, 0) \leq Z_{x0} + Z_{0x}$. Then, using

Lemma 10, we can find $\alpha \in \overline{\mathbb{R}}$ such that $(\leq, \alpha) \leq Z_{x0}$, $(\leq, -\alpha) \leq Z_{0x}$, and $nM < \alpha$. Notice that $\alpha < \infty$ since $Z_{x0} < (<, \infty)$. Further, using Lemma 11, we can get a valuation $v \in Z$ such that $0 - v(x) = \alpha$. Since $nM < \alpha < \infty$, this implies $-\infty < v(x) < -nM$. Let $Z_{x0} = (\triangleleft, c)$. We have $nM < c < \infty$. Using Corollary 9, we can get a valuation $v' \in Z$, such that $-\infty < v'(x) < -c$, a contradiction as it violates the constraint $0 - x \triangleleft c$ of Z. Next, assume that $Z_{xy} < (<, \infty)$ for some $y \in X_H$. Since Z is normal, we have $Z_{x0} \leq Z_{xy} + Z_{y0} < (<, \infty)$ as $Z_{xy} < (<, \infty)$ and $Z_{y0} \leq (\leq, 0)$. We now conclude from the first case that $(\leq, 0) \leq Z_{x0} \leq (\leq, nM)$.

$\dagger_2$ We have to show that either $Z_{0x} = (\leq, -\infty)$ or $(<, -nM) \leq Z_{0x} \leq (\leq, 0)$. Let $Z_{0x} = (\triangleleft, c)$. Suppose $(\leq, -\infty) < Z_{0x} < (<, -nM)$. We have $-\infty < c < -nM$. As before, we can find α such that $(\leq, \alpha) \leq Z_{0x}$, $(\leq, -\alpha) \leq Z_{x0}$ and $\alpha \neq -\infty$. Then, by Lemma 11, we can find $v \in Z$ with $v(x) = \alpha$. We have $-\infty < v(x) \triangleleft c < -nM$. Now, using Corollary 9, we can get a valuation $v' \in Z$ such that $c < v'(x) < -nM$, which leads to a contradiction as it violates the constraint $x - 0 \triangleleft c$ in the zone.

□

Part 2. Equivalence and Finiteness. We introduce below an equivalence relation $\sim_M^n$ of *finite index* on valuations, depending on $n = \max(1, |X_D|)$ and the maximal constant M, and show that, if G is a set of atomic M-bounded integral constraints and if Z is a zone such that its canonical distance graph $\mathbb{G}(Z)$ satisfies ($\dagger$) conditions, then the downward closure $\downarrow_G Z = \{v \in \mathbb{V} \mid \exists v' \in Z \text{ with } v \preceq_G v'\}$ is a union of $\sim_M^n$ equivalence classes.

First, we define $\sim_M$ on $\alpha, \beta \in \overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$ by $\alpha \sim_M \beta$ if $(\alpha \triangleleft c \iff \beta \triangleleft c)$ for all $(\triangleleft, c)$ with $\triangleleft \in \{<, \leq\}$ and $c \in \{-\infty, \infty\} \cup \{d \in \mathbb{Z} \mid |d| \leq M\}$. In particular, if $\alpha \sim_M \beta$ then $(\alpha = -\infty \iff \beta = -\infty)$ and $(\alpha = \infty \iff \beta = \infty)$.

Next, for valuations $v_1, v_2 \in \mathbb{V}$, we define $v_1 \sim_M^n v_2$ by two conditions: $v_1(x) \sim_{nM} v_2(x)$ and $v_1(x) - v_1(y) \sim_{(n+1)M} v_2(x) - v_2(y)$ for all clocks $x, y \in X$. Notice that we use $(n+1)M$ for differences of values. Clearly, $\sim_M^n$ is an equivalence relation of finite index on valuations. Using this, we can show that the zones that are reachable in a safe GTA are unions of $\sim_M^n$-equivalence classes.

Lemma 13. *Let G be a set of X_D-safe M-bounded integral constraints which contains both $x \leq 0$ and $0 \leq x$ for each future clock $x \in X_F$. Let Z be a zone with a canonical distance graph $\mathbb{G}(Z)$ satisfying the ($\dagger$) conditions of Lemma 12. Let $v_1, v_2 \in \mathbb{V}$ be valuations with $v_1 \sim_M^n v_2$. Then, $v_1 \in \downarrow_G Z$ iff $v_2 \in \downarrow_G Z$.*

Finally, from Lemmas 12 and 13, we obtain our main theorem of the section.

Theorem 14. *The simulation relation $\preceq_{\mathcal{A}}$ is finite if $\mathcal{A}$ is safe.*

Proof. Let $(q, Z_0), (q, Z_1), (q, Z_2), \ldots$ be an infinite sequence of *reachable* nodes in the zone graph of $\mathcal{A}$. By Lemma 12, for all i, the distance graph $\mathbb{G}(Z_i)$ in canonical form satisfies conditions ($\dagger$).

The set $\mathcal{G}(q)$ contains only X_D-safe and M-bounded integral constraints. Let G be $\mathcal{G}(q)$ together with the constraints $x \leq 0$ and $0 \leq x$ for each future clock

Table 1. Experimental results obtained by running our prototype implementation and, when possible, the standard reachability algorithm using $\mathcal{G}$-simulation implemented in TCHECKER. Both implementations use a breadth-first search with simulation. For each model, we give the parameters in parenthesis - for ToyECA, we explain the parameterization in [2], while for others, we report the number of concurrent processes. All experiments were run on an Ubuntu machine with an Intel-i5 7th Generation processor and 8 GB RAM, and timeout set to 60 s.

Sl. No.	Models	$\mathcal{G}$-Sim			GTA Reach		
		Visited nodes	Stored nodes	Time in sec	Visited nodes	Stored nodes	Time in sec
1	Dining Phi. (6)	5480	5480	4.911	5480	5480	6.410
2	FDDI (10)	10219	459	10.139	10219	459	16.797
3	Fischer (10)	447598	260998	29.1574	447598	260998	34.6517
4	ToyECA(10000, 4)	150049	49	4.22	3	3	0.0003
5	ToyECA(5000, 6)	315193	193	15.572	3	3	0.0006
6	ToyECA(1000, 100)	TIMEOUT			3	3	0.877
7	ToyECA(50000, 120)	TIMEOUT			3	3	1.52
8	Fire-alarm-pattern(5)	_			46	46	0.027
9	CSMACD-bounded(1)	_			34	26	0.0054
10	CSMACD-bounded(4)	_			4529	2068	2.597
11	ABP-prop1(1)	_			114	114	0.038
12	ABP-prop2(1)	_			168	168	0.026

$x \in X_F$. From Lemma 13 we deduce that for all i, $\downarrow_G Z_i$ is a union of $\sim_M^n$-classes. Since $\sim_M^n$ is of finite index, there are only finitely many unions of $\sim_M^n$-classes. Therefore, we find $i < j$ with $\downarrow_G Z_i = \downarrow_G Z_j$, which implies $Z_j \preceq_G Z_i$. Since $\mathcal{G}(q) \subseteq G$, this also implies $Z_j \preceq_{\mathcal{G}(q)} Z_i$. ☐

8 Experimental Evaluation

We have implemented a prototype that takes as input a GTA, as given in Definition 1, and applies our reachability algorithm, in the open source tool TCHECKER [29]. To do so, we extend TCHECKER to allow clocks to be declared as one of *normal*, *history*, *prophecy*, or *timer*, and extend the syntax of edges to allow arbitrary interleaving of guards and clock changes (reset/release). Our tool, along with the benchmarks used in this paper, is available and can be downloaded from https://github.com/EQuaVe/GTAReach. We present selected results in Table 1, with further details in [2].

First, we consider timed automata models from standard benchmarks [21,34,39]. Despite the overhead induced by our framework (e.g., maintaining general programs on transitions), we are only slightly worse off wrt. running

time than the standard algorithm, while visiting and storing the same number of nodes. We illustrate this in rows 1–3 of Table 1 by providing a comparison of our tool with the implementation of the state-of-the-art zone-based reachability algorithm using $\mathcal{G}$-simulation introduced in [24–26].

Next, we consider models belonging to the class of ECA without diagonal constraints. We remark that ours is the first implementation of a reachability algorithm that can operate on the whole class of ECA directly. We compare against an implementation that first translates the ECA into a timed automaton using the translation proposed in [10], and then runs the state-of-the-art reachability algorithm of [24–26] on this timed automaton. From rows 4–7 of Table 1, we observe significant improvements, both in terms of running time as well as number of visited nodes and stored nodes w.r.t. the standard approach.

Finally, in Rows 8–12, we consider the unified model GTA. As already pointed out, model-checking an event-clock specification φ over a timed automaton model $\mathcal{A}$ can be reduced to the reachability on the product of the TA $\mathcal{A}$ and the ECA representing $\neg\varphi$. In this spirit, our implementation allows the model to use any combination of *normal* clocks, *history* clocks, *prophecy* clocks or *timers* and moreover, permits diagonal guards between any of these clocks. To the best of our knowledge, no existing tool allows all these features. We emphasize this by the $-$ in the $\mathcal{G}$-Sim column of Table 1.

We model simple but useful properties using event-clocks, and check these properties on some standard models from literature such as CSMACD [39], Fire-alarm [35] and Alternating-bit-protocol(ABP) [33]. Note that for the benchmark Fire-alarm-pattern, the specification is modelled using an ECA with diagonals. As a consequence, the product automaton that we check reachability on contains normal clocks and event-clocks. Here, we consider the following ECA specification: no three a's occur within k time units. The negation of this property can be easily modeled by an ECA with two states and a transition on a with the diagonal constraint $\overleftarrow{a} - \overrightarrow{a} \leq k$, where $\overleftarrow{a}$ is the history clock recording time since the previous occurrence of a, and $\overrightarrow{a}$ is a future clock predicting the time to the next a occurrence. When reading an a, the quantity $\overleftarrow{a} - \overrightarrow{a}$ gives the distance between the next and the previous occurrence. This language is used in [19] to observe that ECA with diagonals are more expressive than ECA. Finally, we remark that the model of ABP contains timers. A more detailed discussion of the model and specifications in these benchmarks is provided in [2].

In conclusion, as can be seen from the experimental results in Table 1, we are able to demonstrate the full power of our reachability algorithm for the unified model of generalized timed automata.

9 Conclusion

The success of timed automata verification can safely be attributed to the advances in the zone-based technology over the last three decades. In fact, [22], the precursor to the seminal works [8,9], already laid the foundations for zones by describing the Difference-Bounds-Matrices (DBM) data structure. Our goal

in this work has been to unify timing features defined in different timed models, while at the same time retain the ability to use efficient state-of-the-art algorithms for reachability. To do so, we have equipped the model with two kinds of clocks, history and future, and modified the transitions to contain a program that alternates between a guard and a change to the variables. For the algorithmic part, we have adapted the $\mathcal{G}$-simulation framework to this powerful model. The main challenge was to show finiteness of the simulation in this extended setting. To aid the practical use of this generic model, we have developed a prototype implementation that can answer reachability for GTA. We remark that decidability for GTA comes via zones, and not through regions. In fact, since we generalize event-clock automata, we do not have a finite region equivalence for GTA [28].

We conclude with some interesting avenues for future work. An immediate future work is to use generalized timed automata for model-checking timed specifications over real-time systems. Further, the complexity and expressivity of safe GTA are natural intersting theoretical open questions, but we believe they are not obvious. Both these questions are answered in the timed automata literature using regions. However, we cannot have a region equivalence for our model, since even for the subclass of ECA, it was shown that no finite bisimulation is possible. In particular, it would be interesting to investigate if is possible to have a translation from safe GTA to timed automata. Note that even if such a translation exists, it is likely to incur an exponential blowup since even the translation from ECA to TA costs an exponential. Coming to the complexity of the reachability problem for safe GTA, it is easy to see that our procedure runs in EXPSPACE, as we have shown that each reachable zone is a union of equivalence classes of a finite index (see Lemma 13). On the other hand, PSPACE-hardness is inherited from timed automata [6,8]. Closing the complexity gap is open. We note that even in timed automata, the precise complexity of the simulation based reachability algorithm is difficult to analyze, but its selling point is that it works well in practice. Finally, we would also like to investigate liveness verification for GTA, in particular what future clocks bring us when we consider the setting of ω-words.

References

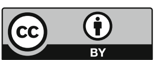
1. Akshay, S., Bollig, B., Gastin, P.: Event clock message passing automata: a logical characterization and an emptiness checking algorithm. Formal Methods Syst. Des. **42**(3), 262–300 (2013)
2. Akshay, S., Gastin, P., Govind, R., Joshi, A.R., Srivathsan, B.: A unified model for real-time systems: Symbolic techniques and implementation. CoRR abs/2305.17824 (2023)
3. Akshay, S., Gastin, P., Govind, R., Srivathsan, B.: Simulations for event-clock automata. In: CONCUR. LIPIcs, vol. 243, pp. 13:1–13:18 (2022)
4. Akshay, S., Gastin, P., Govind, R., Srivathsan, B.: Simulations for event-clock automata. CoRR abs/2207.02633 (2022)

5. Akshay, S., Gastin, P., Prakash, K.R.: Fast zone-based algorithms for reachability in pushdown timed automata. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 619–642. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_30
6. Alur, R.: Techniques for automatic verification of real-time systems. Ph.D. thesis, Stanford University (1991)
7. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Hybrid Systems, pp. 209–229 (1992)
8. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Paterson, M.S. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990). https://doi.org/10.1007/BFb0032042
9. Alur, R., Dill, D.L.: A theory of timed automata. Theoret. Comput. Sci. **126**, 183–235 (1994)
10. Alur, R., Fix, L., Henzinger, T.A.: Event-clock automata: a determinizable class of timed automata. Theor. Comput. Sci. **211**(1–2), 253–273 (1999)
11. de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G.: Real-Time: Theory in Practice: REX Workshop, Mook, The Netherlands. Proceedings, vol. 600 (1992)
12. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
13. Bernstein, A.J., Jr., P.K.H.: Proving real-time properties of programs with temporal logic. In: SOSP, pp. 1–11. ACM (1981)
14. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods Syst. Des. **24**(3), 281–320 (2004)
15. Bouyer, P., Chevalier, F.: On conciseness of extensions of timed automata. J. Autom. Lang. Comb. **10**(4), 393–405 (2005)
16. Bouyer, P., Colange, M., Markey, N.: Symbolic optimal reachability in weighted timed automata. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 513–530. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_28
17. Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Updatable timed automata. Theor. Comput. Sci. **321**(2–3), 291–345 (2004)
18. Bouyer, P., Gastin, P., Herbreteau, F., Sankur, O., Srivathsan, B.: Zone-based verification of timed automata: Extrapolations, simulations and what next? In: FORMATS. LNCS, vol. 13465, pp. 16–42. Springer (2022). https://doi.org/10.1007/978-3-031-15839-1_2
19. Bozzelli, L., Montanari, A., Peron, A.: Taming the complexity of timeline-based planning over dense temporal domains. In: FSTTCS. LIPIcs, vol. 150, pp. 34:1–34:14 (2019)
20. Bozzelli, L., Montanari, A., Peron, A.: Complexity issues for timeline-based planning over dense time under future and minimal semantics. Theor. Comput. Sci. **901**, 87–113 (2022)
21. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The tool Kronos. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 208–219. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0020947
22. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_17

23. D'Souza, D., Tabareau, N.: On timed automata with input-determined guards. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 68–83. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_7
24. Gastin, P., Mukherjee, S., Srivathsan, B.: Reachability in timed automata with diagonal constraints. In: CONCUR. LIPIcs, vol. 118, pp. 28:1–28:17 (2018)
25. Gastin, P., Mukherjee, S., Srivathsan, B.: Fast algorithms for handling diagonal constraints in timed automata. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 41–59. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_3
26. Gastin, P., Mukherjee, S., Srivathsan, B.: Reachability for updatable timed automata made faster and more effective. In: FSTTCS. LIPIcs, vol. 182, pp. 47:1–47:17 (2020)
27. Geeraerts, G., Raskin, J.-F., Sznajder, N.: Event clock automata: from theory to practice. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 209–224. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24310-3_15
28. Geeraerts, G., Raskin, J.-F., Sznajder, N.: On regions and zones for event-clock automata. Formal Methods Syst Design **45**(3), 330–380 (2014). https://doi.org/10.1007/s10703-014-0212-1
29. Herbreteau, F., Point, G.: TChecker. https://github.com/fredher/tchecker (v02 - April 2019)
30. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC '99) (1999)
31. Jonsson, B., Vaandrager, F.: Learning mealy machines with timers. Tech. rep. (2018). https://sws.cs.ru.nl/publications/papers/fvaan/MMT/
32. Koymans, R., Vytopil, J., de Roever, W.P.: Real-time programming and asynchronous message passing. In: PODC, pp. 187–197. ACM (1983)
33. Kurose, J.F., Ross, K.W.: Computer networking - a top-down approach featuring the internet. Addison-Wesley-Longman (2001)
34. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. Theor. Comput. Sci. **345**(1), 27–59 (2005)
35. Muñiz, M., Westphal, B., Podelski, A.: Timed automata with disjoint activity. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 188–203. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33365-1_14
36. Raskin, J., Schobbens, P.: The logic of event clocks - decidability, complexity and expressiveness. J. Autom. Lang. Comb. **4**(3), 247–282 (1999)
37. Sorea, M.: Tempo: A model checker for event-recording automata. Tech. rep., In: Proceedings of RT-Tools'01 (2001)
38. Srivathsan, B.: Reachability in timed automata. ACM SIGLOG News **9**(3), 6–28 (2022)
39. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstracting bisimulations. Formal Methods Syst. Des. **18**(1), 25–68 (2001)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Closed-Loop Analysis of Vision-Based Autonomous Systems: A Case Study

Corina S. Păsăreanu[1,2](✉), Ravi Mangal[2], Divya Gopinath[1], Sinem Getir Yaman[3], Calum Imrie[3], Radu Calinescu[3], and Huafeng Yu[4]

[1] KBR, NASA Ames, Moffett Field, CA 94035, USA
pcorina@andrew.cmu.edu
[2] Carnegie Mellon University, Moffett Field, CA 94035, USA
[3] University of York, York, UK
[4] Boeing Research and Technology, Santa Clara, CA, USA

Abstract. Deep neural networks (DNNs) are increasingly used in safety-critical autonomous systems as perception components processing high-dimensional image data. Formal analysis of these systems is particularly challenging due to the complexity of the perception DNNs, the sensors (cameras), and the environment conditions. We present a case study applying formal probabilistic analysis techniques to an experimental autonomous system that guides airplanes on taxiways using a perception DNN. We address the above challenges by replacing the camera and the network with a compact abstraction whose transition probabilities are computed from the confusion matrices measuring the performance of the DNN on a representative image data set. As the probabilities are estimated based on empirical data, and thus are subject to error, we also compute confidence intervals in addition to point estimates for these probabilities and thereby strengthen the soundness of the analysis. We also show how to leverage local, DNN-specific analyses as run-time guards to filter out mis-behaving inputs and increase the safety of the overall system. Our findings are applicable to other autonomous systems that use complex DNNs for perception.

1 Introduction

Complex autonomous systems, such as autonomous aircraft taxiing systems [31] and autonomous cars [20,25,42], need to perceive and reason about their environments using high-dimensional data streams (such as images) generated by rich sensors (such as cameras). Machine learnt components, specially deep neural networks (DNNs), are particularly capable of the required high-dimensional reasoning and hence, are increasingly used for perception in these systems. While formal analysis of the safety of these systems is highly desirable due to their safety-critical operational settings and the error-prone nature of learned components, in practice this is very challenging because of the complexity of the system components, including the high complexity of the neural networks (which may have thousands or millions of parameters), the complexity of the camera capture

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 289–303, 2023.
https://doi.org/10.1007/978-3-031-37706-8_15

process, and the random and hard to characterize nature of the environment in which the system operates (i.e., the world itself).

In this work, we describe a formal analysis of a closed-loop autonomous system that addresses the above challenges. Our case study is motivated by a real-world application, namely, an experimental autonomous system for guiding airplanes on taxiways developed by Boeing [3,14]. The key idea is to abstract away altogether the perception components, namely, the perception network and the image generator, i.e., the camera taking images of the world, and replace them with a probabilistic component α that maps (abstractions of) the state of the system to state estimates that are used in downstream decision making in the closed-loop system. The resulting system can then be analyzed with standard (probabilistic) model checkers, such as PRISM [34] or Storm [22].

The approach is *compositional*, in the sense that the probabilistic component is computed separately from the rest of the system. The transition probabilities in α are derived based on *confusion matrices* computed for the DNN (measured on representative data sets). Developers routinely use confusion matrices to evaluate machine learning models, so our analysis is closely aligned with existing workflows, facilitating its adoption in practice.

The size of the probabilistic abstraction α is linear in the size of the output of the DNN, and is independent of the number of the DNN parameters or the complexity of the camera and the environment. We also describe how to leverage additional results obtained from analyzing the DNN in isolation to further refine the abstraction and also increase the safety of the closed-loop system through *run-time guards*. In particular, we leverage rules mined from the DNN model [17] to act as run-time guards for the closed-loop analysis, filtering out inputs that likely lead to invalid DNN behavior. Other methods can also be used (e.g. [17,18,21,26,32,35]) to catch adversarial or out-of-distribution inputs.

The probabilities in α are estimated based on empirical data, so they are subject to error. We explore the use of *confidence intervals* in addition to point estimates for these probabilities and thereby strengthen the soundness of the analysis [5,7]. Our technique is applicable to other autonomous systems that use DNN-based perception from high-dimensional data.

Related Work. Formal proofs of closed-loop safety have been obtained for systems with low-dimensional sensor readings [11,12,27–30,40]; however, they become intractable for systems that use rich sensors producing high-dimensional inputs such as images.

Other works address the modeling and scalability challenges by constructing *abstractions* of the perception components [24,33]. To model different environment conditions, these abstract models use *non-deterministic* transitions. The resulting closed-loop systems are analyzed with traditional (non-probabilistic) techniques. The abstractions either lack soundness proofs [33] or come with only probabilistic soundness guarantees [24] which do not translate into probabilistic guarantees over the safety of the overall system. VerifAI [16] can find counter-examples to system safety, but can not provide guarantees.

The recent work in [36] aims to verify the safety of the trajectories of a camera-based autonomous vehicle in a given 3D-scene. The work use invariant

regions over the input space grouped based on the same controller action. However, their abstraction captures only one environment condition (i.e., one scene) and one camera model, whereas our approach is not particular to a camera model and implicitly considers all the possible environment conditions.

In contrast to previous work, we describe a formal analysis that is *probabilistic*, which we believe is natural since the camera images capturing the state of the world are subject to randomness due to the environment; further DNNs are learnt from data and are not guaranteed to be 100% accurate. Recent work [2] also discusses the use of classification metrics, such as confusion matrices, for quantitative system-level analysis with temporal logic specifications. However, the work does not discuss the computation of confidence intervals that is necessary for quantifying the empirical results. Also, it does not incorporate DNN specific analyses as we do here. We build on our previous work DeepDECS [6], where the goal is to perform controller synthesis with safety guarantees, so the formalism is more involved. Furthermore, DeepDECS does not consider confidence interval analysis, which we explore here based on some of our other previous works [5,7]. We analyzed center-line tracking using TaxiNet in [31]. That work focuses on the analysis of the network and not on the overall system.

2 Autonomous Center-Line Tracking with TaxiNet

Boeing is developing an experimental autonomous system for center-line tracking on taxiways in an airport. The system uses a neural network called TaxiNet for perception. TaxiNet is designed to take a picture of the taxiway as input and return the plane's position with respect to the center-line on the taxiway. It returns two outputs; cross track error (`cte`), which is the distance in meters of the plane from the center-line and heading error (`he`), which is the angle in degrees of the plane with respect to the center-line. These outputs are fed to a controller which in turn manoeuvres the plane such that it remains close to the center of the taxiway. This forms a closed-loop system where the perception network continuously receives images as the plane moves on the taxiway. We use this system as a case study and also as a running example throughout the paper.

System Decomposition. The decomposition of this system is illustrated in Fig. 1. The controller sends actions a to the airplane to guide it on the taxiway. The dynamics (which models the movement of the airplane on the airport surface) maps previous state s and action a to the next state s'.[1] Information about the taxiway is provided by the perception network (p), i.e. TaxiNet. The perception network takes high-dimensional images captured with a *camera* (c), and returns its estimation s_{est} of the real state s.

For our application, state $s \in S$ captures the position of the airplane on the surface; S is modeled as $\mathtt{CTE} \times \mathtt{HE}$. The network estimates the state $s := (\mathtt{cte}, \mathtt{he})$ based on images taken with a camera placed on the airplane. If the network is 'perfect', then $s = s_{est}$.[2] However, this does not hold in practice.

[1] Velocity may be provided as feedback to the controller; we ignore here for simplicity.
[2] Assuming the relevant state of the system is recoverable from the input image.

The network is trained on a finite set of images and is not guaranteed to be 100% accurate whereas images observed in operation show a wide variety due to different environment (e.g., light, weather) conditions and imperfections in the camera.

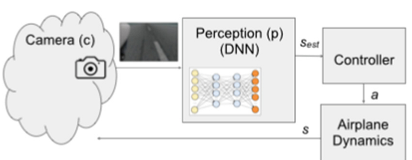


Fig. 1. Closed-loop System

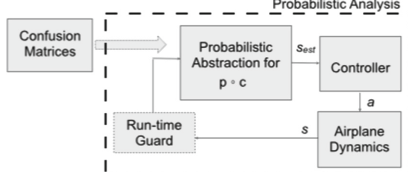


Fig. 2. Abstracted System

Component Modeling. We built a simple discrete model of the airplane dynamics and a discrete-time controller for the system, similar to previous related work [4,23] which also considers discretized control. Since the controller is discretized, we abstract the regression outputs of TaxiNet to view the model as a classifier which predicts the plane's position in discrete states. Treatment of more complex systems with continuous semantics and regression models is left for future work. The main challenge that we address in the paper is the modeling of the perception components (the camera capture process and the network), which we describe in detail in the next section. We model the (abstracted) autonomous system as a Discrete Time Markov Chain (DTMC) [38]; the code for the models is provided in the appendix of an extended version of this paper [37].

Safety Properties. In our study, the goal is to provide *guarantees* for safe behavior with respect to two system-level properties indicated by our industrial partner. The properties specify conditions for safe operation in terms of allowed `cte` and `he` values for the airplane, by using taxiway dimensions. The first property states that the airplane shall never leave the taxiway (i.e., $|\texttt{cte}| \leq 8$ meters). The second property states that the airplane shall never turn more than a prescribed degree (i.e., $|\texttt{he}| \leq 35°$), as it would be difficult to maneuver the airplane from that position. These two properties can be encoded in PCTL [8] as follows.

$$P =?[F(|\texttt{cte}| > 8\,m)] \qquad (\textit{Property 1})$$

$$P =?[F(|\texttt{he}| > 35°)] \qquad (\textit{Property 2})$$

Here $P =?$ indicates that we want to calculate the probability that eventually (F) the system reaches an error state.

TaxiNet DNN. This is a regression model with 24 layers including five convolution layers, and three dense layers (with 100/50/10 ELU neurons) before the output layer. The inputs to the model are RGB color images of size 360 × 200 pixels. We use a representative data set with 11108 images, shared by our industry partner. The model has a Mean Absolute Error (MAE) of 1.185 for

$\texttt{cte}$ and 7.86 for $\texttt{he}$ outputs respectively. The discrete nature of the controller in our DTMCs induces a discretization on TaxiNet's outputs and the treatment of TaxiNet as a classifier for the purpose of our analysis. $\texttt{cte} \in [-8.0\text{ m}, 8.0\text{ m}]$ and $\texttt{he} \in [-35.0°, 35.0°]$ are translated into $\underline{\texttt{cte}} \in \{0, 1, 2, 3, 4\}$ and $\underline{\texttt{he}} \in \{0, 1, 2\}$ as shown below.

$$\underline{\texttt{cte}} = \begin{cases} 3 \text{ if } -8.0\text{ m} <= \texttt{cte} < -4.8\text{ m} \\ 1 \text{ if } -4.8\text{ m} <= \texttt{cte} < -1.6\text{ m} \\ 0 \text{ if } -1.6\text{ m} <= \texttt{cte} <= 1.6\text{ m} \\ 2 \text{ if } 1.6\text{ m} < \texttt{cte} <= 4.8\text{ m} \\ 4 \text{ if } 4.8\text{ m} < \texttt{cte} <= 8.0\text{ m} \end{cases} \qquad \underline{\texttt{he}} = \begin{cases} 1 \text{ if } -35.0° <= \texttt{he} < -11.67° \\ 0 \text{ if } -11.67° <= \texttt{he} <= 11.66° \\ 2 \text{ if } 11.66° < \texttt{he} <= 35.0° \end{cases}$$

We use label "-1" to denote error states, i.e., $\underline{\texttt{cte}} = -1$ iff $|\texttt{cte}| > 8$ m and $\underline{\texttt{he}} = -1$ iff $|\texttt{he}| > 35°$. For simplicity, we use $\texttt{cte}$ and $\texttt{he}$ to denote both the classifier and regression outputs in other parts of the paper (with meaning clear from context). Note that none of the input images are labeled by the classifier as "-1", as the outputs of the network are normalized to be within the prescribed bounds; however, this does not preclude the system from reaching an error.

3 Probabilistic Analysis

In this section, we describe the methodology for abstracting and analyzing an autonomous system leveraging probabilistic model checking. The main idea, which we initially explored in [6], is to replace the composition $p \circ c$ of the camera (denoted as c) and the perception DNN (denoted as p) with a conservative abstraction mapping each system state to every possible estimated state; the transition probabilities are derived empirically based on the confusion matrices computed for the DNN, on a representative data set. We denote this abstraction as $\alpha : S \rightarrow \mathcal{D}(S)$, mapping system states to a discrete distribution over (estimated) system states. Figure 2 depicts the abstracted autonomous system.

We observe that c can be viewed as a map between state $s \in S$ to a distribution over images, denoted as $\mathcal{D}(\texttt{Img})$, where $\texttt{img} \in \texttt{Img}$ and $\texttt{Img}$ is the set of images. For instance, in the TaxiNet system, state s only captures the position of the airplane with respect to the center-line, but there are many different images that correspond to the same position. This is due to uncontrollable environmental conditions, such as temporary sensor failures or different lighting and weather conditions. Consequently, a single state s can map to a number of different images depending on the environment, and this is modeled by considering c to be a probabilistic map of type $S \rightarrow \mathcal{D}(\texttt{Img})$. Given a system state s, $\alpha(s)$ models the probability of $p \circ c$ leading to a particular estimated state s_{est}; α needs to be probabilistic because c itself is probabilistic and p is not perfectly accurate.

We further describe how we can leverage DNN-specific analysis to improve the accuracy of perception and the safety of the overall system, via the optional addition of run-time guards. For the verification of the closed-loop system, we use the PRISM model checking tool [34]. We also explore methods for analysis

of DTMCs with uncertain transition probabilities [5,7], to obtain *probabilistic guarantees* about the validity of our probabilistic safety proofs even though the abstraction probabilities are empirical estimates.

Assumptions. Our analysis assumes that the distribution of inputs to the network remains fixed over time (i.e., it is not subject to distribution shifts). Moreover, the data set of input images used to estimate the probabilities in α is assumed to be *representative*, i.e., constituted of independently drawn samples from this fixed underlying distribution of inputs. Relaxing these assumptions is a challenging but important task for future research.

3.1 Probabilistic Abstractions for Perception

We describe in detail the construction of the probabilistic abstraction $\alpha : S \to \mathcal{D}(S)$. We do not need access to the camera and only require black-box access to the network for constructing our abstraction.[3] We assume S is a finite set such that $\#S = K$ where $\#S$ denotes the cardinality of set S. We use $\alpha(s, s_{est})$ to represent the probability associated with estimated state s_{est}. It is defined as,

$$\alpha(s, s_{est}) := \Pr_{\mathtt{img} \sim c(s)} [p(\mathtt{img}) = s_{est}] \tag{1}$$

We estimate the probabilities in α by means of a confusion matrix. Let $\overline{\mathtt{Img}}_s \subseteq \mathtt{Img}$ denote a *representative test dataset* for images corresponding to state s, i.e., every sample in $\overline{\mathtt{Img}}_s$ is assumed to be an independently drawn sample from $c(s)$. We assume access to representative test datasets corresponding to every state $s \in S$. Let $\overline{\mathtt{Img}} := \bigcup_{s \in S} \overline{\mathtt{Img}}_s$. For any test input $\mathtt{img} \in \overline{\mathtt{Img}}$, let $p^*(\mathtt{img}) \in S$ be the label (i.e., the true underlying state) of $\mathtt{img}$, which is known since $\overline{\mathtt{Img}}$ is a test dataset. For the sake of technical presentation, we assume a bijective map $\mathtt{rep} : S \to [K]$ that maps every state in S to a number in $[K] := \{1, 2, \ldots, K\}$. We evaluate p on the test dataset $\overline{\mathtt{Img}}$ to construct a $K \times K$ confusion matrix $\mathcal{C}$ such that, for any $k, k' \in [K]$, the element in row k and column k' of this matrix is given by the number of inputs from $\overline{\mathtt{Img}}$ with true state $\mathtt{rep}^{-1}(k)$ that the perception network p classifies as state $\mathtt{rep}^{-1}(k')$.

$$\mathcal{C}[k, k'] := \# \left\{ \mathtt{img} \in \overline{\mathtt{Img}} \mid p^*(\mathtt{img}) = \mathtt{rep}^{-1}(k) \wedge p(\mathtt{img}) = \mathtt{rep}^{-1}(k') \right\} \tag{2}$$

Given the confusion matrix $\mathcal{C}$, empirical estimates for the probabilities in α are calculated as follows,

$$\alpha(\mathtt{rep}^{-1}(k), \mathtt{rep}^{-1}(k')) := \frac{\mathcal{C}[k, k']}{\sum_{k'' \in [K]} \mathcal{C}[k, k'']}. \tag{3}$$

[3] Our run-time guard does require white-box access.

TaxiNet Example. For the TaxiNet application, we construct two probabilistic maps, $\alpha_{\texttt{cte}}$ and $\alpha_{\texttt{he}}$, corresponding to each of the state variables `cte` and `he`, using a representative test data set with 11108 samples.[4] Thus, $\alpha_{\texttt{cte}}$ is of type $\texttt{CTE} \rightarrow \mathcal{D}(\texttt{CTE})$ and $\alpha_{\texttt{he}}$ is of type $\texttt{HE} \rightarrow \mathcal{D}(\texttt{HE})$. Table 1 illustrates the confusion matrix for `he`. The mapping $\alpha_{\texttt{he}}$ is computed in a straightforward way: $\alpha_{\texttt{he}}(0,0) = 4748/(4748 + 2139 + 148) = 0.675$, giving the probability of estimating correctly that the value of `he` is zero. Similarly, $\alpha_{\texttt{he}}(1,0) = 91/(91 + 2010) = 0.043$, giving the probability of estimating incorrectly that the value of `he` is zero instead of one. The corresponding DTMC code is as follows:

		Predicted		
	Total = 11108	**0**	**1**	**2**
Actual	**0**	4748	2139	148
	1	91	2010	0
	2	744	211	1017

Table 1. Confusion Matrix for `he`

```
[] he=0 → 0.675: (he_est'=0) + 0.304: (he_est'=1) + 0.021: (he_est'=2);
[] he=1 → 0.043: (he_est'=0) + 0.957: (he_est'=1) + 0.0: (he_est'=2);
[] he=2 → 0.377: (he_est'=0) + 0.107: (he_est'=1) + 0.516: (he_est'=2);
```

A similar computation is performed for constructing $\alpha_{\texttt{cte}}$. The resulting code for the closed-loop system is shown in [37], in the appendix.

3.2 DNN Checks as Run-Time Guards

We use DNN-specific checks as run-time guards to improve the performance of the perception network and therefore the safety of the overall system. We hypothesize that for inputs where the checks pass, the network is more likely to be accurate, and therefore, the system is safer.

For our case study, we distill logical rules from the DNN that characterize misbehavior in terms of intermediate neuron values and use them as run-time guards (as described in Sect. 4). More generally, one can use any off-the-shelf pointwise DNN check, such as local robustness [10,15,19,35,39,41] or confidence checks for well-calibrated networks [21], as run-time guards (provided that they are fast enough to be deployed in practice). For practical reasons (TaxiNet is a regression model, it contains ELU [9] activations, we do not have access to the training data) we can not use off-the-shelf checks here.

Modeling DNN Checks. Let us denote the application of (one or more) DNN-specific checks as a function $\texttt{check} : (\texttt{Img} \rightarrow S) \times \texttt{Img} \rightarrow \mathbb{B}$, such that, for perception network $p \in \texttt{Img} \rightarrow S$ and image $\texttt{img} \in \texttt{Img}$, $\texttt{check}(p, \texttt{img}) = \mathsf{true}$ if p passes the checks at input `img`, and $\texttt{check}(p, \texttt{img}) = \mathsf{false}$ otherwise.

We further assume that a system that uses DNN checks as a run-time guard attempts to read the camera sensor multiple (one or more) times, until the check passes; and aborts (or goes to a fail-safe state) if the number of consecutive failed checks reaches a certain threshold. This logic can be generalized to consider more sophisticated safe-mode operations; for instance, the system can decelerate

[4] To simplify the DTMCs, we model the updates to `cte` and `he` as independent. For more precision, we can compute confusion matrices and α for the pair $(\texttt{cte}, \texttt{he})$.

and/or notify an operator when the threshold is reached, as this could indicate serious sensor failure or adverse weather conditions.

To model the effect of the run-time check in our analysis, we can define β as the probability that an image img generated by the camera c, for *any* state s, satisfies $\texttt{check}(p, \texttt{img}) = \mathsf{true}$;

$$\beta := \Pr_{\texttt{img} \sim D}[\texttt{check}(p, \texttt{img}) = \mathsf{true}] \tag{4}$$

Here D is the distribution obtained by *combining* $c(s)$ for all states $s \in S$.[5] To be more precise we can define a separate β_s for each state s. We estimate β using the representative set of images $\overline{\texttt{Img}}$,

$$\beta := \frac{\#\overline{\texttt{Img}}^{true}}{\#\overline{\texttt{Img}}} \tag{5}$$

where $\overline{\texttt{Img}}^{true} := \{\texttt{img} \in \overline{\texttt{Img}} \mid \texttt{check}(p, \texttt{img}) = \mathsf{true}\}$.

For the overall analysis of the closed-loop system, irrespective of the state s, we can assume that the DNN check will pass with a probability β. Moreover, since the perception network only processes images that pass the DNN check, we construct a refined probabilistic abstraction α^{true} using conditional probability:

$$\alpha^{true}(s, s_{est}) := \Pr_{\texttt{img} \sim c(s)}[p(\texttt{img}) = s_{est} | \texttt{check}(p, \texttt{img}) = \mathsf{true}] \tag{6}$$

We can estimate α^{true} as before, but the confusion matrix is built using only the images that pass the DNN check, i.e., for dataset $\overline{\texttt{Img}}^{true} \subseteq \overline{\texttt{Img}}$.

TaxiNet Example. For TaxiNet, out of 11108 inputs, 9125 inputs (i.e., 82.1%) pass the DNN check resulting in the following code:

```
i:[0..M] init 0;
[]  pc=0 & i<M → 0.821: (v'=1) & (pc'=1) & (i'=0) + 0.179: (v'=0) & (i'=i+1);
```

We model the result of applying the DNN check with variable v; $v = 1$ if the check returns true for an image and $v = 0$ otherwise. M is the number of allowed repeated sensor readings and i is used to count the number of failed DNN checks.

The abstraction for state variables he ($\alpha_{\texttt{he}}$) and cte ($\alpha_{\texttt{cte}}$) is only computed for the inputs that pass the check (i.e., for $v = 1$) based on newly computed confusion matrices. The DTMC code for the closed-loop system with run-time guards is shown in [37], in the appendix.

3.3 Confidence Analysis

The construction of the probabilistic abstractions relies on calculating empirical point estimates of the required probabilities. However, these empirical estimates lack statistical guarantees and can be off by an arbitrary amount from the true probabilities. To address this concern, we experiment with using FACT [5,7]

[5] To simplify the presentation, we omit the precise mathematical formulation for D.

to calculate *confidence intervals* for the probability that the safety properties of the closed-loop system are satisfied. The inputs to FACT are: 1) a parametric DTMC m where each empirically estimated transition probability is represented by a parameter, 2) a PCTL formula ϕ, 3) an error level $\delta \in (0,1)$ and 4) an *observation function* O mapping state s to a tuple representing the number of observations for each outgoing transition from s; in our case, the number of observations can be obtained directly from the computed confusion matrices, i.e., $O(s) = (\mathcal{C}[\texttt{rep}(s), 1], \ldots, \mathcal{C}[\texttt{rep}(s), K])$. FACT synthesizes a $(1-\delta)$-confidence interval $[a,b] \subseteq [0,1]$ for the probability that ϕ is satisfied, given the observations.

TaxiNet Example. The following partial code illustrates the parametric version of the code provided in Sect. 3.1 (with the complete code for the parametric models provided in [37], in the appendix). The first three lines represent the number of observations obtained from the confusion matrix in Table 1.

```
param double x = 4748 2139 148;
param double y = 91 2010;
param double z = 744 211 1017;
...
[] he=0 → x1:(he_est'=0) + x2:(he_est'=1) + (1-x1-x2):(he_est'=2);
[] he=1 → y1:(he_est'=0) + (1-y1):(he_est'=1);
[] he=2 → z1:(he_est'=0) + z2:(he_est'=1) + (1-z1-z2):(he_est'=2);
```

4 Experiments

In this section, we report on the experiments that we conducted as part of our probabilistic safety analysis of the center-line tracking autonomous system.

We built two DTMC models, m_1 and m_2, denoting the closed-loop center-line tracking system without and with a run-time guard, respectively. The airplane dynamics and the controller are identically modeled in the two DTMCs as discrete components. The code for the models (in PRISM syntax) and more details about the analysis are presented in [37], in the appendix.

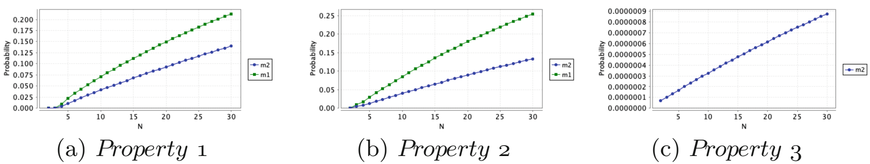
Mining Rules for Run-time Guards. We leverage our prior work [17], to extract rules of the form *Pre* $\implies$ *Post* from the DNN. *Post* is the condition $|\texttt{cte}^* - \texttt{cte}| > 1.0$ m $\vee$ $|\texttt{he}^* - \texttt{he}| > 5°$ on the regression model's outputs and *Pre* is a condition over the neuron values in the three dense layers of TaxiNet ($\texttt{cte}^*$ and $\texttt{he}^*$ denote ground-truth values). The considered *Post* characterizes invalid behavior (as explained in [31]). If an input satisfies *Pre*, the DNN check is considered to have failed on that input. *Pre* can be evaluated efficiently during the forward pass of the model, making it a good run-time guard candidate. Here is an example of a rule for invalid behavior:

$$\begin{aligned} & N_{1,85} <= -0.998 \ \wedge \ N_{2,50} <= 3.31 \ \wedge \ N_{1,84} <= -0.994 \ \wedge \ N_{1,15} > -0.999 \\ \wedge\ & N_{1,21} <= 1.711 \ \wedge \ N_{1,70} <= 11.088 \ \wedge \ N_{1,51} > -0.999 \ \wedge \ N_{1,21} > -0.637 \implies \\ & \qquad |\texttt{cte}^* - \texttt{cte}| > 1.0\ m \ \vee \ |\texttt{he}^* - \texttt{he}| > 5^\circ \end{aligned}$$

$N_{i,j}$ indicates the jth neuron in the ith dense layer. The conditions over neuron values can be checked during the forward pass of the DNN. If an input satisfies the conditions, it is interpreted as failing the check. If the check consecutively

fails M times, the system aborts, meaning that the system stops operating and hands over control to a fail-safe mechanism (such as the pilot). More details on the rules and their deployment as run-time guards are in [37], in the appendix.

Confusion Matrices. The confusion matrices for the classification version of TaxiNet, computed for the two cases (without and with run-time guard) are shown in [37], in the appendix. The tables can be used by developers to better understand the DNN performance. For instance, the results summarized in the confusion matrices indicate that the DNN performs best for inputs lying on the center-line, which can be attributed to training being done mainly using scenarios where the plane follows the center-line. The model appears to perform better when the plane is heading left, as opposed to heading right, which may be due to camera position. These observations can be used by developers to improve the model, by training on more scenarios. Note also that the model does not make 'blatant' errors, mistaking inputs on the *left* as being on the *right* (of center-line) or vice-versa (see e.g., entries with zero observations). Formal proofs can provide guarantees of absence of such transitions.



(a) *Property* 1 (b) *Property* 2 (c) *Property* 3

Fig. 3. Probabilistic model checking results via PRISM

Analysis. We analyzed m_1 and m_2 with respect to the two PCTL properties, $P = ?[F(\texttt{cte} = -1)]$ (*Property 1*), and $P = ?[F(\texttt{he} = -1)]$ (*Property 2*)[6]. The airplane is assumed to start from a initial position on the center-line and heading straight. For m_2, i.e. the model with a run-time guard, we also evaluate the probability of the TaxiNet system going to the abort state using the property, $P = ?[F(v = 0 \;\&\; i = M)$ (*Property 3*), where M is the threshold for the number of consecutive run-time check failures.

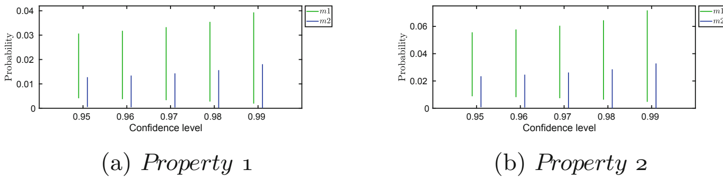
The probabilities of these properties being satisfied, calculated by PRISM, are shown in Fig. 3, where N is a constant in the DTMCs that dictates the length of the finite-time horizon considered for the analysis. Note that the system has an additional planning layer that calculates the waypoints for the airplane's course on the taxiway. The system is only used for controlling the airplane movement between pairs of waypoints, hence a short horizon suffices.

The confidence intervals computed with FACT are shown in Fig. 4, at different confidence levels (0.95 to 0.99), for $N = 4$. For computing the intervals, we ignore the transitions in the DTMCs that were not observed in our data (see [37] for more details).

[6] We rewrote the properties in terms of the discrete values.

The PRISM analysis scales well; e.g., evaluating *Property 1* for model m_2 ($N = 30$) requires less than 0.1 s on an M1 MacBook Pro, 16 GB RAM. The numbers are similar for other queries. However, the confidence analysis does not scale as well; we could not go beyond $N = 4$ for a timeout of two hours, with *Property 1* hardest to check. Newer work, fPMC [13], addresses these scalability challenges but we found it not yet mature enough to be applied to our models.

Discussion and Lessons Learned. The experiments demonstrate the feasibility of our approach, which enables reasoning about a complex DNN interacting with conventional (discrete-time) components via a simple probabilistic abstraction. Our analysis not only provides qualitative (i.e., an error is reachable or not) but also quantitative (i.e., likelihood of error) results, helping developers assess the risk associated with the analyzed scenario.



(a) *Property 1* (b) *Property 2*

Fig. 4. Confidence interval results via FACT

The results highlight the benefit of the run-time guards in improving the safety of the overall system; see Figs. 3(a,b) for lower error probabilities and Figs. 4(a,b) for tighter intervals for m_2. The probability of aborting is very small, indicating the efficacy of the fail-safe mechanism (see Figs. 3(c)). More importantly, since the DNN demonstrates higher accuracy on the inputs where the run-time check passes, the results also indicate that *improved accuracy of the DNN translates into improved safety.* The computed probabilities and confidence intervals can be examined by developers and regulators to ensure that system safety is met at required levels. If the confidence intervals are too large, they can be made tighter by adding more data, as guided by the confusion matrices.

Based on our feedback (confusion matrices) our industrial partner is retraining the perception network. As the system is in its early stages, our industrial partner was more interested in the trends suggested by our analysis rather than the exact probability results. For instance, our results indicate that safety will increase with a better-performing network. The partner was also interested in how the DNN-specific analysis contributes to the system-level analysis. A probabilistic analysis is best viewed as an "average-case" analysis rather than "worst-case". Nevertheless, such analysis is still useful since it conveys whether the system at least behaves safely in the average-case.

5 Conclusion

We demonstrated a method for the analysis of the safety of autonomous systems that use complex DNNs for visual perception. Our abstraction helps separate the concerns of DNN and conventional system development and evaluation. It also enables the integration of heterogeneous artifacts from DNN-specific analysis and system-level probabilistic model checking. The approach produces not only qualitative results but also provides insights that can be used in quantitative safety assessment for AI/DNN-enabled systems. This is, potentially, an important step to fill one of the gaps of quantitative evaluation for future AI certification [1].

Future work involves experimentation with image data sets representing a variety of environment conditions. We also plan to refine our models, inducing finer partitions on the DNN, and validate them through simulations. Another future research direction involves the study of the composition of safety proofs for the system analyzed in different scenarios. Finally, we are working on compositional analysis techniques to achieve worst-case (non-probabilistic) guarantees.

References

1. EASA concept paper: First usable guidance for level 1 machine learning applications (2021). https://www.easa.europa.eu/en/downloads/134357/en
2. Badithela, A., Wongpiromsarn, T., Murray, R.M.: Leveraging classification metrics for quantitative system-level analysis with temporal logic specifications. In: 2021 60th IEEE Conference on Decision and Control (CDC), pp. 564–571. IEEE (2021). https://doi.org/10.1109/CDC45484.2021.9683611
3. Beland, S., et al.: Towards assurance evaluation of autonomous systems. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, 2–5 November 2020, pp. 84:1–84:6. IEEE (2020)
4. Byrne, R., Abdallah, C., Dorato, P.: Experimental results in robust lateral control of highway vehicles. In: Proceedings of 1995 34th IEEE Conference on Decision and Control, vol. 4, pp. 3572–3575 (1995)
5. Calinescu, R., Ghezzi, C., Johnson, K., Pezzé, M., Rafiq, Y., Tamburrelli, G.: Formal verification with confidence intervals to establish quality of service properties of software systems. IEEE Trans. Reliab. **65**(1), 107–125 (2015)
6. Calinescu, R., Imrie, C., Mangal, R., Păsăreanu, C., Santana, M.A., Vázquez, G.: Discrete-event controller synthesis for autonomous systems with deep-learning perception components. arXiv preprint arXiv:2202.03360 (2022)
7. Calinescu, R., Johnson, K., Paterson, C.: FACT: a probabilistic model checker for formal verification with confidence intervals. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 540–546. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_32
8. Ciesinski, F., Größer, M.: On probabilistic computation tree logic. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 147–188. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24611-4_5

9. Clevert, D.A., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (ELUs). arXiv preprint arXiv:1511.07289 (2015)
10. Cohen, J., Rosenfeld, E., Kolter, Z.: Certified adversarial robustness via randomized smoothing. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 97, pp. 1310–1320. PMLR, 09–15 June 2019
11. Dawson, C., Gao, S., Fan, C.: Safe control with learned certificates: a survey of neural Lyapunov, barrier, and contraction methods. arXiv preprint arXiv:2202.11762 (2022)
12. Dawson, C., Lowenkamp, B., Goff, D., Fan, C.: Learning safe, generalizable perception-based hybrid control with certificates. IEEE Rob. Autom. Lett. **7**(2), 1904–1911 (2022)
13. Fang, X., Calinescu, R., Gerasimou, S., Alhwikem, F.: Software performability analysis using fast parametric model checking. arXiv preprint arXiv:2208.12723 (2022)
14. Frew, E., et al.: Vision-based road-following using a small autonomous aircraft. In: 2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720), vol. 5, pp. 3006–3015 (2004)
15. Fromherz, A., Leino, K., Fredrikson, M., Parno, B., Pasareanu, C.: Fast geometric projections for local robustness certification. In: International Conference on Learning Representations (2021)
16. Ghosh, S., Pant, Y.V., Ravanbakhsh, H., Seshia, S.A.: Counterexample-guided synthesis of perception models and control. In: 2021 American Control Conference (ACC), pp. 3447–3454. IEEE (2021)
17. Gopinath, D., Converse, H., Pasareanu, C., Taly, A.: Property inference for deep neural networks. In: International Conference on Automated Software Engineering (ASE), pp. 797–809. IEEE (2019)
18. Gopinath, D., Katz, G., Păsăreanu, C.S., Barrett, C.: DeepSafe: a data-driven approach for assessing robustness of neural networks. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 3–19. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_1
19. Gowal, S., et al.: On the effectiveness of interval bound propagation for training verifiably robust models. arXiv preprint arXiv:1810.12715 (2018)
20. Grigorescu, S.M., Trasnea, B., Cocias, T.T., Macesanu, G.: A survey of deep learning techniques for autonomous driving. CoRR abs/1910.07738 (2019)
21. Guo, C., Pleiss, G., Sun, Y., Weinberger, K.Q.: On calibration of modern neural networks. CoRR abs/1706.04599 (2017)
22. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transfer **24**(4), 589–610 (2022)
23. Hoffmann, G.M., Tomlin, C.J., Montemerlo, M., Thrun, S.: Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing. In: American Control Conference, ACC 2007, New York, NY, USA, 9–13 July 2007, pp. 2296–2301. IEEE (2007)
24. Hsieh, C., Li, Y., Sun, D., Joshi, K., Misailovic, S., Mitra, S.: Verifying controllers with vision-based perception using safe approximate abstractions. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **41**(11), 4205–4216 (2022)
25. Huang, X., et al.: A survey of safety and trustworthiness of deep neural networks: verification, testing, adversarial attack and defence, and interpretability. Comput. Sci. Rev. **37**, 100270 (2020)

26. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. CoRR abs/1610.06940 (2016)
27. Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., Lee, I.: Verisig 2.0: verification of neural network controllers using Taylor model preconditioning. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 249–262. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_11
28. Ivanov, R., Carpenter, T.J., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verifying the safety of autonomous systems with neural network controllers. ACM Trans. Embedded Comput. Syst. (TECS) **20**(1), 1–26 (2020)
29. Ivanov, R., Jothimurugan, K., Hsu, S., Vaidya, S., Alur, R., Bastani, O.: Compositional learning and verification of neural network controllers. ACM Trans. Embedded Comput. Syst. (TECS) **20**(5s), 1–26 (2021)
30. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 169–178 (2019)
31. Kadron, I.B., Gopinath, D., Pasareanu, C.S., Yu, H.: Case study: analysis of autonomous center line tracking neural networks. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, 18–19 October 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, 18–19 July 2021, Revised Selected Papers. LNCS, pp. 104–121 (2021). https://doi.org/10.1007/978-3-030-95561-8_7
32. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
33. Katz, S.M., Corso, A.L., Strong, C.A., Kochenderfer, M.J.: Verification of image-based neural network controllers using generative models. J. Aerosp. Inf. Syst. **19**(9), 574–584 (2022)
34. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
35. Leino, K., Wang, Z., Fredrikson, M.: Globally-robust neural networks. In: International Conference on Machine Learning (ICML) (2021)
36. Habeeb, P., Deka, N., D'Souza, D., Lodaya, K., Prabhakar, P.: Verification of camera-based autonomous systems. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., 1 (2023). https://doi.org/10.1109/TCAD.2023.3240131
37. Pasareanu, C.S., et al.: Closed-loop analysis of vision-based autonomous systems: a case study. CoRR abs/2302.04634 (2023). https://doi.org/10.48550/arXiv.2302.04634
38. Privault, N.: Discrete-Time Markov Chains, pp. 77–94. Springer, Heidelberg (2013)
39. Raghunathan, A., Steinhardt, J., Liang, P.: Certified defenses against adversarial examples. In: International Conference on Learning Representations (2018)
40. Santa Cruz, U., Shoukry, Y.: NNLander-VeriF: a neural network formal verification framework for vision-based autonomous aircraft landing. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods Symposium, pp. 213–230. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06773-0_11
41. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 1–30 (2019)
42. Tabernik, D., Skocaj, D.: Deep learning for large-scale traffic-sign detection and recognition. CoRR abs/1904.00649 (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Hybrid Controller Synthesis for Nonlinear Systems Subject to Reach-Avoid Constraints

Zhengfeng Yang[1], Li Zhang[1], Xia Zeng[2(✉)], Xiaochao Tang[1], Chao Peng[1], and Zhenbing Zeng[3]

[1] Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai, China
{zfyang,cpeng}@sei.ecnu.edu.cn, {lzhang,xctang}@stu.ecnu.edu.cn

[2] School of Computer and Information Science, Southwest University, Chongqing, China
xzeng0712@swu.edu.cn

[3] Department of Mathematics, Shanghai University, Shanghai, China
zbzeng@shu.edu.cn

Abstract. There is a pressing need for learning controllers to endow systems with properties of safety and goal-reaching, which are crucial for many safety-critical systems. Reinforcement learning (RL) has been deployed successfully to synthesize controllers from user-defined reward functions encoding desired system requirements. However, it remains a significant challenge in synthesizing provably correct controllers with safety and goal-reaching requirements. To address this issue, we try to design a special hybrid polynomial-DNN controller which is easy to verify without losing its expressiveness and flexibility. This paper proposes a novel method to synthesize such a hybrid controller based on RL, low-degree polynomial fitting and knowledge distillation. It also gives a computational approach, by building and solving a constrained optimization problem coming from verification conditions to produce barrier certificates and Lyapunov-like functions, which can guarantee every trajectory from the initial set of the system with the resulted controller satisfies the given safety and goal-reaching requirements. We evaluate the proposed hybrid controller synthesis method on a set of benchmark examples, including several high-dimensional systems. The results validate the effectiveness and applicability of our approach.

Keywords: Formal verification · Controller synthesis · Reinforcement learning · Barrier certificate · Lyapunov-like function

1 Introduction

The design of control and decision-making software for autonomous systems is a key part of many industrial applications, such as unmanned aerial vehicles, ground vehicles and general robots, therefore it attracts continued attention in the last decade [7,9,12,14]. Among many research works in this field, a highly

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 304–325, 2023.
https://doi.org/10.1007/978-3-031-37706-8_16

challenging problem is the controller synthesis, i.e., to build control systems that guarantee the safety and the reachability simultaneously. As an emergency approach, the machine learning method has also been developed to tackle this problem in recent years. Several existing techniques focus on learning a control policy from user-defined reward/cost functions for encoding the required properties. A typical way is to use the framework of reinforcement learning (RL) which evaluates and improves the controller's performance by interacting with environments and systems. Because of its strong ability to deal with nonlinear and/or uncertain (or indeterministic) dynamical systems of high dimensions, as well as the universal approximation power of the deep neural networks, the RL-based controller synthesis has been extensively studied, and substantial progress has been made by different research teams [22,23]. However, formal reasoning of the required properties of such DNN-controlled dynamical systems is an arduous and challenging problem which makes the practical use of RL still limited. For safety/reachability verification of the system under the learned controller, one main approach is tracing the reachable sets of the system through computing [8,13,30], which needs to measure the solutions to the ODEs of the system, thus the scalability of these approaches is largely restricted. Another major approach is creating a certificate synthesis through solving the associated SMT problems [6,16,31], which also has limited scalability since the complexity of symbolic computation in the general purpose SMT solvers. In this paper, we will utilize the advantage of RL to train an elaborately designed hybrid controller, which makes the system easier to be verified with safety and goal-reaching requirements while maintaining controllability.

Our proposed hybrid controller is in the form of a lower degree polynomial plus a relatively small size neural network, called a polynomial-DNN controller. The learning-based process of the polynomial-DNN controller synthesis is divided into the following four phases: (1) at first we train a well-performing DNN controller by RL with safety and goal-reaching requirements; (2) then we manage to fit the trained DNN roughly by a polynomial with a prescribed lower degree bound as one part of the hybrid structure; (3) we construct a small and special neural network (NN) with Square activation function on the hidden layer and tanh on the output layer as the supplement for the polynomial part, and subsequently distill an initial polynomial-DNN controller from the original DNN controller; (4) finally, using RL from the distilled one to fine-tune a well-performing polynomial-DNN controller.

Thanks to the hybrid form consisting of a polynomial and a small NN with the special structure, the obtained hybrid controller is easier to verify and maintains its expressiveness and flexibility for two main reasons: (1) considering the verification efficiency, the original DNN is fitted by a lower degree polynomial through coarse approximation which can be easily obtained and significantly reduce the difficulty of formal verification; (2) the NN part compensates for the controller performance loss caused by the coarse polynomial approximation. Benefitting from its feature, the system with the polynomial-DNN controller can be equivalently transformed into a polynomial form via system recasting, which makes post-verification easily solvable.

The necessity of proposing a polynomial-DNN type controller can be explained as follows. Transforming DNN into polynomial form enables the application of efficient polynomial solving techniques for formal verification, but there is no guarantee that a polynomial of a specified degree bound can fit a DNN with high accuracy; meanwhile, the approximation and corresponding verification problem will become quite complicated as the degree of the polynomial increases, which also may result in the failure of the verification. Therefore, we resort to lower degree polynomial approximation simultaneously retrain a small NN as the compensation for loss of accuracy, since a rough approximating polynomial part cannot replace the whole DNN controller, and the verification may fail for the system controlled by the polynomial part. The hybrid controller balances the richness of expressiveness and the ease of formal verification very well. To check the effectiveness of the proposed approach, we have evaluated the hybrid controller synthesis on a set of commonly used benchmark examples. To summarize, the main contributions of this paper are as follows:

- We propose a method to synthesize a hybrid polynomial-DNN controller subject to reach-avoid constraints, via RL incorporated with lower degree polynomial fitting and distillation based retraining, which not only maintains good control performance but also makes post-verification solvable.
- We delicately design a residual network as a compensation of the target controller. The particularity of the differential form of the residual network allows us to cast the differential equations of the control systems into an equivalent polynomial form which is conducive to formal verification.
- We carry out a detailed experimental evaluation on a set of benchmarks to demonstrate the effectiveness of our approach, and the necessity of the controller in such a hybrid form through ablation studies.

1.1 Related Works

Several research works focus on the controller synthesis for the safety requirement, in which a typical way is to use reinforcement learning or supervised learning to build the overall learning framework for synthesizing security certificates (such as control barrier function, CBF) [1,26–29].

For the goal-reaching requirement, most of existing works concentrate on building controllers to drive the system to reach a specified set within a time bound [8,11,13,30]. Some others focus on synthesizing the control policy to make the system asymptotically converge to a specified goal state set, which is called stability requirement. The certificate of Lyapunov functions generation is a practical routine in this aspect [3–5,15,25].

In fact, learning a reach-avoid controller, namely, for both safety and goal-reaching requirements, is a much more complicated problem. An example was given in [10], where a correct-by-construction controller that consists of a reference controller and a tracking controller has been successfully built to derive the actual trajectory according to the reference trajectory, and different reference controllers have been pre-designed for different scenarios.

Recently, a new learning-based approach is implemented in [17], where the safe and goal-reaching policy is constructed by jointly learning two additional certificate functions using supervised learning. Notice that there may exist the risk of synthesizing false certificates, as the certificate constraints are only satisfied at the sampled points. Although one can perform posterior formal verification to overcome this weak-point, it would be difficult to do the verification with several DNNs in the system. By comparison, our synthesized hybrid polynomial-DNN controller has clear advantages on formal verification.

2 Preliminaries

Notations. Let $\mathbb{R}[\mathbf{x}]$ denote the ring of polynomials with coefficients in $\mathbb{R}$ over variables $\mathbf{x} = [x_1, x_1, \ldots, x_n]^T$, and $\mathbb{R}[\mathbf{x}]^n$ denotes the n-dimensional polynomial vector. Let $\Sigma[\mathbf{x}] \subset \mathbb{R}[\mathbf{x}]$ be the set of SOS polynomials. The distance from $\mathbf{x}$ to a set S is defined by $\|\mathbf{x}\|_S = \inf_{s\in S} \|\mathbf{x} - s\|_2$. A continuous function $\alpha : [0, a) \to [0, +\infty)$ for some $a > 0$ is said to belong to class $\mathcal{K}$ if it is strictly increasing and satisfies $\alpha(0) = 0$. A continuous function $\beta : (-b, c) \to (-\infty, +\infty)$ for some $b, c > 0$ is said to belong to extended-class $\mathcal{K}$ if it is strictly increasing and satisfies $\beta(0) = 0$. A continuous function $\gamma : [0, c) \times [0, \infty) \to [0, +\infty)$ for some $c > 0$ belongs to class $\mathcal{KL}$, if for each fixed s, the mapping $\gamma(r, s)$ belongs to class $\mathcal{K}$ with respect to r, and for each fixed r, the mapping $\gamma(r, s)$ is decreasing with respect to s, and $\gamma(r, s) \to 0$ as $s \to \infty$.

This section formulates the safety and goal-reaching controller synthesis problem. A controlled continuous dynamical system is modeled by first-order ordinary differential equations

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), \quad \text{with } \mathbf{u} = \mathbf{k}(\mathbf{x}), \tag{1}$$

where $\mathbf{x} \in \Psi \subseteq \mathbb{R}^n$ are the system states, $\mathbf{u} \in U \subseteq \mathbb{R}^m$ are the control inputs, and $\mathbf{f} \in \mathbb{R}[\mathbf{x}]^n$ is the vector field defined on the state space $D \subseteq \mathbb{R}^n$.

Assume $\mathbf{f}$ satisfies the local Lipschitz condition, which ensures (1) has a unique solution $\mathbf{x}(t, \mathbf{x}_0)$ in D for every initial state $\mathbf{x}_0 \in D$ at $t = 0$. A dynamical system is equipped with a domain $\Psi \subset D$ and an initial set $\Theta \subset \Psi$, represented as a triple $\mathcal{C} \doteq (\mathbf{f}, \Psi, \Theta)$. Given a prespecified unsafe region $X_u \subset D$, we say that the system $\mathcal{C}$ is *safe* if all trajectories starting from Θ can not evolve into the unsafe region X_u, which has been widely investigated in safety critical applications.

Definition 1 (Safety). *For* a *controlled constrained continuous dynamical system (CCDS)* $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$ *and a given unsafe region* X_u*, the system is safe if for all* $\mathbf{x}_0 \in \Theta$*, there does not exist* $t_1 > 0$ *such that*

$$\forall t \in [0, t_1).\mathbf{x}(t, \mathbf{x}_0) \in \Psi \quad \text{and } \mathbf{x}(t_1, \mathbf{x}_0) \in X_u.$$

At the same time, another important property has received much attention which is a generalization of stability and called *goal-reaching*.

Definition 2 (Goal-reaching). *Given a controlled CCDS* $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$ *and a set of goal states* $X_g \subset D$*, the system* $\mathcal{C}$ *is goal-reaching with respect to the goal set* X_g*, if there exists a* $\mathcal{KL}$*-function* γ *such that for any* $\mathbf{x}_0 \in \Theta$*,*

$$\|\mathbf{x}(t)\|_{X_g} \leq \gamma(\|\mathbf{x}(0)\|_{X_g}, t) \quad \textit{for all } t \geq 0.$$

Definition 3 (Safe and Goal-reaching Controller Synthesis). *Given a controlled CCDS* $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$ *with* $\mathbf{f}$ *defined by (1) with an unsafe set* X_u *and a goal set* X_g*, design a locally Lipschitz continuous feedback control law* $\mathbf{k}$ *such that the closed-loop system* $\mathcal{C}$ *with* $\mathbf{f} = \mathbf{f}(\mathbf{x}, \mathbf{k}(\mathbf{x}))$ *is both safe and goal-reaching as per Definition 1 and 2.*

The concept of *barrier certificates* plays an important role in safety verification of continuous systems. The essential idea is to use the zero level set of a barrier certificate $B(\mathbf{x})$ as a barrier to separate all the reachable states from the unsafe region. The following concept of barrier certificate, adapted from [24], can be used to guarantee the safety of a given controlled CCDS.

Theorem 1. *[24] Given a controlled CCDS* $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$*, with* $\mathbf{f}$ *defined by (1), a feedback control law* $\mathbf{u} = \mathbf{k}(\mathbf{x})$*, and the unsafe region* X_u*. Suppose there exists a real-valued function* $B : \Psi \to \mathbb{R}$ *satisfying the following conditions:*

(i) $B(\mathbf{x}) \geq 0 \quad \forall \mathbf{x} \in \Theta$,
(ii) $B(\mathbf{x}) < 0 \quad \forall \mathbf{x} \in X_u$,
(iii) $B(\mathbf{x}) = 0 \Rightarrow \mathcal{L}_f B(\mathbf{x}) > 0 \quad \forall \mathbf{x} \in \Psi$,

where $\mathcal{L}_f B(\mathbf{x})$ *denotes the Lie-derivative of* $B(\mathbf{x})$ *along the vector field* $\mathbf{f}(\mathbf{x})$*, i.e.,* $\mathcal{L}_f B(\mathbf{x}) = \sum_{i=1}^{n} \frac{\partial B}{\partial x_i} \cdot f_i(\mathbf{x})$*, then* $B(\mathbf{x})$ *is a barrier certificate for the closed-loop system* $\mathcal{C}$ *with the control law* $\mathbf{k}(\mathbf{x})$*, and the safety of system* $\mathcal{C}$ *is guaranteed.*

For the goal-reaching controller design, we use a more general Lyapunov-like function which is introduced by the following definition.

Definition 4 (Lyapunov-like function). *Given a continuous system* $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$*, and the set of goal states* $X_g \subseteq \Psi$*, a continuous differentiable real-valued function* $V : \Psi \to \mathbb{R}$ *is said to be a Lyapunov-like function if*

(i) $\{\mathbf{x} | V(\mathbf{x}) \leq 0\} \neq \emptyset$ *and* $\{\mathbf{x} | V(\mathbf{x}) \leq 0\} \subseteq X_g$,
(ii) $\mathcal{L}_f V(\mathbf{x}) \leq -\beta(V(\mathbf{x})) \quad \forall \mathbf{x} \in \Psi$,

where β *is some extended class* $\mathcal{K}$ *function, and* $\mathcal{L}_f V(\mathbf{x}) = \sum_{i=1}^{n} \frac{\partial V}{\partial x_i} \cdot f_i(\mathbf{x})$*.*

As mentioned in [17], the above Lyapunov-like function is more general than the classic one used in [3,4,21,25]. The Lyapunov-like function does not necessarily require that $\mathcal{L}_f V(\mathbf{x})$ has to be always negative-definite, that is, $\mathcal{L}_f V(\mathbf{x}) > 0$ can happen on $\{\mathbf{x} | V(\mathbf{x}) < 0\}$, which will make the function less restrictive.

Theorem 2. *For a controlled CCDS* $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$ *with* $\mathbf{f}$ *defined by (1) and a set of goal states* $X_g \subseteq \Psi$*, if* $V(\mathbf{x})$ *is a Lyapunov-like function as in Definition 4, then the system under* $\mathbf{u} = \mathbf{k}(\mathbf{x})$ *is goal-reaching with respect to* X_g*.*

Combining Theorem 1 and Theorem 2, we obtain the following assertion stating that the existence of barrier certificates and Lyapunov-like functions guarantees the control law is both safe and goal-reachable. Hereafter, we refer to both barrier and Lyapunov-like functions as certificate functions for simplification.

3 Hybrid Polynomial-DNN Controllers Training

For the safe and goal-reaching controller synthesis problem, we design an easy-to-verify control policy with the aid of reinforcement learning (RL) based on barrier certificate and Lyapunov-like function generation. As we know, it is hard for a controller with a simple structure to guarantee the safe and goal-reachable behaviors for large-scale systems. Contrarily, controllers with complex structures can make the system have more flexible behaviors. Unfortunately, it requires much more computation efforts to tackle reach-avoid verification of the system with such a complex controller. To make it amenable, we propose a method to learn a controller with special structure, *hybrid polynomial-DNN controller*, which is easily verifiable, and can be customized to safety and goal-reaching requirement. Specifically, this hybrid controller consists of a polynomial and a small-size neural network with one single hidden layer. Notably, it is expected to exhibit similar behaviors to the original complex DNN controller, but is much easier to be verified thanks to its special structure, which will be elaborated in Sect. 4.

To achieve this, we adopt a low-degree polynomial to roughly approximate the DNN. Then we fix the structure of a small-size neural network and append it to the low-degree polynomial to construct a hybrid form controller, which is retrained using RL. To accelerate the retraining process, we use distillation technology to distill an initialization of the NN part in the hybrid controller. In summary, the learning-based process of the hybrid controller synthesis is divided into the following three stages, as shown in Fig. 1.

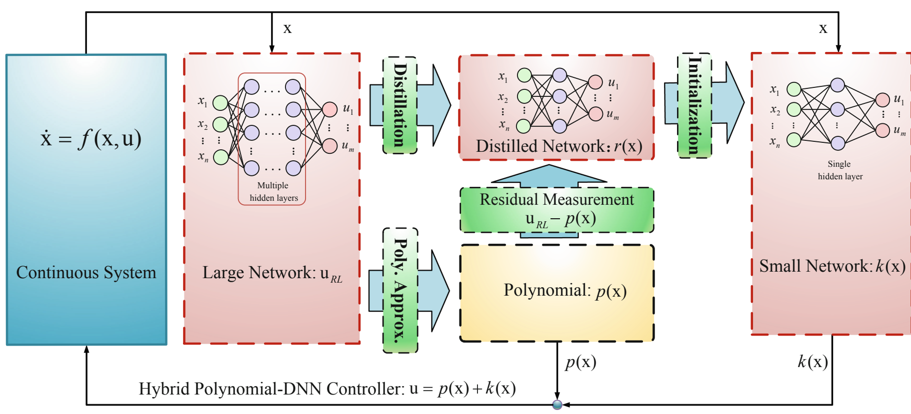


Fig. 1. The diagram of training framework.

– **Train a deep neural network controller via RL.** Based on reinforcement learning, we train a deep neural network (DNN) controller for the given

control system directly. Briefly, the RL procedure continuously uses the current controller to drive the system by interacting with the environment, and updates the relevant parameters of the controller by rewarding and penalizing. Through sufficient simulation and training, we expect to obtain a DNN controller that enables the system behavior to avoid the unsafe set and reach the specified target set with high probability.

- **Fit the DNN controller by a polynomial and distill a residual network by measuring the fitting error.** From the learned DNN controller in the previous process, we reconstruct a hybrid controller consisting of a polynomial and a small neural network with a single hidden layer. Specifically, we approximate the trained DNN controller with an appropriate polynomial by sampling based method. The approximate polynomial is used as the main component of the hybrid controller. We further evaluate the error between the original DNN and the polynomial approximation by distillation learning, which yields a small neural network as a refined module.
- **Generate and retrain a hybrid controller by fine-tuning a small neural network from the distilled network.** We construct a special small NN with square and tanh activation functions on the hidden and output layers respectively, which helps to transform the hard verification problem into a tractable polynomial one. At last, we retrain the hybrid controller consisting of the polynomial part and the small NN template by fine-tuning the small network initialized by the result from the distillation learning.

3.1 Training Well-Performing DNN Controllers Using RL

As illustrated in Fig. 1, the RL method is applied to train a well performed controller, so that the system is able to avoid obstacles and reach the goal region within the time bound.

We construct the reward function through encoding the desired behaviours of the closed-loop system under the DNN controller, which assures unsafe region avoidance and goal region reachability. We hope that the RL helps to synthesize an ideal controller by the designed reward, and all the trajectories of the closed-loop system starting from the initial set Θ cannot evolve into the unsafe region X_u, and reach the desired region X_g under the trained DNN controller. So the reward function design should concern two aspects, i.e., reward the behaviours far away from the unsafe region, and reward the behaviours approaching the goal region. In terms of the safety requirement, the reward function should penalize the behaviours approaching X_u. Thus, the reward function can be defined as a joint Gaussian distribution on the system state, whose expectation and variance are the center and radius of X_u, respectively,

$$reward_u(\mathbf{x}_t) = -e^{-\frac{1}{2}\sum_{i=1}^{n}(\frac{x_i(t)-x_u^i}{\rho_u^i})^2}$$

where $\mathbf{x}_u = (x_u^1, \ldots, x_u^n) \in X_u \subset D$ is the center of X_u and ρ_u is the radius of X_u. Similarly, the reward for the goal-reaching purpose could be defined as a joint Gaussian distribution,

$$reward_g(\mathbf{x}_t) = e^{-\frac{1}{2}\sum_{i=1}^{n}(\frac{x_i(t)-x_g^i}{\rho_g^i})^2}$$

where $\mathbf{x}_g = (x_g^1, \ldots, x_g^n)$ and ρ_g are the center and the radius of X_g, respectively. The entire reward function consists of the above two components, i.e.

$$reward(\mathbf{x}_t) = \lambda \cdot reward_g(\mathbf{x}_t) + (1-\lambda) \cdot reward_u(\mathbf{x}_t),$$

to achieve the task of safety and goal reachability, where $0 < \lambda < 1$ is the parameter to control the weights between $reward_g(\mathbf{x}_t)$ and $reward_u(\mathbf{x}_t)$.

The remaining problem is to train the controller via RL. Here we use Deep Deterministic Policy Gradient (DDPG) [20] which is a popular RL approach suited for continuous control applications. The DDPG algorithm combines the value-based and policy-based methods, and is made up of two neural networks: the critic network and actor network.

To train the desired controller, we first generate a set of initial states from Θ. For each sampled initial state $\mathbf{x}_0$, with the help of $\mathbf{u}_{RL}$, one may yield the associated trajectory as a discrete time state sequence $\{\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_t, \cdots, \mathbf{x}_m\}$ which does not enter the unsafe area, and then collect the transition tuples $(\mathbf{x}_t, \mathbf{x}_{t+1}, \mathbf{u}_t, reward(\mathbf{x}_t))$ to form a replay buffer. Every few time steps, a batch size of data is sampled from the replay buffer to update the parameters of critic network and actor network, and then the new controller is used to simulate the trajectory to collect new data until the controller behaves well.

3.2 Polynomial Approximation

Following the RL training process in Sect. 3.1, one may probably adopt a complex DNN structure to obtain a well-performing DNN controller. For safety critical systems, the properties of such synthesized controllers, such as safety and goal-reaching, need to be formally guaranteed. However, it is a challenging problem to verify specified properties for the closed-loop system under the trained DNN-type controller due to its complexity. Consequently, a high-degree polynomial can be found by approximating the trained DNN with extremely high precision and may be expected as the controller candidate to be verified with polynomial constraint solving. However, it could be an unbearable high computation complexity for the corresponding verification problem with such high-degree polynomial controller, which will be explained in the experiment section.

Based on the trained DNN controller $\mathbf{u}_{RL}$ through RL, we construct an easily verifiable controller with a hybrid form, which could lead the system to be safe and goal-reachable. We firstly roughly approximate the $\mathbf{u}_{RL}$ by a low-degree polynomial, denoted by $p(\mathbf{x})$, as a part. Afterwards, we retrain a small NN, denoted by $k(\mathbf{x})$, with one hidden layer as the compensation for the approximation error between $\mathbf{u}_{RL}$ and $p(\mathbf{x})$. The hybrid polynomial-DNN controller is built, i.e., $p(\mathbf{x}) + k(\mathbf{x})$. The main task of this subsection focuses on how to obtain the approximate polynomial $p(\mathbf{x})$ based on sampling points.

Concretely, a real coefficient vector $\mathbf{c}$ is used to parameterize a polynomial $p(\mathbf{x}, \mathbf{c})$ with a given degree d, i.e., $p(\mathbf{x}, \mathbf{c}) = \sum_j c_j b_j(\mathbf{x})$, where $b_j(\mathbf{x})$ are monomials with total degree $\leq d$. Given the sampling points, we can obtain the coefficient vector $\mathbf{c}^*$ by solving a least squares problems. Thus, the approximate polynomial $p(\mathbf{x}, \mathbf{c}^*)$ is the approximation of $\mathbf{u}_{RL}(\mathbf{x})$ on Ψ, denoted by $p(\mathbf{x})$ for brevity. And the residual function $r(\mathbf{x})$ denotes the error between the approximate polynomial $p(\mathbf{x})$ and the DNN controller, i.e., $r(\mathbf{x}) = \mathbf{u}_{RL}(\mathbf{x}) - p(\mathbf{x})$.

Having $p(\mathbf{x})$, we cannot just regard it as the controller, because the error $r(\mathbf{x})$ between $\mathbf{u}_{RL}(\mathbf{x})$ and $p(\mathbf{x})$ can not be ignorable. To take this into account, we compensate for the error by fitting the residual function $r(\mathbf{x})$, by means of retraining a hybrid controller $p(\mathbf{x}) + k(\mathbf{x}|\theta')$ to rectify the system behavior, where θ' is the parameter to learn the NN part.

3.3 Training the Residual Controller

In this part, we retrain to compensate for the difference in system behavior guided by the polynomial part $p(\mathbf{x})$ versus the original DNN controller $\mathbf{u}_{RL}$.

The Structure of the Residual Network. We design a special neural network as the compensation to make the resulting verification problem tractable. As illustrated in Fig. 2, a typical DNN has a layered architecture and can be represented as a composition of its L layers: $k(\mathbf{x}|\theta') = l_L \circ l_{L-1} \circ \cdots \circ l_1(\mathbf{x})$, where $l_i(\mathbf{x}) = \sigma_i(W_i\mathbf{x} + b_i)$ which is parameterized by a weight matrix W_i and a bias vector b_i, and all the parameters are denoted by θ' for brevity. This work considers σ_i to be square activation on the hidden layers and tanh activation function on the output layer L, as shown in Fig. 2. This special setting has two advantages: i) ability to converge in the training process with the help of normalized output in the range of $[-1, 1]$; ii) ability to transform the control system with NN controller of this type into a polynomial form by system recasting (c.f. 4.1 for more details). Regarding ii), we introduce a new variable x_{n+1} to represent the NN output, i.e., $x_{n+1} := \tanh(h(\mathbf{x}))$, where $h(\mathbf{x}) := l_{L-1} \circ \cdots \circ l_1(\mathbf{x})$ denotes the polynomial part in NN. The main observation that allows us to transform the system with this NN controller into an equivalent polynomial system is the fact that the special NN's derivative can be expressed as

$$\dot{x}_{n+1} = (1 - x_{n+1}^2)\dot{h}. \tag{2}$$

Actually, we construct such small NN with one single hidden layer because it is enough to construct a simple structure neural network further added to the controller as the compensation to control systems well.

The Residual Controller Training. Then we retrain the hybrid controller $p(\mathbf{x}) + k(\mathbf{x}|\theta')$ making use of RL technique as described in the previous subsection. In order to improve training efficiency, the knowledge distillation technique is used to obtain the initialization of the NN part, i.e., $k(\mathbf{x}|\theta')$. It is easy to achieve

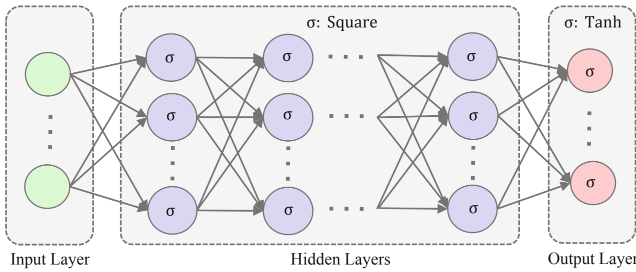


Fig. 2. Structure of the small neural network in the hybrid controller.

this by regarding the residual function $r(\mathbf{x})$ as the ensemble network (also called teacher network) and distilling the knowledge from it into a small model (i.e., student network). The learned student network realizes the knowledge transfer from the teacher network and provides the initial values for the $k(\mathbf{x}|\theta')$ for further training.

We reiterate that the purpose of constructing a hybrid controller by adding $k(\mathbf{x}|\theta')$ to the polynomial part $p(\mathbf{x})$ is to make the hybrid controller drive the system to perform as expected by the compensation. Here we achieve this not by training $k(\mathbf{x}|\theta')$ to satisfy $\mathbf{u}_{RL} = p(\mathbf{x}) + k(\mathbf{x}|\theta')$, but instead we require the controller $p(\mathbf{x}) + k(\mathbf{x}|\theta')$ could drive the following closed system to be safe and goal-reachable essentially: $\dot{\mathbf{x}} = f(\mathbf{x}, p(\mathbf{x}) + k(\mathbf{x}|\theta'))$.

We need to train a hybrid controller $p(\mathbf{x}) + k(\mathbf{x}|\theta')$ for the above system to obtain the parameter θ'. Utilizing the learned parameters of the student network from the knowledge distillation as the initialization for the $k(\mathbf{x}|\theta')$, we simulate the system to collect a dataset of sampled trajectories, and use the DDPG algorithm to achieve the control objective of safety and goal-reaching, by referring to the reward design elaborated in Sect. 3.1. Once the training is completed, we obtain the desired hybrid polynomial-DNN controller $u(\mathbf{x}) = p(\mathbf{x}) + k(\mathbf{x})$, where $p(\mathbf{x})$ is the polynomial part and $k(\mathbf{x})$ is the small neural network.

4 Reach-Avoid Verification with Lyapunov-Like Functions and Barrier Certificates Generation

To ensure the safety and goal-reaching properties for the specified control system under the synthesized controller, a relaxed surrogate is to generate a Lyapunov-like function and a barrier certificate, stated in Theorem 1 and Theorem 2. Note that, to make the computation tractable, the basic idea is to translate the problem of producing barrier certificates and Lyapunov-like function into a solvable polynomial optimization problem. Specifically, we first transform the ODEs $\mathbf{f}$ of the CCDS through system recasting; and then we abstract the initial set Θ, unsafe region X_u, goal set X_g and the system domain Ψ by polynomial expressions. At last, we establish the polynomial optimization problems yielded from the constraints of barrier certificate and Lyapunov-like function,

proceeded by solving the resulted polynomial optimization problem to produce a barrier certificate and Lyapunov-like function, which can guarantee the safety and goal-reaching properties for the system with the hybrid controller, respectively. Notably, Sum-of-Squares (SOS) relaxation technique is applied to encode the polynomial optimization problem as an SOS problem involved with bilinear matrices inequalities (BMI) constraints.

4.1 Constructing Polynomial Simulations of the Controller Network

In the following, we assume the control input $\mathbf{u}$ is one-dimensional for ease of presentation without loss of generality. Given a controlled CCDS $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$ with $\mathbf{f}$ defined by (1) with an unsafe set X_u and a goal set X_g. Suppose the hybrid controller learned for the safety and goal-reaching requirements is $u(\mathbf{x}) = p(\mathbf{x}) + k(\mathbf{x})$. Here $k(\mathbf{x})$ is a small neural network with the square function as its activation function in the hidden layer, and the tanh in the output layer, i.e., $k(\mathbf{x}) = \tanh(h(\mathbf{x}))$ where h is a polynomial which is in fact the composition of an affine function and a square function. We replace the non-polynomial term occurring in the controller part of the vector field $\mathbf{f}(\mathbf{x}, \mathbf{u})$ by introducing $x_{n+1} = \tanh(h(\mathbf{x}))$. Then $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$ is transformed into a polynomial one:

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, p(\mathbf{x}) + x_{n+1}), \\ \dot{x}_{n+1} = (1 - x_{n+1}^2)\dot{h}(\mathbf{x}). \end{cases} \tag{3}$$

For simplicity, we denote (3) as $\hat{\mathbf{f}} \in \mathbb{R}[\mathbf{x}]^{n+1}$.

Besides the vector field, we need to transform the Θ, Ψ, X_u, X_g respectively because of the introduced new variable. For instance, the initial set should be specified by $\bar{\Theta} := \{(\mathbf{x}, x_{n+1}) \in \mathbb{R}^{n+1} \,|\, \mathbf{x} \in \Theta,\, x_{n+1} = \tanh(h(\mathbf{x}))\}$. Actually, $\bar{\Theta}$ can be abstracted by a polynomial inclusion. For the initial set $\bar{\Theta}$, we first compute a hyper-rectangle $I := \{\mathbf{x} \in \mathbb{R}^n | \wedge l_i \leq x_i \leq u_i\}$ as an over-approximation of the bounded compact set Θ through interval analysis, then we could compute a Taylor model for the term $\tanh(h(\mathbf{x}))$ on I and obtain $p_1(\mathbf{x}) - \delta_1 \leq x_{n+1} \leq p_1(\mathbf{x}) + \delta_1$. For $\bar{\Theta}$, we can get the corresponding polynomial abstraction $\hat{\Theta}$. For brevity, let $\hat{\mathbf{x}}$ denote the variable vector with the introduced variable x_{n+1}, i.e., $\hat{\mathbf{x}} = (\mathbf{x}, x_{n+1}) = (x_1, \ldots, x_n, x_{n+1})^T$. Likewise, the other sets Ψ, X_u, X_g can be dealt with in the same manner, and yield the associated polynomial abstractions, $\hat{\Psi}$, $\hat{X}_u$, $\hat{X}_g$. The above polynomial abstractions can be written as following

$$\begin{cases} \hat{\Theta} := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \,|\, \mathbf{x} \in \Theta,\ |x_{n+1} - p_1(\mathbf{x})| \leq \delta_1\}, \\ \hat{\Psi} := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \,|\, \mathbf{x} \in \Psi,\ |x_{n+1} - p_2(\mathbf{x})| \leq \delta_2\}, \\ \hat{X}_u := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \,|\, \mathbf{x} \in X_u,\ |x_{n+1} - p_3(\mathbf{x})| \leq \delta_3\}, \\ \hat{X}_g := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \,|\, \mathbf{x} \in X_g,\ |x_{n+1} - p_4(\mathbf{x})| \leq \delta_4\}. \end{cases} \tag{4}$$

Finally, we obtain a polynomial CCDS $\hat{\mathcal{C}} = (\hat{\mathbf{f}}, \hat{\Psi}, \hat{\Theta})$. Therefore, if $\mathbf{x}(t)$ is a trajectory of system (1) within domain specified by Ψ starting from some initial state $\mathbf{x}(t_0) \in \Theta$, then $\hat{\mathbf{x}}(t)$ is the trajectory of system (3) within the

relaxed domain specified by $\hat{\Psi}$ starting from the initial state $\hat{\mathbf{x}}(t_0) \in \hat{\Theta}$ with $x_{n+1}(t_0) = \tanh(h(\mathbf{x}(t_0)))$.

Theorem 3. *If controlled CCDS $\hat{\mathcal{C}} = (\hat{\mathbf{f}}, \hat{\Psi}, \hat{\Theta})$ with $\hat{\mathbf{f}}$ defined by (3) and with $\hat{\Theta}$, $\hat{\Psi}$, and $\hat{X}_u$ defined by (4) is safe, then the original CCDS $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$ with the given unsafe set X_u is safe. Moreover, if $B(\hat{\mathbf{x}})$ is a barrier certificate of $\hat{\mathcal{C}}$ w.r.t. $\hat{X}_u$, then $B(\mathbf{x}, \tanh(h(\mathbf{x})))$ is also the barrier function of $\mathcal{C}$ w.r.t. X_u.*

Proof. Without loss of generality, let us assume that $\mathbf{x}(t), t > 0$ is one trajectory of the controlled CCDS $\mathcal{C}$ starting from the initial state $\mathbf{x}(t_0) \in \Theta$, then $\hat{\mathbf{x}}(t)$ with $x_{n+1}(t) = \tanh(h(\mathbf{x}(t)))$ is a trajectory of $\hat{\mathcal{C}}$ starting from the initial state $\hat{\mathbf{x}}(t_0) \in \hat{\Theta}$. Then, the safety of $\hat{\mathcal{C}}$ indicates that each trajectory of $\hat{\mathcal{C}}$ from the initial state $\hat{\Theta}$ cannot reach any unsafe state specified by the assertions $\hat{X}_u$, which implies that each trajectory of $\mathcal{C}$ from the initial state $\mathbf{x}(t_0)$ cannot reach any state specified by X_u. Furthermore, the vector field $\hat{\mathbf{f}}$ is yielded from $\mathbf{f}$ by the equivalent transformation, and $\hat{\Theta}$, $\hat{\Psi}$ and $\hat{X}_u$ are the associated polynomial abstractions. Therefore, $B(\mathbf{x}, \tanh(h(\mathbf{x})))$ is the barrier certificate of CCDS $\mathcal{C}$.

Theorem 4. *If controlled CCDS $\hat{\mathcal{C}} = (\hat{\mathbf{f}}, \hat{\Psi}, \hat{\Theta})$ with $\hat{\mathbf{f}}$ defined by (3) and with $\hat{\Theta}$, $\hat{\Psi}$ and $\hat{X}_g$ defined by (4) is goal-reaching, then the original CCDS $\mathcal{C} = (\mathbf{f}, \Psi, \Theta)$ with the given goal set X_g is goal-reaching. Moreover, if $V(\hat{\mathbf{x}})$ is a Lyapunov-like function of $\hat{\mathcal{C}}$ w.r.t. $\hat{X}_g$, then $V(\mathbf{x}, \tanh(h(\mathbf{x})))$ is the Lyapunov-like function of $\mathcal{C}$ w.r.t. X_g.*

Proof. Suppose the CCDS $\mathcal{C}$ is not goal-reaching for the given goal set X_g. Then $\exists \epsilon$ and $\exists \mathbf{x}_0 \in \Theta$ such that $\|\mathbf{x}(t)\|_{X_g} > \epsilon, \forall t > 0$. The state $\hat{\mathbf{x}}(t) \in \hat{\Psi}$ with $x_{n+1}(t) = \tanh(h(\mathbf{x}(t)))$ from the initial state $\hat{\mathbf{x}}(t_0)$ satisfying

$$\|\hat{\mathbf{x}}(t)\|_{\hat{X}_g} > \epsilon, \tag{5}$$

because according to (4), $\hat{X}_g$ is obtained just by involving a new variable and not changing the projection on the first n-dimension , i.e., X_g. Then from the theorem assumption, the CCDS $\hat{\mathcal{C}}$ is goal-reaching, so $\exists T > 0$ such that $\|\hat{\mathbf{x}}(t)\|_{\hat{X}_g} < \epsilon$, which contradicts with (5). Similar to Theorem 3, $V(\mathbf{x}, \tanh(h(\mathbf{x})))$ is the Lyapunov-like function of $\mathcal{C}$ w.r.t. X_g. This completes the proof.

4.2 Producing Barrier Certificate and Lyapunov-Like Function

For simplicity, hereafter we denote $\hat{\Theta}$, $\hat{\Psi}$, $\hat{X}_u$ and $\hat{X}_g$ as follows.

$$\begin{cases} \hat{\Theta} := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \mid \wedge_{i=1}^{m_1} g_i(\hat{\mathbf{x}}) \geq 0\}, & \hat{\Psi} := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \mid \wedge_{j=1}^{m_2} h_j(\hat{\mathbf{x}}) \geq 0\}, \\ \hat{X}_u := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \mid \wedge_{k=1}^{m_3} q_k(\hat{\mathbf{x}}) \geq 0\}, & \hat{X}_g := \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \mid \wedge_{\ell=1}^{m_4} s_\ell(\hat{\mathbf{x}}) \geq 0\}. \end{cases}$$

Barrier Certificate Generation. Assume that the barrier function $B(\hat{\mathbf{x}})$ is a polynomial of degree at most d, whose coefficients form a vector space of dimension $s(d) = \binom{n+1+d}{d}$ with the canonical basis $(\hat{\mathbf{x}}^\alpha)$ of monomials. Suppose

the coefficients are unknown, and denoted by $\mathbf{b} = (b_\alpha) \in \mathbb{R}^{s(d)}$ the coefficient vector of $B(\hat{\mathbf{x}})$, and write

$$B(\hat{\mathbf{x}}, \mathbf{b}) = \sum_{\alpha \in \mathbb{N}_d^n} b_\alpha \hat{\mathbf{x}}^\alpha = \sum_{\alpha \in \mathbb{N}_d^n} b_\alpha \, x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n} x_{n+1}^{\alpha_{n+1}},$$

in the canonical basis. As stated in Theorem 1 and Theorem 3, the controlled CCDS $\mathcal{C}$ is safe under the designed controller if there exists such a barrier certificate $B(\hat{\mathbf{x}}, \mathbf{b})$ for CCDS $\hat{\mathcal{C}}$. Meanwhile, determining the existence of barrier certificate $B(\hat{\mathbf{x}}, \mathbf{b})$, can be represented as the following feasibility problem.

$$\begin{cases} \text{find} \quad \mathbf{b} \\ \text{s.t. } B(\hat{\mathbf{x}}, \mathbf{b}) \geq 0, \;\; \forall \hat{\mathbf{x}} \in \hat{\Theta}, \\ \qquad \mathcal{L}_{\mathbf{f_u}} B(\hat{\mathbf{x}}, \mathbf{b}) > 0, \;\; \forall \mathbf{x} \in \hat{\Psi} \text{ and } B(\hat{\mathbf{x}}, \mathbf{b}) = 0, \\ \qquad B(\hat{\mathbf{x}}, \mathbf{b}) < 0, \;\; \forall \hat{\mathbf{x}} \in \hat{X}_u. \end{cases} \tag{6}$$

Moreover, Sum-of-Squares (SOS) relaxation technique is applied to encode the optimization problem (6) as an SOS program. Given a basic semi-algebraic set $\mathbb{K}$ defined by: $\mathbb{K} = \{\hat{\mathbf{x}} \in \mathbb{R}^{n+1} \,|\, g_1(\hat{\mathbf{x}}) \geq 0, \ldots, g_s(\hat{\mathbf{x}}) \geq 0\}$, where $g_i(\hat{\mathbf{x}}) \in \mathbb{R}[\hat{\mathbf{x}}], 1 \leq i \leq s$, a sufficient condition for the nonnegativity of the given polynomial $f(\hat{\mathbf{x}})$ on the semi-algebraic set $\mathbb{K}$ is provided as

$$f(\hat{\mathbf{x}}) = \sigma_0(\hat{\mathbf{x}}) + \sum_{i=1}^{s} \sigma_i(\hat{\mathbf{x}}) g_i(\hat{\mathbf{x}}), \tag{7}$$

where $\sigma_i(\hat{\mathbf{x}}) \in \Sigma[\hat{\mathbf{x}}]_d$, $1 \leq i \leq s$. Thus, the representation (7) ensures that the polynomial $f(\hat{\mathbf{x}})$ is nonnegative on the given semi-algebraic set $\mathbb{K}$.

Observing (6), the polynomial $\mathcal{L}_{\mathbf{f_u}} B(\hat{\mathbf{x}}, \mathbf{b})$ is involved with the uncertain variable ε in the range $[-\mu^*, \mu^*]$, which can be written as $\hat{h}(\varepsilon) \geq 0$ with

$$\hat{h}(\varepsilon) := (\varepsilon + \mu^*)(\mu^* - \varepsilon).$$

Thus, the problem (6) can be transformed into the following optimization problem through SOS relaxation

$$\begin{cases} \text{find} \quad \mathbf{b} \\ \text{s.t. } B(\hat{\mathbf{x}}, \mathbf{b}) - \sum_i \sigma_i(\hat{\mathbf{x}}) g_i(\hat{\mathbf{x}}) \in \Sigma[\hat{\mathbf{x}}], \\ \qquad \mathcal{L}_{\mathbf{f_u}} B(\hat{\mathbf{x}}, \mathbf{b}) - \lambda(\hat{\mathbf{x}}) B(\hat{\mathbf{x}}, \mathbf{b}) - \sum_j \phi_j(\hat{\mathbf{x}}) h_j(\hat{\mathbf{x}}) - \nu(\hat{\mathbf{x}}, \varepsilon) \hat{h}(\varepsilon) - \epsilon \in \Sigma[\hat{\mathbf{x}}], \\ \qquad -B(\hat{\mathbf{x}}, \mathbf{b}) - \epsilon' - \sum_j \kappa_j(\hat{\mathbf{x}}) q_j(\hat{\mathbf{x}}) \in \Sigma[\hat{\mathbf{x}}], \end{cases} \tag{8}$$

where $\epsilon, \epsilon' > 0$, the entries of $\sigma_i(\hat{\mathbf{x}})$, $\phi_j(\hat{\mathbf{x}})$ $\kappa_j(\hat{\mathbf{x}}) \in \Sigma[\hat{\mathbf{x}}]$, and $\nu(\hat{\mathbf{x}}, \varepsilon) \in \Sigma[\hat{\mathbf{x}}, \varepsilon]$, and $\lambda(\hat{\mathbf{x}}) \in \mathbb{R}[\hat{\mathbf{x}}]$. Note that ϵ, ϵ' are needed to ensure positivity of polynomials as required in the second and third constraints in (6). The feasibility of the constraints in (8) is sufficient to imply the feasibility of the constraints in (6).

Investigating (8), the product of undetermined coefficient parameters from $\lambda(\hat{\mathbf{x}})$ and $B(\hat{\mathbf{x}}, \mathbf{b})$ in the second constraint makes the problem into a bilinear matrix inequalities (BMI) problem, which can be carried out by calling a Matlab package PENBMI solver [18].

Remark that the existence of the feasible solution $\mathbf{b}^*$ to the problem (8) implies that the system is guaranteed to be safe under the designated controller $u(\mathbf{x}) = p(\mathbf{x}) + k(\mathbf{x})$.

Lyapunov-like Function Computation. We wonder that the learned controller is guaranteed to be not only safe but also goal-reaching in a sense of driving the system to converge to the specified goal set. As stated in Theorem 2, the existence of Lyapunov-like function suffices to prove that the system's behaviors asymptotically converge to the specified goal set X_g. In the similar manner, we first formalize the goal-reaching verification for system $\mathcal{C}$ through Theorem 2 and Theorem 4. Assume that the Lyapunov-like function $V(\hat{\mathbf{x}})$ is a polynomial of degree at most d', whose coefficients form a vector space of dimension $s(d') = \binom{n+1+d'}{d'}$ with the canonical basis $(\hat{\mathbf{x}}^\alpha)$ of monomials. We introduce the coefficient parameters of the Lyapunov-like function $V(\hat{\mathbf{x}})$ denoted as the vector $\mathbf{v} = (v_\alpha) \in \mathbb{R}^{s(d')}$, and write

$$V(\hat{\mathbf{x}}, \mathbf{v}) = \sum_{\alpha \in \mathbb{N}^{n+1}_{d'}} v_\alpha \mathbf{x}^\alpha = \sum_{\alpha \in \mathbb{N}^{n+1}_{d'}} v_\alpha \, x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_{n+1}^{\alpha_{n+1}},$$

in the canonical basis. By Theorem 4, the controlled CCDS $\mathcal{C}$ is goal-reaching under the designed controller can be reduced to that the CCDS $\hat{\mathcal{C}}$ is goal-reaching if there exists such a Lyapunov-like function $V(\hat{\mathbf{x}}, \mathbf{v})$. The existence of Lyapunov-like function can be solved by tackling the following feasibility problem:

$$\begin{cases} \text{find} \quad \mathbf{v} \\ \text{s.t.} \quad \emptyset \neq \{\hat{\mathbf{x}} : V(\hat{\mathbf{x}}, \mathbf{v}) \leq 0\} \subseteq \hat{X}_g, \\ \qquad \mathcal{L}_{\mathbf{f_u}} V(\hat{\mathbf{x}}, \mathbf{v}) \leq -\beta(V(\hat{\mathbf{x}}, \mathbf{v})), \ \ \forall \hat{\mathbf{x}} \in \hat{\Psi}. \end{cases} \tag{9}$$

Similarly, we encode the uncertain variable ε in the range $[-\mu, \mu]$ into $\hat{h}(\varepsilon) \geq 0$ with $\hat{h}(\varepsilon) := (\varepsilon + \mu)(\mu - \varepsilon)$, and ε is involved by the controller $\mathbf{u}$ in the polynomial $\mathcal{L}_{\mathbf{f_u}} V(\hat{\mathbf{x}}, \mathbf{v})$. And for the given goal-reaching set $\hat{X}_g$, the constraint $\{\hat{\mathbf{x}} : V(\hat{\mathbf{x}}, \mathbf{v}) \leq 0\} \neq \emptyset$ can be encoded by $V(\hat{\mathbf{x}}_0, \mathbf{v}) \leq 0$ for a point $\hat{\mathbf{x}}_0 \in \hat{X}_g$.

Depending on the above encoding operations, the problem (9) can be transformed into the following constrained polynomial optimization problem

$$\begin{cases} \text{find} \quad \mathbf{v} \\ \text{s.t.} \quad s_i(\hat{\mathbf{x}}) + \sigma'_i(\hat{\mathbf{x}}) V(\hat{\mathbf{x}}, \mathbf{v}) \in \Sigma[\hat{\mathbf{x}}], \\ \qquad -\mathcal{L}_{\mathbf{f_u}} V(\hat{\mathbf{x}}, \mathbf{v}) - \beta(V(\hat{\mathbf{x}}, \mathbf{v})) - \sum_j \phi'_j(\hat{\mathbf{x}}) h_j(\hat{\mathbf{x}}) - \nu'(\hat{\mathbf{x}}, \varepsilon)\hat{h}(\varepsilon) \in \Sigma[\hat{\mathbf{x}}], \\ \qquad -V(\hat{\mathbf{x}}_0, \mathbf{v}) \in \Sigma[\hat{\mathbf{x}}], \end{cases} \tag{10}$$

where $1 \leq i \leq m_4$, $1 \leq j \leq m_2$, the entries of $\sigma'_i(\hat{\mathbf{x}})$, $\phi'_j(\hat{\mathbf{x}}) \in \Sigma[\hat{\mathbf{x}}]$, and $\nu'(\hat{\mathbf{x}}, \varepsilon) \in \Sigma[\hat{\mathbf{x}}, \varepsilon]$. For the sake of simplicity, we consider the extended class $\mathcal{K}$ function $\beta(\cdot)$ is the $\beta(x) = x$ or $\beta(x) = r \cdot x$ $(r > 0)$.

In summary, the safety and goal-reaching verification problem is transformed into a BMI problem by combining (8,10) for the parameters $\mathbf{b}$ and $\mathbf{v}$. The solution $\mathbf{b}^*$ to problem (8) yields a barrier certificate $B(\hat{\mathbf{x}}, \mathbf{b}^*)$. It means that the closed-loop system under the designed controller $u(\mathbf{x}) = p(\mathbf{x}) + k(\mathbf{x})$ is safe. And the solution $\mathbf{v}^*$ to (10) produces a Lyapunov-like function $V(\hat{\mathbf{x}}, \mathbf{v}^*)$, which means that the system asymptotically converges to the specified goal set X_g.

5 Experiments

In this section we first present a nonlinear system to illustrate our approach, and then report an experimental evaluation of our method over a set of benchmark examples and compare with other two different potential methods. All experiments are conducted on 3.2GHz AMD Ryzen 7 3700X CPU under Windows 10 with 16GB RAM.

Example 1. [Academic 3D Model [6]*]* Consider the following continuous dynamical system in the plant:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} z + 8y \\ -y + z \\ -z - x^2 + u \end{bmatrix}.$$

The system domain is $\Psi = \{\mathbf{x} = (x, y, z)^T \in \mathbb{R}^3 \mid -5 \le x, y, z \le 5\}$. Our goal is to design a control law $u = p(\mathbf{x}) + k(\mathbf{x})$ such that all trajectories of the closed-loop system under u starting from the initial set

$$\Theta = \{\mathbf{x} \in \mathbb{R}^3 \,|(x + 0.75)^2 + (y + 1)^2 + (z + 0.4)^2 \le 0.35^2\}$$

will never enter the unsafe region

$$X_u = \{\mathbf{x} \in \mathbb{R}^3 \,|(x + 0.3)^2 + (y + 0.36)^2 + (z - 0.2)^2 \le 0.30^2\},$$

and eventually enter the goal set $X_g = \{\mathbf{x} \in \mathbb{R}^3 \,|x^2 + y^2 + z^2 \le 0.1^2\}$.

For the controller learning process, we attempt to train different NN structures with increasing depth and width as the controller templates, until a desired controller is obtained. We eventually obtained one DNN controller with 5 hidden layers each consisting of 128 neurons, but failed for smaller sizes. Based on this learned DNN controller, we construct a hybrid controller for the system. The polynomial part $p(\mathbf{x})$ is carried out by the sampling-based method as follows:

$$\begin{aligned} p(\mathbf{x}) = 0.125 \, &-3.333x - 5.726y - 10.669z + 1.911x^2 + 1.212xy \\ &+2.138xz - 1.332y^2 - 10.07yz - 12.952z^2. \end{aligned}$$

The hybrid controller is then constructed as $p(\mathbf{x}) + k(\mathbf{x}|\theta')$ where $k(\mathbf{x}|\theta')$ is a small NN with one hidden layer. After retraining by taking $p(\mathbf{x}) + k(\mathbf{x}|\theta')$ into the system, we obtain the NN part with one hidden layer containing 30 neurons.

Under the hybrid controller $p(\mathbf{x}) + k(\mathbf{x})$, the controlled system can be verified to satisfy the safety and goal-reaching properties by the following barrier certificate $B(\mathbf{x}, \tanh(h(\mathbf{x})))$ and Lyapunov-like function $V(\mathbf{x}, \tanh(h(\mathbf{x})))$ respectively,

$$\begin{cases} B = 0.641x^2 - 0.143xy + 0.554y^2 + \cdots + 0.004 \tanh(h(\mathbf{x})) - 0.353z + 0.061, \\ V = -0.09x^2 - 0.311xy + \cdots + 0.0123 \tanh(h(\mathbf{x})) - 0.033x - 0.024z - 0.01, \end{cases}$$

where $h(\mathbf{x}) = 2.248x^2 + 0.962xy + \cdots - 0.389z + 9.051$.

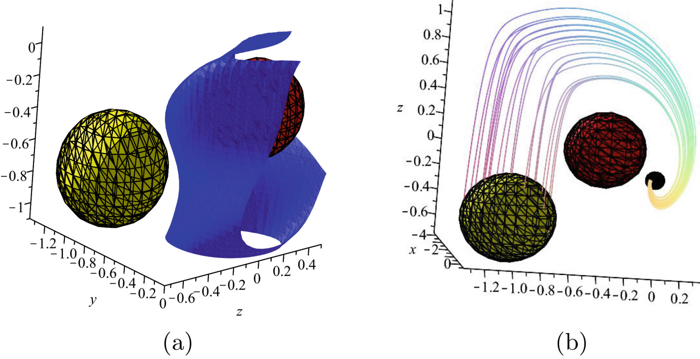


Fig. 3. Phase portrait of the system in Example 1. Subfigure (a) describes the zero level set of the barrier certificate $B(\mathbf{x})$ (the blue surface) separates unsafe region X_u (the red ball) from the initial set Θ(the yellow ball). Subfigure (b) describes all trajectories of different colors from Θ (the yellow ball) can reach X_g (the green ball). (Color figure online)

Figure 3(a) shows the zero level set of the barrier certificate in blue color which separates X_u (the red ball) from all trajectories starting from Θ (the yellow ball), and Fig. 3(b) describes the simulation of different trajectories of the system converges to the goal set X_g (the green ball) under the learned hybrid controller. Therefore, we conclude that the system can be guaranteed to be safe and goal-reachable from the initial set under our learned hybrid controller.

Although DNN policy by RL may appear to work well in many applications, it is difficult to assert any strong and provable claims about its correctness since the neurons, layers, weights and biases are far-removed from the intent of the actual controller. As found in [32], the state-of-the-art neural network verifiers are ineffective for verification of a neural controller over an infinite time horizon with complex system dynamics. So the idea is to learn a controller with formal reasonings of the specified property. *The following part is to conduct the research experiments stated below:*

RE1*: Explore directly learning a polynomial controller to control the system and guarantee its safety and goal-reaching requirements.*

On the verification point, one may think how about directly learning a polynomial controller to control the system (without appealing to the neural policy at all), using reinforcement learning to synthesize its unknown parameters. So the experiment first tried training the controller network with the commonly used Square activation function. Through training on the data set from 250 trajectories with 3000 data points on each, the result was unsuccessful for different network structures (of up to 5 layers and 250 neurons), which means it still fails when simulating the behaviors of the system under the trained polynomial controller. As mentioned in [32], Zhu et al. found that despite many experiments on tuning learning rates and rewards, directly training a linear control program

to conform to their specification with either reinforcement learning (e.g. policy gradient) or random search was unsuccessful because of undesirable overfitting even for an example as simple as the inverted pendulum.

RE2*: Explore the effects of using just a polynomial or a small NN to imitate the original DNN to avoid the hybrid form.*

Our method is based on the RL to obtain a well-performing DNN controller in general form, and then with the guidance of the learned DNN, a hybrid controller is designed which is verifiable for the safety and the goal-reaching properties. The experiment next shows the performance of the hybrid controller synthesis and the comparison of the verification performance with other two RL-guided controller synthesis methods:

(RE2-1) Obtain a polynomial controller by imitating and abstracting the trained DNN controller, and under the guidance of the abstracted polynomial controller the resulting verification of the control system can naturally be encoded to a polynomial constraint solving problem;

(RE2-2) Abstract the DNN controller based on knowledge distillation to obtain a small network that is in simple structure, which is expected to maintain the safety and goal-reaching of the original network (on data set) [11]. Since the posterior verification cannot avoid approximating the neural network with a polynomial, and the upper bound of the error is positively related to the Lipschitz constant, the distilled small network is hopeful to make the verification successful thanks to its smaller Lipschitz constant.

Table 1. Performance Evaluation

Ex	$n_{\mathbf{x}}$	$d_{\mathbf{f}}$	NN *Struc.*		*Hyb. design*		*Poly.* ***(RE2-1)***			*Distil.* ***(RE2-2)***	
			$u_0(\mathbf{x})$	$k(\mathbf{x})$	d_B,d_V	$T_H(s)$	d_P	d_B,d_V	$T_P(s)$	d_B,d_V	$T_D(s)$
C_1 [17]	2	3	2-128(4)-1	2-20-1	2,2	4.953	5	2,2	21.18	2,2	3.507
C_2 [31]	2	3	2-64(4)-1	2-20-1	2,2	4.877	4	2,2	27.61	2,4	8.492
C_3 [32]	2	3	2-64(5)-1	2-20-1	2,2	3.813	×	2,×	×	2,4	10.56
C_4 [3]	2	4	2-64(4)-1	2-20-1	2,2	8.763	5	4,4	82.92	4,2	10.16
C_5 [6]	3	2	3-128(5)-1	3-30-1	2,2	11.70	×	4,×	×	4,×	×
C_6 [6]	3	4	3-128(5)-1	3-30-1	2,2	19.42	5	4,4	103.4	×, ×	×
C_7 [2]	4	1	4-128(5)-2	4-50-2	4,4	49.26	×	×, ×	×	2,×	×
C_8 [17]	4	4	4-128(5)-1	4-40-1	2,2	28.47	6	4,4	229.1	×, ×	×
C_9 [17]	6	3	6-128(5)-2	1-50-2	2,4	64.05	×	×, ×	×	×, ×	×
C_{10} [19]	7	2	7-128(6)-1	7-50-1	2,2	69.73	×	×, ×	×	×, ×	×

We present a detailed experimental evaluation on a set of benchmarks in Table 1. The origins of these 10 widely used examples are provided in the first column; $n_{\mathbf{x}}$ and $d_{\mathbf{f}}$ denote the number of state variables and the maximal degree of the polynomials (or the polynomial abstraction by Taylor model for non-polynomial systems) in the vector fields. The examples are with dimension up

to 7. $u_0(\mathbf{x})$ denotes the network structure of the DNN controller synthesized by RL directly. For example, the trained DNN controller for C_1 has 4 hidden layers with 128 neurons on each. Here, all DNNs are with ReLU activation functions except for tanh on the output layer.

Table 1 has shown the performance of the mentioned three controller synthesis methods with the guidance of the well-trained DNN $u_0(\mathbf{x})$, i.e., hybrid controller design, polynomial controller by imitating (denoted as *Poly.*), NN controller by distillation (denoted as *Distil.*). All the verification process on these methods is carried out through the certificate function generating and the time costs are recorded as T_H, T_P and T_D respectively, when both barrier certificate and Lyapunov-like function have been obtained, and the degrees of the obtained certificate functions are recorded as d_B,d_V; otherwise, '×' is marked when failing to compute any barrier certificate or Lyapunov-like function within the degree bound of 6 and the time bound of 3 hours.

In our hybrid controller design method (i.e., *Hyb. design*), we uniformly choose $p(\mathbf{x})$ of degree 2 and $k(\mathbf{x})$ with one single hidden layer shown in column $k(\mathbf{x})$. d_B and d_V denote the degrees of the computed certificates of barrier function $B(\hat{\mathbf{x}})$ and Lyapunov-like function $V(\hat{\mathbf{x}})$ respectively. T_H in the last column denotes the verification time cost.

The column *Poly.* exhibits the results of the method described in **(RE2-1)** on the benchmarks, intending to further explain the necessity of proposing a hybrid form controller. As an ablation study, we only use polynomial approximations of the original DNNs as surrogate controllers and carry out certificate-based verification of them. Considering the control effect, we increase the degree bound of polynomial templates to 8 to ensure a high precision approximation. d_P denotes the lowest degree of the polynomial surrogate controllers that pass verification and T_P denotes the corresponding time cost; '×' means that no such controller is found. The column *Distil.* provides the results of the method in **(RE2-2)** on the benchmarks. In this ablation study, we have distilled simpler NNs with one single hidden layer from the original DNNs and verify the specified properties using the distilled NN controllers. This process is repeated with the number of neurons of distilled NNs ranging from 20 up to 50 on its hidden layer, until obtaining one satisfying the specified properties whose verification time cost is denoted in T_D, or failing to obtain one such simpler NN, denoted by '×' in T_D.

For all the 10 examples, we have successfully verified the safety and goal-reaching properties of the synthesized hybrid controllers with the certificate generation, while the methods based on polynomial surrogate controllers (i.e., *Poly.*) and distilled NN controllers (i.e., *Distil.*) succeed on 5 and 4 benchmarks, respectively. Moreover, for some examples, *Hyb. design* method can find barrier certificates and Lyapunov-like functions with lower degrees. Consequently, the decision variables of the BMI problems are less than the other methods, which does contribute to improving the effectiveness of the verification procedure.

We compare the efficiency of the methods in terms of the time spent in the verification process for successful examples. On average, the time spent by T_P is 4.3 to 9.5 times as that of T_H on the 5 successful cases of T_P. Meanwhile, the

time cost by T_D is about 8.18 seconds on average, which is 1.46 times more than that of T_H on the four successful cases of T_D. Comparing T_H with T_P and T_D, we conclude that verification of the hybrid controllers is much more efficient.

To summarize, Table 1 shows that all the synthesized hybrid controllers have been efficiently verified to make the systems safe and goal-reachable on a set of commonly used benchmark examples, which demonstrates that our hybrid polynomial-DNN controller synthesis method is quite promising.

6 Conclusion

This paper has presented an approach to synthesize hybrid polynomial-DNN controllers for nonlinear systems such that the closed-loop system can be both well-performing and easily verified upon required properties. Our approach has creatively integrated low degree polynomial fitting and knowledge distillation into RL method during the constructing process. Thanks to the special feature of the hybrid controller, the controlled system can be transformed into the polynomial form. The SOS relaxation based method is applied to generate barrier certificates and Lyapunov-like functions, which can verify the safety and goal-reaching properties of the nonlinear control systems equipped with our synthesized hybrid controllers. Extensive experiments consistently demonstrate the effectiveness and scalability of the proposed approach.

Acknowledgments. This work was supported in part by the National Key Research and Development Project, China under Grant 2022YFA1005100, in part by the National Natural Science Foundation of China under Grants (No. 12171159, No. 62272397, No. 61972385, No. 61902325), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (Grant No. 22510750100).

References

1. Ames, A.D., Coogan, S., Egerstedt, M., Notomista, G., Sreenath, K., Tabuada, P.: Control barrier functions: theory and applications. In: Proceedings of the 18th European Control Conference (ECC), pp. 3420–3431 (2019). https://doi.org/10.23919/ECC.2019.8796030
2. Chan, N., Mitra, S.: Verifying safety of an autonomous spacecraft rendezvous mission. arXiv preprint arXiv:1703.06930 (2017)
3. Chang, Y.C., Roohi, N., Gao, S.: Neural lyapunov control. In: Advances in Neural Information Processing Systems (NeurIPS), pp. 3245–3254 (2019). https://doi.org/10.48550/arXiv.2005.00611
4. Chow, Y., Nachum, O., Duenez-Guzman, E., Ghavamzadeh, M.: A Lyapunov-based approach to safe reinforcement learning. In: Advances in Neural Information Processing Systems (NeurIPS), pp. 8103–8112 (2018). https://doi.org/10.48550/arXiv.1805.07708
5. Delaitre, V., Sivic, J., Laptev, I.: Learning person-object interactions for action recognition in still images. In: Advances in Neural Information Processing Systems (NeurIPS), vol. 24, pp. 234–242 (2011). https://doi.org/10.48550/arXiv.1604.04808

6. Deshmukh, J.V., Kapinski, J.P., Yamaguchi, T., Prokhorov, D.: Learning deep neural network controllers for dynamical systems with safety guarantees. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–7. IEEE (2019). https://doi.org/10.1109/ICCAD45719.2019.8942130
7. Ding, J., Tomlin, C.J.: Robust reach-avoid controller synthesis for switched nonlinear systems. In: 49th IEEE Conference on Decision and Control (CDC), pp. 6481–6486. IEEE (2010). https://doi.org/10.1109/CDC.2010.5717115
8. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 157–168 (2019). https://doi.org/10.1145/3302504.3311807
9. Esfahani, P.M., Chatterjee, D., Lygeros, J.: The stochastic reach-avoid problem and set characterization for diffusions. Automatica **70**, 43–56 (2016). https://doi.org/10.1016/j.automatica.2016.03.016
10. Fan, C., Miller, K., Mitra, S.: Fast and guaranteed safe controller synthesis for nonlinear vehicle models. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 629–652. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_31
11. Fan, J., Huang, C., Li, W., Chen, X., Zhu, Q.: Towards verification-aware knowledge distillation for neural-network controlled systems: invited paper. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8 (2019). https://doi.org/10.1109/ICCAD45719.2019.8942059
12. Fisac, J.F., Chen, M., Tomlin, C.J., Sastry, S.S.: Reach-avoid problems with time-varying dynamics, targets and constraints. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 11–20 (2015). https://doi.org/10.1145/2728606.2728612
13. Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: Reachnn: reachability analysis of neural-network controlled systems. ACM Trans. Embed. Comput. Syst. **18**(5s), 1–22 (2019). https://doi.org/10.1145/3358228
14. Huang, Z., Wang, Y., Mitra, S., Dullerud, G.E., Chaudhuri, S.: Controller synthesis with inductive proofs for piecewise linear systems: an SMT-based algorithm. In: 54th IEEE Conference on Decision and Control (CDC), pp. 7434–7439. IEEE (2015). https://doi.org/10.1109/CDC.2015.7403394
15. Huh, S., Yang, I.: Safe reinforcement learning for probabilistic reachability and safety specifications: a Lyapunov-based approach. arXiv preprint arXiv:2002.10126v1 (2020)
16. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 169–178 (2019). https://doi.org/10.1145/3302504.3311806
17. Jin, W., Wang, Z., Yang, Z., Mou, S.: Neural certificates for safe control policies. arXiv preprint arXiv:2006.08465v1 (2020)
18. Kočvara, M., Stingl, M.: PENBMI user's guide (version 2.0) (2005). http://www.penopt.com
19. Laub, M.T., Loomis, W.F.: A molecular network that produces spontaneous oscillations in excitable cells of dictyostelium. Molecul. Biol. Cell **9**(12), 3521–3532 (1998). https://doi.org/10.1091/mbc.9.12.3521
20. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971v6 (2015)
21. Lyapunov, A.M.: The general problem of the stability of motion. Int. J. Control **55**(3), 531–534 (1992). https://doi.org/10.1115/1.2901415

22. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015). https://doi.org/10.1115/1.4062310
23. Oh, J., Chockalingam, V., Lee, H., et al.: Control of memory, active perception, and action in minecraft. In: International Conference on Machine Learning (ICML), pp. 2790–2799. PMLR (2016). https://doi.org/10.48550/arXiv.1605.09128
24. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Automat. Control **52**(8), 1415–1428 (2007). https://doi.org/10.1109/TAC.2007.902736
25. Richards, S.M., Berkenkamp, F., Krause, A.: The Lyapunov neural network: adaptive stability certification for safe learning of dynamical systems. In: Conference on Robot Learning (CORL), pp. 466–476. PMLR (2018). https://doi.org/10.48550/arXiv.1808.00924
26. Saveriano, M., Lee, D.: Learning barrier functions for constrained motion planning with dynamical systems. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 112–119. IEEE (2019). https://doi.org/10.1109/IROS40897.2019.8967981
27. Srinivasan, M., Dabholkar, A., Coogan, S., Vela, P.A.: Synthesis of control barrier functions using a supervised machine learning approach. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 7139–7145. IEEE (2020). https://doi.org/10.1109/IROS45743.2020.9341190
28. Taylor, A., Singletary, A., Yue, Y., Ames, A.: Learning for safety-critical control with control barrier functions. In: Learning for Dynamics and Control (L4DC), pp. 708–717. PMLR (2020). https://doi.org/10.13140/RG.2.2.21587.60962
29. Wang, L., Theodorou, E.A., Egerstedt, M.: Safe learning of quadrotor dynamics using barrier certificates. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 2460–2465. IEEE (2018). https://doi.org/10.1109/ICRA.2018.8460471
30. Xiang, W., Tran, H.D., Rosenfeld, J.A., Johnson, T.T.: Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In: 2018 Annual American Control Conference (ACC), pp. 1574–1579. IEEE (2018). https://doi.org/10.23919/ACC.2018.8431048
31. Zhao, H., Zeng, X., Chen, T., Liu, Z., Woodcock, J.: Learning safe neural network controllers with barrier certificates. Formal Aspect. Comput. **33**(3), 437–455 (2021). https://doi.org/10.1007/s00165-021-00544-5
32. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 686–701 (2019). https://doi.org/10.1145/3314221.3314638

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Safe Environmental Envelopes of Discrete Systems

Rômulo Meira-Góes[1](✉), Ian Dardik[2], Eunsuk Kang[2], Stéphane Lafortune[3], and Stavros Tripakis[4]

[1] School of EECS, Pennsylvania State University, State College, USA
romulo@psu.edu

[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
{idardik,eunsukk}@andrew.cmu.edu

[3] EECS Department, University of Michigan, Ann Arbor, USA
stephane@umich.edu

[4] Khoury College of Computer Science, Northeastern University, Boston, USA
stavros@northeastern.edu

Abstract. A safety verification task involves verifying a system against a desired safety property under certain assumptions about the environment. However, these environmental assumptions may occasionally be violated due to modeling errors or faults. Ideally, the system guarantees its critical properties even under some of these violations, i.e., the system is *robust* against environmental deviations. This paper proposes a notion of *robustness* as an explicit, first-class property of a transition system that captures how robust it is against possible *deviations* in the environment. We modeled deviations as a set of *transitions* that may be added to the original environment. Our robustness notion then describes the safety envelope of this system, i.e., it captures all sets of extra environment transitions for which the system still guarantees a desired property. We show that being able to explicitly reason about robustness enables new types of system analysis and design tasks beyond the common verification problem stated above. We demonstrate the application of our framework on case studies involving a radiation therapy interface, an electronic voting machine, a fare collection protocol, and a medical pump device.

Keywords: Robustness · Discrete Transition Systems · Model Uncertainty

1 Introduction

A common type of verification task involves verifying a system (C) against a desired property (P) under certain assumptions about the environment (E); i.e., $C||E \models P$. Such assumptions may capture, for example, the expected behavior of a human operator in a safety-critical system, the reliability of the communication

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 326–350, 2023.
https://doi.org/10.1007/978-3-031-37706-8_17

channel in a distributed system, or the capabilities of an attacker. However, the actual environment (E') may occasionally deviate from the original model (E), due to changes or faults in the environment entities (e.g., errors committed by the operator or message loss in the channel). For certain types of deviations, a system that is *robust* would ideally be able to guarantee the property even under the deviated environment; i.e., $C||E' \models P$.

This paper proposes the notion of *robustness* as an explicit, first-class property of a transition system that captures how robust it is against possible *deviations* in the environment. A deviation is modeled as a set of *extra transitions* that may be added to the original environment, resulting in a new, deviated environment E' that has a larger set of behaviors than E does. Then, system C is said to be *robust* to this deviated environment with respect to P if and only if it can still guarantee P even in presence of the deviation. Finally, the overall *robustness* of C with respect to E and P, denoted Δ, is the largest set of deviations that the system is robust against.

Conceptually, Δ defines the safe operating envelopes of the system: As long as the deployment environment remains within these envelopes, the system can guarantee a desired property. Being able to explicitly reason about Δ enables new types of system analysis and design tasks beyond the common verification problem stated above. Given a pair of alternative system designs, C_1 and C_2, one could rigorously compare them with respect to their robustness levels; they both may satisfy property P under the normal operating environment E, but one may be more robust to deviations than the other. Given two properties, P_1 and P_2 (the latter possibly more critical than the former), one could check whether the system would continue to guarantee P_2 under a deviated environment even if it fails to do so for P_1. Finally, given E, P, and a desired level of robustness, Δ, one could *synthesize* machine C to be robust to Δ.

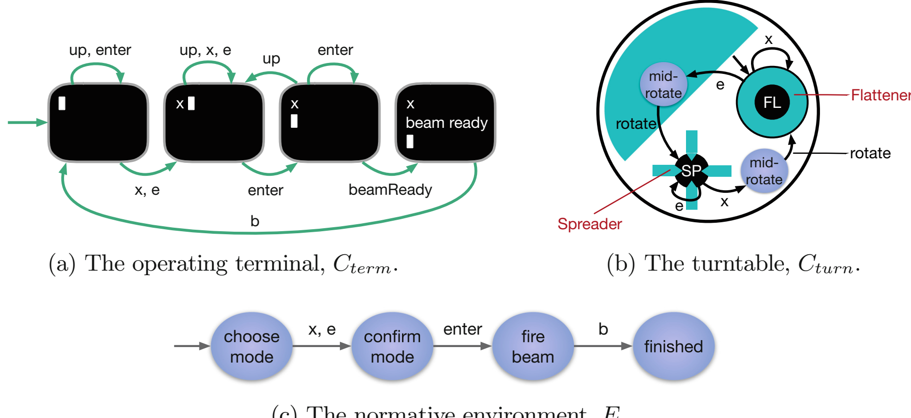
In this paper, we formalize (1) the proposed notion of robustness and (2) the problem of computing Δ for given C, E, and P. One approach to automatically compute Δ is a brute-force method that enumerates all possible sets of deviations; however, as we will show, this approach is impractical, as the number of deviations is exponential in the size of the environment. To mitigate this, we present an approach for computing Δ by reduction to a controller synthesis problem [35,37].

We have built a prototype of the proposed approach for computing robustness and applied it to several case studies, including models of (1) a radiation therapy interface, (2) an electronic voting machine, (3) a public transportation fare collection protocol, and (4) a medical pump device. Our results show that our approach is capable of computing Δ to provide information about deviations under which these systems are able to guarantee their critical safety properties.

The contributions of this paper are as follows: (i) A novel, formal definition of robustness against environmental deviations (Sect. 4); (ii) A simple, brute-force method for computing robustness and a more efficient approach based on controller synthesis (Sect. 5); and (iii) A prototype tool for computing Δ and an experimental evaluation on several case studies (Sect. 6).

2 Motivating Example

As a motivating example, we consider the Therac-25 radiation therapy machine. This machine is infamous for a design flaw that caused radiation overdoses, several of which led to the deaths of patients who received treatment [18]. In this section, we introduce a model for the Therac-25 based on the descriptions in [18] and discuss several methods for analyzing its safety. We show that robustness provides a generally richer analysis than classic verification.



(a) The operating terminal, C_{term}.

(b) The turntable, C_{turn}.

(c) The normative environment, E

Fig. 1. The Therac-25 is modeled as $C_{T25} = C_{term}||C_{beam}||C_{turn}$. C_{beam} is in Fig. 7b.

System. We model the Therac-25 as the composition of the following three finite-state machines: (1) C_{term}, a computer terminal that nurses use to operate the Therac-25, (2) C_{beam}, a beam-emitter that fires a radiation treatment beam in either *X-ray* or *electron* mode, and (3) C_{turn}, a turntable that rotates between two hardware components called the *flattener* and the *spreader*. Formally, we define the Therac-25 as the composition all three machines: $C_{T25} = C_{term}||C_{beam}||C_{turn}$. We show the terminal and turntable in Figs. 1a and 1b respectively. We show the beam in Sect. 6.2 (Fig. 7b), where we present a case study on the Therac-25.

Environment. Nurses operate the Therac-25 by typing at a keyboard connected to a terminal. A nurse begins by choosing a beam mode by typing either an "x" for X-ray or an "e" for electron mode. The nurse then hits the "enter" key and waits for the terminal to display "beam ready" before finally pressing the "b" key to fire the beam. This workflow defines the operating environment which we call E, shown in Fig. 1c.

Safety property. Since the X-ray beams contain a high concentration of radiation, it is imperative that the flattener is in place when the machine fires an X-ray. We capture this key safety property in the following LTL [36] formula:

$$\mathbf{G}(\text{XFIRED} \rightarrow \text{FLATMODE})$$

In this formula, XFIRED is a predicate that is true if an X-ray beam was just fired, while FLATMODE is a predicate that is true when the turn table is in flattener mode. We refer to this safety property as P_{xflat} in this example.

Safety Analyses. Robustness opens our safety analysis beyond classic verification. We discuss several analysis options below.

(1) Standard Verification: We can check that the Therac-25 is safe within the operating environment, that is, $E||C_{T25} \models P_{xflat}$. Standard model checking techniques [2] show that the Therac-25 is indeed safe with respect to E.

(2) Robustness Calculation: Given that the Therac-25 is safe with respect to E, we can calculate its robustness Δ. This calculation identifies the set of safe environmental envelopes of the Therac-25. Importantly, these envelopes reveal the environmental deviations that the Therac-25 can safely handle. For example, in Sect. 6.2, we show that the Therac-25 is robust against the environmental deviations in Fig. 8 in which a nurse repeatedly hits "enter" or the "up" arrow key after choosing a beam mode.

(3) Controller Comparison: Holding the environment E and the property P_{xflat} constant, we can compare the robustness of the Therac-25 against other models. In Sect. 6.2, we introduce the Therac-20 (C_{T20}) and compare the robustness between C_{T25} and C_{T20}. Although both machines are safe with respect to the normative environment, we will find that C_{T25} is strictly less robust than C_{T20}. We will show how contrasting the robustness between the two machines exposes a critical software bug in the Therac-25. Furthermore, we will show that fixing the bug in the Therac-25 causes its robustness to be equivalent to the Therac-20.

(4) Property Comparison: Holding the environment E and the machine C_{T25} constant, we can compare the machine's robustness with respect to P_{xflat} and a second safety property. For example, we could consider a new safety property P' that strengthens P_{xflat} by additionally enforcing the spreader to be in place when a beam is fired in electron mode. The property P' might be of interest to avoid an *underdose*, a situation that might result from the flattener being in place when an electron beam is fired. Because P' is stronger than P_{xflat}, a designer may be interested to compare the robustness between the properties to understand which environmental deviations maintain P_{xflat}, but violate P'.

3 Modeling Formalism

This section describes the underlying formalism used to model the environment, controlled systems, and the properties enforced by them.

Labeled Transition Systems. Given a finite set A, the usual notations $|A|$ and A^* denote the cardinality of A and the set of all finite sequences over A respectively. In this work, we use finite labeled transition systems to model the behavior of the environment, the controller, and the property.

Definition 1. *A labeled transition system (LTS) E is a tuple $\langle Q_E, Act_E, R_E, q_{0,E}\rangle$, where Q_E is a finite set of states, Act_E is a finite set of actions, $R_E \subseteq Q_E \times Act_E \times Q_E$ is the transition relation of E, and $q_{0,E} \in Q_E$ is the initial state.*

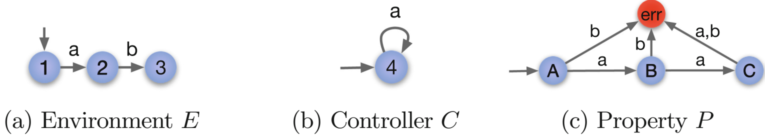
LTS E is said to be deterministic if for any $(q, a, q'), (q, a, q'') \in R_E$, then $q' = q''$; otherwise it is nondeterministic. We extend the transition relation R_E to finite sequences of actions as ${R_E}^* \subseteq Q_E \times {Act_E}^* \times Q_E$ in the usual manner. A *trace* of E is a finite sequence of actions $a_0 \dots a_n$ of E complying with the transition in ${R_E}^*$, i.e., $(q_{0,E}, a_0 \dots a_n, q) \in {R_E}^*$ for some $q \in Q_E$. The set of all traces in E is denoted by $beh(E)$.

Given LTSs E_1 and E_2, the parallel composition $||$ defines standard synchronization of E_1 and E_2 [2,7]. The composed LTS $E_1||E_2 = \langle Q_{E_1} \times Q_{E_2}, Act_{E_1} \cup Act_{E_2}, R_{E_1||E_2}, (q_{0,E_1}, q_{0,E_2})\rangle$ synchronizes over the common actions between E_1 and E_2 and interleaves the remaining actions. Lastly, given LTSs E_1 and E_2, we say that E_1 is a subset of E_2, denoted $E_1 \subseteq E_2$, if $Q_{E_1} \subseteq Q_{E_2}$, $Act_{E_1} = Act_{E_2}$, $R_{E_1} \subseteq R_{E_2}$, and $q_{0,E_1} = q_{0,E_2}$.

Control Strategy. Let an LTS E represent the environmental model to be controlled. A control strategy, or simply *controller*, for E is a function that maps a finite sequence of actions to a set of actions, i.e., $C : {Act_E}^* \rightarrow 2^{Act_E}$. A *controlled trace* of E is a trace of E, $a_0 \dots a_n \in beh(E)$, such that $a_i \in C(a_0 \dots a_{i-1})$ for any $i \leq n$. The set of all controlled runs, denoted by $beh(E/C)$, defines the closed-loop system of C controlling E. For convenience, this closed-loop system is denoted by E/C. In this work, we assume that controller C has finite memory and it can be represented by a deterministic LTS. With an abuse of notation, the LTS controller representation is also denoted by C. For convenience, we define controller $C = \langle Q_C, Act_C, R_C, q_{0,C}\rangle$ to have the same actions as in E, i.e., $Act_C = Act_E$. In this manner, the closed-loop system E/C can be represented by the composition of environment E and controller C: $E/C = E||C$.

Remark 1. We assume that all elements of the set of actions Act_E are "controllable" actions, that can be acted upon by a controller. However, the nondeterministic transition relation of E can be used to model uncontrollable actions of the environment. After an action a is selected by the controller at state q, the environment decides which state the system will be in, similarly to two-player games [15].

Safety Property. In this work, we consider a class of regular linear-time properties called safety properties over an environment E [2]. A safety property P is represented by a deterministic LTS P that defines the set of accepted behaviors. Usually, the LTS P encodes both the traces that satisfy P and those that violate it by including a sink error state. Formally, any trace that reaches the error state $err \in Q_P$ violates the safety property. An LTS E satisfies property P, denoted by $E \models P$, whenever the traces in $beh(E)$ do not reach the error state in P. In this manner, we can test if $E \models P$ by composing $E||P$ and investigating if the err is reached.



(a) Environment E (b) Controller C (c) Property P

Fig. 2. LTSs for the running example

Example 1. We describe a simple example that we use as a running example throughout the paper. Figure 2 depicts the environment E, controller C, and property P considered in this example. The environment E defines that action a is immediately followed by action b. Although controller C in Fig. 2b only shows action a, we assume that $Act_C = \{a, b\}$. In this manner, C only allows action a to occur. Lastly, property P defines that action a should happen at most two times while action b should never happen. It follows that $E/C \models P$ since the controller disables action b and the environment only executes one instance of action a.

4 Robustness Against Environmental Deviations

4.1 Deviations

A deviation is a set of transitions $d \subseteq (Q_E \times Act_E \times Q_E)$ A *deviated system* is defined by augmenting the transitions of environment E with a deviation set:

Definition 2. *Given an LTS* $E = \langle Q_E, Act_E, R_E, q_{0,E}\rangle$ *and a deviation* $d \subseteq Q_E \times Act_E \times Q_E$*. We define the* deviated system E_d *as* $E_d := \langle Q_E, Act_E, R_E \cup d, q_{0,E}\rangle$*.*

A controller C that guarantees property P for environment E, i.e., $E/C \models P$, might violate this property for the deviated environment E_d, i.e., $E_d/C \not\models P$.

Definition 3. *Controller* C *is a* robust controller *with respect to environment* E*, deviation* d*, and property* P *if* $E_d/C \models P$*. Deviation* d *is a* robust deviation *with respect to* E*,* C*, and* P *if* C *is a robust controller with respect to* E*,* d*, and* P*.*

Remark 2. In this paper, we are only interested in ensuring safety properties over the controlled system. For this reason, it is sufficient to only consider adding new transitions to the environment. If a controlled system is safe, then deleting transitions from the environment does not violate the safety property.

4.2 Comparing Deviations

Each deviation set affects the environment in different ways. To reason about the effects of each deviation set, we compare them using a partial order relation over $Q_E \times Act_E \times Q_E$. For deviations d_1 and d_2 such that $d_1 \subseteq d_2$, d_2 deviates

LTS E more than d_1 since $beh(E_{d_1}) \subseteq beh(E_{d_2})$. For this reason, we select the relation $\subseteq$ over $Q_E \times Act_E \times Q_E$ to be the partial order to compare different deviation sets.

Definition 4. *Given E and deviations d_1, d_2, d_1 is* at least as powerful *as d_2 if $d_2 \subseteq d_1$.*

4.3 Robustness

Intuitively, robustness is defined as the set of all possible robust deviations d with respect to the environment E, controller C, and safety property P_{saf}. Additionally, we introduce an environmental constraint, P_{env}, to capture domain knowledge about the system under analysis. P_{env} will filter environment deviations that might not be physically feasible or of interest to analyze. This constraint is captured as a safety property over E, i.e., $E \models P_{env}$ states that the environment satisfies the constraint. Formally, our robustness notions is defined as follows:

Definition 5. *Let environment E, controller C, property P_{saf} such that $E/C \models P_{saf}$, and environment constraint P_{env} such that $E \models P_{env}$ be given. The robustness of controller C with respect to E, P_{saf}, and P_{env}, denoted by $\Delta(E, C, P_{saf}, P_{env})$, is a set of robust deviations $\Delta \subseteq 2^{Q_E \times Act_E \times Q_E}$. Δ is defined to be the (unique) set of robust deviations satisfying the following conditions:*

1. $\forall d \in \Delta.\ E_d/C \models P_{saf}$ [d is robust];
2. $\forall d \subseteq Q_E \times Act_E \times Q_E . E_d/C \models P_{saf} \wedge E_d \models P_{env} \Rightarrow \exists d' \in \Delta . d \subseteq d'$ [d is represented];
3. $\forall d, d' \in \Delta.\ d \neq d' \Rightarrow d \not\subseteq d'$ [unique representation].
4. $\forall d \in \Delta.\ E_d \models P_{env}$ [d is feasible].

When E, C, P_{saf}, and P_{env} are clear from context, we simply write Δ. The set Δ is also denoted as the safety envelope of C with respect to E, P_{saf}, and P_{env}.

Intuitively, the set Δ defines an upper bound on the possible deviations from E that controller C is robust against. In other words, Δ captures the envelopes for which controller C remains safe.

If a designer does not have domain knowledge about the system, then P_{env} can be set to not constrain the environment, i.e., $P_{env} = Act_E^*$. After computing Δ without environmental constraints, a designer can obtain important information about the system and the environment. In the next analysis iteration, this knowledge can be transformed into environmental constraints to enhance the robustness analysis, i.e., $P_{env} \subseteq Act_E^*$.

By definition, Δ is always non-empty since $d = \emptyset$ is always robust. Moreover, due to conditions 2 and 3, only maximal robust deviations are included in Δ. We show that there is a unique set of deviations that satisfies the conditions of Def. 5. The proof of this lemma is available at [27], pg. 23.

Lemma 1. *Given LTS E, controller C, safety property P_{saf}, and environment property P_{env}, there is a unique Δ that satisfies the conditions in Def. 5.*

Example 2. Back to our running example, we investigate robust deviations and Δ. For simplicity, we do not impose any environment constraint, i.e., $P_{env} = Act_E^*$. Figure 3 shows four robust deviations for our running example, where transitions in green are deviations added to the environment. All robust deviations allow at most two transitions with action a, which is the maximum number allowed by the property. In this example, Δ has three robust deviations that are represented in Figs. 3b–3d. Since the robust deviation shown in Fig. 3a is a subset of both deviations in Fig. 3b and Fig. 3c, it is not included in Δ.

(a) A robust deviated environment

(b) Maximal robust deviated environment

(c) Maximal robust deviated environment (d) Maximal robust deviated environment

Fig. 3. Robust deviated environments. Robust transitions $Q_E \times \{b\} \times Q_E$ are omitted.

4.4 Problem Statement

Although Def. 5 has formally introduced our notion of robustness, it does not show how to compute robustness. Therefore, we investigate the problem of computing the set Δ.

Problem 1. Given E, C, P_{saf}, and P_{env} as in Def. 5, compute Δ.

4.5 Comparing Robustness

Our robustness definition also allows us to compare the robustness between different controllers as well as different safety properties.

Comparing Controllers. Holding the environment and safety property constant, we can compare the robustness of the controllers.

Definition 6. *Given an environment E, controllers C_1 and C_2, safety property P_{saf}, and environment constraint P_{env}, controller C_1 is at least as robust as C_2 if and only if for all $d_2 \in \Delta(E, C_2, P_{saf}, P_{env})$ there exists $d_1 \in \Delta(E, C_1, P_{saf}, P_{env})$ such that $d_2 \subseteq d_1$. Equality and strictly less/more robust are defined in the usual manner using $\subseteq$.*

Comparing Safety Properties. Holding the environment and controller constant, we can compare the robustness between safety properties.

Definition 7. *Given an environment E, controllers C, safety properties $P_{saf,1}$ and $P_{saf,2}$, and environment constraint P_{env}, controller C is at least as robust with respect to $P_{saf,1}$ than with respect to $P_{saf,2}$ if and only if for all $d_2 \in \Delta(E, C, P_{saf,2}, P_{env})$, there exists $d_1 \in \Delta(E, C, P_{saf,1}, P_{env})$ such that $d_2 \subseteq d_1$.*

5 Computing Robustness

This section presents two manners of solving Problem 1. One is a brute-force algorithm whereas the second uses control techniques to obtain the solution. Usually when dealing with regular safety properties, one transforms the safety property into an invariance property. This transformation is simply obtained by composing the environment with the safety property; then, an invariance property equivalent to the safety is defined over this composed system [2]. In this composed system, an invariance property is simply defined by a set of safe states. Unfortunately, computing robustness for safety properties does not directly reduce to computing robustness for invariance properties.

When transforming a safety property P_{saf} to an invariance property, we compose the environment and the safety property. Let us assume that there are no environmental constraints. In our scenario, the invariance property P_{inv} is defined based on the composed system $E||C||P_{saf}$, i.e., $P_{inv} \subseteq Q_{E||C||P_{saf}}$. The composed system P_{inv} introduces memory to the environment to differentiate when the safety property is violated or not. This memory addition prevents a simple reduction between invariance and safety properties since robustness is defined with respect to the environment. Robustness defines new transitions in E whereas computing robustness with respect to P_{inv} defines new transitions in $E||C||P_{saf}$. For this reason, we cannot simply reduce the problem of computing Δ with respect to safety properties to the problem of computing Δ with respect to an invariance property.

5.1 Brute-Force Algorithm

One way of solving Problem 1 is via a brute-force algorithm. Intuitively, this algorithm is broken into two parts: (i) finding the set of robust deviations that satisfy the environmental constraint, and (ii) identifying the maximal ones within this set. In part (i), we verify $E_d||C \models P_{saf}$ and $E_d \models P_{env}$ for all deviations $d \subseteq (Q_E \times Act_E \times Q_E) \setminus R_E$, which can be solved using standard model checking techniques [2]. Since this algorithm checks if every deviation set is robust or not, it is clear that it computes Δ.

5.2 Controlling the Deviations Without Environmental Constraints

Due to the lack of scalability of the brute-force algorithm, we search for more efficient ways to compute Δ. For readability purposes, we start by describing our

algorithm in detail assuming no environmental constraints, i.e., unconstrained environment $P_{env} = Act_E^*$. In the next section, we show how to use this algorithm to completely solve Problem 1, i.e., for a possibly constrained environment $P_{env} \subseteq Act_E^*$.

Overview of the Control Algorithm. At a high level, we transform the problem of computing Δ to a problem of controlling environmental transitions to avoid safety violations. Intuitively, we control deviations to force them to be robust, i.e., we take the viewpoint that we can control transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$. Different ways of controlling transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$ provide different robust deviations.

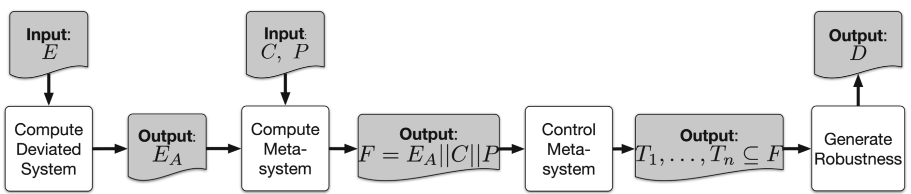


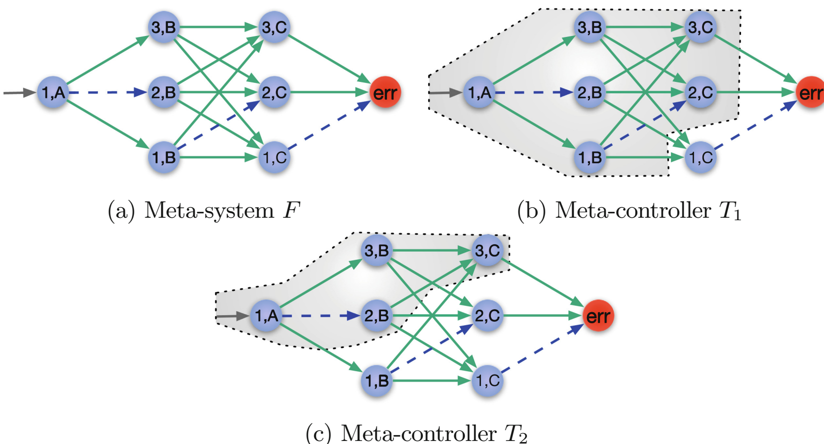
Fig. 4. Overview of our approach to compute robustness for the unconstrained environment. The inputs are the LTSs of environment E, controller C, and property P_{saf}. The set A is the set of all environment transitions, $A = Q_E \times Act_E \times Q_E$. The LTSs $T_1, \ldots, T_n \subseteq F$ represent controlled meta-systems.

Figure 4 provides an overview of our approach. First, we define LTS E_A to be the deviated system with all possible transitions, i.e., $A = Q_E \times Act_E \times Q_E$. The deviated system E_A is the maximally deviated environment since it encompasses every possible deviated system E_d for $d \subseteq Q_E \times Act_E \times Q_E$.

Next, we compose the deviated environment E_A with controller C and property P_{saf}, to create a "meta-system" F. This meta-system provides information about how the deviated environment E_A under the control of C can violate P_{saf}. Following this composition, we pose a control problem over the meta-system to prevent any violation of P_{saf}. There are multiple ways of controlling this composed system; in our approach, we obtain a finite number of controllers encoded as $T_i \subseteq F$. These different ways of controlling the meta-system provide different robust deviations from which we can extract Δ. To make our approach concrete, we describe each step in detail using our running example, shown in Fig. 2.

Constructing the Meta-system. The deviated environment $E_A = E_{Q_E \times Act_E \times Q_E}$ contains the behavior of any other deviated environment. Therefore, we define the meta-system to be the composition of deviated environment E_A, controller C, and property P_{saf}, i.e., $F = E_A||C||P_{saf}$. Figure 5a shows the meta-system F for our running example. Since C only has one state, we omit its state from the state names in Fig. 5a, i.e., states in Fig. 5a are defined as $(q_e, q_p) \in Q_E \times Q_{P_{saf}}$ instead of $(q_e, q_c, q_p) \in Q_E \times Q_C \times Q_{P_{saf}}$. All transitions in F are labeled a, omitted in Fig. 5a, since controller C only enables action a. We

also identify in F which transitions are derived from the environment (dashed blue) and which are derived from deviations (green). For simplicity, we define a single error state in F to capture every $(q_e, q_c, err) \in Q_E \times Q_C \times Q_{P_{saf}}$.



(a) Meta-system F (b) Meta-controller T_1

(c) Meta-controller T_2

Fig. 5. Meta-systems. All transitions have action a since C only enables action a (see Fig. 2b). Dashed blue transitions represent transitions that are feasible in R_E while solid green transitions represent the deviated transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$. The shaded area in Fig. 5b contains all safe states in the meta-system.

Controlling the Meta-system. Once the meta-system is constructed, we pose a meta-control problem over F to ensure that the meta-system avoids the error states, i.e., states $(q_e, q_c, err) \in Q_E \times Q_C \times Q_{P_{saf}}$. These error states represent safety violations in the closed-loop system. For instance, in Fig. 5a, if transition $(2, C) \rightarrow err$ occurs, then the closed-loop system violates P_{saf} since more than two actions a were executed. In this meta-control problem, a meta-controller can disable transitions in F that originated from deviations in E, i.e., transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$.

Problem 2. Given meta-system F, synthesize a meta-controller $T \subseteq F$ such that (1) for any $(q_e, q_c, q_p) \in Q_T$ then state $q_p \neq err$; and (2) for any $\big((q_e, q_c, q_p), a, (q'_e, q'_c, q'_p)\big) \in R_F \setminus R_T$ such that $(q_e, q_c, q_p) \in Q_T$, it follows that $(q_e, a, q'_e) \notin R_E$.

Problem 2 states that the meta-controller is a subset of the meta-system F. We want to maintain the same structure as in F since we need to enforce that the meta-controller does not disable any transition associated with R_E. Condition (1) in Problem 2 ensures that property P_{saf} is not violated. On the other hand, condition (2) guarantees that only transitions assigned to deviations are disabled.

Back to our example, the LTS T described by the shaded area in Fig. 5b demonstrates a possible meta-controller that satisfies Problem 2. Condition (1) is satisfied since the error state is not included in the shaded area. With respect to condition (2), only solid green transitions are disabled. Figure 5c shows another meta-controller.

To solve Problem 2, one can solve a safety game over F using fixed-point computation [15,25]. Due to space limitations, we point the reader to [27], pg. .23 for the solution to this safety game.

Extracting Robust Deviations. Each meta-controller that solves Problem 2 relates to a robust deviation. Intuitively, a meta-controller disables deviations that would violate P_{saf}. For instance, the meta-controller T_1 shown in Fig. 5b disables transition $(3, B) \rightarrow (1, C)$, which relates to disabling transition $3 \xrightarrow{a} 1$ in the environment. Figure 3a depicts the deviated environment related to meta-controller T_1. Similarly, Fig. 3b shows the deviated environment associated with meta-controller T_2.

To extract a robust deviation from a meta-controller, we have to (1) identify the transitions that the meta-controller has disabled; and (2) project the disabled transitions to transitions $Q_E \times Act_E \times Q_E$. Since a meta-controller is a subset of the meta-system, the disabled transitions are obtained by comparing F and T. Intuitively, the disabled transitions are those that escape the shaded area in Fig. 5.

$$Disabled := \{(q,\ a,\ q') \in R_F \mid q \in Q_T\ \wedge\ (q,\ a,\ q') \notin R_T\} \tag{1}$$

For instance, in the case of meta-controller T_1, the transition $((1, B), a, (1, C))$ belongs to the *Disabled* set. Next, based on the disabled transitions, we project them to transitions in $Q_E \times Act_E \times Q_E$, i.e., transitions in the environment.

$$del := \{(q_e,\ a,\ q_e') \in Q_E \times Act_E \times Q_E \mid ((q_e, q_c, q_p), a, (q_e', q_c', q_p')) \in Disabled\} \tag{2}$$

Transitions in *del* are the transitions to be deleted from $Q_E \times Act_E \times Q_E$ such that $(Q_E \times Act_E \times Q_E) \setminus del$ is a robust deviation set. If transitions in *del* are included in a deviation set, they can cause a violation of property P_{saf}. In the case of T_1, the transition $(1, a, 1)$ is included in *del*. If we maintain, for instance, transition $1 \xrightarrow{a} 1$ as part of a deviation set d, then the closed-loop E_d/C violates the property P_{saf} since the path $(1, A) \rightarrow (1, B) \rightarrow (1, C) \rightarrow err$ would be feasible in the meta-controller.

Computing Robustness $\boldsymbol{\Delta}$. Problem 2 searches for meta-controllers that guarantee the satisfaction of property P_{saf}. To compute Δ, we need to obtain a finite number of meta-controllers. Algorithm 1 formalizes our description in Fig. 4. It takes as input the environment E, the controller C, a deviation set d, and a safety property P. From the algorithm overview description in Fig. 2, we have that for the unconstrained environment $d = A = Q_E \times Act_E \times Q_E$ and $P = P_{saf}$.

In Algorithm 1, line 4 computes the largest possible set of invariant states that avoid the error state, i.e., $Inv(Q_F \setminus Err)$ solves the safety game as shown

Algorithm 1. COMPUTE-ROBUSTNESS

Input: LTSs E, C, P and deviation d
Output: Set of deviations D
1: $D \leftarrow \emptyset$
2: $F \leftarrow E_d||C||P$
3: $Err \leftarrow \{(q_e, q_c, q_p) \in Q_F \mid q_p = err\}$
4: $W \leftarrow Inv(Q_F \setminus Err)$
5: **for all** $S \in 2^W \setminus \{\emptyset\}$ **do**
6: $T \leftarrow$ META-CONTROLLER(S, F)
7: $del \leftarrow \{(q_e, a, q_e') \in d \mid \exists((q_e, q_c, q_p),\ a,\ (q_e', q_c', q_p')) \in R_F \setminus R_T$ s.t. $(q_e, q_c, q_p) \in Q_T\}$
8: $D \leftarrow D \cup \{d \setminus del\}$
9: **while** $\exists d_1, d_2 \in D$ s.t. $d_1 \subseteq d_2$ **do**
10: $D \leftarrow D \setminus \{d_1\}$
return D
11: **procedure** META-CONTROLLER(S, F)
12: $S \leftarrow Inv(S)$
13: **if** $q_{0,F} \notin S$ **then**
14: $T \leftarrow \emptyset$
15: **else**
16: $Q_T \leftarrow S$, $Act_T \leftarrow Act_F$, $q_{0,T} \leftarrow q_{0,F}$
17: $R_T \leftarrow \{(q, a, q') \in S \times Act_T \times S \mid (q, a, q') \in R_F\}$
return T

in [27], pg. 23. Based on this invariant set, each iteration in the loop (lines 5–8) computes a meta-controller (line 6) and stores its respective robust deviation (line 8). The meta-controller T is also computed by using the function Inv. The meta-controller solution ensures that $Q_T \subseteq S$. Line 7 computes environmental transitions that must be deleted in order to obtain a robust deviation. The computed robust deviations are stored in Δ. Lastly, the loop in lines 9–10 ensures that only maximal robust deviations are included in Δ.

In more detail, to solve Problem 2, we must guarantee that the meta-system F does not reach any states in $Err := \{(q_e, q_c, q_p) \in Q_F \mid q_p = err\}$. Formally, we compute the set $Inv(Q_F \setminus Err)$, which contains every state in F that does not reach a state in Err via a transition associated with R_E. Based on this invariant set, we can extract any meta-controller that remains within this set. Informally, the META-CONTROLLER(S, F) in line 11 of Algorithm 1 computes a meta-controller that remains within states in S. First, this procedure computes the invariant set of S, i.e., $Inv(S)$ with respect to meta-system F (line 12). In this manner, a meta-controller is defined by projecting the meta-system F to states and transitions in the set of state $Inv(S)$ (lines 16–17).

The following theorem shows that Δ computed via Algorithm 1 is equal to Δ as in Definition 5 when $P_{env} = Act_E^*$, i.e., Algorithm 1 *partially* solves Problem 1.

Theorem 1. *Given LTS E, controller C, and property P_{saf}, Algorithm 1 outputs Δ as in Definition 5 when $P_{env} = Act_E^*$.*

Proof. Sketch. In order to show that Theorem 1 holds, we provide two intermediate lemmas whose proofs are available at [27], pg. 24 (Lemma 2 and Lemma 3). The first lemma states that every meta-controller T produces a robust deviation. In this manner, we show that for every $d \in \Delta$, the deviation d is robust. The second lemma shows that for every maximal robust deviation $d \in \Delta$, there exists a meta-controller T associated with deviation d. Consequently, Algorithm 1 computes every possible maximal robust deviation.

Using Algorithm 1 to compute Δ for our running example, we obtain Δ that contains the three maximal robust deviations shown in Fig. 3. Lastly, we provide the computational complexity of Algorithm 1.

Theorem 2. *Algorithm 1 outputs* Δ *in* $O(2^{|Q_E||Q_C|(|Q_P|-1)})$.

Proof. It follows from the size of 2^W.

Although Algorithm 1 has exponential complexity, we empirically show in Sect. 6 that it scales better than the brute-force algorithm.

Heuristics to Exploit the Structure of F. In Algorithm 1, we compute robust deviations for every possible subset of the largest invariant state set, c.f., line 5. To improve the efficiency of Algorithm 1, we provide a sound and complete heuristic that identifies and skips redundant subsets of $2^W \setminus \emptyset$. The heuristic is based on the observation that sets of states that are not directly connected in F correspond to redundant deletion sets from $Q_E \times Act_E \times Q_E$. As such, the heuristic exploits the structure of F by performing a depth-first search over its state space, hence skipping disconnected groups of states. For instance, the heuristic will skip the subset $\{(1, A), (3, C)\}$ because $(1, A)$ and $(3, C)$ are not connected in F. This subset is redundant because its deletion set $del = \{((1, A), (1, B)), ((1, A), (2, B)), ((1, A), (3, B))\}$ is identical to the deletion set for the subset $\{(1, A)\}$ which is connected. In the worst-case scenario, our heuristic computes the power set of W, i.e., exactly as in line 5.

5.3 Controlling the Deviations with Environmental Constraints

When introducing environmental constraints, we must eliminate the robust deviations that violate these constraints as described in Definition 5. One might think that P_{env} and P_{saf} could be combined as a single safety property for which we then compute Δ. However, this approach does not work since P_{env} must be enforced only by the environment whereas P_{saf} is a property of the closed-loop system. Another approach is to verify if P_{env} is satisfied for each deviation obtained in the for-loop (lines 5–8) in Algorithm 1. Although this approach is feasible, in practice, we want to reduce the number of deviations, using P_{env}, before we compute the robust deviations. For this reason, we describe a sequential algorithm shown in Fig. 6. In this algorithm, Algorithm 1 is used multiple times in this constrained scenario instead of a single time as in the unconstrained scenario (Sect. 5.2).

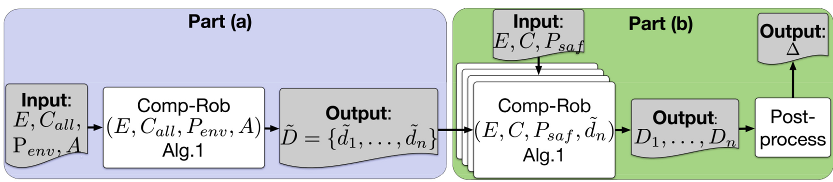


Fig. 6. Overview of our approach to compute robustness for constrained environments.

The algorithm to compute robustness for constrained environments can be broken into two parts: (a) computing all maximal environments $\tilde{d}_i$ that satisfy P_{env}; and (b) computing robust deviations for each deviated environment $E_{\tilde{d}_i}$ found in part (a). Computing the maximal environments that satisfy P_{env} reduces to computing maximal deviations of E with respect to a controller that allows every environment action, C_{all}. Formally, the behavior of C_{all} does not restrain E, $beh(C_{all}) = Act_E^*$; and it can be described by a one-state LTS. Therefore, the output of part (a) is the set of maximal deviations $\tilde{d}_i$ with respect to E, C_{all}, and P_{env}, denoted as maximal environment deviations. Each maximal deviated environment $E_{\tilde{d}_i}$ satisfy the P_{env}.

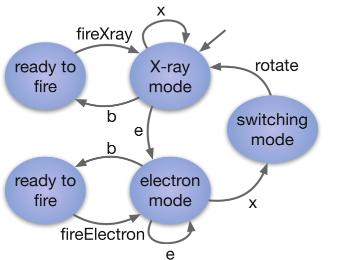
Once we have obtained all maximal environment deviations that satisfy P_{env}, we focus on finding the maximal robust deviations with respect to C and P_{saf}. In other words, we run Algorithm 1 for each maximal deviated environment $E_{\tilde{d}_i}$ together with C and P_{saf}. Since d is a subset of $\tilde{d}_i$, we have that the perturbed system E_d satisfies P_{env}.

Each maximal deviated environment $E_{\tilde{d}_i}$ generates a set of maximal robust deviations D_i with respect to C and P_{saf}. The final step is combining these maximal robust deviations with respect to each $\tilde{d}_i$. Since they are maximal with respect to $\tilde{d}_i$, there could be deviations that are not maximal as defined by Definition 5. The post-processing step combines the deviations and eliminates any non-maximal deviations; and it outputs Δ as in Definition 5. The correctness of this algorithm follows from Theorem 1.

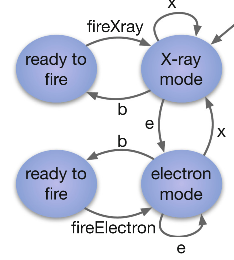
6 Case Studies

6.1 Implementation

We have implemented a prototype tool for computing robustness [28]. The tool accepts a model of an environment, a controller, and a safety property–as well as an optional list of environmental constraints–and outputs Δ. The tool has support for comparing the robustness of two controllers as well as the robustness of a controller with respect to two separate safety properties. Currently, the environment, controller, safety property, and environmental constraints must be encoded in Finite State Process (FSP) notation [23] but this is not a fundamental limitation.



(a) The beam C'_{beam} *with* hardware interlocks used in the Therac-20.



(b) The beam C_{beam} *without* hardware interlocks used in the Therac-25.

Fig. 7. The beam components of the two Therac machines. The hardware interlocks cause C'_{beam} to have a fifth state "switching mode" that will only switch to X-ray mode after the flattener rotates into place.

We wrote the tool in the Kotlin programming language. Our tool includes an implementation of the brute-force algorithm from Sect. 5.1, as well as an implementation of Algorithm 1 and Algorithm 1 with heuristics. In the following case studies, we leverage the tool to calculate and compare the robustness of several systems. We summarize our performance results for each case study in Sect. 6.6.

6.2 Therac-25

Background. In Sect. 2, we introduced the Therac-25 radiation therapy machine. In this section, we present a case study in which we compare the robustness of the Therac-25 to that of its predecessor, the Therac-20. We begin by showing that the Therac-20 is strictly more robust than the Therac-25. We then use this information to identify and fix a critical safety bug in the Therac-25 model.

Therac-20. The Therac-20 is a radiation therapy machine that was designed before the Therac-25. Unlike the Therac-25, the Therac-20 was not known for causing accidents that led to injuries and death. A key difference between the two machines is that the Therac-20 includes hardware *interlocks* in its beam component (Fig. 7a), while the Therac-25 does not (Fig. 7b). The purpose of the hardware interlocks is to provide a layer of security at the hardware level for upholding P_{xflat}. In our model, the interlocks work by ensuring that the flattener is completely rotated into place before allowing an operator to fire an X-ray beam. Unfortunately, hardware interlocks were considered expensive so they were omitted from the design of the later Therac-25 model. In the following section, we compare the robustness between the two Therac machines with respect to the normative environment E and the key safety property P_{xflat}.

Comparing Controllers. Using standard model checking techniques [2], we can confirm that both the Therac-20 and the Therac-25 are safe with respect

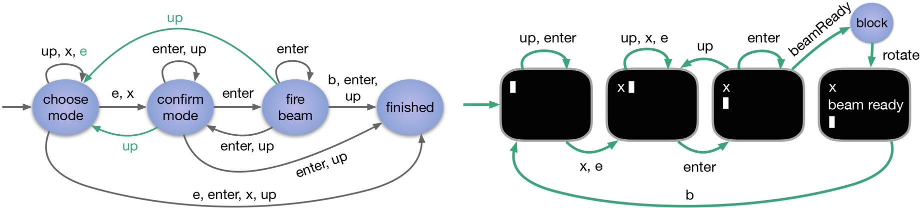


Fig. 8. Visual robustness comparison between the two Therac machines. Both machines are robust against gray transitions, but only the Therac-20 is robust against green transitions. (Color figure online)

Fig. 9. Software fix that eliminates the race condition in the Therac-25.

to E and P_{xflat}. Historically, however, the Therac-20 is known to be safer than the Therac-25. Therefore, we improve our safety analysis by also comparing the robustness between the two machines with respect to E, P_{xlfat}, and an environmental constraint P_{env}. P_{env}, shown in [27], pg. 26, Fig. 11, restricts the environment to firing the beam at most once.

Our tool reports that the Therac-20 is strictly more robust than the Therac-25. To understand this result, we can examine the difference between the robustness for each machine. We show this difference visually by presenting one maximal robust deviation from each machine in Fig. 8. This figure shows that the Therac-20 is robust against the scenario in which the operator 1) types "e" to select electron beam mode, 2) optionally types "enter", 3) presses the "up" arrow key, and finally 4) types "x" to switch the beam into X-ray mode. The Therac-25, however, is not robust against this scenario. We see this in Fig. 8 because the series of actions must pass through at least one green arrow, where a green arrow indicates a transition that the Therac-25 is not robust against. In fact, the Therac-25 does not have *any* maximal robust deviations that allow this scenario.

The Therac-25's lack of robustness to the scenario above represents a race condition that occurs after the operator switches into X-ray mode from electron mode. In this scenario, if the operator types "enter" and fires the X-ray beam before the flattener rotates into place, the beam will fire an unflattened X-ray at the patient. This critical bug was responsible for real-world radiation overdoses, several of which resulted in death [18].

Fixing the Software Bug. In the previous section, we identified a critical software bug in the Therac-25. Our goal in the current section is to fix this bug entirely in the terminal software, thus avoiding an expensive hardware solution.

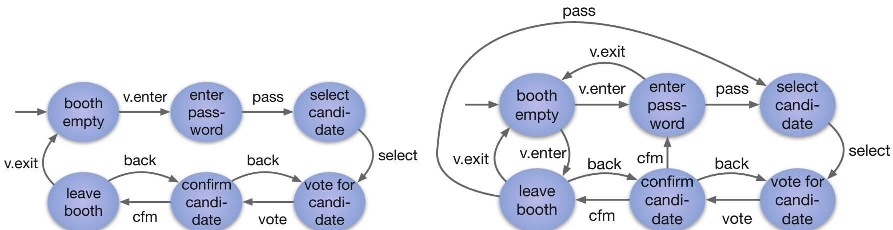
In Fig. 7a, we see that the hardware interlocks prevent a race condition by blocking the operator from typing a "b" until the flattener is rotated into place. Thus we can fix the race condition in software by altering the terminal to block the operator from typing a "b" until the flattener is rotated into place. We implement this fix by redesigning the terminal to block all key strokes from

the instant it issues a "beam ready" message until the turntable rotates into place, as shown in Fig. 9. Finally, we use our tool to evaluate the robustness of the fix. The tool reports that the fixed Therac-25 design is strictly more robust than the original, and equally robust to the Therac-20.

6.3 Voting

Background. In this section, we consider a case study of an electronic voting machine, introduced in [46]. In this case study, we model the voting machine, a voter, and a corrupt election official who attempts to "flip" the voter's choice. We define the voting machine as the composition of a voting booth and a user interface, shown at [27], pg. 26 in Fig. 12a and Fig. 12b respectively.

In the normative environment–shown in Fig. 10a–the voter enters the booth, enters their password, selects a candidate, clicks the vote button, and finally confirms the choice. Unfortunately, some voters may inadvertently skip the confirmation step and leave the booth early. This deviation from the normative behavior presents an opportunity for the election official to "flip" the intended vote: after the voter leaves the booth, the corrupt official can enter the booth, press "back" and change the vote to their liking. This scenario represents an actual election fraud that took place in the US [38].



(a) Normative environment for the voting machine.

(b) The voting machine's robustness is identical with respect to P_{all} and P_{cfm}.

Fig. 10. Models for the voting machine example. In the figures above, the prefix "v" represents actions by the voter.

Comparing Properties. In this case study, we will consider two safety properties, P_{all} and P_{cfm}, both of which imply the absence of vote flipping. P_{all} requires that the election official cannot at any point select, vote, or confirm a candidate. P_{cfm} is weaker, only requiring that the election official cannot at any point confirm a candidate selection.

Using our tool for comparison, we see that the voting machine is equally robust with respect to each property. However, this result is surprising because P_{cfm} is weaker than P_{all}. To understand this result, we examine Fig. 10b where we present the sole maximal robust deviation for each property. In this figure, it is clear that the voting machine is not robust against any deviation in which the voter enters their password and then exits the booth without confirming their vote. The key insight is that, when an election official has the ability to confirm, it *implies* that the official can also select and vote. Therefore, we desire a voting machine without this implication because it will reduce the number of points of failure. For example, we could redesign the voting machine to require a password as part of the confirmation step. In lieu of this insight, a designer could choose to specify a margin of safety into the machine's specification by requiring that it is strictly more robust against P_{cfm} than P_{all}.

6.4 Oyster

Background. The Oyster example was introduced in [41], in which the authors modeled the Oyster card that is used the public transportation system in the United Kingdom. In our model, the controller consists of an *entry gate* and an *exit gate*, where the card holder taps the Oyster card at the start and end of their journey respectively. The environment models the actions of a card holder; in the normative environment, a card holder chooses to tap with either their Oyster card or a credit card, and taps in and out with the chosen card. The key safety property is avoiding an *incomplete journey*, in which a card holder taps in with one card and taps out with a different card.

Calculating Robustness. An incomplete journey is avoided under the normative environment. We calculate the robustness of the system under the two environmental constraints 1) Oyster cards and credit cards give the correct information to the gates and 2) the gates operate correctly and calculate the correct fare when a card is tapped in and out. Unfortunately, the system is not robust to *any* deviations.

6.5 PCA Pump

Background. In this section, we model a patient-controlled analgesia (PCA) pump, originally introduced in [5]. A PCA pump is a medical device that dispenses pain medicine to a patient, offering them partial control over the dose rate. A nurse uses the device interface to program the volume per dosage, as well as a minimum and maximum dose rate to protect the patient from an overdose. The pump includes batteries to power the device in case it is unplugged (e.g., by mistake by the nurse or patient), yet the power may fail if the device runs out of battery. In this case, the device cannot monitor the dosage amount or frequency, which may cause an overdose. Therefore, we define the key safety property P_{pfail}

which requires the PCA pump to abstain from administering medicine after a power failure.

In the normative environment, the nurse operates the pump using the following three step workflow: 1) plug in the pump and turn it on, 2) program the desired dosage parameters into the pump and administer the treatment, and 3) turn off the device and unplug it. The nurse begins with step (1) and ends with step (3), but may omit or repeat step (2) as many times as needed. A diagram of the normative environment is available at [27], pg. 26, Fig. 13. Crucially, the pump is safe with respect to this environment and P_{pfail} because the workflow assumes that the pump is never unplugged in step (2).

Calculating Robustness. We use our tool to calculate the robustness of the pump with respect to the normative environment, P_{pfail}, and an environmental constraint P_{env}. In this case study, P_{env} restricts the environment to actions that are allowed by the pump's interface. A diagram of the sole maximal robust deviation is available at [27], pg. 27, Fig. 14. The tool reports that the pump is robust against four actions, three of which allow the operator to change settings before administering the treatment, and the fourth allows the operator to turn off the device prematurely after programming the dosage parameters. Unfortunately, the pump is not robust against any deviations in which it is unexpectedly unplugged. This poses a key weakness in the pump that the designers may wish to improve upon.

6.6 Results and Discussion

We have run our tool on the examples and case studies above, and we present our results in Table 1. All tests were run on a Mac Book Pro with an M1 Pro chip and 32GB of RAM. In the table, $|Act|$ is the union of Act_E, Act_C, $Act_{P_{saf}}$ and $Act_{P_{env}}$, $|d_{max}|$ is the size of the largest deviation in Δ, and $|W_{P_{env}}|$ is the size of the winning set for each maximal deviation $\tilde{d_i}$ (separated by a comma); NA indicates the absence of an environmental constraint. Furthermore, "Wall Heur" denotes the wall time for running Algorithm 1 with the heuristic, while "Wall Plain" denotes the wall time for running Algorithm 1, and "TO" indicates a time-out after five minutes.

Our results demonstrate that calculating robustness is tractable across several different case studies. In particular, our tool's performance on the larger PCA pump case study shows promising results in terms of scalability. Furthermore, we have shown that Δ is useful as a means for both analysis and comparison of controllers. For example, in the Therac-25 case study, robustness provided a richer analysis than classic verification that helped us discover–and ultimately fix–a critical race condition. Finally, we have also demonstrated in the voting machine case study that robustness provides a means for comparing two properties with respect to a controller and an environment.

Table 1. Summary of results from running our tool.

Example	$\|Act\|$	$\|Q_E\|$	$\|Q_C\|$	$\|Q_P\|$	$\|W\|$	$\|W_{P_{env}}\|$	$\|\Delta\|$	$\|d_{max}\|$	Wall Heur	Wall Plain
Running Example	2	4	2	4	6	NA	3	13	0.433 s	0.431 sec
Therac-25 w/bug	9	5	21	5	62	28,30,31,37	4	21	4.921 sec	TO
Therac-25 w/fix	9	5	19	5	72	18,20,23,25	4	26	0.852 sec	TO
Therac-20	9	5	11	5	40	17,19,21,23	4	26	0.626 sec	TO
Voting wrt. P_{cfm}	9	7	13	3	66	7	1	12	0.469 sec	TO
Voting wrt. P_{all}	9	7	13	3	66	7	1	12	0.426 sec	TO
Oyster	8	4	17	2	15	8	1	4	0.472 sec	TO
PCA Pump	21	11	105	4	1396	34	1	15	1.922 sec	TO

7 Related Work

Quantitative robustness notions for discrete transition systems have been investigated in several works [3,4,8,16,24,32,40,42]. We capture robustness qualitatively, which avoids the need for external cost functions over the discrete transition systems. The problem of synthesizing robust controllers against deviated environments given by a designer is investigated in [45]. Since [45] focuses on synthesizing robust controllers, their framework does not address the analysis of robustness. Moreover, robust controllers are measured via a rank function (quantitatively). Robust linear temporal logic (rLTL) extends the binary view of LTL to a 5-valued semantics to capture different levels of property satisfaction [43]. This work is tangent to ours as it focuses on specifying robustness.

In [17,49], the authors define robustness as a set of environmental behaviors for which a software system can guarantee safety. Defining robustness in the semantic domain–i.e. in terms of behaviors–implicitly describes safe environmental deviations. Our notion of robustness captures safe environmental deviations explicitly in terms of transitions, which offer both syntactic (transitions) and semantic (implied behaviors) information. Transition-based robustness also allows us to capture the safe environmental envelopes of a system; it is not clear how one might efficiently capture this information with only behaviors.

In [29], the authors define robustness also based on additional transitions to the environment. Their definition of robustness compares the perturbed controlled behavior, i.e., $beh(E_d|f)$, instead of directly comparing the additional transitions. In this manner, the partial order used to define robustness in [29] is different from our notion of robustness. Moreover, only an efficient algorithm for invariance properties is presented. Extending the work in [29], the authors explore the relationship between controller robustness and permissiveness for invariance properties [30].

Robust control in discrete event systems is also an active area of research [1,10,19–21,26,31,33,39,44,47,48]. However, they usually deal with specific types of faults such as communication delays, loss of information, or deception

attacks [1,20,21,26,31,39,47]. We capture model uncertainty with our robustness definition, which can be attributed to these faults. Robustness against model uncertainty is tackled in the works of [10,19,44,48]. In these works, deviations are modeled by the behavior generated by the environment. On the other hand, we modeled deviations by the inclusion of extra transitions. In [11], a controller realizability problem is studied for environments modeled as modal transition systems, where a controller satisfies a property in all, some, or none of the LTS family. Our notion of robustness explicitly computes which systems in the LTS family satisfy the property.

Lastly, robustness also relates to fault-tolerance. Fault-tolerance has been studied in the context of distributed systems [13,22,34]. In [6,9,12,14], synthesis of fault-tolerant programs by retrofitting initial fault-intolerant programs. These works focus on specific types of fault models, whereas our robustness model computes the safety envelope the controller is robust against.

8 Conclusion

In this paper, we introduced a new notion of robustness against environmental deviations for discrete-state transition systems. Our notion of robustness is syntactically defined by additional transitions and semantically defined by the controlled behavior generated by these additional transitions. We provided two methods to compute robustness: a brute-force algorithm, and an algorithm based on a controller synthesis problem. We implemented these methods in a prototype tool which we used to analyze several case studies. In these case studies, we demonstrated that our robustness analysis provides crucial information by identifying the environmental envelopes in which the system can guarantee its safety properties.

As part of future work, we plan to extend our work to investigate robustness in the context of partially observable systems as well as in stochastic systems such as Markov decision processes (MDPs). We also plan to investigate the benefit of considering additional environmental states–as well as additional transitions–in our robustness analysis. Finally, we plan to extend our work beyond safety properties, e.g. including liveness.

Acknowledgements. This project was supported by the US NSF Awards CCF-2144860, CNS-1801342, CNS-1801546, CCF-1918140, and ECCS-2144416.

References

1. Alves, M.V.S., da Cunha, A.E.C., Carvalho, L.K., Moreira, M.V., Basilio, J.C.: Robust supervisory control of discrete event systems against intermittent loss of observations. Int. J. Control 1–13 (2019)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
3. Bloem, R., et al.: Synthesizing robust systems. Acta Inf. **51**(3–4), 193–220 (2014)
4. Bloem, R., Greimel, K., Henzinger, T.A., Jobstmann, B.: Synthesizing robust systems. In: 2009 Formal Methods in Computer-Aided Design, pp. 85–92 (2009)

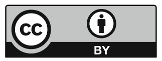
5. Bolton, M.L., Bass, E.J.: Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In: 2011 IEEE International Conference on Systems, Man, and Cybernetics, pp. 1788–1794 (2011). https://doi.org/10.1109/ICSMC.2011.6083931
6. Bonakdarpour, B., Kulkarni, S.S.: SYCRAFT: a tool for synthesizing distributed fault-tolerant programs. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 167–171. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_16
7. Introduction to Discrete Event Systems. Lecture Notes in Computer Science, Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72274-6_9
8. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S.: Proving programs robust. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011), pp. 102–112. Association for Computing Machinery (2011)
9. Cheng, C.-H., Rueß, H., Knoll, A., Buckl, C.: Synthesis of fault-tolerant embedded systems using games: from theory to practice. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 118–133. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_10
10. Cury, J., Krogh, B.: Robustness of supervisors for discrete-event systems. IEEE Trans. Automat. Control **44**(2), 376–379 (1999)
11. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: The modal transition system control problem. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 155–170. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_15
12. Ebnenasir, A., Kulkarni, S.S., Arora, A.: FTSyn: a framework for automatic synthesis of fault-tolerance. Int. J. Softw. Tools Technol. Transf. **10**(5), 455–471 (2008)
13. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput. Surv. **31**(1), 1–26 (1999)
14. Girault, A., Rutten, E.: Automating the addition of fault tolerance with discrete controller synthesis. Formal Method. Syst. Des. **35**, 190–225 (2009)
15. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4
16. Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz robustness of finite-state transducers. In: Raman, V., Suresh, S.P. (eds.) 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 29, pp. 431–443. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2014)
17. Kang, E.: Robustness analysis for secure software design. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment (SEAD 2020), pp. 19–25. Association for Computing Machinery (2020)
18. Leveson, N., Turner, C.: An investigation of the therac-25 accidents. Computer **26**(7), 18–41 (1993). https://doi.org/10.1109/MC.1993.274940
19. Lin, F.: Robust and adaptive supervisory control of discrete event systems. IEEE Trans. Automat. Control **38**(12), 1848–1852 (1993)
20. Lin, F.: Control of networked discrete event systems: dealing with communication delays and losses. SIAM J. Control Optimiz. **52**(2), 1276–1298 (2014)
21. Lin, L., Zhu, Y., Su, R.: Towards bounded synthesis of resilient supervisors. In: 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 7659–7664 (2019)

22. WDAG 1996. LNCS, vol. 1151. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61769-8_9
23. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley and Sons Inc, USA (2000)
24. Majumdar, R., Render, E., Tabuada, P.: Robust discrete synthesis against unspecified disturbances. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control (HSCC 2011), pp. 211–220. Association for Computing Machinery (2011)
25. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic **65**(2), 149–184 (1993)
26. Meira-Góes, R., Marchand, H., Lafortune, S.: Towards resilient supervisors against sensor deception attacks. In: 2019 IEEE 58th Annual Conference on Decision and Control (CDC) (2019)
27. Meira-Góes, R., Dardik, I., Kang, E., Lafortune, S., Tripakis, S.: Safe environmental envelopes of discrete systems. Zenodo (2023). https://doi.org/10.5281/zenodo.7999482
28. Meira-Goes, R., Dardik, I., Kang, E., Lafortune, S., Tripakis, S.: Transitional robustness github repository (2023). https://github.com/cmu-soda/transitional-robustness. Accessed 29 May 2023
29. Meira-Góes, R., Kang, E., Lafortune, S., Tripakis, S.: On tolerance of discrete systems with respect to transition perturbations. arXiv:2110.04200 [eess.SY] (2021)
30. Meira-Góes, R., Kang, E., Lafortune, S., Tripakis, S.: On synthesizing tolerable and permissive controllers for labeled transition systems. In: 16th IFAC Workshop on Discrete Event Systems WODES 2022, vol. 55, no. 28, pp. 158–164 (2022)
31. Meira-Goes, R., Lafortune, S., Marchand, H.: Synthesis of supervisors robust against sensor deception attacks. IEEE Trans. Automat. Control **66**(10), 4990–4997 (2021)
32. Neider, D., Weinert, A., Zimmermann, M.: Synthesizing optimally resilient controllers. Acta Inf. **57**(1), 195–221 (2020)
33. Paoli, A., Lafortune, S.: Safe diagnosability for fault-tolerant supervision of discrete-event systems. Automatica **41**(8), 1335–1347 (2005)
34. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
35. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1989), pp. 179–190. Association for Computing Machinery (1989)
36. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), pp. 46–57 (1977)
37. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
38. U.S. Attorney's Office Eastern District of Kentucky. Clay county officials and residents convicted on racketeering and voter fraud charges (2010). https://archives.fbi.gov/archives/louisville/press-releases/2010/lo032510.htm
39. Rohloff, K.: Bounded sensor failure tolerant supervisory control. In: 11th IFAC Workshop on Discrete Event Systems, vol. 45, no. 29, pp. 272–277 (2012)
40. Samanta, R., Deshmukh, J.V., Chaudhuri, S.: Robustness analysis of string transducers. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 427–441. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_30

41. Sempreboni, D., Viganò, L.: X-men: a mutation-based approach for the formal analysis of security ceremonies. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 87–104 (2020). https://doi.org/10.1109/EuroSP48549.2020.00014
42. Tabuada, P., Balkan, A., Caliskan, S.Y., Shoukry, Y., Majumdar, R.: Input-output robustness for discrete systems. In: Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT 2012), pp. 217–226. Association for Computing Machinery (2012)
43. Tabuada, P., Neider, D.: Robust Linear Temporal Logic. In: Talbot, J.M., Regnier, L. (eds.) 25th EACSL Annual Conference on Computer Science Logic (CSL 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 62, pp. 10:1–10:21. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2016)
44. Takai, S.: Maximizing robustness of supervisors for partially observed discrete event systems. Automatica **40**(3), 531–535 (2004)
45. Topcu, U., Ozay, N., Liu, J., Murray, R.M.: On synthesizing robust discrete controllers under modeling uncertainty. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2012), pp. 85–94. Association for Computing Machinery (2012)
46. Tun, T.T., Bennaceur, A., Nuseibeh, B.: Oasis: weakening user obligations for security-critical systems. In: 2020 IEEE 28th International Requirements Engineering Conference (RE), pp. 113–124 (2020). https://doi.org/10.1109/RE48521.2020.00023
47. Wang, F., Shu, S., Lin, F.: Robust networked control of discrete event systems. IEEE Trans. Automat. Sci. Eng. **13**(4), 1528–1540 (2016)
48. Young, S., Garg, V.K.: Model uncertainty in discrete event systems. SIAM J. Control Optimiz. **33**(1), 208–226 (1995)
49. Zhang, C., Garlan, D., Kang, E.: A behavioral notion of robustness for software systems. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), pp. 1–12. Association for Computing Machinery (2020)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Verse: A Python Library for Reasoning About Multi-agent Hybrid System Scenarios

Yangge Li(✉), Haoqing Zhu(✉), Katherine Braught, Keyi Shen, and Sayan Mitra(✉)

Coordinated Science Laboratory, University of Illinois Urbana-Champaign, Champaign, USA
{li213,haoqing3,braught2,keyis2,mitras}@illinois.edu

Abstract. We present the Verse library with the aim of making hybrid system verification more usable for multi-agent scenarios. In Verse, decision making agents move in a map and interact with each other through sensors. The decision logic for each agent is written in a subset of Python and the continuous dynamics is given by a black-box simulator. Multiple agents can be instantiated, and they can be ported to different maps for creating scenarios. Verse provides functions for simulating and verifying such scenarios using existing reachability analysis algorithms. We illustrate capabilities and use cases of the library with heterogeneous agents, incremental verification, different sensor models, and plug-n-play subroutines for post computations.

Keywords: Scenario verification · Reachability · Hybrid Systems

1 Introduction

Automatic verification tools for hybrid systems have been used to analyze linear models with thousands of continuous dimensions [1,5,6] and nonlinear models inspired by industrial applications [6,14]. The state of the art and the challenges are discussed in a recent survey [11]. Despite the potentially large user base, currently this technology is inaccessible without formal methods training. Automatic hybrid verification tools [10,13,17,25,31] require the input model to be written in a tool-specific language. Tools like C2E2 [15] attempt to translate models from Simulink/Stateflow, but the language-barrier goes down to the underlying math models. The verification algorithms are based on variants of the hybrid automaton [3,21,24] which requires the discrete states (or *modes*) to be spelled out explicitly as a graph, with guards and resets labeling the transitions. We discuss related works in more detail in Sect. 6, including recently developed libraries that address usability barrier [5,7,8].

This research was funded in part by NASA University Leadership Initiative grant (80NSSC22M0070) Robust and Resilient Autonomy for Advanced Air Mobility.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 351–364, 2023.
https://doi.org/10.1007/978-3-031-37706-8_18

In this paper, we present Verse, a Python library that aims to make hybrid technologies more usable for multi-agent scenarios. The key features implemented are as follows: (1) In Verse, users write scenarios in Python. User-defined functions can be used to create complex *agents*, invariant requirements can be written as `assert` statements, and scenarios can be created by instantiating multiple agents, all using the standard Python syntax. Verse parses this scenario and constructs an internal representation of the hybrid automaton for simulation and analysis. (2) Verse introduces an additional structure, called *map,* for defining the modes and the transitions of a hybrid system. Map contains *tracks* that can capture geometric objects (e.g., lanes or waypoints) that make it possible to create new scenarios just by instantiating agents on new maps. With track modes, users do not have to explicitly write different modes for a vehicle following different waypoint segments. Finally, (3) Verse comes with functions for simulation and safety verification via reachability analysis. Developers can implement new functions, plug-in existing tools, or implement advanced algorithms, e.g., for incremental verification. In this tool paper, we illustrate use cases with heterogeneous agents and different scenario setups, the flexibility of plugging in different reachability algorithms and the ability to develop more advanced algorithms (Sect. 5). Verse is available at https://github.com/AutoVerse-ai/Verse-library.

2 Overview of Verse

We will highlight the key features of Verse with an example. Consider two drones flying along three parallel ∞-shaped tracks that are vertically separated in space (shown by black lines in Fig. 1). Each drone has a simple collision avoidance logic: if it gets too close to another drone on the same track, then it switches to either the track above or the one below. A drone on `T1` has both choices. Verse enables creation, simulation, and verification of such scenarios using Python, and provides a collection of powerful functions for building new analysis algorithms.

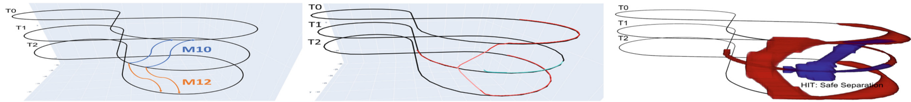


Fig. 1. *Left:* A 3-d ∞-shaped map with example track mode labels. *Center:* Simulation of a red drone nearing the blue drone on `T1` and nondeterministically moving to `T0` or `T2`. Both branches are computed by Verse's `simulate` function. *Right:* Computed reachable sets of the two drones cover more possibilities: either drones can switch tracks when they get close. All four branches are explored by Verse. The branch for blue drone moving downwards violates safety as it may collide with the red drone following T1.

Creating Scenarios. Agents like the drones in this example are described by a *simulator* and a *decision logic* in an expressive subset of Python (see code in Fig. 2 and [26] for more details). The decision logic for an ego agent takes as input its current state and the (observable) states of the other agents, and

updates the discrete state or the *mode* of the ego agent. For example, in lines 41–43 of Fig. 2 an agent updates its mode to begin a track change if there is any agent near it. It may also update the continuous state of the ego agent. The *mode* of an agent, as we shall see later in Sect. 3, has two parts—a *tactical mode* corresponding to agent's decision or discrete state, and a *track mode* that is determined by the map. Using the `any` and `all` functions, the agent's decision logic can quantify over other agents in the scene. User defined functions are also allowed (`is_close`, Fig. 2 line 41). Verse will parse this decision logic to create an internal representation of the transition graph of the hybrid model with guards and resets. The simulator can be written in any language and is treated as a black-box[1]. For the examples discussed in this paper, the simulators are also written in Python. Safety requirements can be specified using `assert` statements (see Fig. 5).

```
38  def decisionLogic(ego: State, others: List[State], track_map):
39    next = copy.deepcopy(ego)
40    if ego.tactical_mode == TacticalMode.Normal:
41      if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
      ↪  others):
42        next.tactical_mode = TacticalMode.MoveDown
43        next.track_mode = track_map.Tg(ego.track_mode, ego.tactical_mode,
        ↪  TacticalMode.MoveDown)
44      if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
      ↪  others):
45        next.tactical_mode = TacticalMode.MoveUp
46    # ...
47    if ego.tactical_mode == TacticalMode.MoveUp:
48      if in_interval(track_map.altitude(ego.track_mode)-ego.z, -1, 1):
49        next.tactical_mode = TacticalMode.Normal
50        next.track_mode = track_map.Tg(ego.track_mode, ego.tactical_mode,
        ↪  TacticalMode.Normal)
51    # ...
```

Fig. 2. Decision Logic Code Snippet from drone_controller.py.

Maps and Sensors. The map of a scenario specifies the tracks that the agents can follow. While a map may have infinitely many tracks, they fall in a finite number of *track modes*. For example, in this ∞-shaped map, each layer is assigned to a track mode (`T0-2`) and all the tracks between each pair of layers are also assigned to a track mode (`M10`, `M01` etc.). When an agent makes a decision and changes its tactical mode, the map object determines the new track mode for the agent. The map abstraction makes scenarios succinct and enables portability of agents across different maps. Besides creating from scratch, Verse provides functions for generating map objects from OpenDRIVE [4] files.

[1] This design decision for Verse is relatively independent. For reachability analysis, Verse currently uses black-box statistical approaches implemented in DryVR [14] and NeuReach [35]. If the simulator is available as a white-box model, such as differential equations, then Verse could use model-based reachability analysis.

The *sensor* function defines which variables from an agent are visible to other agents. The default sensor function allows all agents to see all variables; we discuss how the sensor function can be modified to include bounded noise in Sect. 5. A map, a sensor and a collection of (compatible) agents together define a scenario object (Fig. 3). In the first few lines, the drone agents are created, initialized, and added to the scenario object. A scenario can have heterogeneous agents with different decision logics.

```
32    scenario = Scenario()
33    drone_red = DroneAgent('drone_red', file_name='drone_controller.py')
34    drone_red.set_initial([init_l_1, init_u_1],(CraftMode.Normal, TrackMode.T1))
35    scenario.add_agent(drone_red)
36    drone_blue = DroneAgent('drone_blue', file_name='drone_controller.py')
37    scenario.add_agent(drone_blue)
38    # ...
39    scenario.set_map(M6())
40    scenario.set_sensor(BaseSensor())
41    #traces = scenario.simulate(40, time_step)
42    traces = scenario.verify(40, time_step)
```

Fig. 3. Scenario specification snippet.

Simulation and Reachability. Once a scenario is defined, Verse's `simulate` function can generate simulation(s) of the system, which can be stored and plotted. As shown in Fig. 1(Center), a simulation from a single initial state explores all possible branches that can be generated by the decision logics of the interacting agents, upto a specified time horizon. Verse verifies the safety assertions of a scenario by computing the over-approximations of the *reachable sets* for each agent, and checking these against the predicates defined by the assertions. Figure 1(Right) visualizes the result of such a computation performed using the `verify` function. In this example, the safety condition is violated when the blue drone moves downward to avoid the red drone. The other branches of the scenario are proved to be safe. The `simulate` and `verify` functions save a copy of the resulting execution tree, which can be loaded and traversed to analyze the sequences modes and states that leads to safety violations. Verse makes it convenient to plug in different reachability subroutines. It also provides powerful functions to implement advanced verification algorithms, such as incremental verification.

3 Scenarios in Verse

A *scenario* in Verse is specified by a map, a collection of agents in that map, and a sensor function that defines the part of each agent visible to other agents. We describe these components below, and in Sect. 4, we will discuss how they formally define a hybrid system.

Tracks, Track Modes, and Maps. A *workspace* W is an Euclidean space in which the agents reside (For example, a compact subset of $\mathbb{R}^2$ or $\mathbb{R}^3$). An agent's continuous dynamics makes it roughly follow certain continuous curves

in W, called *tracks*, and occasionally the agent's decision logic changes the track. Formally, a *track* is simply a continuous function $\omega : [0, 1] \rightarrow W$, but not all such functions are valid tracks. A map $\mathcal{M}$ defines the set of tracks $\Omega_{\mathcal{M}}$ it permits. In a highway map, some tracks will be aligned along the lanes while others will correspond to merges and exits.

We assume that an agent's decision logic does not depend on exactly which of the infinitely many tracks it is following, but instead, it depends only on which type of track it is following or the *track mode*. In the example in Sect. 2, the track modes are `T0`, `T1`, `M01`, etc. Every (blue) track for transitioning from point on T0 to the corresponding point on T1 has track mode `M01`. A map has a finite set of track modes $L_{\mathcal{M}}$ and a labeling function $V_{\mathcal{M}} : \Omega_{\mathcal{M}} \rightarrow L_{\mathcal{M}}$ that maps the track to a track mode. It also has a mapping $g_{\mathcal{M}} : W \times L_{\mathcal{M}} \rightarrow \Omega_{\mathcal{M}}$ that maps a track mode and a specific position in the workspace to a specific track.

Finally, a Verse agent's decision logic can change its internal mode or *tactical mode* P (E.g., `Normal` to `MoveUp`). When an agent changes its tactical mode, it may also update the track it is following and this is encoded in the track graph function: $T_{g_{\mathcal{M}}} : L_{\mathcal{M}} \times P \times P \rightarrow L_{\mathcal{M}}$ which takes the current track mode, the current and the next tactical mode, and generates the new track mode the agent should follow. For example, when the tactical mode of a drone changes from `Normal` to `MoveUP` while it is on `T1`, this map function $T_{g_{\mathcal{M}}}(\texttt{T1}, \texttt{Normal}, \texttt{MoveUp}) = \texttt{M10}$ informs that the agent should follow a track with mode `M10`. These sets and functions together define a Verse map object $\mathcal{M} = \langle L_{\mathcal{M}}, V_{\mathcal{M}}, g_{\mathcal{M}}, T_{g_{\mathcal{M}}} \rangle$. We will drop the subscript $\mathcal{M}$ when the map being used is clear from context.

Agents. A Verse *agent* is defined by modes and continuous state variables, a decision logic that defines (possibly nondeterministic) discrete transitions, and a flow function that defines continuous evolution. An agent $\mathcal{A}$ is *compatible* with a map $\mathcal{M}$ if the agent's tactical modes P are a subset of the allowed input tactical modes for T_g. This makes it possible to instantiate the same agent on different compatible maps. The *mode space* for an agent instantiated on map $\mathcal{M}$ is the set $D = L \times P$, where L is the set of track modes in $\mathcal{M}$ and P is the set of tactical modes of the agent. The *continuous state space* is $X = W \times Z$, where W is the workspace (of $\mathcal{M}$) and Z is the space of other continuous state variables. The (full) *state space* is the Cartesian product $Y = X \times D$. In the two-drone example in Sect. 2, the continuous states variables are the positions and velocities along the three axes of the workspace. The modes are $\langle \texttt{Normal}, \texttt{T1} \rangle$, $\langle \texttt{MoveUp}, \texttt{M10} \rangle$, etc.

An *agent* $\mathcal{A}$ in map $\mathcal{M}$ with $k-1$ other agents is defined by a tuple $\mathcal{A} = \langle Y, Y^0, G, R, F \rangle$, where Y is the state space, $Y^0 \subseteq Y$ is the set of initial states. The guard G and reset R functions jointly define the discrete transitions. For a pair of modes $d, d' \in D$, $G(d, d') \subseteq X^k$ defines the condition under which a transition from d to d' is enabled. The $R(d, d') : X^k \rightarrow X$ function specifies how the continuous states of the agent are updated when the mode switch happens. Both of these functions take as input the sensed continuous states of all the other $k-1$ agents in the scenario. The G and the R functions are not defined separately,

but are extracted by the Verse parser from a block of structured Python code as shown in Fig. 2. The discrete states in `if` conditions and assignments define the source and destination of discrete transitions. `if` conditions involving continuous states define guards for the transitions and assignments of continuous states define resets. Expressions with `any` and `all` functions are unrolled to disjunctions and conjunctions according to the number of agents k.

For example in Fig. 2, Lines 47–50 define transitions $\langle \texttt{MoveUp}, \texttt{M10} \rangle$ to $\langle \texttt{Normal}, \texttt{T0} \rangle$ and $\langle \texttt{MoveUp}, \texttt{M21} \rangle$ to $\langle \texttt{Normal}, \texttt{T1} \rangle$. The change of track mode is given by the T_g function. The guard for this transition comes from the `if` condition at Line 48, $G(\langle \texttt{MoveUp}, \texttt{M10} \rangle, \langle \texttt{Normal}, \texttt{T0} \rangle) = \{x \mid -1 < \texttt{T0}.pz - x.pz < 1\}$ for $x \in X$ given by user defined `in_interval` function. Here continuous states remain unchanged after transition.

The final component of the agent is the *flow* function $F : X \times D \times \mathbb{R}^{\geq 0} \rightarrow X$ which defines the continuous time evolution of the continuous state. For any initial condition $\langle x^0, d^0 \rangle \in Y$, $F(x^0, d^0)(\cdot)$ gives the continuous state of the agent as a function of time. In this paper, we use F as a black-box function (see Footnote 1).

Sensors and Scenarios. For a scenario with k agents, a *sensor* function $\mathcal{S} : Y^k \rightarrow Y^k$ defines the continuous observables as a function of the continuous state. For simplifying exposition, in this paper we assume that observables have the same type as the continuous state Y, and that each agent i is observed by all other agents identically. This simple, overtly transparent sensor model, still allows us to write realistic agents that only use information about nearby agents. In a highway scenario, the observable part of agent j to another agent i may be the relative distance $y_j = x_j - x_i$, and vice versa, which can be computed as a function of the continuous state variables x_j and x_i. A different sensor function which gives nondeterministic noisy observations, appears in Sect. 5.

A Verse *scenario* SC is defined by (a) a map $\mathcal{M}$, (b) a collection of k agent instances $\{\mathcal{A}_1 ... \mathcal{A}_k\}$ that are compatible with $\mathcal{M}$, and (c) a sensor $\mathcal{S}$ for the k agents. Since all the agents are instantiated on the same compatible map $\mathcal{M}$, they share the same workspace. Currently, we require agents to have identical state spaces, i.e., $Y_i = Y_j$, but they can have different decision logics and different continuous dynamics.

4 Verse Scenario to Hybrid Verification

In this section, we define the underlying hybrid system $H(SC)$, that a Verse scenario SC specifies. The verification questions that Verse is equipped to answer are stated in terms of the behaviors or *executions* of $H(SC)$. Verse's notion of a hybrid automaton is close to that in Definition 5 of [14]. The only uncommon aspect in [14] is that the continuous flows may be defined by a black-box simulator functions, instead of white-box analytical models (see Footnote 1).

Given a scenario with k agents $SC = \langle \mathcal{M}, \{\mathcal{A}_1, ... \mathcal{A}_k\}, \mathcal{S}, P \rangle$, the corresponding hybrid automaton $H(SC) = \langle \mathbf{X}, \mathbf{X}^0, \mathbf{D}, \mathbf{D}^0, \mathbf{G}, \mathbf{R}, \mathbf{TL} \rangle$, where

1. $\mathbf{X} := \prod_i X_i$ is the *continuous state space*. An element $\mathbf{x} \in \mathbf{X}$ is called a *state*. $\mathbf{X}^0 := \prod_i X_i^0 \subseteq \mathbf{X}$ is the set of *initial continuous states*.
2. $\mathbf{D} := \prod_i D_i$ is the *mode space*. An element $\mathbf{d} \in \mathbf{D}$ is called a *mode*. $\mathbf{D}^0 := \prod_i D_i^0 \subseteq \mathbf{D}$ is the finite set of *initial modes*.
3. For a mode pair $\mathbf{d}, \mathbf{d}' \in \mathbf{D}$, $\mathbf{G}(\mathbf{d}, \mathbf{d}') \subseteq \mathbf{X}$ defines the continuous states from which a transition from $\mathbf{d}$ to $\mathbf{d}'$ is enabled. A state $\mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}')$ iff there exists an agent $i \in \{1, ..., k\}$, such that $\mathbf{x}_i \in G_i(\mathbf{d}_i, \mathbf{d}'_i)$ and $\mathbf{d}_j = \mathbf{d}'_j$ for $j \neq i$.
4. For a mode pair $\mathbf{d}, \mathbf{d}' \in \mathbf{D}$, $\mathbf{R}(\mathbf{d}, \mathbf{d}') : \mathbf{X} \to \mathbf{X}$ defines the change of continuous states after a transition from $\mathbf{d}$ to $\mathbf{d}'$. For a continuous state $\mathbf{x} \in \mathbf{X}$, $\mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x}) = R_i(\mathbf{d}_i, \mathbf{d}'_i)(\mathbf{x})$ if $\mathbf{x} \in G_i(\mathbf{d}_i, \mathbf{d}'_i)$, otherwise $= \mathbf{x}_i$.
5. $\mathbf{TL}$ is a set of pairs $\langle \xi, \mathbf{d} \rangle$, where the *trajectory* $\xi : [0, T] \to \mathbf{X}$ describes the evolution of continuous states in mode $\mathbf{d} \in \mathbf{D}$. Given $\mathbf{d} \in \mathbf{D}, \mathbf{x}^0 \in \mathbf{X}$, ξ should satisfy $\forall t \in \mathbb{R}^{\geq 0}, \xi_i(t) = F_i(\mathbf{x}_i^0, \mathbf{d}_i)(t)$.

We denote by $\xi.fstate$, $\xi.lstate$, and $\xi.ltime$ the initial state $\xi(0)$, the last state $\xi(T)$, and $\xi.ltime = T$. For a sampling parameter $\delta > 0$ and a length m, a δ-*execution* of a hybrid automaton $H = H(SC)$ is a sequence of m labeled trajectories $\alpha := \langle \xi^0, \mathbf{d}^0 \rangle, ..., \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$, such that (1) $\xi^0.fstate \in \mathbf{X}^0, \mathbf{d}^0 \in \mathbf{D}^0$, (2) For each $i \in \{1, ..., m-1\}$, $\xi_i.lstate \in \mathbf{G}(\mathbf{d}^i, \mathbf{d}^{i+1})$ and $\xi^{i+1}.fstate = \mathbf{R}(\mathbf{d}^i, \mathbf{d}^{i+1})(\xi^i.lstate)$, and (3) For each $i \in \{1, ..., m-1\}$, $\xi^i.ltime = \delta$ for $i \neq m-1$ and $\xi^i.ltime \leq \delta$ for $i = m-1$.

We define first and last state of an execution $\alpha = \langle \xi^0, \mathbf{d}^0 \rangle, ..., \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$ as $\alpha.fstate = \xi^0.fstate$, $\alpha.lstate = \xi^{m-1}.lstate$ and the first and last mode as $\alpha.fmode = \mathbf{d}^0$ and $\alpha.lmode = \mathbf{d}^{m-1}$. The set of reachable states is defined by $Reach_H := \{\alpha.lstate \mid \alpha \text{ is an execution of } H\}$. In addition, we denote the reachable states in a specific mode $\mathbf{d} \in \mathbf{V}$ as $Reach_H(\mathbf{d})$ and $Reach_H(T)$ to be the set of reachable states at time T. Similarly, denoting the unsafe states for mode $\mathbf{d}$ as $\mathbf{U}(\mathbf{d})$, the safety verification problem for H can be solved by checking whether $\forall \mathbf{d} \in \mathbf{D}$, $Reach_H(\mathbf{d}) \cap \mathbf{U}(\mathbf{d}) = \emptyset$. Next, we discuss Verse functions for verification via reachability.

Verification Algorithms in Verse. The Verse library comes with several built-in verification algorithms, and it provides functions that users can use to implement powerful new algorithms. We describe the basic algorithm and functions in this section.

Consider a scenario SC with k agents and the corresponding hybrid automaton $H(SC)$. For a pair of modes, $\mathbf{d}, \mathbf{d}'$ the standard discrete $post_{\mathbf{d},\mathbf{d}'} : \mathbf{X} \to \mathbf{X}$ and continuous $post_{\mathbf{d},\delta} : \mathbf{X} \to \mathbf{X}$ operators are defined as follows: For any state $\mathbf{x}, \mathbf{x}' \in \mathbf{X}$, $post_{\mathbf{d},\mathbf{d}'}(\mathbf{x}) = \mathbf{x}'$ iff $\mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}')$ and $\mathbf{x}' = \mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x})$; and, $post_{\mathbf{d},\delta}(\mathbf{x}) = \mathbf{x}'$ iff $\forall i \in 1, ..., k$, $\mathbf{x}'_i = F_i(\mathbf{x}_i, \mathbf{d}_i, \delta)$. These operators are also lifted to sets of states in the usual way. Verse provides `postCont` to compute $post_{\mathbf{d},\delta}$ and `postDisc` to compute $post_{\mathbf{d},\mathbf{d}'}$. Instead of computing the exact post, `postCont` and `postDisc` compute over-approximations using improved implementations of the algorithms in [14]. Verse's `verify` function implements a reachability analysis algorithm using these post operators. The algorithm constructs an execution tree $Tree = \langle V, E \rangle$ up to depth m in breadth first order. Each vertex $\langle \mathbf{S}, \mathbf{d} \rangle \in V$

is a pair of a set of states and a mode. The root is $\langle \mathbf{X}^0, \mathbf{d}^0 \rangle$. There is an edge from $\langle \mathbf{S}, \mathbf{d} \rangle$ to $\langle \mathbf{S}', \mathbf{d}' \rangle$, iff $\mathbf{S}' = post_{\mathbf{d}',\delta}(post_{\mathbf{d},\mathbf{d}'}(\mathbf{S}))$. The safety conditions are checked when the tree is constructed. Currently, Verse implements only bounded time reachability, however, basic unbounded time analysis with fixed-point checks could be added following [14,32].

5 Experiments and Use Cases

We evaluate key features and algorithms in Verse through examples. We consider two types of agents: a 4-d ground vehicle with bicycle dynamics and the Stanley controller [22] and a 6-d drone with a NN-controller [23]. Each of these agents can be fitted with one of two types of decision logic: (1) a collision avoidance logic (CA) by which the agent switches to a different available track when it nears another agent on its own track, and (2) a simpler non-player vehicle logic (NPV) by which the agent does not react to other agents (and just follows its own track at constant speed). We denote the car agent with CA logic as agent C-CA, drone with NPV as D-NPV, and so on. We use four 2-d maps ($\mathcal{M}$1-4) and two 3-d maps $\mathcal{M}$5-6. $\mathcal{M}$1 and $\mathcal{M}$2 have 3 and 5 parallel straight tracks, respectively. $\mathcal{M}$3 has 3 parallel tracks with circular curve. $\mathcal{M}$4 is imported from OpenDRIVE. $\mathcal{M}$6 is the figure-8 map used in Sect. 2.

Safety Analysis with Multiple Drones in a 3-d Map. The first example is a scenario with two drones—D-CA agent (red) and D-NPV agent (blue)—in map $\mathcal{M}$5. The safety assertion requires agents to always separate by at least 1 m. Figure 4(*left*) shows the computed reachable set, its projection on x-position, and on z position. Since the agents are separated in space-time, the scenario is verified safe. These plots are generated using Verse's plotting functions.

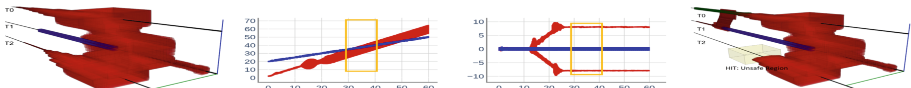


Fig. 4. Left to right: (1) Computed reachtubes for a 2-drone scenario; (2) same reachtube projected on x-dimension, and (3) on z-dimension. Since there is no overlap in space-time, no collision. (4) Reachtube for a 3-drone scenario, the red drone violates the safety condition by entering the unsafe region after moving downward. (Color figure online)

Checking Multiple Safety Assertions. Verse supports multiple safety assertions specified using `assert` statements. For example, the user can specify unsafe regions (Line 77–78) or safe separation between agents (Line 79–82) as shown in Fig. 5. We add a second D-NPV to the previous scenario and both safety assertions. The result is shown in the rightmost Fig. 4. In this scenario, D-CA violates the safety property by entering the unsafe region after moving downward to avoid collision. The behavior of D-CA after moving upward is not influenced. There is no violation of safe separation. Verse allow users to extract the set of reachable states and mode transitions that leads to a safety violation.

```
77         assert not (ego.x > 40 and ego.x < 50 and \
78             ego.y > -5 and ego.y < 5 and ego.z > -10 and ego.z < -6), "Unsafe Region"
79         assert not any(ego.x-other.x < 1 and ego.x-other.x > -1 and \
80             ego.y-other.y < 1 and ego.y-other.y > -1 and \
81             ego.z-other.z < 1 and ego.z-other.z > -1 \
82             for other in others), "Safe Separation"
```

Fig. 5. Safety assertions for three drone scenario.

Changing Maps. Verse allows users to easily create scenarios with different maps and port agents across compatible maps. We start with a scenario with one C-CA agent (red) and two C-NPV agents (blue, green) in $\mathcal{M}1$. The safety assertion is that the vehicles should be at least 1m apart in both x and y-dimensions. Figure 6(*left*) shows the verification result and safety is not violated. However, if we switch to map $\mathcal{M}3$ by changing one line in the scenario definition, a reachability analysis shows that a safety violation can happen after C-CA merges left Fig. 6(*center*). In addition, Verse allows importing map from OpenDRIVE [4] format. An example is included in the extended version of the paper [26].

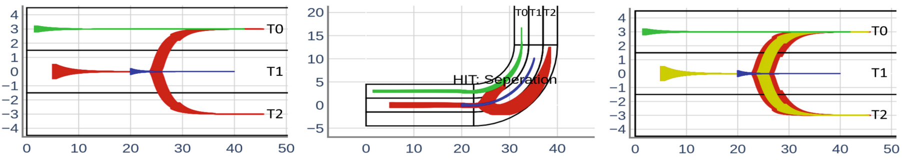


Fig. 6. *Left:* running the three car scenario on map with parallel straight lanes. *Center:* same scenario with a curved map. *Right:* same scenario with a noisy sensor. (Color figure online)

Adding Noisy Sensors. Verse supports scenarios with different sensor functions. For example, the user can create a noisy sensor function that mimics a realistic sensor with bounded noise. Such sensor functions are easily added to the scenario using the `set_sensor` function.

Figure 6(*right*) shows exactly the same three-car scenario with a noisy sensor, which adds ± 0.5 m noise to the perceived position of all other vehicles. Since the sensed values of other agents only impacts the checking of the guards (and hence the transitions) of the agents, Verse internally bloats the reachable set of positions for the other agents by ± 0.5 while checking guards. Compared with the behavior of the same agent with no sensor noise (shown in yellow in Fig. 6(*right*)), the sensor noise enlarges the region over which the transition can happen, causes enlarged reachtubes for the red agent.

Plugging in Different Reachability Engines. With a little effort, Verse allows users to plug in different reachability tools for the `postCont` computation. The user will need to modify the interface of the reachability tool so that given a set of initial states, a mode, and a non negative value δ, the reachability tool can output the set of reachable states over a δ-period represented by a set of

timed hyperrectangles. Currently, Verse implements computing `postCont` using DryVR [14], NeuReach [35] and Mixed Monotone Decomposition [12]. A scenario with two car agents in map $\mathcal{M}1$ verified using NeuReach and DryVR is included in the extended version of the paper [26].

Incremental Verification. We implemented an incremental verification algorithm in Verse called `verifyInc`. This algorithm improves `verify` by caching and reusing reachtubes, and can be effective when analyzing a sequence of slightly different scenarios. The function `verifyInc` avoids re-computing $post_{\mathbf{d},\mathbf{d}'}$ and $post_{\mathbf{d},\delta}$ when constructing the execution tree by reusing earlier execution runs. Experiments show that `verifyInc` reduces running time by 10x for two identical runs and 2x when the decision logic is slightly modified. (More details are provided in the extended version of paper [26]). This exercise illustrates a usage of Verse in creating alternative analysis algorithms.

Table 1 summarizes the running time of verifying all the examples in this section. We additionally include three standard benchmarks: van-der-pol (Agent V) [20], spacecraft rendezvous (Agent S) [20], and gearbox (Agent G) [2]. As expected, the running times increase with the number of discrete mode transition. However, for complicated scenario with 7 agents and 37 transitions, the verification can still finish in under 6 mins, which suggests some level of scalability. The choice of reachability engine can also impact running time. For the same scenario in rows 2, 3 and 10, 11, Verse with NeuReach[2] as the reachability engine takes more time than using DryVR as the reachability engine.

Table 1. Runtime for verifying examples in Sect. 5. Columns are: number of agents (#$\mathcal{A}$), agent type ($\mathcal{A}$), map used (Map), reachability engine used (`postCont`), sensor type (N$\mathcal{S}$), number of mode transitions #TR, and the total run time (Rt). N/A for not available.

#$\mathcal{A}$	$\mathcal{A}$	Map	`postCont`	N$\mathcal{S}$	#Tr	Rt (s)	#$\mathcal{A}$	$\mathcal{A}$	Map	`postCont`	Noisy $\mathcal{S}$	#Tr	Rt (s)
2	D	$\mathcal{M}6$	DryVR	No	8	55.9	2	D	$\mathcal{M}5$	DryVR	No	5	18.7
2	D	$\mathcal{M}5$	NeuReach	No	5	1071.2	3	D	$\mathcal{M}5$	DryVR	No	7	39.6
7	C	$\mathcal{M}2$	DryVR	No	37	322.7	3	C	$\mathcal{M}1$	DryVR	No	5	23.4
3	C	$\mathcal{M}3$	DryVR	No	4	34.7	3	C	$\mathcal{M}4$	DryVR	No	7	118.3
3	C	$\mathcal{M}1$	DryVR	Yes	5	29.4	2	C	$\mathcal{M}1$	DryVR	No	5	21.6
2	C	$\mathcal{M}1$	NeuReach	No	5	914.9	1	V	N/A	DryVR	N/A	1	0.33
1	S	N/A	DryVR	N/A	3	2.3	1	G	N/A	DryVR	N/A	3	67.14

6 Related Work

Automatic hybrid verification tools typically require the input model to be written in a tool-specific language [10,13–15,17,25]. Libraries like JuliaReach [7]

[2] Runtime for NeuReach includes training time.

Hylaa [5] and HyPro [8] share our motivation to reduce the usability barrier by providing reachability analysis APIs for popular programming languages. Verse is distinct in this family in that it supports creation and analysis of multi-agent scenarios. The work in [33] also supports multiple agents, however, Verse significantly improves usability with maps, scenarios and decision logics written in Python.

Interactive theorem provers have been used for modeling and verification of multi-agent and hybrid systems [16,19,27,29]. KeYmeraX [19] uses quantified differential dynamic logic for specifying multi-agent scenarios and supports proof search and user defined tactics. Isabelle/HOL [16], PVS [27], and Maude [29] have also been used for limited classes of hybrid systems. These approaches are geared for a different user segment in that they provide higher expressive and analytical power to expert users. Verse is inspired by widely used tools for simulating multi-agent scenarios [9,18,28,30,36]. While the models created in these tools can be flexible and expressive, currently they are not amenable to formal verification.

7 Conclusions and Future Directions

In this paper, we presented the new open source Verse library for broadening applications of hybrid system verification technologies to scenarios involving multiple interacting decision-making agents. There are several future directions for Verse. Verse currently assumes all agents interact with each other only through the sensor in the scenario and all agents share the same sensor. This restriction could be relaxed to have different types of asymmetric sensors. Functions for constructing and systematically sampling scenarios could be developed. Functions for post-computation for white-box models by building connections with existing tools [1,10,15] would be a natural next step. Those approaches could obviously utilize the symmetry property of agent dynamics as in [32,34], but beyond that, new types of symmetry reductions should be possible by exploiting the map geometry.

References

1. Althoff, M.: An introduction to CORA 2015. In: Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
2. Althoff, M., et al.: Arch-comp20 category report: continuous and hybrid systems with linear continuous dynamics. In: Frehse, G., Althoff, M. (eds.) 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20), ARCH20. EPiC Series in Computing, vol. 74, pp. 16–48. EasyChair (2020). https://doi.org/10.29007/7dt2
3. Alur, R., et al.: The algorithmic analysis of hybrid systems. Theoret. Comput. Sci. **138**(1), 3–34 (1995)
4. Association for Standardization of Automation and Measuring Systems (ASAM): Open dynamic road information for vehicle environment, August 2021. https://www.asam.net/standards/detail/opendrive/

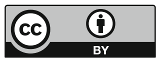
5. Bak, S., Duggirala, P.S.: HyLAA: a tool for computing simulation-equivalent reachability for linear systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, pp. 173–178. ACM (2017)
6. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, pp. 23–32. Association for Computing Machinery, New York (2019)
7. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: a toolbox for set-based reachability. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 39–44 (2019)
8. Ábrahám, E., Schupp, S., Chen, X., Kowalewski, S., Makhlouf, I., Sankaranarayanan, S.: HyPro: a C++ library for the representation of state sets for the reachability analysis of hybrid systems (2023)
9. Brittain, M., Alvarez, L.E., Breeden, K., Jessen, I.: AAM-Gym: artificial intelligence testbed for advanced air mobility (2022)
10. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
11. Chen, X., Sankaranarayanan, S.: Reachability analysis for cyber-physical systems: are we there yet? In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods, NFM 2022. LNCS, vol. 13260. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06773-0_6
12. Coogan, S.: Mixed monotonicity for reachability and safety in dynamical systems. In: 2020 59th IEEE Conference on Decision and Control (CDC), pp. 5074–5085 (2020)
13. Devonport, A., Khaled, M., Arcak, M., Zamani, M.: PIRK: scalable interval reachability analysis for high-dimensional nonlinear systems. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 556–568. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_27
14. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: DryVR: data-driven verification and compositional reasoning for automotive systems. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 441–461. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_22
15. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 531–538. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_29
16. Foster, S., Huerta y Munive, J.J., Gleirscher, M., Struth, G.: Hybrid systems verification with Isabelle/HOL: simpler syntax, better models, faster proofs. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 367–386. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_20
17. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
18. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 63–78. Association for Computing Machinery, New York (2019)

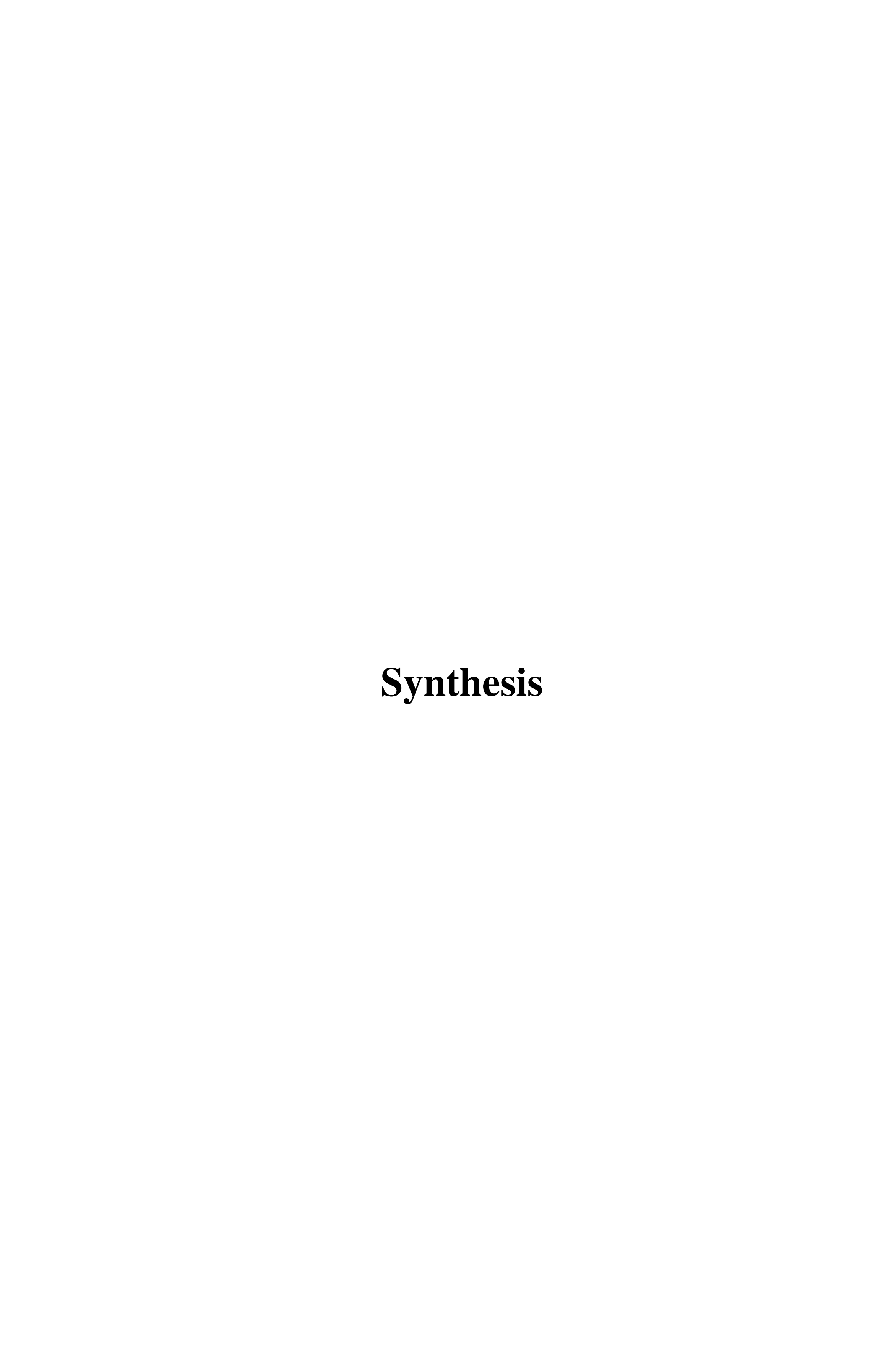
19. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
20. Geretti, L., et al.: Arch-comp20 category report: continuous and hybrid systems with nonlinear dynamics. In: Frehse, G., Althoff, M. (eds.) 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20). EPiC Series in Computing, vol. 74, pp. 49–75. EasyChair (2020). https://doi.org/10.29007/zkf6
21. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998)
22. Hoffmann, G.M., Tomlin, C.J., Montemerlo, M., Thrun, S.: Autonomous automobile trajectory tracking for off-road driving: controller design, experimental validation and racing. In: 2007 American Control Conference, pp. 2296–2301 (2007)
23. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 169–178 (2019)
24. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science (2011). https://doi.org/10.1007/978-3-031-02003-2. Morgan Claypool (November 2005), also available as Technical Report MIT-LCS-TR-917, MIT
25. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
26. Li, Y., Zhu, H., Braught, K., Shen, K., Mitra, S.: Verse: a Python library for reasoning about multi-agent hybrid system scenarios (2023)
27. Lim, H., Kaynar, D., Lynch, N., Mitra, S.: Translating timed I/O automata specifications for theorem proving in PVS. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 17–31. Springer, Heidelberg (2005). https://doi.org/10.1007/11603009_3
28. Lopez, P.A., et al.: Microscopic traffic simulation using SUMO. In: The 21st IEEE International Conference on Intelligent Transportation Systems. IEEE (2018)
29. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoret. Comput. Sci. **285**(2), 359–405 (2002). rewriting Logic and its Applications
30. Jiang, M., et al.: GRAIC: a simulator framework for autonomous racing. https://popgri.github.io/Race/ (2021)
31. Ray, R., Gurung, A., Das, B., Bartocci, E., Bogomolov, S., Grosu, R.: XSpeed: accelerating reachability analysis on multi-core processors. In: Piterman, N. (ed.) HVC 2015. LNCS, vol. 9434, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26287-1_1
32. Sibai, H., Li, Y., Mitra, S.: SceneChecker: boosting scenario verification using symmetry abstractions. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 580–594. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_28
33. Sibai, H., Mokhlesi, N., Fan, C., Mitra, S.: Multi-agent safety verification using symmetry transformations. In: TACAS 2020. LNCS, vol. 12078, pp. 173–190. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_10

34. Sibai, H., Mokhlesi, N., Mitra, S.: Using symmetry transformations in equivariant dynamical systems for their safety verification. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 98–114. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_6
35. Sun, D., Mitra, S.: NeuReach: learning reachability functions from simulations. In: TACAS 2022. LNCS, vol. 13243, pp. 322–337. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_17
36. Wu, C., Kreidieh, A., Parvate, K., Vinitsky, E., Bayen, A.M.: Flow: Architecture and benchmarking for reinforcement learning in traffic control. arXiv:1710.05465 (2017)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Synthesis

Counterexample Guided Knowledge Compilation for Boolean Functional Synthesis

S. Akshay(✉), Supratik Chakraborty(✉), and Sahil Jain

Indian Institute of Technology Bombay, Mumbai, India
{akshayss,supratik}@cse.iitb.ac.in,
sahil.jain1.c2022@iitbombay.org

Abstract. Given a specification as a Boolean relation between inputs and outputs, Boolean functional synthesis generates a function, called a Skolem function, for each output in terms of the inputs such that the specification is satisfied. In general, there may be many possibilities for Skolem functions satisfying the same specification, and criteria to pick one or the other may vary from specification to specification.

In this paper, we develop a technique to represent the space of Skolem functions in a criteria-agnostic form that makes it possible to subsequently extract Skolem functions for different criteria. Our focus is on identifying such a form and on developing a compilation algorithm for this form. Our approach is based on a novel counter-example guided strategy for existentially quantifying a subset of variables from a specification in negation normal form. We implement this technique and compare our performance with those of other knowledge compilation approaches for Boolean functional synthesis, and show promising results.

1 Introduction

Manually designing systems that satisfy complex user-provided specifications can be notoriously tricky. *Automated synthesis* has therefore attracted significant attention of researchers over the past few decades [1–5]. In this paradigm, a user describes the desired behaviour of a system as a relational specification between its inputs and outputs, and an algorithm automatically generates an implementation, such that the specification is provably satisfied. In this paper, we focus only on systems with Boolean inputs and outputs with relational specifications given as Boolean formulas. The synthesis problem in this setting is also called *Boolean functional synthesis*. Formally, let $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ be a Boolean formula representing the specification, where $\boldsymbol{X} = (x_1, \ldots x_m)$ is a vector of Boolean inputs and $\boldsymbol{Y} = (y_1, \ldots y_n)$ a vector of Boolean outputs of the system. Boolean functional synthesis requires us to generate a vector of Boolean functions $\boldsymbol{\Psi}(\boldsymbol{X}) = \big(\psi_1(\boldsymbol{X}), \ldots \psi_n(\boldsymbol{X})\big)$ such that $\forall \boldsymbol{X} \big(\exists \boldsymbol{Y} \varphi(\boldsymbol{X}, \boldsymbol{Y}) \Leftrightarrow \varphi(\boldsymbol{X}, \boldsymbol{\Psi}(\boldsymbol{X}))\big)$.

Authors names are in alphabetical order of last names

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 367–389, 2023.
https://doi.org/10.1007/978-3-031-37706-8_19

For each $i \in \{1, \ldots n\}$, the function $\psi_i(\boldsymbol{X})$ is called a Skolem function for y_i in $\varphi(\boldsymbol{X}, \boldsymbol{Y})$, and $\boldsymbol{\Psi}(\boldsymbol{X})$ is called a Skolem function vector.

There are several interesting applications of Boolean functional synthesis, including automated program synthesis, circuit repair and debugging, cryptanalysis and the like [2,6–10]. This has motivated researchers to develop novel algorithms for solving increasingly larger and more complex synthesis benchmarks [11–19]. Each such algorithm generates *a single Skolem function vector* for a given relational specification, thereby providing an implementation of the system. However, there may be many alternative function vectors that also serve as Skolem function vectors for the same specification. Some of these may yield system implementations that are more "desirable" than those obtained from other Skolem function vectors, when non-functional metrics like size of program/circuit needed for implementation, ease of understandability etc. are considered. Therefore, having a tool output a single Skolem function vector (chosen by the tool, without any user agency in the choice) can be restrictive in terms of implementation choices available to the user.

One way to address the above problem is to use a knowledge compilation approach, i.e. to compile the specification to a *special normal form* from which it is relatively easy to use downstream logic synthesis tools to generate any Skolem function vector optimizing user-specified criteria. Unfortunately, earlier work on knowledge compilation for Boolean functional synthesis [13,14,20] does not allow us to do this easily. They simply allow efficient synthesis of one (among possibly many) Skolem function vector from the compiled representation. Moreover, the user has no agency in choosing which Skolem function vector is synthesized; all choices are made implicitly deep inside heuristics of the compilation algorithms. For example, if we compile a relational specification to wDNNF [14] or SynNNF [13], the only guarantee we have is that the so-called GACKS Skolem functions (see [14]) can be efficiently synthesized from the compiled forms. But what if these functions are not the user's preferred choice of Skolem functions for an application? Unfortunately, not much can be done if we compile the specification to wDNNF or SynNNF. Similarly, the compilation approach proposed in [20] allows efficient synthesis of Skolem functions of yet another form, but even here, the user hardly has any agency in choosing which (among many alternative) Skolem function vectors is actually output. Existing algorithms therefore effectively restrict the *semantic choice* of Skolem functions with hardly any way for the user to influence this choice. Once the semantic choice has been made by the compiler, the only agency the user has is in *optimizing the implementation of this semantic choice*. We believe the inability of existing compilation approaches to allow the user semantic choice of Skolem functions is a limiting factor in practical usage of these works. In this paper, we take a first step towards remedying this problem.

The central question we ask in this paper is: *Can we compile a Boolean relational specification to a representation that does not restrict the semantic choice of Skolem functions, and yet allows easy deployment of downstream logic synthesis tools to obtain Skolem functions customized to user-provided criteria?* Our main result is an affirmative answer to this question. We also design and imple-

ment an algorithm that compiles a given specification in negation normal form to such a representation form, We emphasize that our goal in this paper is not to identify specific optimization criteria or to synthesize Skolem functions that optimize some specific criteria. Instead, we focus on developing a representation that makes it possible to use downstream logic optimization tools to synthesize Skolem functions satisfying user-provided criteria. Our experiments show that our approach is competitive performance-wise to earlier approaches that severely restrict the semantic choice of Skolem functions.

The primary contributions of this paper can be summarized as follows.

- We formalize the problem of symbolically and compactly representing all Skolem function vectors for a Boolean relational specification in such a way that it is amenable to downstream optimization by logic synthesis tools.
- We propose a candidate for this representation as a set of pairs of functions, one for each output, which we call the *Skolem basis vector*. We show that the Skolem basis vector is guaranteed to exist for any specification and is unique with respect to an ordering of the output variables.
- For single-output specifications, we show that the Skolem basis vector can be computed easily, as a pair of (semantically unique) Boolean functions. For multi-output specifications, we relate the problem of generating Skolem basis vector to the question of performing efficient quantification of outputs.
- We investigate two properties, namely *unateness and conflict-freeness of outputs*, that permit efficient quantification of outputs. This, in turn, allows a Skolem basis vector to be generated in polynomial time in special cases.
- We present a novel counterexample-guided algorithm for transforming a specification to one where a designated output variable is conflict-free. We call this process *rectification* of the output.
- We present an overall algorithm that takes a specification and generates a Skolem basis vector by successively rendering outputs unate or conflict-free.
- We present a tool implementing our algorithm, and report experimental results on a suite of publicly available benchmarks.

Related Work. In knowledge compilation, the general goal is to represent a problem specification in a form that allows specific questions to be answered efficiently (see e.g., [21–23]). In [22,24], representation forms for Boolean functions were proposed that allow efficient enumeration of all satisfying assignments of the function. However, this idea cannot be easily extended to enumerate Skolem functions, since the space of functions is doubly exponentially large in the number of variables. For Boolean functional synthesis, [13,20,25,26] provide normal forms and present compilers that render synthesis of a single Skolem function vector easy. However, they do not provide the user any agency in choosing the Skolem function vector. In fact, the optimizations used in [13] preclude generation of all Skolem function vectors for reasons of efficiency. In the current work, our focus is on symbolically representing the space of all Skolem function vectors, without necessarily converting the given specification to a semantically equivalent one in special normal form. Thus, the problem addressed in this paper is technically different from those addressed in [13,20,25,26]. Nevertheless, our work can be viewed as knowledge representation for all Skolem functions.

2 A Motivating Example

We start with a simple example that illustrates some of the problems we wish to address. Suppose we are designing a memoryless arbiter that must arbitrate requests from three users for a shared resource. Let the arbiter inputs be Boolean variables r_1, r_2, r_3, where r_i is true iff there is a request from user i. Let the corresponding arbiter outputs be g_1, g_2, g_3, where g_i is true iff access is granted to user i. We want the arbiter to satisfy the following properties: (a) at most one user must be granted access at a time, (b) if some user has requested access, some user must be granted access, and (c) a user should be granted access only if she has requested. The above properties can be encoded as a specification $\varphi \equiv \varphi_1 \wedge \varphi_2 \wedge \varphi_3$, where $\varphi_1 \equiv \big(g_1 \Rightarrow \neg(g_2 \vee g_3)\big) \wedge \big(g_2 \Rightarrow \neg(g_1 \vee g_3)\big) \wedge \big(g_3 \Rightarrow \neg(g_1 \vee g_2)\big)$, $\varphi_2 \equiv (r_1 \vee r_2 \vee r_3) \Rightarrow (g_1 \vee g_2 \vee g_3)$, and $\varphi_3 \equiv (g_1 \Rightarrow r_1) \wedge (g_2 \Rightarrow r_2) \wedge (g_3 \Rightarrow r_3)$.

It turns out that there are many different Skolem function vectors $\boldsymbol{\Psi} = (\psi_1, \psi_2, \psi_3)$ for the above specification, where each ψ_i gives a Skolem function for g_i. We ran two state-of-the-art Boolean functional synthesis tools, viz. Manthan2 [17] and BFSS [14], on this specification. BFSS required us to also specify a linear order of outputs (we will shortly see why), and we used $g_1 \prec g_2 \prec g_3$. Both tools solved the problem in no time, and each reported a Skolem function vector *without any room for the user to influence the choice of Skolem functions.* Specifically, the Skolem functions returned by Manthan2 can be represented by the And-Inverter Graph (AIG) shown in Fig. 1a. Here, each circle represents a two-input AND gate, and each dotted (resp. solid) edge represents a connection with (resp. without) logical negation. Thus, the Skolem functions are: $\psi_2 \equiv r_2 \wedge \neg r_1 \wedge \neg r_3$, $\psi_1 \equiv r_1 \wedge \neg r_3 \wedge \neg g_2$ and $\psi_3 \equiv r_3 \wedge \neg g_1 \wedge \neg g_2$. Running BFSS on the same specification yields Skolem functions represented by the AIG in Fig. 1c. Here, $\psi_3 \equiv r_3 \wedge \neg r_1 \wedge \neg r_2$, $\psi_2 \equiv r_2 \wedge \neg g_3$ and $\psi_1 \equiv r_1 \wedge \neg g_2 \wedge \neg g_3$.

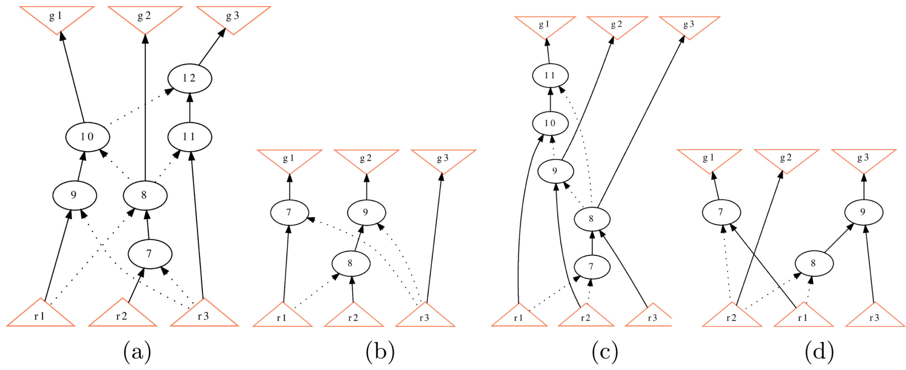


Fig. 1. Unoptimized and optimized AIGs of Skolem functions

Are the Skolem functions generated by the two tools in their simplest forms, and did they miss out some possibilities of optimization? To answer this, we used

a widely used logic optimization tool, viz. abc [27], to simplify the two AIGs using commands to minimize the AND gate count and to balance lengths of paths in the AIGs. The resulting simplified AIGs are shown in Fig. 1b (obtained from Fig. 1a) and Fig. 1d (obtained from Fig. 1c). Thus, Manthan2's solution is equivalent to $\psi_3 \equiv r_3$, $\psi_2 \equiv r_2 \wedge \neg r_1 \wedge \neg r_3$, $\psi_1 \equiv r_1 \wedge \neg r_3$, while BFSS' solution is equivalent to $\psi_2 \equiv r_2$, $\psi_1 \equiv r_1 \wedge \neg r_2$, $\psi_3 \equiv r_3 \wedge \neg r_1 \wedge \neg r_2$. Note that the two solutions are semantically equivalent modulo permutaton of indices (although this wasn't obvious prior to optimization).

There are some important take-aways from this simple experiment. First, neither Manthan2 nor BFSS gave the user any agency in the semantic choice of the synthesized Skolem functions. The use of the abc tool with user-provided optimization criteria at the end simply gave us choice of implementation for the Skolem functions already determined by each tool. Significantly, there are choices of Skolem function vectors, viz. $\psi_1 \equiv r_1 \wedge (\neg r_2 \vee \neg r_3)$, $\psi_2 \equiv r_2 \wedge (\neg r_1 \vee r_3)$, $\psi_3 \equiv (\neg r_1 \wedge \neg r_2 \wedge r_3)$, that are *ignored* by both Manthan2 and BFSS (and by other tools like CADET [11]). This can lead to ignoring "better" Skolem function vectors in general. The user's criteria for desirability of Skolem functions may differ from one problem instance to another, and may be completely different from what is hard-coded in the innards of a tool like Manthan2/BFSS. For example, the new Skolem function vector considered above admits an AIG representation in which input-to-output shortest (resp. longest) path lengths are equal across all outputs. This may indeed be a desirable feature in some application where variability of output delays matters. However, there is currently no way to influence BFSS/Manthan2 to arrive at Skolem functions optimized per such criteria.

The above example also illustrates the important role played by logic optimization in obtaining efficient implementations of Skolem functions generated by state-of-the-art synthesis tools. However, using logic optimization as a post-processor can only provide a better implementation of already chosen (semantically) Skolem functions. Fortunately, more than five decades of research in logic optimization has resulted in mature (even commercial) tools that can do much more than just implementation optimization. Specifically, don't-care based optimizations [28] can search within a specified space of (semantically distinct) functions to choose one that is optimized according to a given user criteria. Such a choice involves a combined optimization across semantic and implementation choices. Given this capability of logic optimizers, and their indispensable use in synthesis flows, we posit that logic optimizers are the right engines to choose between alternative semantic choices of Skolem functions, in addition to optimizing their implementation. Of course, this requires specifying the semantic space of all (Skolem) functions in a form that can be easily processed by logic optimizers. State-of-the-art logic optimizers already allow specifying a family of functions using *on-sets* and *don't-care sets* [29]. Therefore, we propose to use this representation for representing the space of Skolem functions as well.

Before presenting the details of on-sets and don't-care sets for Skolem functions in our example, we note that Skolem functions for different outputs cannot

be chosen independently in general. For example, $\psi_3 \equiv r_3$ is generated by Manthan2, and $\psi_2 \equiv r_2$ is generated by BFSS. However, there is no Skolem function vector with $\psi_2 \equiv r_2$ and $\psi_3 \equiv r_3$), since this would lead to $g_2 = g_3 = 1$ when $r_2 = r_3 = 1$. Therefore, any representation of the semantic space of all Skolem function vectors *must necessarily* take into account dependence between Skolem functions for different outputs. One way to achieve this is to impose a linear order on the outputs, and to represent the set of Skolem functions for an output in terms of Skolem functions for preceding (in the order) outputs. With this approach, the semantic space of Skolem functions for each output can be expressed by two functions: one representing the set of assignments for which every Skolem function in the represented space must evaluate to 1 (i.e. on-set), and the other representing assignments for which it is ok for a Skolem function to evaluate to either 0 or 1 (i.e. don't-care set).

The above representation is analogous to representing vector spaces using a small set of mutually orthogonal basis vectors, where every vector in the space can be expressed as a linear combination of these basis vectors. In a similar manner, let A denote the on-set of a family of Skolem functions, and B denote the don't-care set for the same family. Let $\mathsf{GenImpl}(B)$ denote the set of all *generalized implicants* of B, i.e. all formulas ν such that $\nu \Rightarrow B$. Every Skolem function in the represented space can then be obtained (modulo semantic equivalence) as $A \vee \nu$ where $\nu \in \mathsf{GenImpl}(B)$. Specifically, for our example, with $g_1 \prec g_2 \prec g_3$ of outpus (same as that given to BFSS), we have $A_1 \equiv (\neg r_3 \wedge \neg r_2 \wedge r_1)$, $B_1 \equiv (r_3 \vee r_2) \wedge r_1$, $A_2 \equiv (\neg r_3 \wedge r_2 \wedge \neg g_1)$, $B_2 \equiv r_3 \wedge r_2 \wedge \neg g_1$, $A_3 \equiv r_3 \wedge \neg g_2 \wedge \neg g_1$, $B_3 = 0$. The Karnaugh-maps shown below depict how the space of all Skolem function vectors can be visualized in terms of A_i and B_i. To obtain a specific Skolem function vector, we must place a 1 in each A_i-cell, choose a subset of the B_i cells and place 1's in those cells and 0's in the balance B_i cells. Each such choice provides a semantically distinct Skolem function vector, and every Skolem function vector corresponds to one such choice. Specifically, the Skolem function vector missed by Manthan2/BFSS can now be easily obtained by choosing the red and blue B_1 cells and the teal B_2 cell to be 1 in the Karnaugh-maps. Similarly, Manthan2's solution is obtained by choosing the blue B_1 cell and teal B_2 cell to be 1, and BFSS' solution is obtained by choosing the red B_1 cell and teal B_2 cell to be 1. Allowing a logic optimizer to optimize Skolem functions with the spaces represented by $(A_1, B_1, A_2, B_2, A_3, B_3)$ therefore makes it possible to synthesize each of these Skolem function vectors. This motivates compiling a given specification into an (A_i, B_i) pair for the Skolem functions for each output y_i.

$r_2r_3 \rightarrow$ / $r_1 \downarrow$	00	01	11	10
0	0	0	0	0
1	A_1	B_1	B_1	B_1

Space of Sk fns for g_1

$r_2r_3 \rightarrow$ / $g_1 \downarrow$	00	01	11	10
0	0	0	B_2	A_2
1	0	0	0	0

Space of Sk fns for g_2

$g_2r_3 \rightarrow$ / $g_1 \downarrow$	00	01	11	10
0	0	A_3	0	0
1	0	0	0	0

Space of Sk fns for g_3

3 Preliminaries and Notation

Let $\boldsymbol{Z} = (z_1, \ldots, z_n)$ be a vector of Boolean variables. A *literal* is a variable (z_i) or its complement ($\neg z_i$), a *clause* is a disjunction of literals and a *cube* is a conjunction of literals. For $1 \leq i \leq j \leq n$, we use $\boldsymbol{Z}_i^j$ to denote the *slice* $(z_i, \ldots z_j)$ of the vector $\boldsymbol{Z}$. An n-input Boolean function is a mapping from $\{0,1\}^n$ to $\{0,1\}$. A Boolean formula $\varphi(\boldsymbol{Z})$ is a syntactic object whose semantics is given by a mapping from $\{0,1\}^n$ to $\{0,1\}$. Thus, every Boolean formula represents a unique Boolean function, and every Boolean function can be represented by a (not necessarily unique) Boolean formula. Henceforth, we refer to Boolean formulas and Boolean functions interchangeably.

The *support* of $\varphi(\boldsymbol{Z})$, denoted $\mathsf{sup}(\varphi)$, is the set of variables in $\boldsymbol{Z}$. For ease of exposition, we will abuse notation and use $\boldsymbol{Z}$ to denote either a vector or the underlying set of elements, depending on the context. A complete (resp. partial) *assignment* π for $\boldsymbol{Z}$ is a complete (resp. partial) mapping from $\boldsymbol{Z}$ to $\{0,1\}$. The value of variable z_i assigned by π is denoted $\pi[z_i]$. A complete assignment π of $\boldsymbol{Z}$ is a *satisfying assignment* for $\varphi(\boldsymbol{Z})$ if the Boolean function represented by φ evaluates to 1 when all variables in $\mathsf{sup}(\varphi)$ are assigned values given by π. In this case, we say that $\pi \models F$. A formula $\varphi(\boldsymbol{Z})$ is *satisfiable* if it has at least one satisfying assignment; otherwise it is *unsatisfiable*. We say that two formulas on n variables are *equivalent* if they represent the same semantic mapping from $\{0,1\}^n$ to $\{0,1\}$. Given Boolean formulas φ and α with $z_i \in \mathsf{sup}(\varphi)$, we use $\varphi[z_i \mapsto \alpha]$ to denote the formula obtained by substituting α for every occurrence of z_i in φ. We use $\varphi\big|_{z_i=1}$ (resp. $\varphi\big|_{z_i=0}$) to denote the formula obtained by setting z_i to 1 (resp. 0) in the formula $\varphi(\boldsymbol{Z})$. The resulting formulas are also called positive (resp. negative) co-factors of φ w.r.t. z_i. For notational convenience, we use $\varphi\big|_{\pi}$ to denote the formula obtained by repeatedly co-factoring φ using the (possibly partial) assignment of variables given by π. As discussed in Sect. 2, we say that a function $\varphi'(\boldsymbol{Z})$ is a *generalized implicant* of $\varphi(\boldsymbol{Z})$ if $\varphi'(\boldsymbol{Z}) \Rightarrow \varphi(\boldsymbol{Z})$. This generalizes the notion of implicants used in the literature, which are restricted to be cubes. The set of all generalized implication of φ is denoted $\mathsf{GenImpl}(\varphi)$.

A Boolean formula $\varphi(\boldsymbol{Z})$ can be represented as a circuit or a Directed Acyclic Graph (DAG) consisting of $\neg$, $\wedge$ and $\vee$ gates, with literals at leaves. Further, it can be converted to a semantically equivalent formula in *Negation Normal Form (NNF)*, i.e., with no $\neg$-labelled internal nodes, in time linear in the size of the circuit. We consider formulas to be given in NNF unless mentioned otherwise, and interchangeably refer to a Boolean formula and the circuit representing it. If an NNF formula in *Conjunctive Normal Form (CNF)*, i.e., as conjunction of clauses, is unsatisfiable, then there is a subset of its clauses whose conjunction is unsatisfiable. This set is called its *unsatisfiable core*, and a *minimal unsatisfiable core* is one without any proper subset that is also an unsatisfiable core.

The *Boolean functional synthesis* problem, and notions of *Skolem functions* and *Skolem function vectors* have already been defined in Sect. 1. Let $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ be a Boolean relational specification over inputs $\boldsymbol{X}$ and outputs $\boldsymbol{Y}$. A commonly used approach, adopted by several Boolean functional synthesis algorithms [6, 14–16], works as follows. Without loss of generality, let $y_1 \prec \cdots \prec y_n$ be a

linear ordering of the outputs in $\boldsymbol{Y}$. We first define a set of derived specifications $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_i^n)$ for all $i \in \{1, \ldots n\}$, where $\varphi^{(i)} \Leftrightarrow \exists \boldsymbol{Y}_1^{i-1} \varphi(\boldsymbol{X}, \boldsymbol{Y})$. Next, for each $i \in \{1, \ldots n\}$, we find a Skolem function for y_i from the derived specification $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_i^n)$, by treating y_i as the sole output and all of $\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n$ as inputs in $\varphi^{(i)}$. Let $\psi_i(\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n)$ denote the Skolem function for y_i thus obtained. Finally, we substitute the Skolem functions $\psi_{i+1}, \ldots \psi_n$ for $y_{i+1}, \ldots y_n$ respectively in the Skolem function ψ_i obtained above. This gives a Skolem function for y_i only in terms of $\boldsymbol{X}$. By repeating the above process for all i in decreasing order from $n-1$ to 1, we obtain a Skolem function vector for φ.

4 A New Knowledge Representation for Skolem Functions

We start with a key definition that is motivated by the desire to represent the entire space of Skolem functions arising from a specification compactly, and in a form that is easily amenable to well-established logic synthesis and optimization workflows. Recall from Sect. 2 that for a multi-output specification, Skolem functions for different outputs may be dependent on each other. Hence, the set of Skolem function vectors cannot be expressed as a Cartesian product of sets of Skolem functions for individual outputs. Instead, we impose a linear order on the outputs, and express the Skolem function for one output in terms of the inputs and other outputs that precede it in the order. Such a linear order may be automatically generated, user-provided, or even generated with guidance from the user, e.g., if the user provides a partial order on the outputs. We assume the availability of such an order $\prec$ in the definition below.

Definition 1. *Let $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ be a specification over a linearly ordered set of outputs $\boldsymbol{Y} = \{y_1, \ldots, y_n\}$. We say that output y_i has a Skolem basis in φ if there exists a pair of functions (A_i, B_i) over $\boldsymbol{X} \cup \boldsymbol{Y}_{i+1}^n$ such that*

1. *$A_i \wedge B_i$ is unsatisfiable, and*
2. *any Skolem function $\psi_i(\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n)$ for y_i in the derived specification $\varphi^{(i)}$ can be written as $\psi_i \equiv A_i \vee g$ for some $g \in \mathsf{GenImpl}(B_i)$.*

We call the vector of pairs $\langle (A_i, B_i) \rangle_{1 \leq i \leq n}$ the Skolem basis vector for φ wrt $\prec$.

The Skolem basis vector can be seen as a succinct representation of the Skolem function space, i.e., the set of all Skolem function vectors of φ. A natural question that arises at this point is: *Given a specification φ and order $\prec$ of outputs, does there always exist a Skolem basis for φ wrt $\prec$*? Fortunately, as we show in this paper, the answer is a resounding "Yes". Not only that, the Skolem basis for a given φ and $\prec$ is unique upto semantic equivalence of the basis functions. It is important to note that not every set of functions can be represented using just two basis functions. This is easy to see via a counting argument: the number of sets of Boolean functions over m inputs is $2^{2^{2^m}}$. However, the number of sets that admit a Skolem basis is (loosely) upper bounded by $2^{2 \cdot 2^m}$. Skolem

functions are therefore special, since we show that the space of all Skolem functions for every output in every specification always admits representation by two basis functions, regardless of the order $\prec$. Interestingly, though the definition of Skolem basis vector needs us to specify an order $\prec$ on the outputs, somewhat surprisingly, the Skolem function space itself does not depend on the order.

Proposition 1. *Suppose $\boldsymbol{\Psi}$ is a Skolem function vector for the outputs $\boldsymbol{Y}$ in terms of inputs $\boldsymbol{X}$ in φ. Then, for any order $\prec$, $\boldsymbol{\Psi}$ can be generated using the Skolem basis vector of φ wrt $\prec$, and then substituting, for each $i \in \{1, \ldots n\}$, the Skolem functions ψ_j for y_j where $i < j \leq n$, in the Skolem function for ψ_i.*

Proof Sketch: With ordering $y_1 \prec y_2 \prec \ldots y_n$, let $\langle (A_i, B_i) \rangle$ be the corresponding Skolem basis vector. The support of A_n, B_n are only the inputs $\boldsymbol{X}$, while the support of A_i, B_i (for $i > 1$) are $\boldsymbol{X} \cup \{y_{i+1}, ...y_n\}$. Let $\boldsymbol{\Psi} = (\psi_1, \ldots \psi_n)$ be an arbitrary Skolem function vector, where each ψ_i is a function of $\boldsymbol{X}$. By definition of Skolem basis, since ψ_n is a Skolem function for y_n, it can be obtained from A_n and B_n (each of which has support $\boldsymbol{X}$). Now consider ψ_i for $1 \leq i < n$. By definition of Skolem basis, every Skolem function for y_i in terms of $\boldsymbol{X} \cup \{y_{i+1}, ...y_n\}$ can be obtained from A_i and B_i. In particular, if we set y_{i+1} to ψ_{i+1} and so on until y_n to ψ_n, every Skolem function for y_i in terms of $\boldsymbol{X}$ can be obtained from A_i and B_i. $\square$

Another interesting property about Skolem basis vector is that, when it exists, it is unique. Later we will show (constructively) that it always exists and hence we would have also constructed the unique one.

Proposition 2. *For any y_i in φ, its Skolem basis, when it exists, is unique.*

Proof. Fix i. Let S be the set of all Skolem functions for y_i in $\varphi^{(i)}$. From Definition 1, we know that for all $f \in S$, $A_i \Rightarrow f$. Hence, $A_i \Rightarrow \bigwedge_{f \in S} f$. However, we also know that $A_i \in S$ (corresponds to choosing the generalized implicant 0 from $\mathsf{GenImpl}(B_i)$). Therefore, $\left(\bigwedge_{f \in S} f\right) \Rightarrow A_i$. It follows from the two implications that $A \Leftrightarrow \bigwedge_{f \in S} f$.

In a similar manner, Definition 1 implies that for all $f \in S$, $f \Rightarrow A_i \vee B_i$. Hence $\left(\bigvee_{f \in S} f\right) \Rightarrow A_i \vee B_i$. However, we know that $A_i \vee B_i \in S$ (corresponds to choosing the generalized implicant B from $\mathsf{GenImpl}(B)$). Therefore, $A_i \vee B_i \Rightarrow \bigvee_{f \in S} f$. It follows from the two implications that $B_i \Leftrightarrow \bigvee_{f \in S} f$. $\square$

Finally, we explain how our new representation of Skolem functions using a Skolem basis vector naturally lends itself to easy processing by downstream logic synthesis and optimization tools. Thus, a Skolem basis vector is not just an arbitrary way to represent the space of all Skolem function vectors; instead, it is strongly motivated by the way modern logic synthesis and optimization tools work to search the semantic space of partially specified functions (i.e. functions specified with on-sets and don't-care sets). Specifically, in logic synthesis and optimization parlance [29], A_i is the *on-set* and B_i is the *don't-care set* for Skolem functions for y_i in φ. In other words, A_i describes all assignments for which every Skolem function for y_i must evaluate to 1 while B_i describes those assignments

on which a Skolem function can evaluate to either 1 or 0 without violating the requirement of being a Skolem function for y_i in φ. Thus, every semantically distinct Skolem function for y_i in φ can be obtained by choosing a distinct subset of satisfying assignments of B_i and choosing the Skolem function to evaluate on this subset of assignments in addition to those determined by A_i. Indeed, state-of-the-art logic synthesis and optimization tools (such as abc [27]) use on-sets and don't care sets expressed as Boolean functions to represent the space of all realizations of a partially specified function. The don't cares are then used to optimize the semantic and implementation choices when choosing the optimal realization of such a partially specified function, as per user provided criteria like area, gate count, delay, power consumption, balance of delays across paths etc. Indeed, the following guarantee follows rather trivially from Proposition 1.

Proposition 3. *Suppose we have access to a logic optimization tool that finds the optimal semantic and implementation choice of a partially specified function as per user criteria. Using this tool on the Skolem basis vector of φ wrt $\prec$ yields the optimal choice among all Skolem functions, where optimality of Skolem function for y_i is conditioned on the choice of Skolem functions for y_j, for $1 \leq j < i$.*

Having defined and motivated the Skolem basis vector as our new knowledge representation, in the rest of the paper we will show how it can actually be computed, *in theory and in practice.*

5 Towards Synthesizing the Skolem Basis Vector

The Single Output Case: First, we consider the case of a singleton output and show that here the existence of Skolem basis is easy to establish, and the basis is also easy to compute.

Theorem 1. *For a single-output specification $\varphi(\boldsymbol{X}, y)$, the Skolem basis for y in φ is given by $A \equiv \varphi(\boldsymbol{X}, 1) \wedge \neg\varphi(\boldsymbol{X}, 0)$ and $B \equiv \varphi(\boldsymbol{X}, 1) \leftrightarrow \varphi(\boldsymbol{X}, 0)$. Thus, in this case, the Skolem basis vector for φ can be computed in time/space linear in size of the circuit representing φ.*

Proof. Let $2^{|\boldsymbol{X}|}$ denote the set of all complete assignments π of $\boldsymbol{X}$. Define $S_1 = \{\pi \mid \pi \in 2^{|\boldsymbol{X}|},\ \pi \models \varphi(\boldsymbol{X}, 1)\}$ and $S_0 = \{\pi \mid \pi \in 2^{|\boldsymbol{X}|},\ \pi \models \varphi(\boldsymbol{X}, 0)\}$. By definition of S_0 and S_1, (with $\overline{S_i}$ denoting complement of set S_i), we have:

- $\pi \in S_1 \cup S_0$ iff $\pi \models \exists y\, \varphi(\boldsymbol{X}, y)$.
- $\pi \in S_1 \cap S_0$ iff $\pi \models \forall y\, \varphi(\boldsymbol{X}, y)$.
- $\pi \in \overline{S_1} \cap \overline{S_0}$ iff $\pi \models \forall y\, \neg\varphi(\boldsymbol{X}, y)$.
- For every $\pi \in S_1 \cap \overline{S_0}$, the only value of y that makes $\varphi(\pi, y)$ true is 1.
- For every $\pi \in S_0 \cap \overline{S_1}$, the only value of y that makes $\varphi(\pi, y)$ true is 0.

Now let $\psi(\boldsymbol{X})$ be an arbitrary Skolem function for y in $\varphi(\boldsymbol{X})$. Recall that by definition a Skolem function satisfies $\forall \boldsymbol{X} \left(\exists y\, \varphi(\boldsymbol{X}, y) \Leftrightarrow \varphi(\boldsymbol{X}, \psi(\boldsymbol{X}))\right)$. It then follows from the above observations that if $\pi \in S_1 \cap \overline{S_0}$, $\psi(\pi)$ must evaluate to

1. Similarly, if $\pi \in (S_1 \cap S_0) \cup (\overline{S_1} \cap \overline{S_0})$, it makes no difference whether $\psi(\pi)$ evaluates to 0 or 1. Finally, if $\pi \in S_0 \cap \overline{S_1}$, $\psi(\pi)$ must evaluate to 0. Since ψ was an arbitrary Skolem function for y in φ, we infer that the Skolem basis for $\mathsf{AllSk}(\varphi)$ is (A, B), where $A \equiv \varphi(\boldsymbol{X}, 1) \wedge \neg\varphi(\boldsymbol{X}, 0)$ represents the set $S_1 \cap \overline{S_0}$, and $B \equiv \big(\varphi(\boldsymbol{X}, 0) \Leftrightarrow \varphi(\boldsymbol{X}, 1)\big)$ represents the set $(S_1 \cap S_0) \cup (\overline{S1} \cap \overline{S_0})$. □

We next consider the multiple output case, where our strategy (as done usually for Skolem *function* synthesis) is to reduce to the one-output case above.

Multiple Outputs and Existential Quantification: When we have multiple outputs, from the definition of Skolem basis vector (Definition 1), it follows that the problem reduces to the single output case, if we can compute the derived specifications $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_{i+1}^n)$. Unfortunately, computing $\varphi^{(i)}(\boldsymbol{X}, \boldsymbol{Y}_i^n)$ cannot always be done efficiently, even when $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and the order $\prec$ on $\boldsymbol{Y}$ are given. We compute $\varphi^{(i)}$ from a given $\varphi^{(i-1)}$, where the variable y_i to be quantified is either chosen on-the-fly (giving a dynamic computation of $\prec$) or determined as per a statically provided order. Since $\varphi^{(i+1)} \Leftrightarrow \exists \boldsymbol{Y}_1^i \varphi \Leftrightarrow \exists y_i \varphi^{(i)}$ for all $i \in \{1, \ldots n-1\}$, we first consider how a single output variable can be quantified from a derived specification.

The conceptually simplest way to compute $\exists y_i\, \varphi^{(i)}$ is as $\varphi^{(i)}\big|_{y_i=1} \vee \varphi^{(i)}\big|_{y_i=0}$. Unfortunately, this doubles the size of the circuit representation. An alternative is to find a Skolem function, say ψ_i, for y_i in $\varphi^{(i)}$, and then use $\varphi^{(i)}[y_i \mapsto \psi_i]$. This works well when ψ_i can be represented compactly. However, an NNF representation of ψ_i can be as large as that of $\varphi^{(i)}$ (e.g. if $\psi_i \equiv \varphi^{(i)}\big|_{y_i=1}$), in which case we may double the circuit size. We therefore ask *if it is possible to compute* $\exists y_i\, \varphi^{(i)}$ *by simply substituting a constant (not necessarily a Skolem function) for* y_i *in an NNF formula of almost the same size as* $\varphi^{(i)}$. It turns out that this is possible in two practically relevant cases. In other cases, we transform the circuit to permit such constant substitutions. For notational convenience, in the rest of this section, we omit i and use y and φ for y_i and $\varphi^{(i)}$.

The Case of Unates: A variable y is *positive (resp. negative) unate* in φ if $\varphi\big|_{y=0} \Rightarrow \varphi\big|_{y=1}$ (resp. $\varphi\big|_{y=1} \Rightarrow \varphi\big|_{y=0}$). A variable is *unate* in φ if it is either positive or negative unate in φ. Then, we have: easily proved.

Lemma 1. *If y is positive unate in φ, then $\exists y\ \varphi \Leftrightarrow \varphi\big|_{y=1}$. Similarly, if y is negative unate in φ, then $\exists y\ \varphi \Leftrightarrow \varphi\big|_{y=0}$.*

Proof. The proof immediately from the definition of positive and negative unateness, and from the fact that $\exists y\, \varphi \Leftrightarrow \varphi\big|_{y=0} \vee \varphi\big|_{y=1}$. □

As an example, consider $\varphi \equiv (x \wedge (y_1 \vee y_2)) \vee (\neg x \wedge \neg y_2)$. Here, y_1 is positive unate in φ, but y_2 is not unate in φ. However, y_2 is negative unate in $\varphi\big|_{y_1=1}$, which by Lemma 1 is equivalent to $\exists y_1\, \varphi$. This shows that even if a variable is not unate to begin with, it may become unate after some variables are quantified. If we use the order $y_1 \prec y_2$ in our example, both $\exists y_1\, \varphi$ and $\exists y_1 \exists y_2\, \varphi$ can be computed by substituting for y_1 and y_2 in φ. This is however not true for $y_2 \prec y_1$.

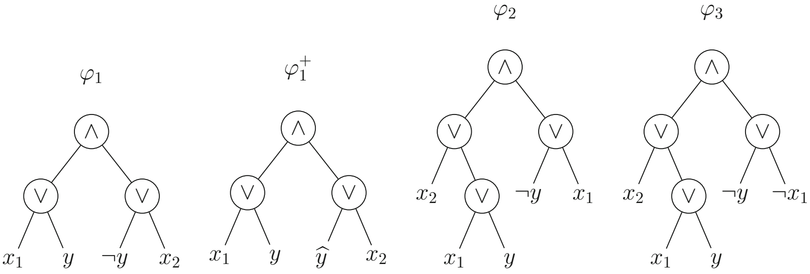


Fig. 2. NNF circuit representations of formula $\varphi_1, \varphi_1^+, \varphi_2, \varphi_3$.

In general, given a specification $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and a linear ordering $\prec$ of outputs, if each output y_i is unate in the derived specification $\varphi^{(i)} \equiv \exists \boldsymbol{Y}_1^{i-1} \varphi$, then we can apply Lemma 1, Definition 1 and Theorem 1 to synthesize the entire Skolem basis vector for φ w.r.t. $\prec$ efficiently. This also suggests a heuristic for finding a (partial) order on the outputs $\boldsymbol{Y}$. Specifically, given a derived specification $\varphi^{(i)}$, we try to find an output variable y in its support such that y is unate in $\varphi^{(i)}$. If such a variable exists, we use it as the next variable in the $\prec$ order, and obtain $\varphi^{(i+1)}$ by using Lemma 1 to compute $\exists y\, \varphi^{(i)}$. As our experiments show (see Sect. 7) and has also been observed elsewhere [14], this approach is surprisingly effective for finding Skolem functions for many benchmarks.

The Case of No Conflicts: Next, we consider another case where quantification can be achieved by substituing constants for variables.

Definition 2. *Let φ be an NNF formula, $y \in \mathsf{sup}(\varphi)$. Suppose we replace every occurence of $\neg y$ in φ by a fresh variable $\widehat{y}$ $(\widehat{y} \notin \mathsf{sup}(\varphi))$. The resulting formula is called the y-*positive form *of φ and is denoted φ^{+y}. The variable y is said to be* in conflict *in φ if there exists an assignment $\pi : \mathsf{sup}(\varphi) \setminus \{y\} \rightarrow \{0, 1\}$ such that $\varphi^{+y}\big|_\pi \Leftrightarrow y \wedge \widehat{y}$. Otherwise, we say that y is* conflict-free *in φ^{+y}.*

The assignment π in the above definition is called a *counterexample to conflict-freeness of y in φ*. It is easy to see that both y and $\widehat{y}$ are positive unate in φ^{+y}. Henceforth, we use φ^+ instead of φ^{+y} when y is clear from the context.

We illustrate conflicts and conflict-freeness in Fig. 2. The y-positive form of φ_1 is shown as φ_1^+, where $\widehat{y}$ is a fresh variable. Clearly, y is in conflict in φ_1 since $\varphi_1^+\big|_\pi \Leftrightarrow y \wedge \widehat{y}$ for $\pi : x_1 \mapsto 0, x_2 \mapsto 0$. Similarly, y is in conflict in φ_2 (as seen with $\pi : x_1 \mapsto 0, x_2 \mapsto 0$). However, y is not in conflict in φ_3 as there is no assignment π of x_1, x_2 for which $\varphi_3^+\big|_\pi \Leftrightarrow y \wedge \widehat{y}$.

Lemma 2. *If y is conflict-free in φ, then $\exists y\, \varphi \Leftrightarrow \varphi^+\big|_{y=1,\widehat{y}=1}$.*

Proof. Since y is conflict-free in φ, it follows that $\varphi^+\big|_{y=1,\widehat{y}=1} \Rightarrow \big(\varphi^+\big|_{y=1,\widehat{y}=0} \vee \varphi^+\big|_{y=0,\widehat{y=1}}$. Since all internal nodes in φ^+ are labeled by either $\wedge$ or $\vee$, it also follows that y and $\widehat{y}$ are positive unate in φ^+. Therefore, $\big(\varphi^+\big|_{y=1,\widehat{y}=0} \vee$

$\varphi^+|_{y=0,\widehat{y}=1} \Rightarrow \varphi^+|_{y=1,\widehat{y}=1}$. The proof is completed by observing that by definition $\exists y\, \varphi \Leftrightarrow \big(\varphi|_{y=0} \vee \varphi|_{y=1}\big) \Leftrightarrow \big(\varphi^+|_{y=0,\widehat{y}=1} \vee \varphi^+|_{y=1,\widehat{y}=0}\big)$. □

A notion similar to conflict as defined above was used in [13,20] for defining normal forms for synthesis. The difference is that unlike in [13,20], we do not require a pre-specified subset of the support to be set to 1 in the assignment π. To identify conflicts, we define a *conflict formula* $\kappa_{\varphi,y}$ as $\big(\varphi^+|_{y=1,\widehat{y}=1} \wedge \neg\varphi^+|_{y=1,\widehat{y}=0} \wedge \neg\varphi^+|_{y=0,\widehat{y}=1}\big)$. By Definition 2, y is conflict-free in φ iff $\kappa_{\varphi,y}$ is unsatisfiable.

Proposition 4. *For $1 \leq i \leq 4$, there exist φ_i with $y_i \in \sup(\varphi_i)$ s.t., (i) y_1 is neither unate nor conflict-free in φ_1, (ii) y_2 is unate but not conflict-free in φ_2, (iii) y_3 is conflict-free but not unate in φ_3, (iv) y_4 is unate, conflict-free in φ_4.*

The formulas $\varphi_1, \varphi_2, \varphi_3$ from Fig. 2 satisfy conditions (i), (ii) and (iii) respectively. For (iv), we consider $\varphi_4 \equiv x \wedge y$, in which y is unate and conflict-free. Lemmas 1, 2 and Proposition 4 show that both unateness and conflict-freeness are independently useful, and hence combining we directly obtain:

Theorem 2. *Given $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and a linear order $\prec$ on $\boldsymbol{Y}$, if y_i is either unate or conflict-free in $\varphi^{(i)}$ for all $i \in \{1, \ldots n\}$, then we can effectively synthesize the Skolem basis vector in time linear in size of φ.*

We remark that the implications of Theorem 2 go beyond what can be achieved by earlier work on normal forms for synthesis [13,20]. Indeed, there are formulas that are neither in SynNNF nor SAUNF but for which Theorem 2 applies.

Finally, unateness is a semantic property; hence if y is not unate in φ, it is not unate in every μ such that $\varphi \Leftrightarrow \mu$. However, conflict-freeness has a representational aspect. If y is in conflict in φ, we can *always* find another NNF formula μ such that (i) $\mu \Leftrightarrow \varphi$, and (ii) y is conflict-free in μ. To see why, note that if $\mu \equiv (y \wedge \varphi|_{y=1}) \vee (\neg y \wedge \varphi|_{y=0})$, i.e. Shannon expansion of φ w.r.t. y, then $\mu \Leftrightarrow \varphi$ and y is conflict-free in μ. However, taking the Shannon expansion may not always be the best way to render an output conflict-free, as it often leads to blow-up in the size of the expanded formula. In the next section, we give a counterexample guided algorithm to obtain μ from φ and y, that works much more efficiently than Shannon expansion in practice.

6 Counterexample-Guided Rectification

Recall from the previous section that if y is in conflict in $\varphi(\boldsymbol{X}, \boldsymbol{Y})$, then there exists a counterexample (assignment) $\pi : \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\} \to \{0,1\}$ such that $\varphi^+|_{\pi} \Leftrightarrow y \wedge \widehat{y}$. In this section, we discuss how we can use such counterexamples to transform $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ to a specification $\mu(\boldsymbol{X}, \boldsymbol{Y})$ such that $\mu \Leftrightarrow \varphi$ and y is conflict-free in μ. We call such a transformation *rectification* of φ w.r.t y, and the resulting formula μ is said to be *rectified* w.r.t. y.

Lemma 3. *Let π be a counterexample to conflict-freeness of y in $\varphi(\boldsymbol{X}, \boldsymbol{Y})$ and let ξ be a formula satisfying (a) $\mathsf{sup}(\xi) \subseteq \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\}$, (b) $\varphi \Rightarrow \xi$, and (c)*

$\xi\big|_\pi$ is unsatisfiable. Define $\tau \equiv \varphi \wedge \xi$ and let τ^+ denotes the positive form of τ w.r.t. y. Then the following hold: (i) $\tau \Leftrightarrow \varphi$, (ii) π is not a counterexample to conflict-freeness of y in τ, and (iii) every counterexample to conflict-freeness of y in τ is also a counterexample to conflict-freeness of y in φ.

Algorithm 1 RectifyOneOutput($\varphi(\boldsymbol{X}, \boldsymbol{Y})$, y)

```
1: μ := φ
2: repeat
3:     res := SatSolve(κ_{μ,y})  ▷ κ_{μ,y} is confict formula for y in μ
4:     if res is SAT then
5:         Let π : X ∪ Y \ {y} → {0,1} be a satisfying assignment of κ_{μ,y}
6:         ξ := PartialRectifier(μ, π)
7:         μ := μ ∧ ξ
8: until res is UNSAT  ▷ All counterexample to conflict-freeness of y in μ are removed
9: return μ
```

Proof. Since $\varphi \Rightarrow \xi$, it follows that $\tau \Leftrightarrow \varphi \wedge \xi \Leftrightarrow \varphi$. This proves claim (i) of Lemma 3. Next, note that since π is a counterexample to conflict-freeness of y in φ, we must have $\varphi^+\big|_\pi \Leftrightarrow (y \wedge \widehat{y})$. Since ξ does not have y in its support, it follows that $\tau^+ \Leftrightarrow \varphi^+ \wedge \xi$. Therefore, $\tau^+\big|_\pi \Leftrightarrow \varphi^+\big|_\pi \wedge \xi\big|_\pi \Leftrightarrow (y \wedge \widehat{y}) \wedge \xi\big|_\pi$. However, from the premise of Lemma 3, we know that $\xi\big|_\pi$ is unsatisfiable. Hence $\tau^+\big|_\pi$ is false. Specifically, $\tau^+\big|_\pi \not\Leftrightarrow (y \wedge \widehat{y})$, and hence π is not a counterexample to conflict-freeness of y in τ. This proves claim (ii) of Lemma 3. Finally, let $\pi' : \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\} \to \{0, 1\}$ be a counterexample to conflict-freeness of y in τ. By definition, $\tau^+\big|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$. However, $\tau^+\big|_{\pi'} \Leftrightarrow \varphi^+\big|_{\pi'} \wedge \xi\big|_{\pi'}$. Since all variables in support of ξ are assigned by π', we must have $\xi\big|_{\pi'}$ being equivalent to either 0 or 1. If $\xi\big|_{\pi'}$ is 0, then $\tau^+\big|_{\pi'}$ must also be 0, a contradiction of $\tau^+\big|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$. Therefore, we must have $\xi\big|_{\pi'}$ equivalent to 1, and hence $\varphi^+\big|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$ for $\tau^+\big|_{\pi'}$ to be equivalent to $(y \wedge \widehat{y})$. It follows that π' must be a counterexample to conflict-freeness of y in φ. This proves claim (iii) of Lemma 3. □

Henceforth, we call a formula ξ satisfying conditions (a), (b) and (c) of Lemma 3 a *partial rectifier* of φ w.r.t. y. Given π, it is easy to find a partial rectifier.

Lemma 4. *For all $v \in \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\}$, let $\ell_{v,\pi}$ denote v if $\pi[v] = 1$, and $\neg v$ if $\pi[v] = 0$. Let ξ_π be $\neg\big(\bigwedge_{v \in \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\}} l_{v,\pi}\big)$. Then ξ_π satisfies conditions (a), (b) and (c) of Lemma 3.*

The proof follows immediately from the observations: (i) π is the only satisfying assignment of $\neg\xi_\pi$, and (ii) $\varphi\big|_\pi \Leftrightarrow \big(\varphi^+[\widehat{y} \mapsto \neg y]\big)\big|_\pi \Leftrightarrow (y \wedge \widehat{y})[\widehat{y} \mapsto \neg y] \Leftrightarrow 0$. Consequently, $\neg\xi_\pi \Rightarrow \neg\varphi$. Although Lemma 4 gives a partial rectifier, it prevents only the assignment π from being a counterexample to conflict-freeness of y in

τ. Later we will see a partial rectifier that prevents many more assignments from being counterexamples. For the time being, however, we assume that we have access to a procedure PARTIALRECTIFIER that takes as inputs φ and π and outputs a partial rectifier that satisfies conditions (a), (b) and (c) of Lemma 3.

The above discussion suggests a simple algorithm, shown as Algorithm RECTIFYONEOUTPUT below, for rectifying a specification φ w.r.t. an output y.

The algorithm first initializes a temporary formula μ to φ. It then invokes a propositional satisfiability (SAT) solver to obtain a satisfying assignment π of the conflict formula $\kappa_{\mu,y}$ (defined in Sect. 5 just before Proposition 4). The assignment π serves as a counterexample to conflict-freeness of y in μ, and is used to obtain a partial rectifier ξ of μ w.r.t. y. The formula μ is then updated by conjoining it with ξ. Lemma 3 guarantees that this gives a specification semantically equivalent to φ, while removing π from the set of counterexamples to conflict-freeness of y in μ. By repeating the process with the updated formula μ, all counterexamples to conflict-freeness of y in μ are eventually removed.

Theorem 3. *Algorithm* RECTIFYONEOUTPUT *always terminates with a formula μ s.t. $\mu \Leftrightarrow \varphi$ and y is conflict-free in μ.*

Proof. The following inductive invariants hold at end of every iteration of the loop in lines 2–8, thanks to Lemma 3: (i) $\mu \Leftrightarrow \varphi$, (ii) the set of counterexamples to conflict-freeness of y in μ has strictly fewer elements than at the start of the iteration. Since the set of counterexamples is finite (at most $2^{|\boldsymbol{X}|+|\boldsymbol{Y}|-1}$ elements), eventually this set must become empty. By definition of the conflict formula, $\kappa_{\mu,y}$ must be unsatisfiable when this happens. Hence, the algorithm eventually exits the loop in lines 2–8 and terminates. Since there are no counterexamples to conflict-freeness of y in μ on termination, y is indeed conflict-free in μ. □

Rectification by Counterexample Generalization: The idea of counterexample generalization is best illustrated by an example. Consider the specification $\varphi(\boldsymbol{X}, y) \equiv \big((x_1 \wedge x_2) \vee ((x_2 \wedge x_3) \vee y)\big) \wedge \big(\neg y \vee (\neg x_3 \wedge x_4)\big)$, wherein y is in conflict. To see why this is so, consider φ^{+y} (henceforth called φ^{+}) represented as a NNF circuit in Fig. 3. Let π be an assignment that assigns 1 to x_1, x_3 and 0 to x_2, x_4. The values in red below the leaves in Fig. 3 represent this assignment. If we propagate these values upstream to the root of the circuit, we get the values/formulas shown in red adjacent to internal nodes, as shown in Fig. 3. This process is akin to *constant/symbol propagation* in symbolic simulation [30]. Note that the root of the circuit is assigned $y \wedge \widehat{y}$ by this process, indicating that $\varphi^{+}\big|_{\pi} \Leftrightarrow (y \wedge \widehat{y})$. Hence, y is in conflict in φ and π is a counterexample to conflict-freeness of y in φ.

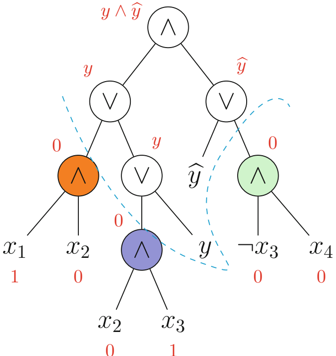


Fig. 3. Circuit representing φ^{+y}

Interestingly, the constant/symbol propagation discussed above can yield many more counterexamples beyond π. Specifically, let N denote the set of coloured nodes in the figure. Suppose we cut the circuit at the nodes in N, as shown by the dotted line in Fig. 3. Let the sub-circuit above the cut be denoted C_N. Notice that the leaf nodes of C_N are either nodes in N or leaf nodes of the original circuit corresponding to y or $\widehat{y}$. Now consider any assignment $\pi' : \{x_1, x_2, x_3, x_4\} \rightarrow \{0, 1\}$ s.t. when we propagate constants/symbols in the original circuit starting with π' at the leaves, we get the same values as in Fig. 3 at all nodes in N. This ensures that all leaves of C_N have the same constant/symbol as in Fig. 3. Therefore, further constant/symbol propagation must assign exactly the same constant/symbol/formula at every internal node of C_N as in Fig. 3. Specifically, the root node is assigned $y \wedge \widehat{y}$, implying that π' is a counterexample to conflict-freeness of y in φ.

Can we characterize all the counterexamples π' obtainable by the above method? It turns out we can do this. First, note from Fig. 3 that the sub-circuits rooted at the orange, purple and green nodes represent the Boolean formulas $x_1 \wedge x_2$, $x_2 \wedge x_3$ and $(\neg x_3 \wedge x_4)$ respectively. Hence, the set of all counterexamples π' obtained above are precisely the satisfying assignment of the formula $\beta \equiv \neg(x_1 \wedge x_2) \wedge \neg(x_2 \wedge x_3) \wedge \neg(\neg x_3 \wedge x_4)$. Notice that there are many assignments beyond π that satisfy β, e.g. $x_1x_2x_3x_4 = 0000$ or 0010 or 1000, and so on. Thus, we have truly *generalized* the counterexample π.

In general, given a specification $\varphi(\boldsymbol{X}, \boldsymbol{Y})$, an output variable y and a counterexample $\pi : \boldsymbol{X} \cup \boldsymbol{Y} \setminus \{y\} \rightarrow \{0, 1\}$ to conflict-freeness of y in φ, we first construct an NNF circuit representing φ^+. For every node n in the circuit, let φ_n^+ denote the sub-formula represented by the sub-circuit rooted at n. Next, we assign values given by π to the leaves of the circuit representing φ^+ and propagate these values to the root of the circuit. Let $v_{n,\pi}$ denote the constant/symbol/formula assigned to node n in the circuit by this process. In other words, $v_{n,\pi} \Leftrightarrow \varphi_n^+\big|_\pi$. We now choose a subset N of nodes n such that (i) $\mathsf{sup}(\varphi_n^+) \cap \{y, \widehat{y}\} = \emptyset$, (ii) $v_{n,\pi}$ is a constant, and (iii) every path from a non-y, non-$\widehat{y}$ leaf to the root passes through a node in N. Such a set N can always be found, for example, by choosing N to be the set of non-y, non-$\widehat{y}$ leaves. However, as Fig. 3 shows, N need not include only leaf nodes. Let $\beta_{\pi,N}$ denote the formula $\bigwedge_{n \in N} \left(\varphi_n^+ \Leftrightarrow v_{\pi,n}\right)$.

Lemma 5. *Every satisfying assignment of $\beta_{\pi,N}$ is a counterexample to conflict-freeness of y in φ. Moreover, $\neg\beta_{\pi,N}$ satisfies the three conditions required for a partial rectifier as specified in Lemma 3.*

Proof. Since every path from a non-y, non-$\widehat{y}$ leaf to the root passes through a node in N, we can use nodes in N and the leaves corresponding to y and $\widehat{y}$ to cut

the circuit (as shown in Fig. 3). Let C_N denote the sub-circuit above this cut. Let π' be a satisfying assignment (not necessarily same as π) of $\beta_{\pi,N}$. By definition of $\beta_{\pi,N}$, constant/symbol propagation starting from π' assigns the constant value $v_{\pi,n}$ to every node $n \in N$. It follows that for all leaf nodes l of the sub-circuit C_N, $v_{\pi',l} = v_{\pi,l}$. Hence, every internal node m of C_N must also have $v_{\pi',m} = v_{\pi,m}$. In particular the root node gets assigned the same value/symbol/formula that it had when we did constant/symbol propagation starting from π. In other words, $\varphi^+\big|_{\pi'} \Leftrightarrow \varphi^+\big|_{\pi}$. However, Since π is a counterexample to conflict-freeness of y in φ, we know $\varphi^+\big|_{\pi} \Leftrightarrow (y \wedge \widehat{y})$. Therefore, $\varphi^+\big|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$ and π' is a counterexample to conflict-freeness of y in φ^+.

To see $\neg\beta_{\pi,N}$ satisfies the conditions required of a partial rectifier in Lemma 3, note that $\mathsf{sup}(\varphi_n^+) \cap \{y, \widehat{y}\} = \emptyset$. Therefore, $\mathsf{sup}(\neg\beta_{\pi,N}) \cap \{y, \widehat{y}\}$ is also empty. Next, by definiton, if an assignment $\pi' \models \beta_{\pi,N}$, every node $n \in N$ in the circuit φ^+ gets assigned the constant value $v_{\pi,n}$. Using the same argument as in the first part of the proof, we can then show that $\varphi^+\big|_{\pi'} \Leftrightarrow (y \wedge \widehat{y})$. Hence $\varphi\big|_{\pi'} \Leftrightarrow \varphi^+[\widehat{y} \mapsto \neg y]\big|_{\pi'}$ $\Leftrightarrow y \wedge \widehat{y}[\widehat{y} \mapsto \neg y] \Leftrightarrow 0$. This shows that $\beta_{\pi,N} \Rightarrow \neg\varphi$. In other words, $\varphi \Rightarrow \neg\beta_{\pi,N}$. Finally, $\beta_{\pi,N}\big|_{\pi} \Leftrightarrow \bigwedge_{n \in N} \left(\varphi_n^+\big|_{\pi} \Leftrightarrow v_{\pi,n}\right)$. However, $v_{\pi,n} \Leftrightarrow \varphi_n^+\big|_{\pi}$ by definition. Hence $\beta_{\pi,N}\big|_{\pi} \Leftrightarrow 1$ and hence $\neg\beta_{\pi,N}\big|_{\pi}$ is unsatisfiable. □

The above lemma allows us to use $\neg\beta_{\pi,N}$ as a partial rectifier of φ w.r.t. y in Algorithm RectifyOneOutput. Significantly, this eliminates in one shot all counterexamples to conflict-freeness of y in φ that are satisfying assignments of $\beta_{\pi,N}$, thereby reducing the number of iterations of the loop in Algorithm RectifyOneOutput. As seen in the example above, $\beta_{\pi,N}$ can indeed have many more satisfying assignments beyond π. We use this technique to implement the subroutine PartialRectifier in Algorithm RectifyOneOutput. Specifically, we choose the set N such that the longest path of each node $n \in N$ from a leaf of C_μ is within an empirically determined threshold (20 in our experiments).

Generalizing Using Unsatisfiable Cores: It turns out that we can generalize counterexamples even beyond what was achieved above. To see a concrete example, consider the specification $\gamma(\boldsymbol{X}, y) \equiv \varphi(\boldsymbol{X}, y) \wedge \big(\neg y \vee (x_1 \wedge x_2)\big)$, where $\varphi(\boldsymbol{X}, y)$ is the same specification considered in Fig. 3. The NNF circuit representing γ^{+y} (or γ^+ for short) is the same as that shown in Fig. 3 with an additional $\wedge$-gate that feeds the root node, and that is fed by the $\widehat{y}$ leaf and output of the orange node. The same assignment π as considered earlier serves as a counterexample to conflict-freeness of y in γ, and the same set N can be chosen to obtain the same partial rectifier $\neg\beta$, where $\beta \equiv \neg(x_1 \wedge x_2) \wedge \neg(x_2 \wedge x_3) \wedge \neg(\neg x_3 \wedge x_4)$. Note, however, that in the circuit for γ^+, if the orange and purple nodes are assigned the value 0 by constant propagation starting from an assignment π', the root node must be assigned $y \wedge \widehat{y}$, *regardless of the value assigned to the green node*. Therefore, we could have used $\beta' \equiv \neg(x_1 \wedge x_2) \wedge \neg(x_2 \wedge x_3)$, which represents a larger set of counterexamples than β. Specifically, $x_1x_2x_3x_4 = 1001$ does not satisfy β but satisfies β'. It follows that rectification using $\neg\beta'$ eliminates more counterexamples in one go than rectification using $\neg\beta$.

In general, given φ, y, π and N as in our previous discussion, let s_n be a fresh variable for every node $n \in N$, and define the formula $\rho_{\pi,N} \equiv \varphi \wedge$

$\bigwedge_{n\in N}\big((s_n \Rightarrow (\varphi_n^+ \Leftrightarrow v_{\pi,n})) \wedge s_n\big)$. Since $\varphi \Rightarrow \neg\beta_{\pi,N}$ (see Lemma 5) and since $\beta_{\pi,N} \equiv \bigwedge_{n\in N}(\varphi_n^+ \Leftrightarrow v_{\pi,n})$, it follows that $\rho_{\pi,N}$ is unsatisfiable. Assuming φ is satisfiable (otherwise the synthesis problem is itself trivial), every unsatisfiable core of $\rho_{\pi,N}$ must set a subset of the s_n variables to 1. Let $U \subseteq N$ be the set of nodes n s.t. $s_n = 1$ in a minimal unsatisfiable core of ρ. Then $\rho_{\pi,U} \equiv \varphi \wedge \bigwedge_{n\in U}\big((s_n \Rightarrow (\varphi_n^+ \Leftrightarrow v_{\pi,n})) \wedge s_n\big)$ is unsatisfiable.

Lemma 6. *Lemma 5 holds with $\beta_{\pi,N}$ replaced by $\beta_{\pi,U}$. Moreover, $\beta_{\pi,N} \Rightarrow \beta_{\pi,U}$.*

Algorithm 2 FindSkBasisVec($\varphi(\boldsymbol{X}, \boldsymbol{Y})$)

1: $\alpha := \varphi$; $i := 1$ ▷ *Assume $|\boldsymbol{Y}| = n$*
2: **repeat**
3: $y_i :=$ Next output variable to find Skolem basis
4: $A_i := \alpha\big|_{y_i=1} \wedge \neg\alpha\big|_{y_i=0}$
5: $B_i := \alpha\big|_{y_i=1} \Leftrightarrow \alpha\big|_{y_i=0}$
6: **if** y_i is positive unate in α **then**
7: $\alpha := \alpha\big|_{y_i=1}$ ▷ *Existentially quantify y_i*
8: **else if** y_i is negative unate in α **then**
9: $\alpha := \alpha\big|_{y_i=0}$ ▷ *Existentially quantify y_i*
10: **else**
11: $\mu :=$ RectifyOneOutput(α, y_i) ▷ *y_i is conflict-free in μ*
12: $\alpha := \mu^{+y_i}\big|_{y_i=1,\widehat{y_i}=1}$ ▷ *Existentially quantify y_i*
13: $i := i + 1$
14: **until** all outputs processed
15: **return** $\langle(A_i, B_i)\rangle_{1\le i\le n}$

Overall Algorithm: We are now present Algorithm FindSkBasisVec. The algorithm initializes a running specification α to φ. It then repeatedly chooses the next output y_i for whose Skolem functions a Skolem basis needs to be computed. The choice of y_i can be as per a static order, or as determined on-the-fly heuristically. The algorithm then finds Skolem basis (A_i, B_i) using Theorem 1 by treating y_i as the sole output in the specification α. It next updates the running specification α by existentially quantifying y_i from α. In order to do this, it first checks if y_i is unate in α, and if so, substitutes an appropriate constant for y_i in α to quantify it out. Otherwise, the algorithm invokes Algorithm Rectify-OneOutput. Thanks to Theorem 3, we can effectively and efficiently quantify y_i from α by setting $y_i = 1$ and $\widehat{y_i} = 1$ in the positive form of the formula μ returned by RectifyOneOutput. Once all outputs are processed, the algorithm outputs the vector of (A_i, B_i) pairs computed as the Skolem basis vector.

Theorem 4. *Algorithm* FindSkBasisVec *terminates with a Skolem basis vector for the specification $\varphi(\boldsymbol{X}, \boldsymbol{Y})$.*

Proof. The proof of termination follows immediately from Theorem 3. The proof of correctness follows from Definition 1, Theorems 1, 3, and Lemmas 1, 2. □

Though we developed rectification as a technique for rendering a variable conflict free with the objective of generating Skolem basis vectors, it can be

independently used to compile a Boolean formula to a form that allows efficient quantifier elimination. However, a performance evaluation of rectification versus other quantification techniques in such applications is beyond the scope of this paper.

7 Implementation and Experiments

We implemented the above algorithms in C++ using the abc package [27] and ran our tool on a set of 602 Boolean functional synthesis benchmarks (also used in [12,14]). We used an Intel(R) Xeon(R) CPU E5-2660 v2@2.20GHz machine with 40 cores in single-threaded mode (multiple cores used only to run experiments in parallel). We set an overall timeout of 3600 seconds, within which the timeout for unate-check was 1000 seconds.

Detailed Analysis of Our Results. We did an ablation study to understand which part of our approach was most successful in compiling the benchmarks. Our results are summarized in Fig. 4. Here, "Total solves" denotes the number (out of 602) benchmarks for which Algorithm FindSkBasisVec completed within the timeout. "PAR2 score" is a widely used weighted performance score, computed as sum of time taken (in seconds) for each solved instances and double of timeouts (3600 s)s) for each unsolved instance. For benchmarks that were rectified, *for each application of rectification, we verified (using a SAT solver) that the rectified circuit was semantically equivalent* to the original. The time for this verification is included when computing PAR2 scores. In row 3, we note the "Average time" taken (including for verification), in seconds, over all solved instances. In rows 4, 5 and 6, we count, respectively, the number of solved benchmarks, where (i) all variables were unate (ii) some but not all were unate and (iii) no variables were unate (these add up to row 1). In row 7, we list the number of solved benchmarks for which there was at least one conflict, i.e., a call to the rectification algorithm was needed. Row 8 lists the solved benchmarks with at least one output that was not unate but no outputs having conflicts. The other rows are self-explanatory.

		DO	SO	CDO	CSO
1	Total Solves	287	298	299	308
2	PAR2 Scores	3839.56	3672.65	3696.90	3565.01
3	Average time	151.28	74.29	146.94	95.24
4	allUnates	98	98	98	98
5	someUnates	146	157	151	160
6	noUnates	43	43	50	50
7	fixedConflicts	71	19	73	21
8	noConflicts	118	181	128	189
9	fixedConflicts ∩ someUnates	68	16	69	17
10	noConflicts ∩ someUnates	78	141	82	143
11	fixedConflicts ∩ noUnates	3	3	4	4
12	noConflicts ∩ noUnates	40	40	46	46

Fig. 4. Table of results

Order Dependence. Since a Skolem basis vector depends on the ordering of outputs, we considered two order variants. In the first, we considered a heuristically

determined static order (denoted SO), taken as is from [14]. Then, we tried a heuristic dynamic order (denoted DO): after each output variable is processed, the next is obtained on-the-fly by applying the heuristic from [14].

Conflict Optimization in Calculating Skolem Basis Vector. We found several problem instances where the specification is not realizable, i.e., there exist input values for which no output values can make the specification true. For such instances, it is reasonable to restrict the computation of Skolem basis vector to a set F of Skolem functions, such that for any Skolem function $\psi \notin F$, there exists $\psi' \in F$ such that ψ and ψ' differ only on the space of input assignments for which no assignment of outputs would satisfy the specification. It turns out that this can be easily encoded in Algorithm 1 by modifying the conflict formula $\kappa_{\mu,y}$ to $\kappa_{\mu,y} \wedge \varphi(\boldsymbol{X}, \boldsymbol{Y'})$, where $\boldsymbol{Y'}$ is a fresh set of variables. Doing this, along with the static/dynamic ordering gives us the "CSO" and "CDO" columns in Fig. 4.

Observations. With either SO or DO, without conflict optimization, we are able to *compute Skolem basis vectors for 299 of 602 benchmarks* (286 were solved by both, 1 by only DO and 12 by only SO). Interestingly, the static order (SO) had fewer conflicts compared to the dynamic order (DO), when we had to rectify more often. Further, in the presence of conflict optimization, we are able to compute Skolem basis vectors for 309 out of 602 benchmarks. Note is that even though the PAR2 score is large, the average time taken is less than 2.5 min, including time taken for verification. In other words, *when we are able to compute Skolem basis vectors, we are able to do so in remarkably short duration.*

Comparison with Other Tools/Approaches. There are no existing tools that synthesize a represention of the space of all Skolem function vectors. Knowledge compilation tools e.g., C2Syn [13], NNF2SDD [25,31] come closest as they try to obtain a single circuit that is semantically equivalent to the original and is in a normal form: the SynNNF form for C2Syn and the SDD form for NNF2SDD. Skolem functions hence could be potential alternative approaches. In practice, C2Syn does refinement (see [13]) operations for performance boosting, thereby restricting the space of Skolem function vectors. Even with this optimization for C2Syn it can compile only 218 (out of 602) benchmarks, while NNF2SDD compiles only 142 to SDD on the same computing platform.

An apples-to-apples performance comparison of Boolean functional synthesis tools (that synthesize a single Skolem function vector) with our tool (that computes Skolem basis vectors for all Skolem function vectors) is not possible, since two different problems are being solved. Nevertheless, to understand the performance penalty incurred in computing a representation of all Skolem function vectors, we observe from [12] that with a 7200 s s timeout and using a more powerful cluster, Manthan [12] (resp. BFSS [14]) could synthesize a single Skolem function vector for ∼356 (resp. 247) out of the same 602 benchmarks. In comparison, with 3600 s s timeout, we are able to compute Skolem basis vector for

$\sim$ 300 benchmarks. In [17], an improved and highly engineered tool Manthan2 was developed, which could synthesize a single Skolem function vector for 502 benchmarks within 7200 s.s. Interestingly, *we are able to compute Skolem basis vectors for 22 benchmarks (out of which 13 have non-unate variables), for which even Manthan2 [17] fails to synthesize a single Skolem function vector.*

8 Conclusion

In this work, we have introduced a representation for the space of Skolem functions, using the notion of Skolem basis vector. Our representation itself is criteria-agnostic, but allows the use of other existing techniques to optimize Skolem functions wrt different criteria. We develop a compilation algorithm that uses a combination unate and conflict-detection along with generalized counter-example guided approach to synthesize the Skolem basis vector. Our next step would be to identify specific problem contexts and optimization criteria and integrate our approach with the state-of-the-art logic synthesis tools to synthesize specific Skolem functions satisfying the given criteria.

References

1. Jacobs, S., et al.: The second reactive synthesis competition (SYNTCOMP 2015). In: Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015, pp. 27–57 (2015)
2. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 1362–1369 (2017)
3. Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. Int. J. Softw. Tools Technol. Transf. **7**(2), 118–128 (2005). https://doi.org/10.1007/s10009-004-0179-0
4. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. STTT **15**(5–6), 497–518 (2013)
5. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013)
6. Balabanov, V., Jiang, J.-H.R.: Resolution proofs and skolem functions in QBF evaluation and applications. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 149–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_12
7. Balabanov, V., Jiang, J.-H.R.: Unified QBF certification and its applications. Form. Methods Syst. Des. **41**(1), 45–65 (2012)
8. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 430–435. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_33
9. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. SIGPLAN Not. **45**(6), 316–329 (2010)
10. Jo, S., Matsumoto, T., Fujita, M.: Sat-based automatic rectification and debugging of combinational circuits with lut insertions. In: Proceedings of the 2012 IEEE 21st Asian Test Symposium, ATS 2012, pp. 19–24. IEEE Computer Society (2012)

11. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 375–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_23
12. Golia, P., Roy, S., Meel, K.S.: Manthan: a data-driven approach for Boolean function synthesis. Comput. Aided Verificat. **12225**, 611–633 (2020)
13. Akshay, S., Arora, J., Chakraborty, S., Krishna, S., Raghunathan, D., Shah, S.: Knowledge compilation for boolean functional synthesis. In: Proceedings of of Formal Methods in Computer Aided Design (FMCAD), 2019
14. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: Boolean functional synthesis: hardness and practical algorithms. Formal Methods Syst. Des. **57**(1), 53–86 (2021)
15. Akshay, S., Chakraborty, S., John, A.K., Shah, S.: Towards parallel boolean functional synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 337–353. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_19
16. John, A., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: FMCAD, pp. 73–80 (2015)
17. Golia, P., Slivovsky, F., Roy, S., Meel, K.S.: Engineering an efficient Boolean functional synthesis engine. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, 1–4 November 2021, pp. 1–9. IEEE (2021)
18. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_22
19. Tabajara, L.M., Vardi, M.Y.: Factored Boolean functional synthesis. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 124–131 (2017)
20. Shah, P., Bansal, A., Akshay, S., Chakraborty, S.: A normal form characterization for efficient boolean skolem function synthesis. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, 29 June–2 July 2021, pp. 1–13. IEEE (2021)
21. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Int. Res. **17**(1), 229–264 (2002)
22. Darwiche, A.: Decomposable negation normal form. J. ACM **48**(4), 608–647 (2001)
23. Darwiche, A.: Tractable Boolean and arithmetic circuits. In: Hitzler, P., Sarker, M.K. (eds.) Neuro-Symbolic Artificial Intelligence: The State of the Art, vol. 342 of Frontiers in Artificial Intelligence and Applications, pp. 146–172. IOS Press (2021)
24. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. J. Appl. Non-Classical Logics **11**(1–2), 11–34 (2001)
25. Shi, W., Shih, A., Darwiche, A., Choi, A.: On tractable representations of binary neural networks. In: Calvanese, D., Erdem, E., Thielscher, M. (eds.) Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, 12–18 September 2020, pp. 882–892 (2020)
26. Somenzi, F.: Binary decision diagrams. In: Calculational System Design, vol. 173 of NATO Science Series F, pp. 303–366. IOS Press (1999)
27. Abc: A system for sequential synthesis and verification
28. Mishchenko, A., Brayton, R.K., Jiang, J.H.R., Jang, S.: Scalable don't-care-based logic optimization and resynthesis. ACM Trans. Reconfigurable Technol. Syst. **4**(4), 34:1-34:23 (2011)

29. De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill Higher Education, Boston (1994)
30. Bryant, R.E.: Symbolic simulation-techniques and applications. In: 27th ACM/IEEE Design Automation Conference, pp. 517–521 (1990)
31. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: Walsh, T. (ed.) IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, 16–22 July 2011, pp. 819–826. IJCAI/AAAI (2011)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Guessing Winning Policies in LTL Synthesis by Semantic Learning

Jan Křetínský[1,2](✉), Tobias Meggendorfer[1,3], Maximilian Prokop[1,2], and Sabine Rieder[1]

[1] Technical University of Munich, Munich, Germany
jan.kretinsky@tum.de
[2] Masaryk University, Brno, Czech Republic
[3] Institute of Science and Technology, Klosterneuburg, Austria
tobias.meggendorfer@cit.tum.de

Abstract. We provide a learning-based technique for guessing a winning strategy in a parity game originating from an LTL synthesis problem. A cheaply obtained guess can be useful in several applications. Not only can the guessed strategy be applied as best-effort in cases where the game's huge size prohibits rigorous approaches, but it can also increase the scalability of rigorous LTL synthesis in several ways. Firstly, checking whether a guessed strategy is winning is easier than constructing one. Secondly, even if the guess is wrong in some places, it can be fixed by strategy iteration faster than constructing one from scratch. Thirdly, the guess can be used in on-the-fly approaches to prioritize exploration in the most fruitful directions.

In contrast to previous works, we (i) reflect the highly structured logical information in game's states, the so-called semantic labelling, coming from the recent LTL-to-automata translations, and (ii) learn to reflect it properly by learning from previously solved games, bringing the solving process closer to human-like reasoning.

1 Introduction

LTL Synthesis. [38] is a framework for automatic construction of reactive systems specified by formulae of linear temporal logic (LTL) [37]. Since LTL is a prominent logic in the area of safety-critical and provably reliable dynamic systems, LTL synthesis is a very tempting option to construct such systems since it avoids error-prone manual implementation; instead it is replaced with the need for a complete specification of the system (which is not trivial either, but in some cases easier). However, there is also an important computational caveat: the problem of LTL synthesis is 2-EXPTIME complete. Despite the infeasibility in the worst-case, many heuristics have been designed that can cope with practical problems, as documented by the yearly progress in the synthesis competition

This research was funded in part by the German Research Foundation (DFG) project 427755713 *Group-By Objectives in Probabilistic Verification (GOPro).*

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 390–414, 2023.
https://doi.org/10.1007/978-3-031-37706-8_20

SYNTCOMP [18], which has an LTL track for a number of years. Yet, many reasonable instances even in the benchmark set of SYNTCOMP still remain practically unsolvable. In this paper, we aim at *guessing a solution* through a machine-learning model, even for hard cases, thus possibly providing an applicable answer, in a sense, without reading the input formula. We achieve that by learning from other games and by reflecting *semantic* information, bringing the process closer to human reasoning.

The classic technique for solving LTL synthesis is to

1. turn the LTL formula into a deterministic parity automaton (DPA),
2. turn the DPA (and the partitioning of atomic propositions into system variables and environment variables) into a parity game (PG) between the system and the environment players, and
3. solve the PG; any winning strategy of the system player then directly induces a system policy (also representable as a circuit) satisfying the LTL formula.

Due to the worst-case doubly-exponential blowup in the first step and the practically bad performance of (Safra's [39] and others' [36,40]) determinization procedures, this option was rarely used practically until direct, more practical translations were given [8,12]. The significantly smaller automata [20] have made this approach feasible and, in fact, winning in SYNTCOMP since then. The approach is implemented in the tool `Strix` [33], which additionally constructs the DPA/PG *only partially*, on-the-fly until it finds a winning strategy for one of the players. This helps to overcome some more cases where the DPA is still very large; yet, more complex specifications often remain out of reach.

Semantic Labelling. The key difficulty in the on-the-fly exploration is a good heuristic that prioritizes exploration in promising directions, so that a solution can be obtained quickly, without constructing "irrelevant" parts of the game.

In a concrete state of a PG, is it better to go left or right? While this question obviously does not have a simple answer in general, we take a step back and instead of a PG we solve the LTL synthesis problem. For instance, consider a state of a PG corresponding to satisfying $\mathbf{G}a$, i.e. "always a holds". Then, the letter $\{a\}$ is clearly a better choice (for the system) than $\emptyset$. The former leads to the obligation of satisfying again $\mathbf{G}a$; the latter to the obligation `ff` (falsifying the formula). Taking the former edge does not guarantee winning, but the chances are certainly higher than giving up directly. In order to estimate the chances of winning with some obligation, we can evaluate it by randomly assigning truth values to temporal subformulae; intuitively, $\mathbf{G}a$ can be true or false, so its "trueness" is 0.5, `ff` has trueness 0. *Trueness* is examined in [22] and utilized in newer versions of `Strix` [31] as guidance.

Does every state correspond to a goal in LTL? And if so, can we determine which continuation brings us closer to satisfying it? Recall that the classic translations of LTL to non-deterministic Büchi automata (NBA), stemming from [43], label the states of the NBA with a conjunction of LTL formulae, which are the current goals in this state. For deterministic automata, the situation is inevitably more complex. While the determinization procedures obfuscated any possible such semantic labelling, the more recent approach re-established it, e.g., [8] with

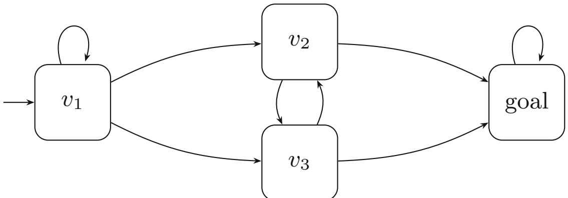


Fig. 1. Simple game where it is not clear which edges are "winning".

[26], or [42] with [9]. Beside the overall goal, it is necessary to also monitor the *progress of subgoals*. For example, consider $\mathbf{GF}(a \wedge \mathbf{X}b)$ "infinitely often a is followed by b". No matter what happens, the goal remains the same. However, whenever a, we are progressing with the subgoal of seeing the $a - b$ sequence once, yielding a subgoal b, which is regarded as promising.

Our Aim. In this paper, we aim at *better guessing of winning decisions* than in [22,31]. While the previous work only reflected trueness of the main goal, which is just the percentage of truth assignments leading to satisfaction of a Boolean formula, our approach reflects also (i) the temporal structure of the formulae, (ii) the monitored subgoals, and (iii) learns from previously solved games. On the technical level, we design over 200 *structural features* instead of just trueness, learn an *SVM* classifier comparing which edge is most promising, and use *data from previously solved games*, i.e. which edges are "winning". As it turns out, defining this notion already is surprisingly tricky: We cannot simply use the output of classical strategy improvement algorithms, as there may be multiple, incompatible solutions. Indeed, already for reachability, there are no maximal permissive strategies [3], see Fig. 1. Here the edge (v_2, v_3) is winning iff (v_3, v_2) is not used, and vice versa; using both makes them losing. Nevertheless, they are "better" than, e.g., the self-loop on v_1, which is always losing. Thus, we want to value both edges between v_2 and v_3 equally, and higher than the self loop on v_1.

Our Contribution can be summarized as follows:

- We learn a model predicting which edge has better chances to be winning. To this end, we define features on the semantic labelling in Sect. 5.1, introduce a way to measure the degree of "winning" of an edge in Sect. 4, and apply learning of support vector machines using our novel ground truth in Sect. 5.2.
- We evaluate "how winning" the suggested strategy is, i.e. how many wrong choices it made, on several inputs in Sect. 6.2. Surprisingly, this value often is 0, i.e. our strategy is often winning even for complex formulae, and even without reading them (meaning that our strategy is of constant size, *independent of the formula*, as opposed to a decision table in the concrete game; it can be run on the fly with no pre-computation, and decisions depend only on the labelling of the current state).
- Besides, while `Strix`'s architecture and interface ask for a significantly different type of advice (not just for the better of two edges), we show

Strix already profits from our advice and—modulo our unoptimized advice implementation—speeds up significantly, as we see in Sect. 6.3.

Usage of our Results:

- We provide an immediate solution (without even reading the input formula), which is often winning; moreover, it is applicable even to games too huge to be analyzed in any way. Besides, it is even of a constant size, i.e. independent of the size of the state space.
- Our approach opens the way to (i) a solver based on the semantic labelling, for instance, based on strategy iteration only quickly fine-tuning the already almost correct guess, and (ii) an on-the-fly-exploration advisor to Strix, with the proven potential to be the most efficient among the current techniques.

Related Work. To the best of our knowledge, there is only one other approach to using machine learning in LTL-synthesis. Here, the authors train a very powerful model (a hierarchical transformer) in order to directly predict a controller or counter example solely off the LTL specification [41]. Further, if their prediction is refuted by a classical model checking algorithm, they train a separated hierarchical transformer to repair it [5] until it is correct. While this turns out to be an overall competitive approach that also manages to solve some instances where classical synthesis tools as Strix [33] fail, this does not yield a complete procedure, as the repair loop is not guaranteed to ever terminate. In this work, we aim to improve existing, complete procedures such as implemented in Strix by means of machine learning based heuristics.

2 Preliminaries

We introduce notation and provide an overview of necessary background knowledge. Due to space constraints, we only briefly comment on several topics and refer the interested reader to the respective literature.

We use $\mathbb{N}$ to denote the set of non-negative integers. The constants tt and ff denote *true* and *false*, respectively.

2.1 Synthesis & Games

The synthesis problem in its general form asks whether a system can be controlled such that it satisfies a given specification under any (possible) environment. Moreover, one often is interested in obtaining a witness to this query, i.e. some *controller* or *strategy* which specifies the system's actions.

Parity Games are a standard formalism used in synthesis. A *parity game* is a tuple $\mathcal{G} = ((V, E), v_0, P, \mathfrak{p})$, where (V, E) is a finite digraph, $v_0 \in V$ a *starting vertex*, $P : V \to \{\mathcal{S}, \mathcal{E}\}$ a *player mapping*, and $\mathfrak{p} : V \to \mathbb{N}$ a *priority assignment*. Each vertex belongs to one of the two players $\mathcal{S}$ (called *system*) and $\mathcal{E}$ (called *environment*). In other words, the set of vertices is partitioned into player $\mathcal{S}$'s vertices $V_\mathcal{S}$ and player $\mathcal{E}$'s vertices $V_\mathcal{E}$. See Fig. 2 for an example.

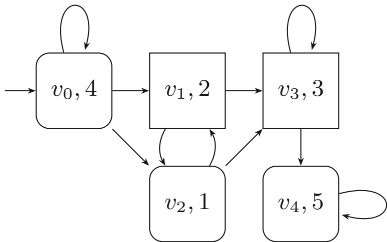


Fig. 2. An example parity game, taken from [22]. Rounded rectangles belong to the system $\mathcal{S}$ and normal rectangles to the environment $\mathcal{E}$. The vertices are additionally labelled with their priorities.

Remark 1. In our implementation priorities are assigned to edges instead of vertices, as this allows for a much more concise representation and suits most translations better. However, for ease of presentation, we consider *state-based acceptance* instead of *transition-based*.

Playing. To play the game, a token is placed in the initial vertex v_0. Then, the player owning the token's current vertex moves the token along an outgoing edge of the current vertex. This is repeated infinitely, giving rise to an infinite sequence of vertices containing the token $\rho = v_0 v_1 v_2 \cdots \in V^\omega$, called a *play*. We write ρ_i to refer to the i-th vertex in a play. A play ρ is *winning* (for the system player) if the smallest priority occurring infinitely often is odd. (Using "maximal" instead of "minimal" or "even" instead of "odd" does not fundamentally change the problem at hand.) Formally, we define $\inf(\rho) = \{v \in V \mid \forall j.\ \exists k \geq j.\ \rho_j = v\}$ as the set of infinitely occurring states. Since the game graph is finite, this set always is non-empty. The smallest priority occurring infinitely often is given as $\mathfrak{p}(\rho) = \min\{\mathfrak{p}(v) \mid v \in \inf(\rho)\}$ and system wins the play ρ if $\mathfrak{p}(\rho)$ is odd.

Strategies. A strategy of player p is a mapping $\sigma_p : V_p \to E$ assigning to each of p's vertices an appropriate edge along which the token will be moved, i.e. $(v, \sigma_p(v)) \in E$ for all $v \in V_p$.[1] Once both players fix a strategy, the game is fully determined and a unique run is induced. We call a strategy of system $\sigma_\mathcal{S}$ *winning* if for *all* strategies of the environment $\sigma_\mathcal{E}$ the induced play is winning, i.e. system wins no matter what the environment does.

For example, consider again the game depicted in Fig. 2. Fixing the strategies $\sigma_\mathcal{S} = \{v_0 \mapsto (v_0, v_2), v_2 \mapsto (v_2, v_3), v_4 \mapsto (v_4, v_4)\}$ and $\sigma_\mathcal{E} = \{v_1 \mapsto (v_1, v_2), v_3 \mapsto (v_3, v_3)\}$ induces the play $v_0 v_2 v_3 v_3 \cdots$. The set of infinitely often seen priorities equals $\{3\}$, hence the system player wins with these strategies. Moreover, the strategy σ_0 is winning, since the play always ends up in either v_3 or v_4.

Synthesis. With these notions, we can compactly define the synthesis question: *Given a parity game $\mathcal{G}$, does there exist a winning strategy for the system player?* In the example above, σ_0 is a witness to this question.

[1] Strategies may be more complex, e.g., by using memory. However, "positional" strategies are sufficient for parity games, thus we omit the general definition.

This problem is still intensely studied due to its broad applications. It also is one of the few problems which canonically lie in **NP** $\cap$ **coNP** (even in **UP** $\cap$ **coUP** [19]), with recent breakthroughs achieving quasi-polynomial algorithms [4,14,28].

Extensive-Form Game. A common notion in game theory is the *extensive-form* game. Intuitively, this means completely "unrolling" the game into an explicit representation. See e.g. [34, Chp. 5–7] for details. In our case, we consider the *game tree*, where each node corresponds to a simple path in the game $\mathcal{G}$. Suppose we are in state $s = (v_1, \ldots, v_i)$ of the game tree. Then, the successors of s are determined by all successors of v_i in the game, i.e. $\{u \mid (v_i, u) \in E\}$ as follows. Suppose such a successor u already occurs along s, i.e. a loop is closed, we check if the corresponding play is winning or losing. In that case, the choice leads to a corresponding winning or losing leaf of the tree, respectively. Otherwise, i.e. when no loop is closed by the choice, it leads to $s \circ u$. Essentially, this game tree represents all potential simple paths (and thus, intuitively, all potential positional strategies) that can arise in the game, and each edge corresponds to a particular move of a player (also called *ply* in game theory). In particular, it is finite, however of potentially exponential size. Note that we can restrict to simple paths only because positional strategies are sufficient.

Minimax Game Solving. A fundamental way to solve games is the *minimax decision* rule, which intuitively corresponds to exhaustively exploring the extensive-form game (also discussed in [34]). Suppose we assign a value of 0 to "losing" leaves of the game tree and a value of 1 to the "winning" leaves. Then, we can "back-propagate" values by setting $V(s)$ the maximum of all successors of s if it currently is the turn of the system player and the minimum if instead it is environment's turn (which wants the system to lose). The game is winning if the value in the initial state of the game tree is 1. This approach is also called *backward induction* or *retrograde analysis*: starting from the winning / losing positions of the game, we consider all moves which could lead to such situations.

Strategy Improvement (or *strategy iteration*, abbreviated by *SI*) is the most prominent practical way of solving parity games, i.e. answering the synthesis question. It received significant attention due to recent practical advances [13,15,17,32] and modern tool developments [6,33]. We explain the approach briefly, since its details are not important for this work. Intuitively, SI starts from arbitrary initial strategies for each player, and then performs the following steps in a loop. First, we check whether either strategy is winning. If yes, the algorithm exits, returning this strategy. Otherwise, one of the strategies is improved by changing its choices in some vertices. If an improvement is not possible, there exists no winning strategy for the respective player. Otherwise, the process is repeated with the new strategy.

This algorithm converges to the correct result in finite time for any initial strategy. However, if this initial strategy is chosen "close" to a winning strategy, then SI intuitively needs to perform fewer steps to converge to an optimal

one. Thus, a heuristic which often comes up with a "good" initial strategy may improve the runtime significantly over arbitrary or random initialization.

2.2 Linear Temporal Logic and Reactive Synthesis

Linear Temporal Logic (LTL) [37] is a standard logic used to specify desired behaviour of a system. The syntax usually is given by

$$\phi ::= \mathtt{ff} \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \phi \,\mathbf{U}\, \phi,$$

where $a \in \mathsf{AP}$ is an *atomic proposition*, inducing the *alphabet* $\Sigma = 2^{\mathsf{AP}}$. These formulae are interpreted over infinite sequences $w \in \Sigma^\omega$ called ω-words. A word $w = w_0 w_1 \cdots \in \Sigma^\omega$ satisfies the *next* operator $\mathbf{X}\phi$ iff ϕ is satisfied in the next step. Similarly, the *until* operator $\phi\mathbf{U}\psi$ is satisfied iff ϕ holds until ψ is eventually satisfied. Usual abbreviations are defined as *finally* $\mathbf{F}\phi \equiv \mathtt{tt}\,\mathbf{U}\,\phi$ and *globally* $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$, which require that ϕ holds at least once or always, respectively. Moreover, the construction underlying our work also considers *strong release* $\phi\,\mathbf{M}\,\psi \equiv \psi\,\mathbf{U}\,(\psi \wedge \phi)$, *(weak) release* $\phi\,\mathbf{R}\,\psi \equiv \mathbf{G}\psi \vee (\phi\,\mathbf{M}\,\psi)$, and *weak until* $\phi\,\mathbf{W}\,\psi \equiv \mathbf{G}\phi \vee (\phi\,\mathbf{U}\,\psi)$. Considering these additional operators allows formulas to be represented in *negation normal form*, i.e. the negation $\neg$ only appears in front of atomic propositions. In the interest of space, we refer to [12] for precise definition on the semantics and discussion of these subtleties. Understanding these issues is however not required for this work.

LTL Synthesis is an instance of the general synthesis problem, where the specification to be satisfied is given in form of an LTL formula [38]. Due to recent advances [11,12,16,20,21,25], the *automata-based approach* [43] to LTL synthesis received significant attention. In particular, the tool `Strix` [33], built on top of `Owl` [24], which in turn implements these ideas, won several iterations of the synthesis competition SYNTCOMP [18]. Essentially, the given LTL formula is translated into an ω-automaton, which in turn is transformed into a parity game. Solving the resulting game yields a solution to the original synthesis question.

This game is obtained by "splitting" the automaton, as follows. The set of atomic propositions is split into system- and environment-controlled propositions, i.e. $\mathsf{AP} = \mathsf{AP}_\mathcal{S} \cup \mathsf{AP}_\mathcal{E}$, and the players' actions correspond to choosing which of their propositions to enable. Once both players chose their propositions' values, the automaton moves to the next vertex according to the players' choices. Concretely, for an automaton state p, the environment can choose to move into (p, v) where $v \subseteq 2^{\mathsf{AP}_\mathcal{E}}$, and from there, system can move to any automaton state $q = \delta(p, v' \cup v)$ where $v' \subseteq 2^{\mathsf{AP}_\mathcal{S}}$ and δ is the transition function of the automaton. In particular, this means that the obtained game is *alternating*, i.e. system and environment take turns in alternation. Moreover, by convention the environment moves first. See e.g. [33] for more details on this approach.

Semantic Translations from LTL to automata are the key ingredient to our approach. On top of providing a parity game, they also give a *semantic labelling*,

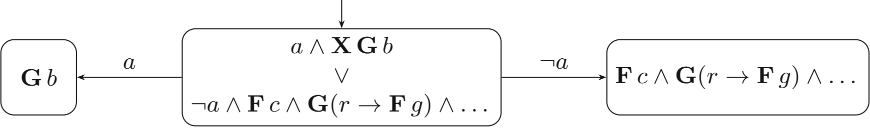


Fig. 3. Motivational example to provide guidance through semantic labelling.

i.e. interpretable meaning, to the game's vertices. In particular, the approach introduced in [8] (see also [10–12]) and implemented in `Owl` [25] intuitively yields for each vertex a list of LTL formulae, which roughly correspond to (sub-)goals which still have to be fulfilled, possibly repetitively.

2.3 Our Goal

In this work, we want to demonstrate that this semantic labelling can be efficiently exploited for reactive synthesis. For a motivational example to consider semantic labelling, we display a (vastly simplified) labelled game in Fig. 3. We are offered with the choice of choosing a or $\neg a$. While it is not completely clear that choosing a is indeed better, it certainly seems to be more promising, as the subsequent labelling seems much "easier" to handle. Thus, faced with a choice, we likely would first try to win with a. Observe that without the semantic labelling, our best option in this situation would be a random guess. In [22], the authors used a simple, manually designed mechanism trying to capture this notion, called *trueness*. Motivated by the (surprisingly good) results of this approach, we want to tackle this problem by more sophisticated means. Concretely, we want to make meaningful decisions based on the labelling. However, while the theory underpinning semantic translations is quite clean and pleasant [12], the actual labellings appearing in practice are quite complex. To further complicate things, the highly optimized implementation thereof [25] employs several subtle optimizations and special cases. We provide an example to showcase the complexity of this labelling in practice later in Sect. 5, kept brief in the interest of space, and a small real-world example in [23, Appendix A.1]. Since we have a simple intuition which however seems difficult to formalize, we opt to tackle this problem through means of machine learning.

3 Previous Approaches and Their Limitations

In this section, we briefly summarize the ideas of [22] and the inherent problems associated with them. The primary motivation of [22] is to exploit the semantic labelling provided by [25], which gives us an indication of the long term goals in the game. As an analogy, consider the game of chess. Here, the "semantic labelling" is given by the board state, i.e. the position of each piece. This labelling provides us with a reasonable indication of (i) our current situation and (ii) which moves might be better than others. In particular, understanding

and evaluating the semantics of the game is what allows humans to have a good intuition about the quality of moves, without thinking through the intractably large game tree. Likewise, this understanding is what enabled algorithms to perform beyond human capabilities.

3.1 Parity Game Solving by Trueness

A central notion of [22] is *trueness*, an approximation of how close a formula is to being satisfied, i.e. `tt`. The intuition is that the semantic labelling of states effectively describes "goals" of the system player. If the formula is `tt`, the system has satisfied all goals and consequently won the game. Likewise, increasing the trueness is indicative for a good move. Remaining with the analogy of chess, trueness somewhat corresponds to counting the number of pieces on the board (or rather the difference between our and the opponent's pieces): If no enemy pieces remain, we certainly have won, and a change of this difference, i.e. capturing an enemy piece or avoiding capture of own pieces, is a good indicator for the quality of a move. In particular, this prohibits us from taking moves which immediately lead to a piece being taken.

In [22], the authors propose two ideas. First, they suggest to use a trueness-maximizing strategy as initial one for strategy iteration, i.e. in each state select the edge which maximizes (or minimizes, in the case of $\mathcal{E}$) the obtained trueness. Second, they use *Q-Learning*, a popular reinforcement learning approach, as a solver for parity games, i.e. as competitor to strategy iteration, using three different reward signals. There, each edge is given a reward, which is mostly based on (the change of) trueness, and these values then are back-propagated until choosing optimal rewards in each step yields a winning strategy.

While they also show Q-Learning to be an interesting avenue, we primarily focus on the "initializing strategy iteration" approach, since our goal is to augment exiting strategy iteration solvers. Moreover, the experimental evaluation of [22] suggests that Q-Learning scales poorly to large real-world formulae.

3.2 Problems

We now outline two key issues of this approach.

Myopic Trueness The primary heuristic in [22] is trueness. While this approach already performs surprisingly well, especially for so called *safety* and *co-safety* formulae, it fails to take into account temporal dependencies; trueness is myopic. Again, considering chess, while counting the change of pieces does help us avoid "obviously stupid" moves, it does not stop us from moving pieces into positions where they are effectively guaranteed to be taken eventually and does not allow for sacrificing a piece in exchange for a long-term advantage.

Manual Design Their reward functions were defined manually, in contrast to being obtained from a learning process. While the intuition behind these definitions is reasonable, obtaining a guidance heuristic as a result of an optimization process is a much more principled approach.

We proceed to outline how we tackle these issues by a more sophisticated approach.

4 A New Hope

We want to improve reactive synthesis by applying machine learning. As already motivated by [22], we want to approach this problem by identifying "promising" edges, choosing those as initial strategy for SI. Naturally, as a first step, we need training data for our learning approach. In particular, we need to identify which actually are the actual good choices in games, i.e. the *ground truth*. As it turns out, this is more complicated than one might expect.

4.1 Obtaining Training Data with SI

As SI allows us to solve a game and determine winning edges, one might try to employ SI for obtaining a ground truth (as we did initially). However, SI actually provides us with potentially misleading or even conflicting information! As we already hinted in the introduction through Fig. 1, SI cannot give us a canonical ground truth. In the example, one edge is winning iff the other is not used, and vice versa. Thus, SI will yield a strategy which does not take both edges and we would consider one of them losing. Moreover, note that there is no fundamental reason to prefer one edge over the other, so SI might in one run classify the edge from v_2 to v_3 as good and in a second run (or on a similar game) do the opposite or even consider neither winning. The underlying problem is that parity games do not allow for a unique *maximally permissive* strategy (see e.g. [3]), thus we cannot derive the "suitability" of an edge from a single solution strategy.

4.2 Solving the Game Tree

Instead of using a particular strategy obtained from SI, we therefore propose to identify "all" solutions, i.e. all edges which are part of a winning strategy. More formally, for each vertex v we want to determine the value of each outgoing edge in the corresponding game tree rooted at v. To prefer "shorter" solutions over larger, we add a beta-decay to the value. Concretely, suppose we consider the game tree state $s = (v_1, \ldots, v_i)$ which ends in a system state v_i. Then, the value of s is defined by $\mathrm{val}(s) = \beta \cdot \max_{s' \in \mathrm{successors}(s)} \mathrm{val}(s')$ for a fixed $0 < \beta < 1$.

As we already mentioned, evaluating this tree is intractably large, namely exponential in the size of the game, which itself is already doubly-exponential in the input formula [27,38]. Thus, we employ a classical technique of game theory.

4.3 Monte Carlo Tree Search (MCTS)

Intuitively, we explicitly unfold the tree up to a specified depth, e.g. 7 plies, and then assign the results of (guided) random sampling to the occurring leaves, approximating the (beta-decayed) value of the game in these vertices.

We describe our method to approximate the value of a node $s = (v_1, \ldots, v_i)$ in the game tree. In essence, starting from v_i, we randomly select successors, with the following restrictions for each player. The environment plays *optimally*, i.e. if a state is winning for the environment (which we can determine beforehand through classical approaches) we immediately stop sampling and return a value of 0. Otherwise, the environment heuristically tries to delay the play as long as possible (decreasing the value the system player obtains due to beta-decay). In contrast, the system player checks in a one-step lookahead if a choice is trivially winning, i.e. leading to a state labelled `tt`, always choosing such an edge if one exists. Otherwise, the system randomly chooses among edges which are not trivially losing, i.e. lead to a `ff` state. If either player closes a loop, i.e. selects a successor which already occurs along the path, we determine the value by checking if the loop is winning or losing. A loss yields a value of 0, while a win yields β^{length}. In summary, we approximate the probability of winning by playing randomly (avoiding obvious mistakes) against an optimal opponent, under-approximating the true value. We deliberately opt for this random-choice approach to prefer regions where there is less potential for error.

4.4 Optimizations

While MCTS makes approximation of the game tree value feasible, we added several further technical improvements to arrive at a practically viable method.

SCC Decomposition. We exploit the structure of the game by decomposing it into its strongly connected components (SCCs) and put them in reverse topological order. Computing (or approximating) the value in that order allows for caching: Once a run in the game tree leaves an SCC, it can only reach SCCs further down in the topological order, and, since we compute values in this order, the value of the reached state is already known, allowing us to re-use it immediately.

Pruning. In addition to employing the MCTS values as game values in the tree expansion, we also use it to prune the game tree. In particular, once we computed the Monte Carlo values for each state, we restrict the choice of the environment to the successors which yield (close to) the lowest Monte Carlo value (recall that the environment prefers lower values). We empirically chose 0.02 as a threshold, i.e. we only keep those edges for the environment which are within 0.02 value of the lowest decision. While in theory this might remove crucial paths due to statistical fluctuations of MCTS, in practice it allows for a much deeper game tree, which in our experiments heavily outweighed the theoretical downside.

5 Handling the Truth

We introduced a way how to obtain a well-founded notion of "value" (to be precise, an approximation thereof) for a choice, i.e. an indication how good this choice is. As such, we can rank edges by their value in each state. Intuitively,

picking an edge which is ranked very highly should correspond to a good chance of winning. A high value means that even against an optimal player we can very likely close a winning loop, and, due to beta decay, do so quickly, thus minimizing the chance for an error.

Recall that our goal is to provide a good initial strategy. Thus, the exact values actually are irrelevant, since we only want to give the best edge as initial choice. Instead of trying to predict the exact value, we therefore want to learn this relative ranking. Formally, suppose we consider a system vertex $v \in V_{\mathcal{S}}$ with edges $E_v = \{(v, u) \mid (v, u) \in E\}$. A ranking of edges effectively corresponds to a (total) order $\prec_v \subseteq E_v \times E_v$. The principle of *pairwise ranking* [30] suggests that we learn a function $f : E_v \times E_v \to \{-1, 1\}$ that classifies pairs of edges depending on which one is the better choice, i.e. $f(e, e') = 1$ if $e' \prec_v e$ and -1 otherwise. However, such a function might not be perfect. For example, we could get $f(e_1, e_2) = 1$, $f(e_2, e_3) = 1$, and $f(e_3, e_1) = 1$, which is incompatible with any order. Thus, learning to rank suggests to determine an ordering $\prec$ that minimizes the *inversions* w.r.t. f, i.e. the number of cases where $f(e, e') = 1$ but $e \prec_v e'$. This problem, called rank aggregation, is known to be **NP**-hard, and we employ a greedy approximation as suggested by [30].

Our concrete goal thus now is to learn such a function f based on the semantic labelling of the start and end vertices of the two edges. We want to employ machine learning for this purpose: While the high-level intuition of the semantic labelling is rather clear, the actual implementation used to obtain the games [24] employs numerous optimizations, separate cases, etc. To provide the reader with a sense of the complexity, we display a single edge in the automaton obtained for a simple formula in Fig. 4, and a real-world scenario in [23, Appendix A.1].

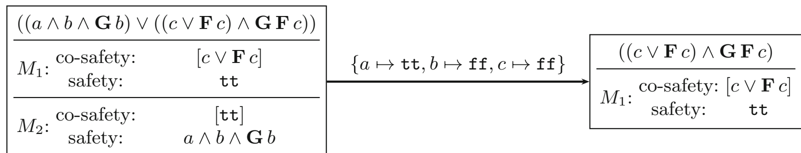


Fig. 4. A single transition in the automaton computed for the formula $(a \wedge \mathbf{G}b) \vee \mathbf{GF}c$.

We proceed to describe (i) (some of) the features we use, i.e. which quantities we extract from the labelling, (ii) the model we employ, and (iii) the dataset and methodology used to train our model.

5.1 Features

In total, we have defined over 200 different features to convert the edges into a usable vector of reals. In the interest of space we only present the high-level ideas of a small subset which covers most interesting ideas.

Since most information is contained in the states rather than in the edges themselves, the majority of our features are defined for the former. An edge is

then either associated with the feature value of its successor or with the change in a feature value between its predecessor and successor. As indicated in Fig. 4, the semantic labelling comprises several formulae, namely a "master" formula, which intuitively indicates the global state, and several "monitors" (which themselves comprise several formulae), monitoring repeating sub-goals. We define *base features*, which convert a single formula to a single number. These features can then be applied to both the master as well as monitor formulae, where further aggregation is necessary. Some notable base features are the following:

Number of Conjuncts We count the number of conjuncts if the top level operator is a conjunction and otherwise default to 1. The intuition behind this feature is that less conjuncts tend to correspond to a less constrained formula. Further, reducing the number of conjuncts along an edge often means that sub-goals have been achieved. (We consider several further syntactic features such as the number of disjuncts, the height of the syntax tree, or the number of temporal operators, which all follow similar ideas.)

Trueness Since this has proven to be a solid heuristic on its own, we again incorporate it as a feature.

System Control This feature (and variations thereof) incorporate the information of the variable partitioning by approximating how much impact the choice of the system variables has on the truth value of the formula. Intuitively, a higher system control is desirable. Further, this feature also counteracts false positives of, e.g., trueness, as high values of trueness are worth much less if the system has no control on whether one of the many satisfying assignments is played.

Obligation Set This group of features is based on the idea of *obligation sets* as introduced by [29]. In essence, an obligation set for a formula φ is an assignment that, if played indefinitely, satisfies the formula. Using the inductive definition of [29], we can compute a formula φ' whose satisfying assignments are exactly the obligation sets of φ, see [23, Appendix A.2]. Using this new formula, we can obtain numerous new features by applying other base features to φ'. In particular, we are interested in the new formulas trueness as this indicates how many obligation sets exist. Further, we are interested in its system control, as a higher value makes it more likely that the system can enforce at least one obligation set.

In addition to the base features, we define the following edge-specific features:

Priority As priorities are crucial for winning a play, it is only natural to incorporate that information in our features. However, as SVMs struggle with parity information, we reorder the priorities by how beneficial they are for the system and map them to $[-1, 1]$ (similar to [22]). In particular, the smallest odd priority gets mapped to 1 and the smallest even priority to -1. For this normalization, we use an a-priori upper bound provided by the underlying automaton construction.

Progress This feature is rather similar to [22]'s progress feature. We compute the percentage of already succeeded sub-goals of a monitor (instead of

their trueness) and aggregate by weighted average (rather than maximum). Additionally, we introduce punishments for failing monitors. Intuitively, this encourages long-term progress for temporal goals.

One Step Here, the idea is to recommend an assignment that is to be played in the current state by traversing the syntax tree and propagating recommendations upwards, which is inspired by message passing in graph neural networks. For example, if we see $a \wedge b$ we strongly recommend playing a and b, if we see $\mathbf{F}(a \wedge b)$ we take the previous recommendation and tune it down, since $\mathbf{F}$ is "less urgent". The feature value is obtained by measuring how well the valuation of an edge aligns with the recommended assignment.

5.2 Pair Classification by Support Vector Machines

To instantiate our pair classification function f, we opt for support vector machines. In principle, one could employ any binary classifier, which is why we also experimented with other models such as decision trees, random forests or gradient boosted trees. However, SVMs proved to perform best, which we attribute to their great ability to generalize due to their margin maximizing nature [30]. Additionally, SVMs are rather simple (compared to our other options) and provide us with extra information known as *confidence*. Given by the distance of the predicted sample to the decision hyperplane, its magnitude can be interpreted as how confident the SVM is in its prediction. We denote the confidence of a pair (e_1, e_2) by $c(e_1, e_2)$ and use it to slightly alter the greedy ranking algorithm from literature. To rank the edges of a vertex v, each edge $e \in E_v$ gets assigned a score $s(e) = \sum_{e' \in E_v, e' \neq e} c(e, e')$. Recall that if we predict $e \prec_v e'$, the confidence is negative. Finally, we rank the edges according to their score, where a higher score corresponds to a better edge, and the recommended strategy is obtained by playing the highest ranked edge for each state.

5.3 Further Notes on Implementation

In addition to the feature extraction, there are several other engineering aspects, which are crucial for the final performance. In this section, we comment on the three most important ones.

Statewise Feature Normalization. Before passing the features to the model, we proceed to normalize them. Due to possible future applications in on-the-fly solvers, we only consider feature values of edges from the same state for this normalization. The crucial observation is that this already introduces comparative information in the features. A normalized trueness value of 1, for example, means this edge has the best trueness among all other edges from their state although it does not tell us anything about its absolute value. While the latter might also be important in theory, we observed that in practice the statewise normalized value is more important with only a few exceptions.

State Classification. We observed several significantly different behaviours required in different states. For example, in some states we need to exclusively focus on the master formula, while in others only the monitors play a role. This also relates to the underlying principles of the automaton construction. It is very difficult, especially for a simple model like an SVM, to switch between different behaviours. We divide states into three groups which approximate the different classes, and train separate models for each class. The three classes we suggest are (i) states without monitors, (ii) states where the master formula does not change in any successor, (iii) and states that fall into neither category. In addition to having the separate models learn separate behaviours, we can also provide them with separate feature sets that only include relevant information. For example, the first class only requires features of the master formula, whereas these can be neglected in the second one.

Complement Construction. The underlying automaton construction uses the fact that the system being able to enforce satisfaction of a formula φ is equivalent to the environment being able to enforce falsification of $\neg\varphi$. In other words, solving the game for the negated formula and swapped roles yields the same result. However, in the game obtained for $\neg\varphi$ the role of "system", the player who choses second and for which we learnt the recommendation, i.e. for transitions from states (p, v) to q, now corresponds to the original environment. This drastically changes the meaning of features. For example, a trueness of 0 suddenly is very desirable. We tackle this by training separate models for both cases. Together with state classification, this yields a total of 6 different models that we assemble for our heuristic.

5.4 Training the Model

With these ideas at hand, we conclude this section by discussing our dataset, in particular how we preprocess it, and how we train our model.

Dataset and Preprocessing. As one of our goals is to exploit human bias in writing LTL formulae, the foundation of our dataset is given by the LTL benchmarks of SYNTCOMP.[2]. To further augment the data, we mutate these formulae by randomly replacing temporal operators. This yields new (random) samples that syntactically resemble the original, human-written structure. For practical reasons, we only consider formulae which can be converted to a DPA within 10 min. Ultimately, this leaves us with 405 original and 514 mutated formulae, of which we use 60% each for training, 20% for validation, and 20% for evaluation.

Obtaining the edge pairs for training requires several further steps. First of all, we exclude trivial cases that can easily be detected by simple rules (see Sect. 4.3), allowing our model to focus on complicated cases. Further, we exclude pairs where the ground truth value happens to be equal, as it is unclear which edge the model should predict. In particular, we exclude all edges originating in losing

[2] Available on GitHub https://github.com/SYNTCOMP/benchmarks.

states (since there is no sensible action to recommend). Finally, we only include a limited amount of pairs per game in the training set: Pairs of the same game tend to look similar, thus a few disproportionately large games would result in a very unbalanced dataset. All remaining edge pairs are added in both orders, i.e. $((e_1, e_2), y)$ and $((e_2, e_1), -y)$, where $y \in \{1, -1\}$ determines which edge is better, in order to prioritize teaching symmetry to the model.

Training. For each of the 6 models, we first compute mean and standard deviation of the respective training set and use them to standardize the input to $\mathcal{N}(0, 1)$. Further, we perform recursive feature elimination for each state class individually, adapted to features appearing twice (once for each input edge). For each state class, we ended up with 30–40 features.

For the actual training process, we performed an extensive grid search for several model types (decision trees, random forests, etc., see Sect. 5.2) in order to determine suitable values for the hyper-parameters. As mentioned earlier, we ultimately opted for the SVMs due to their simplicity and generalization abilities.

6 Experimental Evaluation

In this section, we present experimental evaluation of our tool `SemML`. The model was learnt by communicating the relevant data to a Python process running `scikit-learn` [35]. We then extracted the learnt weights and, based on them, implemented the recommendation procedure in Java, on top of `Owl` [24]. The artifact can be found at [1], which references a slightly improved version from the one we submitted to the artifact evaluation [2].

6.1 Evaluation Goals

Our primary goal in this work is to show that our approach, enabled by our new ground truth, can be used to solve more complicated instances than the approach of [22], in particular formulae going beyond pure (co-)safety. Thus, our first evaluation goal is the following:

> *Research Question 1:* How much does our model based on SVM and the game tree ground truth outperform the trueness-based initial strategy recommendation approach of [22]?

We refer to the trueness-based initial strategy of [22] as `TrueSI`.

Although not the focus of this work, we ultimately want to improve synthesis through meaningful exploration guidance, in particular, by suggesting likely winning edges. Thus, we are interested how our prototype performs in a real-world scenario.

> *Research Question 2:* How do initial strategies recommended by our approach synergize with state-of-the-art synthesis tools?

We address both questions separately.

6.2 RQ1: Quality of Initial Strategy

Datasets. To fairly compare to [22], we consider the same dataset, i.e. randomly generated LTL formulae, split into three categories: "(Co-)Safety", "Near (Co-)Safety", and "Parity". See [22] for details on how these are obtained. In essence, the tool `randltl` [7] is used to generate random formulae with different biases. Then, we filter out formulae which need more than 10 min to be translated to a parity automaton. As a second dataset, we also use some (original and mutated) SYNTCOMP formulae (the test set described in Sect. 5.4). We only consider formulae where the corresponding game can be won by system. We do this simply because we can only recommend on games which are winning – otherwise there is no preference on edges since every action is losing by definition. In total, this leaves 262 randomly generated formulae and 123 from SYNTCOMP.

Metrics. We consider two metrics for our comparison. Firstly, similar to [22], we consider the fraction of *immediately solved* games, i.e. games where following actions recommended by `SemML` or `TrueSI` directly yields a winning strategy. In light of our motivation to augment SI solvers, we want to measure how "close" the recommended strategy is to being correct in case is not immediately winning. To this end, we feed it to (a modified version of) the parity game solver `Oink` [6] and compute the *(relative) distance* of the obtained strategy, as follows. We count the number of (reachable) states in which the winning strategy determined by `Oink` differs from the recommended one, i.e. how many "wrong" choices were recommended, and divide it by the total amount of (reachable) states. We note that this unfortunately induces a slight bias that we cannot measure: `Oink` may potentially change winning decisions because of internal details of the algorithm. Ideally, we would want to obtain the minimal distance over all winning strategies; however this quantity is intractable to compute due to the exponential size of the strategy space. Nevertheless, we believe that this measure strongly correlates with the quality of the strategy.

We argue that simply measuring the number of iterations required by strategy iteration to converge is a too crude metric: On the one hand, even a "very wrong" strategy can be changed to a winning strategy in a single iteration by changing the choice in every single state. On the other hand, even a nearly correct strategy, requiring only a hand full of changes, may need as many iterations. Moreover, this additionally induces the same bias as above.

Table 1. Summary of our comparison between `TrueSI`, the approach of [22], and our tool `SemML`. We first list the fraction of immediately winning strategies (larger is better), followed by the geometric mean of the relative distance, i.e. the fraction of states in which the decision was adapted by `Oink` to obtain a winning strategy (smaller is better). For the first comparison, we also consider random initialization as a baseline. For this second comparison to be fair, we only consider games where neither tool yielded an immediately winning strategy.

Tool	(Co-)Safety	Near (Co-)Safety	Parity	SYNTCOMP
Immediately Solving				
`TrueSI`	100%	85%	66%	44%
`SemML`	99%	95%	88%	85%
Random	7%	2%	5%	3%
Relative Distance				
`TrueSI`	–	75%	45%	29%
`SemML`	–	52%	28%	16%
Ratio of both	–	1.4	1.6	1.8

Fig. 5. A detailed comparison on SYNTCOMP formulae. The left plot compares how many games were immediately solved, grouped by size and considering the (arithmetic) mean in each group. `SemML`'s values are displayed by crosses, `TrueSI` by circles. The right plot compares the relative distance of `SemML`'s and `TrueSI`'s solutions.

Expectations. Since our approach incorporates trueness as one of its many features, we expect that our approach should be at least on par with the previous one of [22]. As we also consider long-term temporal information beyond trueness, we particularly expect to outperform `TrueSI` on larger, more complicated instances.

Results. We ran this evaluation on consumer hardware (Intel Core i7-8565U with 16GB RAM). We summarize our findings in Table 1. Clearly, our approach vastly outperforms the previous one. In particular, while `TrueSI` perfectly handles (co-)safety formulae, its performance quickly drops when going to more complicated formulae. In comparison, the `SemML` solves the vast majority of formulae immediately, even on the quite complicated SYNTCOMP dataset. We note that these findings are not "absolute" (as to be expected from machine

learning approaches). There are few instances where the previous approach does perform better. Our baseline comparison to a random initialization approach validates that both approaches indeed solve a non-trivial problem.

Since we are particularly interested in complex, "human written" formulae, we investigate the SYNTCOMP dataset more closely. In Fig. 5, we provide a more detailed view on our two metrics. First, we investigate how the "immediately solving" performance evolves in comparison to the size of the game, which intuitively correlates with the difficulty of the synthesis question. We observe that `SemML` solves practically all smaller games and still performs well on larger games, compared to `TrueSI`, which quickly falls off. The second plot displays the relative distances for each instance which neither recommendation solved immediately. We clearly see that the strategies recommended by `SemML` are better in almost all cases.

This positively answers our first question. Aside from the direct comparison to the previous approach, the significant percentage of immediately solved games gives us an interesting implication: If `SemML` solves many games immediately, we can use `SemML` as a best-effort guidance tool for reactive synthesis questions which are intractably large to solve. Moreover, `SemML` thus presents us with a constant size representation of a winning strategy for many games, effectively described by approximately a few hundred SVM weights compared to a decision table for thousands of states in *each* game.

6.3 RQ2: On-the-fly SemML

In our second experiment, we evaluate the suitability of `SemML` for real-world parity game solving by using it as guidance tool for the state-of-the-art reactive synthesis tool `Strix` [33].

Strix' Anatomy. We first briefly describe how `Strix` works and how it uses guidance heuristics. In essence, `Strix` builds the parity game on-the-fly, i.e. iteratively constructs parts of the game it deems important. Then, two strategy improvements are running in parallel, one for either player. Not yet explored states are treated as losing for both. In this way, if we find a winning strategy for either player on the constructed part of the game, it is winning for the complete game. Otherwise, we need to explore further. Here, a key ingredient for practical efficiency is a heuristic to decide which states should be explored first: If we explore states reachable under the "smallest" winning strategy, we naturally find this strategy as quickly as possible. In its current form, `Strix` employs trueness for this guidance and selects an *automaton* edge with the *globally* highest trueness for exploration. (Dually, edges with the lowest trueness are also followed, since these are "promising" for the environment.)

Integration. We integrate `SemML` with `Strix` as follows. Suppose we are asked to compute a global score for an automaton edge $e = (p, q)$ (recall that `SemML` gives *local* advice on edges in the *game*). We explicitly build up the game between the automaton states p and q, i.e. all choices of the environment in p followed by the

respective system choices. For each occurring system state s, we compute the SemML ranking score as explained in Sect. 5.2, i.e. the confidence based score. This only gives us local information: the magnitude of our score only reflects the preference relative to actions available in the system state $s = (p, v)$. Since the previously used trueness proved to be a good indicator for global progress, we multiply our local score by this global value. Finally, to obtain a value for the automaton edge, we take the minimal value of all arising system states, since the environment chooses first. We additionally apply straightforward rules such as assigning values of 0 and 1 values to ff and tt states, respectively. Finally, Strix by default employs a decomposition approach, which does not build a single DPA. Therefore, SemML would not be applicable, and we disable it for the purpose of evaluation.

Dataset. We considered 188 randomly selected formulae of SYNTCOMP (which were not used in the training of the model), also including unrealizable ones.

Metrics. We evaluate the total required time to solve the game and compare to Strix in its normal configuration. Since we expect the unoptimized computation of SemML's advice to take considerable time, we separately measure the required time and additionally perform a comparison with this time subtracted. Since our scoring function is a straightforward SVM, we strongly believe that by tailoring the evaluation to Strix' requirements, it can be significantly sped up. In particular, our advice computation re-constructs information which is computed during the exploration of the automaton but difficult to access without significant changes to both Strix and Owl.

Expectations. We do not expect this approach to work to its full potential because Strix architecture does not exactly fit our approach (recall that our primary motivation was to compare to [22]). We discuss these differences and possible ways to address them later. Moreover, as we construct the intermediate game states for every recommendation and evaluate the recommender SVM several times, we expect that significant time is spent computing the advice of SemML.

Results. We conducted our experiments on a server with an Intel Xeon E5-2630 v4 processor with 256GiB of RAM and employed a 10 min timeout per execution. We summarize our findings in Fig. 6. Strikingly, our approach already performs favourably, despite the differences in architecture, hardly optimized advice computation, and no specific re-training for the task at hand. Excluding the time spent for advice computation, our approach performs significantly better in practically all instances. This answers our second question positively, too.

Adapting SemML to Strix In order to adapt our underlying approach, we require several non-trivial changes to SemML. We discuss the "mismatches" between the current approach and how they could be addressed. First, Strix selects a globally optimal edge to explore while SemML suggest actions locally. In particular,

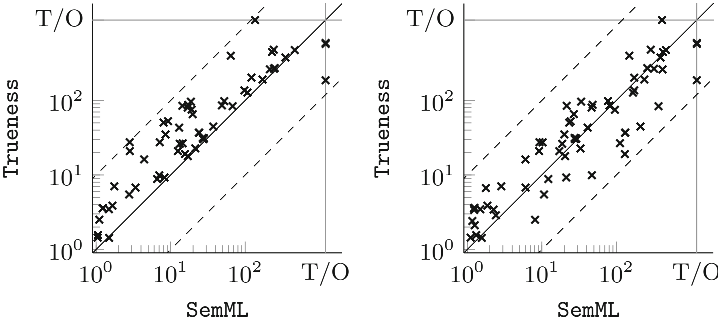


Fig. 6. Scatter plot comparing `Strix` with guidance provided by `SemML` and the default `Trueness`. On the left, we depict the total runtime excluding time spent for computing the guidance, and on the right we show the total time. We plot all models for which at least one method produced a result and count timeouts as 20 min (twice the timeout of 10 min). Note that the plot is logarithmic. The dashed lines denote a 10x difference.

our scoring is not trained to compare edges of two different states. While trueness seems to be a good compromise for the time being, we believe that (through significant engineering effort) `Strix` can be modified to accommodate local recommendations, or, alternatively, a more sophisticated indicator of a state's global relevance can be learnt. Second, `Strix` performs two searches, one for the environment and one for the system player. However, the parity games we deal with are not entirely symmetric – environment always moves first. Thus, we cannot directly apply `SemML`'s ranking to environment states, as they have a different structure. Here, we believe that the best solution is to train a separate model for the environment (or rather, six further models). Thirdly, `Strix` only constructs the automaton explicitly and computes the game implicitly. As such, `Strix` requests scoring information only for edges in the automaton and not in the game. This can be addressed by closely integrating the scoring computation with the exploration of the automaton – instead of rebuilding the game for each edge (p, q), we can compute all scores for all outgoing edges of p at once. Finally, as we mentioned, `Strix` by default applies a decomposition approach which builds several sub-automata. These also are equipped with semantic labelling, however with a different meaning – enough to create a significant hurdle for our learning approach. We note that `Strix` actually builds automata by communicating with `Owl` through a highly optimized interface between Java and C++, significantly complicating passing information back and forth between the processes.

7 Conclusion

We demonstrated that semantic labelling can be exploited for practical gains in LTL synthesis. Our experimental evaluation shows that we vastly outperform the simple approach of [22], the first step in this direction. Moreover, despite several

mismatches, our approach shows promising results for real world applications of this idea, i.e. when combined with the state-of-the-art tool Strix.

Future Work. As discussed above, the main point for future work is a tight, tailored integration with Strix. In particular, we want to modify our approach to be applicable to the decomposition methods of Strix, modify Strix to consider local guidance, and actually learn for the precise task required by Strix.

Aside from this, we believe that there might be further interesting features (hand-crafted or learnt) which could provide us with additional insights. In particular, we want to employ automated feature extraction, through more sophisticated model architectures such as *transformers* or *graph neural networks*.

References

1. Artifact for “Guessing Winning Policies in LTL Synthesis by Semantic Learning". Zenodo (2023). https://doi.org/10.5281/zenodo.7876095
2. Artifact for “Guessing Winning Policies in LTL Synthesis by Semantic Learning". Zenodo (2023). https://doi.org/10.5281/zenodo.7876096
3. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. RAIRO Theor. Inform. Appl. **36**(3), 261–275 (2002). https://doi.org/10.1051/ita:2002013
4. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasi-polynomial time. SIAM J. Comput. **51**(2), 17–152 (2022). https://doi.org/10.1137/17m1145288
5. Cosler, M., Schmitt, F., Hahn, C., Finkbeiner, B.: Iterative circuit repair against formal specifications. arXiv preprint arXiv:2303.01158 (2023)
6. Dijk, T.: Oink: an implementation and evaluation of modern parity game solvers. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 291–308. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_16
7. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
8. Esparza, J., Křetínský, J.: From LTL to deterministic automata: a safraless compositional approach. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 192–208. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_13
9. Esparza, J., Křetínský, J., Raskin, J.-F., Sickert, S.: From LTL and limit-deterministic büchi automata to deterministic parity automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 426–442. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_25
10. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From linear temporal logic and limit-deterministic büchi automata to deterministic parity automata. Int. J. Softw. Tools Technol. Transf. **24**(4), 635–659 (2022). https://doi.org/10.1007/s10009-022-00663-1
11. Esparza, J., Kretínský, J., Sickert, S.: One theorem to rule them all: a unified translation of LTL into ω-automata. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, 09–12 July 2018, pp. 384–393. ACM (2018). https://doi.org/10.1145/3209108.3209161

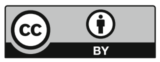
12. Esparza, J., Kretínský, J., Sickert, S.: A unified translation of linear temporal logic to ω-automata. J. ACM **67**(6), 33:1-33:61 (2020). https://doi.org/10.1145/3417995
13. Fearnley, J.: Efficient parallel strategy improvement for parity games. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 137–154. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_8
14. Fearnley, J., Jain, S., Schewe, S., Stephan, F., Wojtczak, D.: An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In: Erdogmus, H., Havelund, K. (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, 10–14 July 2017, pp. 112–121. ACM (2017). https://doi.org/10.1145/3092282.3092286
15. Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04761-9_15
16. Gaiser, A., Křetínský, J., Esparza, J.: Rabinizer: small deterministic automata for LTL(**F**,**G**). In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 72–76. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_7
17. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 455–459. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_34
18. Jacobs, S., et al.: The reactive synthesis competition (SYNTCOMP): 2018–2021. arXiv preprint arXiv:2206.00251 (2022)
19. Jurdzinski, M.: Deciding the winner in parity games is in UP \cap co-up. Inf. Process. Lett. **68**(3), 119–124 (1998). https://doi.org/10.1016/S0020-0190(98)00150-1
20. Komárková, Z., Křetínský, J.: Rabinizer 3: safraless translation of LTL to small deterministic automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 235–241. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_17
21. Křetínský, J., Garza, R.L.: Rabinizer 2: small Deterministic Automata for LTL/GL. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 446–450. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_32
22. Křetínský, J., Manta, A., Meggendorfer, T.: Semantic labelling and learning for parity game solving in LTL synthesis. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 404–422. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_24
23. Kretinsky, J., Meggendorfer, T., Prokop, M., Rieder, S.: Guessing winning policies in LTL synthesis by semantic learning (2023)
24. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: a library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 543–550. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_34
25. Křetínský, J., Meggendorfer, T., Sickert, S., Ziegler, C.: Rabinizer 4: from LTL to your favourite deterministic automaton. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 567–577. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_30
26. Kretínský, J., Meggendorfer, T., Waldmann, C., Weininger, M.: Index appearance record with preorders. Acta Inform. **59**(5), 585–618 (2022). https://doi.org/10.1007/s00236-021-00412-y
27. Kupferman, O., Rosenberg, A.: The blow-up in translating LTL to deterministic automata. In: van der Meyden, R., Smaus, J.-G. (eds.) MoChArt 2010. LNCS (LNAI), vol. 6572, pp. 85–94. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20674-0_6

28. Lehtinen, K., Parys, P., Schewe, S., Wojtczak, D.: A recursive approach to solving parity games in quasipolynomial time. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:8)2022
29. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL satisfiability checking revisited. In: Sánchez, C., Venable, K.B., Zimányi, E. (eds.) 2013 20th International Symposium on Temporal Representation and Reasoning, Pensacola, 26–28 September 2013, pp. 91–98. IEEE Computer Society (2013). https://doi.org/10.1109/TIME.2013.19
30. Liu, T.: Learning to rank for information retrieval. Found. Trends Inf. Retr. **3**(3), 225–331 (2009). https://doi.org/10.1561/1500000016
31. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Inform. 3–36 (2019). https://doi.org/10.1007/s00236-019-00349-3
32. Meyer, P.J., Luttenberger, M.: Solving mean-payoff games on the GPU. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 262–267. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_17
33. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 578–586. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_31
34. Osborne, M.J.: An introduction to game theory (2004)
35. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
36. Piterman, N.: From nondeterministic buchi and streett automata to deterministic parity automata. In: Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12–15 August 2006, Seattle, pp. 255–264. IEEE Computer Society (2006). https://doi.org/10.1109/LICS.2006.28
37. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
38. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0035790
39. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, 24–26 October 1988, pp. 319–327. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21948
40. Schewe, S.: Tighter bounds for the determinisation of Büchi automata. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1_13
41. Schmitt, F., Hahn, C., Rabe, M.N., Finkbeiner, B.: Neural circuit synthesis from specification patterns. In: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, 6–14 December 2021, Virtual, pp. 15408–15420 (2021). https://proceedings.neurips.cc/paper/2021/hash/8230bea7d54bcdf99cdfe85cb07313d5-Abstract.html
42. Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 312–332. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_17

43. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the Symposium on Logic in Computer Science (LICS 1986), Cambridge, Massachusetts, 16–18 June 1986, pp. 332–344. IEEE Computer Society (1986)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Policy Synthesis and Reinforcement Learning for Discounted LTL

Rajeev Alur[1], Osbert Bastani[1], Kishor Jothimurugan[1], Mateo Perez[2](✉), Fabio Somenzi[2], and Ashutosh Trivedi[2]

[1] University of Pennsylvania, Philadelphia, PA, USA
{alur,obastani,kishor}@seas.upenn.edu
[2] University of Colorado Boulder, Boulder, USA
{mateo.perez,fabio,ashutosh.trivedi}@colorado.edu

Abstract. The difficulty of manually specifying reward functions has led to an interest in using linear temporal logic (LTL) to express objectives for reinforcement learning (RL). However, LTL has the downside that it is sensitive to small perturbations in the transition probabilities, which prevents probably approximately correct (PAC) learning without additional assumptions. Time discounting provides a way of removing this sensitivity, while retaining the high expressivity of the logic. We study the use of discounted LTL for policy synthesis in Markov decision processes with unknown transition probabilities, and show how to reduce discounted LTL to discounted-sum reward via a reward machine when all discount factors are identical.

1 Introduction

Reinforcement learning [39] (RL) is a sampling-based approach to synthesis in systems with unknown dynamics where an agent seeks to maximize its accumulated reward. This reward is typically a real-valued feedback that the agent receives on the quality of its behavior at each step. However, designing a reward function that captures the user's intent can be tedious and error prone, and misspecified rewards can lead to undesired behavior, called *reward hacking* [5].

Due to the aforementioned difficulty, recent research [8,17,23,31,35] has shown interest in utilizing high-level logical specifications, particularly linear temporal logic [7] (LTL), to express intent. However, a significant challenge arises due to the sensitivity of LTL, similar to other infinite-horizon objectives like average reward and safety, to small changes in transition probabilities. Even slight modifications in transition probabilities can lead to significant impacts on the value, such as enabling previously unreachable states to become reachable. Without additional information on the transition probabilities, such as the minimum nonzero transition probability, LTL is proven to be not probably approximately correct (PAC) [29] learnable [3,43]. Ideally, it is desirable to maintain PAC learnability while still keeping the benefits of a highly expressive temporal logic.

This research was partially supported by ONR award N00014-20-1-2115, NSF grant CCF-2009022, and NSF CAREER award CCF-2146563.

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 415–435, 2023.
https://doi.org/10.1007/978-3-031-37706-8_21

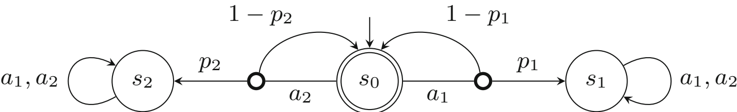


Fig. 1. Example showing non-robustness of safety specifications.

Discounting can serve as a solution to this problem. Typically, discounting is used to encode time-sensitive rewards (i.e., a payoff is worth more today than tomorrow), but it has a useful secondary effect that payoffs received in the distant future have small impact on the accumulated reward today. This insensitivity enables PAC learning without requiring any prior knowledge of the transition probabilities. In RL, discounted reward is commonly used and has numerous associated PAC learning algorithms [29].

In this work, we examine the discounted LTL of [2] for policy synthesis in Markov decision processes (MDPs) with unknown transition probabilities. We refer to such MDPs as "unknown MDPs" throughout the paper. This logic maintains the syntax of LTL, but discounts the temporal operators. Discounted LTL gives a quantitative preference to traces that satisfy the objective sooner, and those that delay failure as long as possible. The authors of [2] examined discounted LTL in the model checking setting. Exploring policy synthesis and learnability for discounted LTL specifications is novel to this paper.

To illustrate how discounting affects learnability, consider the example [32] MDP shown in Fig. 1. It consists of a safe state s_0, two sink states s_1, s_2, and two actions a_1, a_2. Taking action a_i in s_0 leads to a sink state with probability p_i and stays in s_0 with probability $1 - p_i$. Suppose we are interested in learning a policy to make sure that the system always stays in the state s_0. Now consider two scenarios—one in which $p_1 = 0$ and $p_2 = \delta$ and another in which $p_2 = 0$ and $p_1 = \delta$ where $\delta > 0$ is a small positive value. In the former case, the optimal policy is to always choose a_1 in s_0 and in the latter case, we need to choose a_2 in s_0. Furthermore, it can be shown that a near-optimal policy in one case is not near-optimal in another. However, we cannot select a finite number of samples needed to distinguish between the two cases (with high probability) without knowledge of δ. In contrast, the time-discounted semantics of the safety property evaluates to $1 - \lambda^k$ where k is the number of time steps spent in the state s_0. Then, for sufficiently small δ, any policy achieves a high value w.r.t. the discounted safety property in both scenarios. In general, small changes to the transition probabilities do not have drastic effects on the nature of near-optimal policies for discounted interpretations of LTL properties.

Contributions. Table 1 summarizes results of this paper in the context of known results regarding policy synthesis for various classes of specifications. We consider three key properties of specifications, namely, (1) whether there is a finite-state optimal policy and whether there are known algorithms for (2) computing an optimal policy when the MDP is known, as well as for (3) learning a near-optimal

Table 1. Policy synthesis in MDPs for different classes of specifications.

Specification	Memory	Policy Synthesis Algorithm	
		Known MDP	PAC Learning
Reward Machines	Finite [24,34]	Exists [24,34]	Exists [38]
LTL	Finite [7]	Exists [7]	Impossible [3,43]
Discounted LTL	Infinite	Open	Exists
Uniformly Discounted LTL	Finite	Exists	Exists

policy when the transition probabilities are unknown (without additional assumptions). The classes of specifications include reward machines with discounted-sum rewards [24], linear temporal logic (LTL) [7], discounted LTL and a variant of discounted LTL in which all discount factors are identical, which we call *uniformly* discounted LTL. In this paper, we show the following.

- In general, finite-memory optimal policies may not exist for discounted LTL specifications.
- There exists a PAC learning algorithm to learn policies for discounted LTL specifications.
- There is a reward machine for any uniformly discounted LTL specification such that the discounted-sum rewards capture the semantics of the specification. From this we infer that for any given MDP finite-memory optimal policies exist and can be computed.

Related Work. Linear temporal logic (LTL) is a popular and expressive formalism to unambiguously express qualitative safety and progress requirements of Kripke structures and MDPs [7]. The standard approach to model check LTL formulas against MDPs is the *automata-theoretic* approach where the LTL formulas are first translated to a class of good-for-MDP automata [20], such as limit-deterministic Büchi automata [18,36,37,40], and then, efficient graph-theoretic techniques (computing accepting end-component and then maximizing the probability to reach states in such components) [13,30,40] over the product of the automaton with the MDP can be used to compute optimal satisfaction probabilities and strategies. Since LTL formulas can be translated into (deterministic) automata in doubly exponential time, the probabilistic model checking problem is in 2EXPTIME with a matching lower bound [11].

Several variants of LTL have been proposed that provide discounted temporal modalities. De Alfaro et al. [15] proposed an extension of μ-calculus with discounting and showed [14] the decidability of model-checking over finite MDPs. Mandrali [33] introduced discounting in LTL by taking a discounted sum interpretation of logic over a trace. Littman et al. [32] proposed geometric LTL as a logic to express learning objectives in RL. However, this logic has unclear semantics for nesting operators. Discounted LTL was proposed by Almagor, Boker, and

Kupferman [2], which considers discounting without accumulation. The decidability of the policy synthesis problem for discounted LTL against MDPs is an open problem.

An alternative approach to discounting that ensuring PAC learnability is to introduce a fixed time horizon, along with a temporal logic for finite traces. In this setting, the logic LTL_f is the most popular [10,16]. Using LTL_f with a finite horizon yields simple algorithms [41], finite automata suffice for checking properties, but at the expense of the expressivity of the logic, formulas like $\mathbf{GF}p$ and $\mathbf{FG}p$ both mean that p occurs at the end of the trace.

There has been a lot of recent work on reinforcement learning from temporal specifications [1,9,16,19,21,22,24–28,31,32,42,44]. Such approaches often lack strong convergence guarantees. Some methods have been developed to reduce LTL properties to discounted-sum rewards [8,19] while preserving optimal policies; however they rely on the knowledge of certain parameters that depend on the transition probabilities of the unknown MDP. Recent work [3,32,43] has shown that PAC algorithms that do not depend on the transition probabilities do not exist for the class of LTL specifications. There has also been work on learning algorithms for LTL specifications that provide guarantees when additional information about the MDP (e.g., the smallest nonzero transition probability) is available [6,12,17].

2 Problem Definition

An alphabet Σ is a finite set of letters. A finite word (resp. ω-word) over Σ is defined as a finite sequence (resp. ω-sequence) of letters from Σ. We write Σ^* and Σ^ω for the set of finite and ω-words over Σ.

A *probability distribution* over a finite set S is a function $d\colon S \to [0,1]$ such that $\sum_{s \in S} d(s) = 1$. Let $\mathcal{D}(S)$ denote the set of all discrete distributions over S.

Markov Decision Processes. A Markov Decision Process (MDP) is a tuple $\mathcal{M} = (S, A, s_0, P)$, where S is a finite set of states, s_0 is the initial state, A is a finite set of actions, and $P : S \times A \to \mathcal{D}(S)$ is the transition probability function. An *infinite run* $\psi \in (S \times A)^\omega$ is a sequence $\psi = s_0 a_0 s_1 a_1 \ldots$, where $s_i \in S$ and $a_i \in A$ for all $i \in \mathbb{Z}_{\geq 0}$. For any run ψ and any $i \leq j$, we let $\psi_{i:j}$ denote the subsequence $s_i a_i s_{i+1} a_{i+1} \ldots a_{j-1} s_j$. Similarly, a *finite run* $h \in (S \times A)^* \times S$ is a finite sequence $h = s_0 a_0 s_1 a_1 \ldots a_{t-1} s_t$. We use $\mathcal{Z}(S, A) = (S \times A)^\omega$ and $\mathcal{Z}_f(S, A) = (S \times A)^* \times S$ to denote the set of infinite and finite runs, respectively.

A policy $\pi : \mathcal{Z}_f(S, A) \to \mathcal{D}(A)$ maps a finite run $h \in \mathcal{Z}_f(S, A)$ to a distribution $\pi(h)$ over actions. We denote by $\Pi(S, A)$ the set of all such policies. A policy π is deterministic if, for all finite runs $h \in \mathcal{Z}_f(S, A)$, there is an action $a \in A$ with $\pi(h)(a) = 1$.

Given a finite run $h = s_0 a_0 \ldots a_{t-1} s_t$, the *cylinder* of h, denoted by $\mathtt{Cyl}(h)$, is the set of all infinite runs with prefix h. Given an MDP $\mathcal{M}$ and a policy $\pi \in \Pi(S, A)$, we define the probability of the cylinder set by $\mathcal{D}_\pi^{\mathcal{M}}(\mathtt{Cyl}(h)) = \prod_{i=0}^{t-1} \pi(h_{0:i})(a_i) P(s_i, a_i, s_{i+1})$. It is known that $\mathcal{D}_\pi^{\mathcal{M}}$ can be uniquely extended

to a probability measure over the σ-algebra generated by all cylinder sets. Let $\mathcal{P}$ be a finite set of atomic propositions and $\Sigma = 2^{\mathcal{P}}$ denote the set of all valuations of propositions in $\mathcal{P}$. An infinite word $\rho \in \Sigma^\omega$ is a map $\rho : \mathbb{Z}_{\geq 0} \to \Sigma$.

Definition 1 (Discounted LTL). *Given a set of atomic propositions $\mathcal{P}$,* discounted LTL *formulas over $\mathcal{P}$ are given by the grammar*

$$\varphi := b \in \mathcal{P} \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}_\lambda \varphi \mid \varphi \ \mathbf{U}_\lambda \varphi$$

where $\lambda \in [0,1)$. Note that, in general, different temporal operators within the same formula may have different discount factors λ. For a formula φ and a word $\rho = \sigma_0\sigma_1 \ldots \in (2^{\mathcal{P}})^\omega$, the semantics $[\![\varphi, \rho]\!] \in [0,1]$ is given by

$$\begin{aligned}
[\![b, \rho]\!] &= \mathbb{1}\big(b \in \sigma_0\big) \\
[\![\neg\varphi, \rho]\!] &= 1 - [\![\varphi, \rho]\!] \\
[\![\varphi_1 \vee \varphi_2, \rho]\!] &= \max\big\{[\![\varphi_1, \rho]\!], [\![\varphi_2, \rho]\!]\big\} \\
[\![\mathbf{X}_\lambda \varphi, \rho]\!] &= \lambda \cdot [\![\varphi, \rho_{1:\infty}]\!] \\
[\![\varphi_1 \mathbf{U}_\lambda \varphi_2, \rho]\!] &= \sup_{i \geq 0} \left\{ \min \left\{ \lambda^i [\![\varphi_2, \rho_{i:\infty}]\!], \min_{0 \leq j < i} \{\lambda^j [\![\varphi_1, \rho_{j:\infty}]\!]\} \right\} \right\}
\end{aligned}$$

where $\rho_{i:\infty} = \sigma_i\sigma_{i+1}\ldots$ denotes the infinite word starting at position i.

Conjunction is defined using $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$. We use $\mathbf{F}_\lambda \varphi = \texttt{true}\mathbf{U}_\lambda \varphi$ and $\mathbf{G}_\lambda \varphi = \neg\mathbf{F}_\lambda \neg\varphi$ to denote the discounted versions of *finally* and *globally* operators respectively. Note that when all discount factors equal 1, the semantics corresponds to the usual semantics of LTL.

For this paper, we consider the case of strict discounting, where $\lambda < 1$. We refer to the case where the discount factor is the same for all temporal operators as *uniform discounting*. Our definition differs from [2] in two ways: 1) we discount the next operator, and 2) we enforce strict, exponential discounting.

Example Discounted LTL Specifications. To develop an intuition of the semantics of discounted LTL, we now present a few example formulas and their meaning.

- $\mathbf{F}_\lambda\, p$ obtains a value of λ^n where n is the first index where p becomes true in a trace, and 0 if p is never true. An optimal policy attempts to reach a p-state as soon as possible.
- $\mathbf{G}_\lambda\, p$ obtains a value of $1 - \lambda^n$ where n is the first index that a $\neg p$ occurs in a trace, and 1 if p always holds. An optimal policy attempts to delay reaching a $\neg p$-state as long as possible.
- $\mathbf{X}_\lambda\, p$ obtains a value of λ if p is in the second position and 0 otherwise.
- $p \vee \mathbf{X}_\lambda\, q$ obtains a value of 1 if p is in the first position of the trace, a value of λ if the trace begins with $\neg p$ followed by q, and a value of 0 otherwise.
- $\mathbf{F}_\lambda\, p \wedge \mathbf{G}_\lambda\, q$ evaluates to the minimum of λ^n and $(1-\lambda^m)$, where n is the first position where p becomes true in a trace and m is the first position where q becomes false. If $n^* = log_\lambda 0.5$ is the index where these two competing objectives coincide, then the optimal policy attempts to stay within q-states for the first n^* steps and then attempts to reach a p-state as soon as possible.

- Consider the formula $\mathbf{F}_{\lambda_1}\mathbf{G}_{\lambda_2}p$. Given a trace, consider a p-block of length m starting at position n, that is, p holds at all positions from n to $n+m-1$, and does not hold at position $n-1$ (or n is the initial position). The value of such a block is $\lambda_1^n(1-\lambda_2^m)$. The value of the trace is then the maximum over values of all such p-blocks. The optimal policy attempts to have as long a p-block as possible as early as possible. The discount factor λ_1 indicates the preference for the p-block to occur sooner and the discount factor λ_2 indicates the preference for the p-block to be longer.
- $\mathbf{G}_{\lambda_1}\mathbf{F}_{\lambda_2}p$ obtains a value equivalent to $\neg\mathbf{F}_{\lambda_1}\mathbf{G}_{\lambda_2}\neg p$. Traces which contain more p's at shorter intervals are preferred. The discount factor λ_1 indicates the preference for the total number of p's to be larger and λ_2 indicates the preference for the interval between the consecutive p's to be shorter.

Policy Synthesis Problem. Given an MDP $\mathcal{M} = (S, A, s_0, P)$, we assume that we have access to a *labelling function* $L : S \to \Sigma$ that maps each state to the set of propositions that hold true in that state. Given any run $\psi = s_0a_0s_1a_1\dots$ we can define an infinite word $L(\psi) = L(s_0)L(s_1)\dots$ that denotes the corresponding sequence of labels. Given a policy π for $\mathcal{M}$, we define the value of π with respect to a discounted LTL formula φ as

$$\mathcal{J}^{\mathcal{M}}(\pi,\varphi) = \mathop{\mathbb{E}}_{\rho\sim\mathcal{D}_\pi^{\mathcal{M}}}[\![\varphi,\rho]\!] \tag{1}$$

and the optimal value for $\mathcal{M}$ with respect to φ as $\mathcal{J}^*(\mathcal{M},\varphi) = \sup_\pi \mathcal{J}^{\mathcal{M}}(\pi,\varphi)$. We say that a policy π is optimal for φ if $\mathcal{J}^{\mathcal{M}}(\pi,\varphi) = \mathcal{J}^*(\mathcal{M},\varphi)$. Let $\Pi_{\text{opt}}(\mathcal{M},\varphi)$ denote the set of optimal policies. Given an MDP $\mathcal{M}$, a labelling function L and a discounted LTL formula φ, the policy synthesis problem is to compute an optimal policy $\pi \in \Pi_{\text{opt}}(\mathcal{M},\varphi)$ when one exists.

Reinforcement Learning Problem. In reinforcement learning, the transition probabilities P are unknown. Therefore, we need to interact with the environment to learn a policy for a given specification. In this case, it is sufficient to learn an ε-optimal policy π that satisfies $\mathcal{J}^{\mathcal{M}}(\pi,\varphi) \geq \mathcal{J}^*(\mathcal{M},\varphi) - \varepsilon$. We use $\Pi_{\text{opt}}^{\varepsilon}(\mathcal{M},\varphi)$ to denote the set of ε-optimal policies. Formally, a learning algorithm $\mathcal{A}$ is an iterative process which, in every iteration n, (i) takes a step in $\mathcal{M}$ from the current state, (ii) outputs a policy π_n and (iii) optionally resets the current state to s_0. We are interested in probably-approximately correct (PAC) learning algorithms.

Definition 2 (PAC-MDP). *A learning algorithm $\mathcal{A}$ is said to be PAC-MDP for a class of specifications $\mathcal{C}$ if, there is a function η such that for any $p > 0$, $\varepsilon > 0$, MDP $\mathcal{M} = (S, A, s_0, P)$, labelling function L, and specification $\varphi \in \mathcal{C}$, taking $N = \eta(|S|, |A|, |\varphi|, \frac{1}{p}, \frac{1}{\varepsilon})$, with probability at least $1-p$, we have*

$$\left|\left\{n \mid \pi_n \notin \Pi_{opt}^{\varepsilon}(\mathcal{M},\varphi)\right\}\right| \leq N.$$

It has been shown that there does not exist PAC-MDP algorithms for LTL specifications. Therefore, we are interested in the class of discounted LTL specifications that are strictly discounted, i.e. $\lambda < 1$ for every temporal operator.

3 Properties of Discounted LTL

In this section, we discuss important properties of discounted LTL regarding the nature of optimal policies. We first show that, under uniform discounting, the amount of memory required for the optimal policy may increase with the discount factor. We then show that, in general, allowing multiple discount factors may result in optimal policies requiring infinite memory. This motivates our restriction to the uniform discounting case in Sect. 4. We end this section by introducing a PAC learning algorithm for discounted LTL.

3.1 Nature of Optimal Policies

It is known that for any (undiscounted) LTL formula φ and any MDP $\mathcal{M}$, there exists a *finite memory* policy that is optimal—i.e., the policy stores only a finite amount of information about the history. Formally, given an MDP $\mathcal{M} = (S, A, s_0, P)$, a finite memory policy $\pi = (M, \delta_M, \mu, m_0)$ consists of a finite set of memory states M, a transition function $\delta_M : M \times S \times A \to M$ and an action function $\mu : M \times S \to \mathcal{D}(A)$. Given a finite run $h = s_0 a_0 \dots s_t = h' s_t$, the policy's action is sampled from $\mu(\delta_M(m_0, h'), s_t)$ where δ_M is also used to represent the transition function extended to sequences of state-action pairs. We use $\Pi_f(S, A)$ to denote the set of finite memory policies. In this paper, we will show that uniformly discounted LTL admits finite memory optimal policies, but that infinite memory may be required for the general case.

Unlike (undiscounted) LTL, discounted LTL allows a notion of satisfaction quality. In discounted LTL, traces which satisfy a reachability objective sooner are given a higher value, and are thus preferred. If an LTL formula cannot be satisfied, the corresponding discounted LTL formula will assign higher values to traces which delay failure as long as possible. These properties of discounted LTL are desirable for enabling notions of promptness, but may yield more complex strategies which try to balance the values of multiple competing subformulas.

Example 1. Consider the discounted LTL formula $\varphi = \mathbf{G}_\lambda p \wedge \mathbf{F}_\lambda \neg p$. This formula contains two competing objectives that cannot both be completely satisfied. Increasing the value of $\mathbf{G}_\lambda p$ by increasing the number of p's at the beginning of the trace before the first $\neg p$ decreases the value of $\mathbf{F}_\lambda \neg p$. Under the semantics of conjunction, the value of φ is the minimum of the two subformulas. Specifically, the value of φ w.r.t. a word ρ is

$$\begin{aligned}
[\![\mathbf{G}_\lambda p \wedge \mathbf{F}_\lambda \neg p, \rho]\!] &= [\![\neg \mathbf{F}_\lambda \neg p \wedge \mathbf{F}_\lambda \neg p, \rho]\!] \\
&= [\![\neg(\mathbf{F}_\lambda \neg p \vee \neg \mathbf{F}_\lambda \neg p), \rho]\!] \\
&= 1 - \max\{[\![\mathbf{F}_\lambda \neg p, \rho]\!], [\![\neg \mathbf{F}_\lambda \neg p, \rho]\!]\} \\
&= 1 - \max\left\{\sup_{i \geq 0}\{\lambda^i [\![\neg p, \rho_{i:\infty}]\!]\}, 1 - \sup_{i \geq 0}\{\lambda^i [\![\neg p, \rho_{i:\infty}]\!]\}\right\}.
\end{aligned}$$

where $\rho_{i:\infty}$ is the trace starting from index i. Now consider a two state (deterministic) MDP with two states $S = \{s_1, s_2\}$ and two actions $A = \{a_1, a_2\}$ in

which the agent can decide to either stay in s_1 or move to s_2 at any step and the system stays in s_2 upon reaching s_2. This MDP can be seen in Fig. 2. We have one proposition p which holds in state s_1 and not in s_2. Note that all runs produced by the example MDP are either of the form s_1^ω or $s_1^k s_2^\omega$. The discounted LTL value of runs of the form s_1^ω is 0. The value of runs of the form $\psi = s_1^k s_2^\omega$ is

$$v(k) = [\![\varphi, L(\psi)]\!] = 1 - \max\{\lambda^k, 1 - \lambda^k\} \ .$$

A finite memory policy stays in s_1 for k steps will yield this value. Since λ^k is decreasing in k and $1-\lambda^k$ is increasing in k, the integer value of k that maximizes $v(k)$ lies in the interval $[\gamma - 1, \gamma + 1]$ where $\gamma \in \mathbb{R}$ satisfies $\lambda^\gamma = 1 - \lambda^\gamma$. Figure 2 shows this graphically. We have that $\gamma = \frac{\log(0.5)}{\log(\lambda)}$ which is increasing in λ. Hence, the amount of memory required increases with increase in λ.

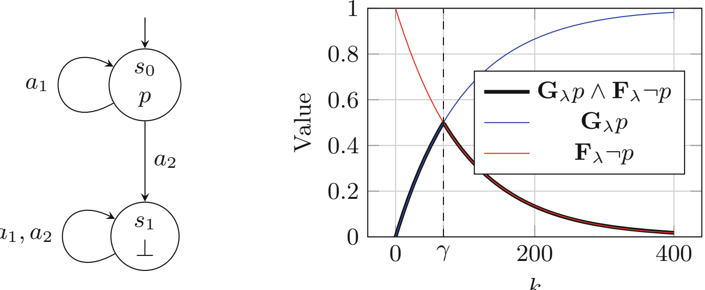


Fig. 2. An example showing that memory requirements for optimal policies may depend on the discount factor. The red line is λ^k, the blue line is $1 - \lambda^k$ and the solid black line is $v(k) = 1 - \max\{\lambda^n, 1 - \lambda^n\}$, where k is the number of time steps one remains in s_0. The dashed vertical line shows the value γ where $v(k)$ is maximized. We have set $\lambda = 0.99$. Note that changing the value of λ corresponds to rescaling the x-axis. (Color figure online)

The optimal strategy in the example above tries to balance the value of two competing subformula. We will now show that extending this idea to the general case of multiple discount factors requires balancing quantities that are decaying at different speeds. This balancing may require remembering an arbitrarily long history of the trace—infinite memory is required.

Theorem 1. *There exists an MDP $\mathcal{M} = (S, A, s_0, P)$, a labelling function L and a discounted LTL formula φ such that for all $\pi \in \Pi_f(S, A)$ we have $J^{\mathcal{M}}(\pi, \varphi) < \mathcal{J}^*(\mathcal{M}, \varphi)$.*

Proof. Consider the MDP $\mathcal{M}$ depicted in Fig. 3. It consists of three states $S = \{s_0, s_1, s_2\}$ and two actions $A = \{a_1, a_2\}$. The edges are labelled with actions and the corresponding transition probabilities. There are two propositions $\mathcal{P} = \{p_1, p_2\}$ and p_1 holds true in state s_1 and p_2 holds true in state s_2. The specification is given by $\varphi = \mathbf{F}_{\lambda_1}\mathbf{G}_{\lambda_2} p_1 \wedge \mathbf{F}_{\lambda_2} p_2$ where $\lambda_1 < \lambda_2 < 1$.

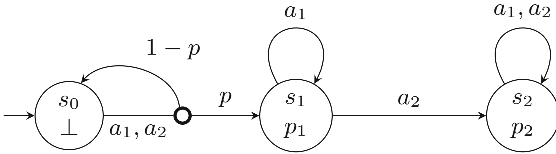


Fig. 3. The need for infinite memory for achieving optimality in discounted LTL.

For any run ψ that never visits s_2, we have $[\![\varphi, L(\psi)]\!] = 0$ since $[\![\mathbf{F}_{\lambda_2} p_2, L(\psi)]\!] = 0$. Otherwise the run has the form $\psi = s_0^{k_0} s_1^{k_1} s_2^{\omega}$ where k_0 is stochastic and k_1 is a strategic choice by the agent. To show that this requires an infinite amount of memory to play optimally, one just has to show that the optimal choice of k_1 increases with k_0. This means that the agent must remember k_0, the number of steps spent in the initial state, via an unbounded counter. Note that every value of k_0 has a non-zero probability in $\mathcal{M}$ and therefore choosing a suboptimal k_1 for even a single value of k_0 causes a decrease in value from the policy that always chooses optimal k_1.

The value of the run ψ is $[\![\varphi, L(\psi)]\!] = \min(\lambda_1^{k_0}(1 - \lambda_2^{k_1}), \lambda_2^{k_0+k_1})$. Note that $\lambda_1^{k_0}(1 - \lambda_2^{k_1})$ increases with increase in k_1 and $\lambda_2^{k_0+k_1}$ decreases with increase in k_1. Therefore taking $\gamma \in \mathbb{R}$ to be such that $\lambda_1^{k_0}(1 - \lambda_2^{\gamma}) = \lambda_2^{k_0+\gamma}$, the optimal choice of k_1 lies in the interval $[\gamma-1, \gamma+1]$. Now γ satisfies $1 = \big((\lambda_2/\lambda_1)^{k_0}+1\big)\lambda_2^{\gamma}$. Since $\lambda_1 < \lambda_2 < 1$ we must have that γ increases with increase in k_0. Therefore, k_1 also increases with increase in k_0. □

3.2 PAC Learning

In the above discussion, we showed that one might need infinite memory to act optimally w.r.t a discounted LTL formula. However, it can be shown that for any MDP $\mathcal{M}$, labelling function L, discounted LTL formula φ and any $\varepsilon > 0$, there is a finite-memory policy π that is ε-optimal for φ. In fact, we can show that this class of discounted LTL formulas admit a PAC-MDP learning algorithm.

Theorem 2 (Existence of PAC-MDP). *There exists a PAC-MDP learning algorithm for discounted LTL specifications.*

Proof (sketch). Our approach to compute ε-optimal policies for discounted LTL is to compute a policy which is optimal for T steps. The policy will depend on the entire history of atomic propositions that has occured so far.

Given discounted LTL specification φ, the first step of the algorithm is to determine T. We select T such that for any two infinite words α and β where the first $T+1$ indices match, i.e. $\alpha_{0:T} = \beta_{0:T}$, we have that $\big|[\![\varphi, \alpha]\!] - [\![\varphi, \beta]\!]\big| \leq \varepsilon$. Say that the maximum discount factor appearing in all temporal operators is $\lambda_{\max}$. Due to the strict discounting of discounted LTL, selecting $T \geq \frac{\log \varepsilon}{\log \lambda_{\max}}$ ensures that $\big|[\![\varphi, \alpha]\!] - [\![\varphi, \beta]\!]\big| \leq \lambda^n \leq \varepsilon$.

Now we unroll the MDP for T steps. We include the history of the atomic proposition sequence in the state. Given an MDP $\mathcal{M} = (S, A, s_0, P)$ and a labeling $L : S \to \Sigma$, the unrolled MDP $\mathcal{M}_T = (S', A', s'_0, P')$ is such that

$$S' = \bigcup_{t=0}^{T} S \times \underbrace{\Sigma \times \ldots \times \Sigma}_{t \text{ times}} ,$$

$A' = A$, $P'((s, \sigma_0, \ldots, \sigma_{t-1}), a, (s', \sigma_0, \ldots, \sigma_{t-1}, \sigma_t)) = P(s, a, s')$ if $0 \leq t \leq T$ and $\sigma_t = L(s')$, and is 0 otherwise (the MDP goes to a sink state if $t > T$). The leaves of the unrolled MDP are the states where T timesteps have elapsed. In these states, there is an associated finite word of length T. For a finite word of length T, we define the value of any formula φ to be zero beyond the end of the trace, i.e. $[\![\varphi, \rho_{j:\infty}]\!] = 0$ for any $j > T$. We then compute the value of the finite words associated with the leaves which is then considered as the reward at the final step. We can use existing PAC algorithms to compute an ε-optimal policy w.r.t. this reward for the finite horizon MDP $\mathcal{M}_T$ from which we can obtain a 2ε-optimal policy for $\mathcal{M}$ w.r.t the specification φ. $\square$

4 Uniformly Discounted LTL to Reward Machines

In general, optimal strategies for discounted LTL require infinite memory (Theorem 1). However, producing such an example required the use of multiple, varied discount factors. In this section, we will show that finite memory is sufficient for optimal policies under uniform discounting, where the discount factors for all temporal operators in the formula are the same. We will also provide an algorithm for computing these strategies.

Our approach is to reduce uniformly discounted LTL formulas to *reward machines*, which are finite state machines in which each transition is associated with a reward. We show that the value of a given discounted LTL formula φ for an infinite word ρ is the discounted-sum reward computed by a corresponding reward machine.

Formally, a reward machine is a tuple $\mathcal{R} = (Q, \delta, r, q_0, \lambda)$ where Q is a finite set of states, $\delta : Q \times \Sigma \to Q$ is the transition function, $r : Q \times \Sigma \to \mathbb{R}$ is the reward function, $q_0 \in Q$ is the initial state, and $\lambda \in [0, 1)$ is the discount factor. With any infinite word $\rho = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$, we can associate a sequence of rewards $c_0c_1 \ldots$ where $c_t = r(q_t, \sigma_t)$ with $q_t = \delta(q_{t-1}, \sigma_{t-1})$ for $t > 0$. We use $\mathcal{R}(\rho)$ to denote the discounted reward achieved by ρ,

$$\mathcal{R}(\rho) = \sum_{t=0}^{\infty} \lambda^t c_t,$$

and $\mathcal{R}(w)$ to denotes the partial discounted reward achieved by the finite word $w = \sigma_0\sigma_1 \ldots \sigma_T \in \Sigma^*$—i.e., $\mathcal{R}(w) = \sum_{t=0}^{T} \lambda^t c_t$ where c_t is the reward at time t.

Given a reward machine $\mathcal{R}$ and an MDP $\mathcal{M}$, our objective is to maximize the expected value $\mathcal{R}(\rho)$ from the reward machine reading the word ρ produced

by the MDP. Specifically, the value for a policy π for $\mathcal{M}$ is

$$\mathcal{J}^{\mathcal{M}}(\pi, \mathcal{R}) = \mathop{\mathbb{E}}_{\rho \sim \mathcal{D}_{\pi}^{\mathcal{M}}}[\mathcal{R}(\rho)]$$

where π is optimal if $\mathcal{J}^{\mathcal{M}}(\pi, \mathcal{R}) = \sup_{\pi} \mathcal{J}^{\mathcal{M}}(\pi, \mathcal{R})$. Finding such an optimal policy is straightforward: we consider the product of the reward machine $\mathcal{R}$ with the MDP $\mathcal{M}$ to form a product MDP with a discounted reward objective. In the corresponding product MDP, we can compute optimal policies for maximizing the expected discounted-sum reward using standard techniques such as policy iteration and linear programming. If the transition function of the MDP is unknown, this product can be formed on-the-fly and any RL algorithm for discounted reward can be applied. Using the state space of the reward machine as memory, we can then obtain a finite-memory policy that is optimal for $\mathcal{R}$.

We have the following theorem showing that we can construct a reward machine $\mathcal{R}_{\varphi}$ for every uniformly discounted LTL formula φ.

Theorem 3. *For any uniformly discounted LTL formula φ, in which all temporal operators use a common discount factor λ, we can construct a reward machine $\mathcal{R}_{\varphi} = (Q, \delta, r, q_0, \lambda)$ such that for any $\rho \in \Sigma^{\omega}$, we have $\mathcal{R}_{\varphi}(\rho) = [\![\rho, \varphi]\!]$.*

We provide the reward machine construction for Theorem 3 in the next subsection. Using this theorem, one can use a reward machine $\mathcal{R}_{\varphi}$ that matches the value of a particular uniformly discounted LTL formula φ, and then apply the procedure outlined above for computing optimal finite-memory policies for reward machines.

Corollary 1. *For any MDP $\mathcal{M}$, labelling function L and a discounted LTL formula φ in which all temporal operators use a common discount factor λ, there exists a finite-memory optimal policy $\pi \in \Pi_{opt}(\mathcal{M}, \varphi)$. Furthermore, there is an algorithm to compute such a policy.*

4.1 Reward Machine Construction

For our construction, we examine the case of uniformly discounted LTL formula with positive discount factors $\lambda \in (0, 1)$. This allows us to divide by λ in our construction. We note that the case of uniformly discounted LTL formula with $\lambda = 0$ can be evaluated after reading the initial letter of the word, and thus have trivial reward machines.

The reward machine $\mathcal{R}_{\varphi}$ constructed for the uniformly discounted LTL formula φ exhibits a special structure. Specifically, all edges within any given strongly-connected component (SCC) of $\mathcal{R}_{\varphi}$ share the same reward, which is either 0 or $1 - \lambda$, while all other rewards fall within the range of $[0, 1 - \lambda]$. We present an inductive construction of the reward machines over the syntax of discounted LTL that maintains these invariants.

Lemma 1. *For any uniformly discounted LTL formula φ there exists a reward machine $\mathcal{R}_{\varphi} = (Q, \delta, r, q_0, \lambda)$ such that following hold:*

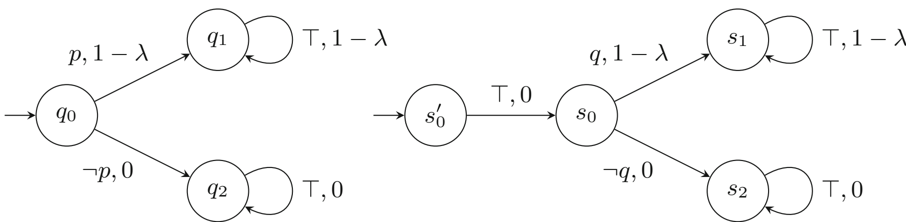


Fig. 4. Reward machines for $\varphi = p$ (left) and $\varphi = \mathbf{X}_\lambda q$ (right). The transitions are labeled by the guard and reward.

I_1. *For any* $\rho \in \Sigma^\omega$, *we have* $\mathcal{R}_\varphi(\rho) = [\![\rho, \varphi]\!]$.
I_2. *There is a partition of the states* $Q = \bigcup_{\ell=1}^{L} Q_\ell$ *and a type mapping* $\chi : [L] \to \{0, 1-\lambda\}$ *such that for any* $q \in Q_\ell$ *and* $\sigma \in \Sigma$,
  (a) $\delta(q, \sigma) \in \bigcup_{m=\ell}^{L} Q_m$, *and*
  (b) if $\delta(q, \sigma) \in Q_\ell$ *then* $r(q, \sigma) = \chi(\ell)$.
I_3. *For any* $q \in Q$ *and* $\sigma \in \Sigma$, *we have* $0 \le r(q, \sigma) \le 1 - \lambda$.

Our construction proceeds inductively. We define the reward machine for the base case of a single atomic proposition, i.e. $\varphi = p$, and then the construction for negation, the next operator, disjunction, the eventually operator (for ease of presentation), and the until operator. The ideas used in the constructions for disjunction, the eventually operator, and the until operator build off of each other, as they all involve keeping track of the maximum/minimum value over a set of subformulas. We use properties I_1 and I_3 to show correctness, and properties I_2 and I_3 to show finiteness. A summary of the construction and detailed proofs can be found in the full version of this paper [4].

Atomic Propositions. Let $\varphi = p$ for some $p \in \mathcal{P}$. The reward machine $\mathcal{R}_\varphi = (Q, \delta, r, q_0, \lambda)$ for φ is such that $Q = \{q_0, q_1, q_2\}$ and $\delta(q, \sigma) = q$ for all $q \in \{q_1, q_2\}$ and $\sigma \in \Sigma$. The reward machine is shown in Fig. 4 where edges are labelled with propositions and rewards. If $p \in \sigma$, $\delta(q_0, \sigma) = q_1$ and $r(q_0, \sigma) = 1 - \lambda$. If $p \notin \sigma$, $\delta(q_0, \sigma) = q_2$ and $r(q_0, \sigma) = 0$. Finally, $r(q_1, \sigma) = 1 - \lambda$ and $r(q_2, \sigma) = 0$ for all $\sigma \in \Sigma$. It is clear to see that I_1, I_2, and I_3 hold.

Negation. Let $\varphi = \neg\varphi_1$ for some LTL formula φ_1 and let $\mathcal{R}_{\varphi_1} = (Q, \delta, r, q_0, \lambda)$ be the reward machine for φ_1. Notice that the reward machine for φ can be constructed from $\mathcal{R}_{\varphi_1}$ by simply replacing every reward c with $(1 - \lambda) - c$ as $\sum_{i=0}^{\infty} \lambda^i (1 - \lambda) = 1$. Formally, $\mathcal{R}_\varphi = (Q, \delta, r', q_0, \lambda)$ where $r'(q, \sigma) = (1 - \lambda) - r(q, \sigma)$ for all $q \in Q$ and $\sigma \in \Sigma$. Again, assuming that invariants I_1, I_2, and I_3 hold for $\mathcal{R}_{\varphi_1}$, it easily follows that they hold for $\mathcal{R}_\varphi$.

Next Operator. Let $\varphi = \mathbf{X}_\lambda \varphi_1$ for some φ_1 and let $\mathcal{R}_{\varphi_1} = (Q, \delta, r, q_0, \lambda)$ be the reward machine for φ_1. The reward machine for φ can be constructed from

$\mathcal{R}_{\varphi_1}$ by adding a new initial state q_0' and a transition in the first step from it to the initial state of $\mathcal{R}_{\varphi_1}$. From the next step $\mathcal{R}_\varphi$ simulates $\mathcal{R}_{\varphi_1}$. This has the resulting effect of skipping the first letter, and decreasing the value by λ. Formally, $\mathcal{R}_\varphi = (\{q_0'\} \sqcup Q, \delta', r', q_0', \lambda)$ where $\delta'(q_0', \sigma) = q_0$ and $\delta'(q, \sigma) = \delta(q, \sigma)$ for all $q \in Q$ and $\sigma \in \Sigma$. Similarly, $r'(q_0', \sigma) = 0$ and $r'(q, \sigma) = r(q, \sigma)$ for all $q \in Q$ and $\sigma \in \Sigma$. Assuming that invariants I_1, I_2, and I_3 hold for $\mathcal{R}_{\varphi_1}$, it follows that they hold for $\mathcal{R}_\varphi$.

Disjunction. Let $\varphi = \varphi_1 \vee \varphi_2$ for some φ_1, φ_2 and let $\mathcal{R}_{\varphi_1} = (Q_1, \delta_1, r_1, q_0^1, \lambda)$ and $\mathcal{R}_{\varphi_2} = (Q_2, \delta_2, r_2, q_0^2, \lambda)$ be the reward machines for φ_1 and φ_2, respectively. The reward machine $\mathcal{R}_\varphi = (Q, \delta, r, q_0, \lambda)$ is constructed $\mathcal{R}_{\varphi_1}$ and $\mathcal{R}_{\varphi_2}$ such that for any finite word it maintains the invariant that the discounted reward is the maximum of the reward provided by $\mathcal{R}_{\varphi_1}$ and $\mathcal{R}_{\varphi_2}$. Moreover, once it is ascertained that the reward provided by one machine cannot be overtaken by the other for any suffix, $\mathcal{R}_\varphi$ begins simulating the reward machine with higher reward.

The construction involves a product construction along with a real-valued component that stores a scaled difference between the total accumulated reward for φ_1 and φ_2. In particular, $Q = (Q_1 \times Q_2 \times \mathbb{R}) \sqcup Q_1 \sqcup Q_2$ and $q_0 = (q_0^1, q_0^2, 0)$. The reward deficit ζ of a state $q = (q_1, q_2, \zeta)$ denotes the difference between the total accumulated reward for φ_1 and φ_2 divided by λ^n where n is the total number of steps taken to reach q. The reward function is defined as follows.

- For $q = (q_1, q_2, \zeta)$, we let $f(q, \sigma) = r_1(q_1, \sigma) - r_2(q_2, \sigma) + \zeta$ denote the new (scaled) difference between the discounted-sum rewards accumulated by $\mathcal{R}_{\varphi_1}$ and $\mathcal{R}_{\varphi_2}$. The current reward depends on whether $f(q, \sigma)$ is positive (accumulated reward from $\mathcal{R}_{\varphi_1}$ is higher) or negative and whether the sign is different from ζ. Formally,

$$r(q, \sigma) = \begin{cases} r_1(q_1, \sigma) + \min\{0, \zeta\} & \text{if } f(q, \sigma) \geq 0 \\ r_2(q_2, \sigma) - \max\{0, \zeta\} & \text{if } f(q, \sigma) < 0 \end{cases}$$

- For a state $q_i \in Q_i$ we have $r(q_i, \sigma) = r_i(q_i, \sigma)$.

Now we need to make sure that ζ is updated correctly. We also want the transition function to be such that the (reachable) state space is finite and the reward machine satisfies I_1, I_2 and I_3.

- First, we make sure that, when the difference ζ is too large, the machine transitions to the appropriate state in Q_1 or Q_2. For a state $q = (q_1, q_2, \zeta)$ with $|\zeta| \geq 1$, we have

$$\delta(q, \sigma) = \begin{cases} \delta_1(q_1, \sigma) & \text{if } \zeta \geq 1 \\ \delta_2(q_2, \sigma) & \text{if } \zeta \leq -1. \end{cases}$$

- For states with $|\zeta| < 1$, we simply advance both the states and update ζ accordingly. Letting $f(q, \sigma) = r_1(q_1, \sigma) - r_2(q_2, \sigma) + \zeta$, we have that for a

state $q = (q_1, q_2, \zeta)$ with $|\zeta| < 1$,

$$\delta(q, \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma), f(q, \sigma)/\lambda). \tag{2}$$

– Finally, for $q_i \in Q_i$, $\delta(q_i, \sigma) = \delta_i(q_i, \sigma)$.

Finiteness. We argue that the (reachable) state space of $\mathcal{R}_\varphi$ is finite. Let $Q_i = \bigcup_{\ell=1}^{L_i} Q_\ell^i$ for $i \in \{1, 2\}$ be the SCC decompositions of Q_1 and Q_2 that satisfy property I_2 for $\mathcal{R}_{\varphi_1}$ and $\mathcal{R}_{\varphi_2}$ respectively. Intuitively, if $\mathcal{R}_\varphi$ stays within $Q_\ell^1 \times Q_m^2 \times \mathbb{R}$ for some $\ell \leq L_1$ and $m \leq L_2$, then the rewards from $\mathcal{R}_{\varphi_1}$ and $\mathcal{R}_{\varphi_2}$ are constant; this enables us to infer the reward machine ($\mathcal{R}_{\varphi_1}$ and $\mathcal{R}_{\varphi_2}$) with the higher total accumulated reward in a finite amount of time after which we transition to Q_1 or Q_2. Hence the set of all possible values of ζ in a reachable state $(q_1, q_2, \zeta) \in Q_\ell^1 \times Q_m^2 \times \mathbb{R}$ is finite. This can be shown by induction.

Property I_1. Intuitively, it suffices to show that $\mathcal{R}_\varphi(w) = \max\{\mathcal{R}_{\varphi_1}(w), \mathcal{R}_{\varphi_2}(w)\}$ for every finite word $w \in \Sigma^*$. We show this property along with the fact that for any $w \in \Sigma^*$ of length n, if the reward machine reaches a state (q_1, q_2, ζ), then $\zeta = (\mathcal{R}_{\varphi_1}(w) - \mathcal{R}_{\varphi_2}(w))/\lambda^n$. This can be proved using induction on n.

Property I_2. This property is true if and only if for every SCC $\mathcal{C}$ of $\mathcal{R}_\varphi$ there is a type $c \in \{0, 1-\lambda\}$ such that if $\delta(q, \sigma) = q'$ for some $q, q' \in \mathcal{C}$ and $\sigma \in \Sigma$, we have $r(q, \sigma) = c$. From the definition of the transition function δ, $\mathcal{C}$ cannot contain two states where one is of the form $(q_1, q_2, \zeta) \in Q_1 \times Q_2 \times \mathbb{R}$ and the other is $q_i \in Q_i$ for some $i \in \{1, 2\}$. Now if $\mathcal{C}$ is completely contained in Q_i for some i, we can conclude from the inductive hypothesis that the rewards within $\mathcal{C}$ are constant (and they are all either 0 or $1 - \lambda$). When all states of $\mathcal{C}$ are contained in $Q_1 \times Q_2 \times \mathbb{R}$, they must be contained in $\bar{Q}_1 \times \bar{Q}_2 \times \mathbb{R}$ where $\bar{Q}_i$ is some SCC of $\mathcal{R}_{\varphi_i}$. In such a case, we can show that $|\mathcal{C}| = 1$ and in the presence of a self loop on a state within $\mathcal{C}$, the reward must be either 0 or $1 - \lambda$.

Property I_3. We now show that all rewards are bounded between 0 and $(1 - \lambda)$. Let $q = (q_1, q_2, \zeta)$ and $f(q, \sigma) = r_1(q_1, \sigma) - r_2(q_2, \sigma) + \zeta$. We show the bound for the case when $f(q, \sigma) \geq 0$ and the other case is similar. If $\zeta \geq 0$, then $r(q, \sigma) = r_1(q_1, \sigma) \in [0, 1 - \lambda]$. If $\zeta < 0$, then $r(q, \sigma) \leq r_1(q_1, \sigma) \leq 1 - \lambda$ and

$$r(q, \sigma) = r_1(q_1, \sigma) + \zeta = f(q, \sigma) + r_2(q_2, \sigma) \geq 0.$$

This concludes the construction for $\varphi_1 \vee \varphi_2$.

Eventually Operator. For ease of presentation, we treat the until operator as a generalization of the *eventually* operator $\mathbf{F}_\lambda$ and present it first. We have that $\varphi = \mathbf{F}\varphi_1$ for some φ_1. Let $\mathcal{R}_{\varphi_1} = (Q_1, \delta_1, r_1, q_0^1, \lambda)$ be the reward machine for φ_1. Let $\mathbf{X}_\lambda^i$ denote the operator $\mathbf{X}_\lambda$ applied i times. We begin by noting that

$$\mathbf{F}_\lambda\varphi_1 \equiv \bigvee_{i \geq 0} \mathbf{X}_\lambda^i \varphi_1 = \varphi_1 \vee \mathbf{X}_\lambda\varphi_1 \vee \mathbf{X}_\lambda^2\varphi_1 \vee \dots .$$

The idea of the construction is to keep track of the unrolling of this formula up to the current timestep n,

$$\mathbf{F}_\lambda^n \varphi_1 = \bigvee_{n \geq i \geq 0} \mathbf{X}_\lambda^i \varphi_1 = \varphi_1 \vee \mathbf{X}_\lambda \varphi_1 \vee \mathbf{X}_\lambda^2 \varphi_1 \vee \ldots \vee \mathbf{X}_\lambda^n \varphi_1.$$

For this, we will generalize the construction for disjunction. In the disjunction construction, there were states of the form (q_1, q_2, ζ) where ζ was a bookkeeping parameter that kept track of the difference between $\mathcal{R}_{\varphi_1}(w)$ and $\mathcal{R}_{\varphi_2}(w)$, namely, $\zeta = (\mathcal{R}_{\varphi_1}(w) - \mathcal{R}_{\varphi_2}(w))/\lambda^n$ where $w \in \Sigma^*$ is some finite word of length n. To generalize this notion to make a reward machine for $\max\{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$, we will have states of the form $\{(q_1, \zeta_1), \ldots, (q_n, \zeta_n)\}$ where $\zeta_i = (\mathcal{R}_i(w) - \max_j \mathcal{R}_j(w))/\lambda^n$. When $\zeta_i \leq -1$ then $\mathcal{R}_i(w) + \lambda^n \leq \max_j \mathcal{R}_j(w)$ and we know that the associated reward machine $\mathcal{R}_i$ cannot be the maximum, so we drop it from our set. We also note that the value of $\mathbf{X}_\lambda^i \varphi_1$ can be determined by simply waiting i steps before starting the reward machine $\mathcal{R}_{\varphi_1}$, i.e. $\lambda^i \mathcal{R}_{\varphi_1}(\rho_{i:\infty}) = \mathcal{R}_{\mathbf{X}_\lambda^i \varphi_1}(\rho)$. This allows us to perform a subset construction for this operator.

For a finite word $w = \sigma_0 \sigma_1 \ldots \sigma_n \in \Sigma^*$ and a nonnegative integer k, let $w_{k:\infty}$ denote the subword $\sigma_k \ldots \sigma_n$ which equals the empty word ϵ if $k>n$. We use the notation $[\![\mathbf{X}_\lambda^k \varphi_1, w]\!] = \lambda^k \mathcal{R}_{\varphi_1}(w_{k:\infty})$ and define $[\![\mathbf{F}_\lambda^k \varphi_1, w]\!] = \max_{k \geq i \geq 0} [\![\mathbf{X}_\lambda^k \varphi_1, w]\!]$ which represents the maximum value accumulated by the reward machine of some formula of the form $\mathbf{X}_\lambda^i \varphi_1$ with $i \leq k$ on a finite word w. The reward machine for $\mathbf{F}_\lambda \varphi_1$ will consist of states of the form (v, S), containing a value v for bookkeeping and a set S that keeps track of the states of all $\mathcal{R}_{\mathbf{X}_\lambda^i \varphi_1}$ that may still obtain the maximum given a finite prefix w of length n, i.e. reward machine states of all subformulas $\mathbf{X}_\lambda^i \varphi_1$ for $n \geq i \geq 0$ that satisfy $[\![\mathbf{X}_\lambda^i \varphi_1, w]\!] + \lambda^n > [\![\mathbf{F}_\lambda^n \varphi_1, w]\!]$ since λ^n is the maximum additional reward obtainable by any $\rho \in \Sigma^\omega$ with prefix w. The subset S consists of elements of the form $(q_i, \zeta_i) \in S$ where $q_i = \delta_1(q_0^1, w_{i:\infty})$ and $\zeta_i = ([\![\mathbf{X}_\lambda^i \varphi_1, w]\!] - [\![\mathbf{F}_\lambda^n \varphi_1, w]\!])/\lambda^n$ corresponding to each subformula $\mathbf{X}_\lambda^i \varphi_1$. The value $v = \max\{-1, -[\![\mathbf{F}_\lambda^n \varphi_1, w]\!]/\lambda^n\}$ is a bookkeeping parameter used to initialize new elements in the set S and to stop adding elements to S when $v \leq -1$. We now present the construction formally.

We form a reward machine $\mathcal{R}_\varphi = (Q, \delta, r, q_0, \lambda)$ where $Q = \mathbb{R} \times 2^{Q_1 \times \mathbb{R}}$ and $q_0 = (0, \{(q_0^1, 0)\})$. We define a few functions that ease defining our transition function. Let $f(\zeta, q, \sigma) = r_1(q, \sigma) + \zeta$ and $m(S, \sigma) = \max\limits_{(q_i, \zeta_i) \in S} f(\zeta_i, q_i, \sigma)$. For the subset construction, we define

$$\Delta(S, \sigma) = \bigcup_{(q,\zeta) \in S} \{(\delta_1(q, \sigma), \zeta') : \zeta' = \big((f(\zeta, q, \sigma) - m(S, \sigma))/\lambda\big) > -1\}$$

The transition function is

$$\delta((v, S), \sigma) = \begin{cases} \Big(v'(S, v, \sigma),\ \Delta(S, \sigma) \sqcup \big(q_0^1, v'(S, v, \sigma)\big)\Big) & \text{if } v'(S, v, \sigma) > -1 \\ (-1,\ \Delta(S, \sigma)) & \text{if } v'(S, v, \sigma) \leq -1 \end{cases}$$

where $v'(S, v, \sigma) = (v - m(S, \sigma))/\lambda$. The reward function is $r((v, S), \sigma) = m(S, \sigma)$.

We now argue that $\mathcal{R}_\varphi$ satisfies properties I_1, I_2 and I_3 and the set of reachable states in $\mathcal{R}_\varphi$ is finite assuming $\mathcal{R}_{\varphi_1}$ satisfies I_1, I_2 and I_3.

Finiteness. Consider states of the form $(v, S) \in Q$. If $v = 0$, then it must be that $\zeta_i = 0$ for all $(q_i, \zeta_i) \in S$ since receiving a non-zero reward causes the value of v to become negative. There are only finitely many such states. If $-1 < v < 0$, then we will reach a state $(v', S') \in Q$ with $v' = -1$ in at most n steps, where n is such that $v/\lambda^n \leq -1$. Therefore, the number of reachable states (v, S) with $-1 < v < 0$ is also finite. Also, the number of states of the form $(-1, S)$ that can be initially reached (via paths consisting only of states of the form (v, S') with $v > -1$) is finite. Furthermore, upon reaching such a state $(-1, S)$, the reward machine is similar to that of a disjunction (maximum) of $|S|$ reward machines. From this we can conclude that the full reachable state space is finite.

Property I_1. The transition function is designed so that the following holds true: for any finite word $w \in \Sigma^*$ of length n and letter $\sigma \in \Sigma$, if $\delta(q_0, w) = (v, S)$, then $m(S, \sigma) = (\llbracket \mathbf{F}_\lambda^{n+1}\varphi_1, w\sigma \rrbracket - \llbracket \mathbf{F}_\lambda^{n}\varphi_1, w \rrbracket)/\lambda^n$. Since $r((v, S), \sigma) = m(S, \sigma)$, we get that $\mathcal{R}_\varphi(w) = \llbracket \mathbf{F}_\lambda^{n}\varphi_1, w \rrbracket$. Thus, $\mathcal{R}_\varphi(\rho) = \llbracket \mathbf{F}_\lambda\varphi_1, \rho \rrbracket$ for any infinite word $\rho \in \Sigma^\omega$. This property for $m(S, \sigma)$ follows from the preservation of all the properties outlined in the above description of the construction.

Property I_2. Consider an SCC $\mathcal{C}$ in $\mathcal{R}_\varphi$ such that $(v, S) = \delta((v, S), w)$ for some $(v, S) \in \mathcal{C}$ and $w \in \Sigma^*$ of length $n > 0$. Note that if $-1 < v < 0$, then $(v', S') = \delta((v, S), w)$ is such that $v' < v$. Thus, it must be that $v = 0$ or $v = -1$. If $v = 0$, then all the reward must be zero, since any nonzero rewards result in $v < 0$. If $v = -1$, then it must be that for any $(q_i, \zeta_i) \in S$, q_i is in an SCC $\mathcal{C}_1^i$ in $\mathcal{R}_{\varphi_1}$ with some reward type $c_i \in \{0, 1 - \lambda\}$. For all ζ_i to remain fixed (which is necessary as otherwise some ζ_i strictly increases or decreases), it must be that all c_i are the same, say c. Thus, the reward type in $\mathcal{R}_{\varphi_1}$ for SCC $\mathcal{C}$ equals c.

Property I_3. We can show that for any finite word $w \in \Sigma^*$ of length n and letter $\sigma \in \Sigma$, if $\delta(q_0, w) = (v, S)$, then the reward is $r((v, S), \sigma) = m(S, \sigma) = (\llbracket \mathbf{F}_\lambda^{n+1}\varphi_1, w\sigma \rrbracket - \llbracket \mathbf{F}_\lambda^{n}\varphi_1, w \rrbracket)/\lambda^n$ using induction on n. Since property I_3 holds for $\mathcal{R}_{\varphi_1}$, we have that $0 \leq (\llbracket \mathbf{F}_\lambda^{n+1}\varphi_1, w\sigma \rrbracket - \llbracket \mathbf{F}_\lambda^{n}\varphi_1, w \rrbracket) \leq (1 - \lambda)\lambda^n$.

Until Operator. We now present the until operator, generalizing the ideas presented for the eventually operator. We have that $\varphi = \varphi_1 \mathbf{U}_\lambda \varphi_2$ for some φ_1 and φ_2. Let $\mathcal{R}_{\varphi_1} = (Q_1, \delta_1, r_1, q_0^1, \lambda)$ and $\mathcal{R}_{\varphi_2} = (Q_2, \delta_2, r_2, q_0^2, \lambda)$. Note that

$$\begin{aligned}\varphi_1 \mathbf{U}_\lambda \varphi_2 &= \bigvee_{i \geq 0} (\mathbf{X}_\lambda^i \varphi_2 \wedge \varphi_1 \wedge \mathbf{X}_\lambda \varphi_1 \wedge \ldots \wedge \mathbf{X}_\lambda^{i-1} \varphi_1) \\ &= \varphi_2 \vee (\mathbf{X}_\lambda \varphi_2 \wedge \varphi_1) \vee (\mathbf{X}_\lambda^2 \varphi_2 \wedge \varphi_1 \wedge \mathbf{X}_\lambda \varphi_1) \vee \ldots.\end{aligned}$$

The goal of the construction is to keep track of the unrolling of this formula up to the current timestep n,

$$\varphi_1 \mathbf{U}_\lambda^n \varphi_2 = \bigvee_{n \geq i \geq 0} (\mathbf{X}_\lambda^i \varphi_2 \wedge \varphi_1 \wedge \mathbf{X}_\lambda \varphi_1 \wedge \ldots \wedge \mathbf{X}_\lambda^{i-1} \varphi_1) = \bigvee_{n \geq i \geq 0} \psi_i.$$

Each ψ_i requires a subset construction in the style of the eventually operator construction to maintain the minimum. We then nest another subset construction in the style of the eventually operator construction to maintain the maximum over ψ_i. For a finite word $w \in \Sigma^*$, we use the notation $[\![\psi_i, w]\!]$ and $[\![\varphi_1 \mathbf{U}_\lambda^k \varphi_2, w]\!]$ for the value accumulated by reward machine corresponding to these formula on the word w, i.e. $[\![\psi_i, w]\!] = \min\{[\![\mathbf{X}_\lambda^i \varphi_2]\!], \min_{i>j\geq 0}\{[\![\mathbf{X}_\lambda^j \varphi_1, w]\!]\}$ and $[\![\varphi_1 \mathbf{U}_\lambda^k \varphi_2, w]\!] = \max_{k\geq i\geq 0} [\![\psi_i, w]\!]$.

Let $\mathcal{S} = 2^{(Q_1 \sqcup Q_2)\times\mathbb{R}}$ be the set of subsets containing (q, ζ) pairs, where q may be from either Q_1 or Q_2. The reward machine consists of states of the form $(v, I, \mathcal{X})$ where the value $v \in \mathbb{R}$ and the subset $I \in \mathcal{S}$ are for bookkeeping, and $\mathcal{X} \in 2^{\mathcal{S}}$ is a subset of subsets for each ψ_i. Specifically, each element of $\mathcal{X}$ is a subset S corresponding to a particular ψ_i which may still obtain the maximum, i.e. $[\![\psi_i, w]\!] + \lambda^n > [\![\varphi_1 \mathbf{U}_\lambda^n \varphi_2, w]\!]$. Each element of S is of the form (q, ζ). We have that $q \in Q_2$ for at most one element where $q = \delta_2(q_0^2, w_{k:\infty})$ and $\zeta = ([\![\mathbf{X}_\lambda^k \varphi_2, w]\!] - [\![\varphi_1 \mathbf{U}_\lambda^n \varphi_2, w]\!])/\lambda^n$. For the other elements of S, we have that $q \in Q_1$ with $q = \delta_1(q_0^1, w_{k:\infty})$ and $\zeta = ([\![\mathbf{X}_\lambda^k \varphi_1, w]\!] - [\![\varphi_1 \mathbf{U}_\lambda^n \varphi_2, w]\!])/\lambda^n$. If for any of these elements, the value of its corresponding formula becomes too large to be the minimum for the conjunction forming ψ_i, i.e. $[\![\psi_i, w]\!] + \lambda^n \leq [\![\varphi_1 \mathbf{U}_\lambda^n \varphi_2, w]\!] + \lambda^n \leq [\![\mathbf{X}_\lambda^k \varphi_t, w]\!]$ which occurs when $\zeta \geq 1$, that element is dropped from S. In order to update $\mathcal{X}$, we add a new S corresponding to ψ_n on the next timestep. The value $v = \max\{-1, [\![\varphi_1 \mathbf{U}_\lambda^n \varphi_2, w]\!]\}$ is a bookkeeping parameter for initializing new elements in the subsets and for stopping the addition of new elements when $v \leq -1$. The subset I is a bookkeeping parameter that keeps track of the subset construction for $\bigwedge_{n>i\geq 0} \mathbf{X}_\lambda^i \varphi_1$, which is used to initialize the addition of a subset corresponding to $\psi_n = \mathbf{X}_\lambda^n \varphi_2 \wedge (\bigwedge_{n>i\geq 0} \mathbf{X}_\lambda^i \varphi_1)$. We now define the reward machine formally.

We define a few functions that ease defining our transition function. We define $\delta_*(q, \sigma) = \delta_i(q, \sigma)$ and $f_*(\zeta, q, \sigma) = r_i(q, \sigma) + \zeta$ if $q \in Q_i$ for $i \in \{1, 2\}$. We also define $n(S, \sigma) = \min_{(q_i,\zeta_i)\in S} f_*(\zeta_i, q_i, \sigma)$ and $m(\mathcal{X}, \sigma) = \max_{S\in\mathcal{X}} n(S, \sigma)$. For the subset construction, we define

$$\Delta(S, \sigma, m) = \bigcup_{(q,\zeta)\in S} \{(\delta_*(q, \sigma), \zeta') : \zeta' < 1\}$$

where $\zeta' = (f_*(\zeta, q, \sigma) - m)/\lambda$ and

$$T(\mathcal{X}, \sigma, m) = \bigcup_{S\in\mathcal{X}} \{\Delta(S, \sigma, m) : n(S, \sigma) > -1\}.$$

We form a reward machine $\mathcal{R}_\varphi = (Q, \delta, r, q_0, \lambda)$ where $Q = \mathbb{R} \times \mathcal{S} \times 2^{\mathcal{S}}$ and $q_0 = (0, \emptyset, \{\{(q_0^2, 0)\}\})$. The transition function is

$$\delta((v, I, \mathcal{X}), \sigma) = \begin{cases} \Big(v',\ I',\ T(\mathcal{X}, \sigma, m) \sqcup \big(I' \sqcup (q_0^2, v')\big)\Big) & \text{if } v' > -1 \\ (-1,\ \emptyset,\ T(\mathcal{X}, \sigma, m)) & \text{if } v' \leq -1 \end{cases}$$

where $m = m(\mathcal{X}, \sigma)$, $v' = (v - m)/\lambda$, and $I' = \Delta(I \sqcup (q_0^1, v'), \sigma, m)$. The reward function is $r((v, I, \mathcal{X}), \sigma) = m(\mathcal{X}, \sigma)$.

We now show a sketch of correctness, which mimics the proof for the eventually operator closely.

Finiteness. Consider states of the form $(v, I, \mathcal{X}) \in Q$. If $v = 0$, then for all $S \in \mathcal{X}$ and $(q_i, \zeta_i) \in S$ it must be that $\zeta_i = 0$ since receiving a non-zero reward causes the value of v to become negative. Similarly, all $\zeta_i = 0$ for $(q_i, \zeta_i) \in I$ when $v = 0$. There are only finitely many such states. If $-1 < v < 0$, then we will reach a state $(v', I', \mathcal{X}') \in Q$ with $v' = -1$ in at most n steps, where n is such that $v/\lambda^n \leq -1$. Therefore, the number of reachable states $-1 < v < 0$ is also finite. Additionally, the number of states where $v = -1$ that can be initially reached is finite. Upon reaching such a state $(-1, \emptyset, \mathcal{X}')$, the reward machine is similar to that of the finite disjunction of reward machines for finite conjunctions.

Property I_1. The transition function is designed so that the following holds true: for any finite word $w \in \Sigma^*$ of length n and letter $\sigma \in \Sigma$, if $\delta(q_0, w) = (v, I, \mathcal{X})$, then $m(\mathcal{X}, \sigma) = (\llbracket \varphi_1 \mathbf{U}_\lambda^{n+1} \varphi_2, w\sigma \rrbracket - \llbracket \varphi_1 \mathbf{U}_\lambda^{n} \varphi_2, w \rrbracket)/\lambda^n$. Since $r((v, I, \mathcal{X}), \sigma) = m(\mathcal{X}, \sigma)$, we get that $\mathcal{R}_\varphi(w) = \llbracket \varphi_1 \mathbf{U}_\lambda^{n} \varphi_2, w \rrbracket$. Thus, $\mathcal{R}_\varphi(\rho) = \llbracket \varphi_1 \mathbf{U}_\lambda \varphi_2, \rho \rrbracket$ for any infinite word $\rho \in \Sigma^\omega$. This property for $m(\mathcal{X}, \sigma)$ follows from the properties outlined in the construction, which can be shown inductively.

Property I_2. Consider an SCC $\mathcal{C}$ of $\mathcal{R}_\varphi$ and a state $(v, I, \mathcal{X}) \in \mathcal{C}$. If $v = 0$, then we must receive zero reward because non-zero reward causes the value of v to become negative. It cannot be that $-1 < v < 0$ since if $v < 0$, we reach a state $(v', I', \mathcal{X}') \in Q$ with $v' = -1$ in at most n steps, where n is such that $v/\lambda^n \leq -1$. If $v = -1$, then we have a state of the form $(-1, \emptyset, \mathcal{X})$. For this to be an SCC, all elements of the form $(q_k, \zeta_k) \in S$ for $S \in \mathcal{X}$ must be such that q_k is in an SCC of its respective reward machine (either $\mathcal{R}_{\varphi_1}$ or $\mathcal{R}_{\varphi_2}$) with reward type $t_k \in \{0, 1 - \lambda\}$. Additionally, there cannot be a $t'_k \neq t_k$ otherwise there would be a ζ_k that changes following a cycle in the SCC $\mathcal{C}$. Thus, the reward for this SCC $\mathcal{C}$ is t_k.

Property I_3. This property can be shown by recalling the property above that $r((v, I, \mathcal{X}), \sigma) = m(\mathcal{X}, \sigma) = (\llbracket \varphi_1 \mathbf{U}_\lambda^{n+1} \varphi_2, w\sigma \rrbracket - \llbracket \varphi_1 \mathbf{U}_\lambda^{n} \varphi_2, w \rrbracket)/\lambda^n$.

5 Conclusion

This paper studied policy synthesis for discounted LTL in MDPs with unknown transition probabilities. Unlike LTL, discounted LTL provides an insensitivity to small perturbations of the transitions probabilities which enables PAC learning without additional assumptions. We outlined a PAC learning algorithm for discounted LTL that uses finite memory. We showed that optimal strategies for discounted LTL require infinite memory in general due to the need to balance the values of multiple competing objectives. To avoid this infinite memory, we examined the case of uniformly discounted LTL, where the discount factors for all temporal operators are identical. We showed how to translate uniformly discounted LTL formula to finite state reward machines. This construction shows that finite memory is sufficient, and provides an avenue to use discounted reward

algorithms, such as reinforcement learning, for computing optimal policies for uniformly discounted LTL formulas.

References

1. Aksaray, D., Jones, A., Kong, Z., Schwager, M., Belta, C.: Q-learning for robust satisfaction of signal temporal logic specifications. In: Conference on Decision and Control (CDC), pp. 6565–6570. IEEE (2016)
2. Almagor, S., Boker, U., Kupferman, O.: Discounting in LTL. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 424–439 (2014)
3. Alur, R., Bansal, S., Bastani, O., Jothimurugan, K.: A Framework for transforming specifications in reinforcement learning. In: Raskin, J.F., Chatterjee, K., Doyen, L., Majumdar, R. (eds.) Principles of Systems Design. LNCS, vol. 13660, pp. 604–624. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-22337-2_29
4. Alur, R., Bastani, O., Jothimurugan, K., Perez, M., Somenzi, F., Trivedi, A.: Policy synthesis and reinforcement learning for discounted LTL. arXiv preprint arXiv:2305.17115 (2023)
5. Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., Mané, D.: Concrete problems in AI safety. arXiv preprint arXiv:1606.06565 (2016)
6. Ashok, P., et al.: PAC statistical model checking for Markov decision processes and stochastic games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 497–519. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_29
7. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
8. Bozkurt, A.K., Wang, Y., Zavlanos, M.M., Pajic, M.: Control synthesis from linear temporal logic specifications using model-free reinforcement learning. In: 2020 IEEE International Conference on Robotics and Automation (ICRA), pp. 10349–10355. IEEE (2020)
9. Brafman, R., De Giacomo, G., Patrizi, F.: LTLf/LDLf non-Markovian rewards. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32 (2018)
10. Camacho, A., Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: LTL and beyond: formal languages for reward function specification in reinforcement learning. In: IJCAI, vol. 19, pp. 6065–6073 (2019)
11. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
12. Daca, P., Henzinger, T.A., Kretinsky, J., Petrov, T.: Faster statistical model checking for unbounded temporal properties. ACM Trans. Comput. Logic (TOCL) **18**(2), 1–25 (2017)
13. De Alfaro, L.: Formal Verification of Probabilistic Systems. Stanford University (1998)
14. de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: Model checking discounted temporal properties. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 77–92. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_6
15. de Alfaro, L., Henzinger, T.A., Majumdar, R.: Discounting the future in systems theory. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1022–1037. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_79

16. De Giacomo, G., Iocchi, L., Favorito, M., Patrizi, F.: Foundations for restraining bolts: reinforcement learning with LTLf/LDLf restraining specifications. In: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 29, pp. 128–136 (2019)
17. Fu, J., Topcu, U.: Probably approximately correct MDP learning and control with temporal logic constraints. arXiv preprint arXiv:1404.7073 (2014)
18. Hahn, E.M., Li, G., Schewe, S., Turrini, A., Zhang, L.: Lazy probabilistic model checking without determinisation. arXiv preprint arXiv:1311.2928 (2013)
19. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 395–412. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_27
20. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Good-for-MDPs automata for probabilistic analysis and reinforcement learning. In: TACAS 2020. LNCS, vol. 12078, pp. 306–323. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_17
21. Hasanbeig, M., Kantaros, Y., Abate, A., Kroening, D., Pappas, G.J., Lee, I.: Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In: Conference on Decision and Control (CDC), pp. 5338–5343 (2019)
22. Hasanbeig, M., Abate, A., Kroening, D.: Logically-constrained reinforcement learning. arXiv preprint arXiv:1801.08099 (2018)
23. Hasanbeig, M., Kantaros, Y., Abate, A., Kroening, D., Pappas, G.J., Lee, I.: Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In: 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 5338–5343. IEEE (2019)
24. Icarte, R.T., Klassen, T., Valenzano, R., McIlraith, S.: Using reward machines for high-level task specification and decomposition in reinforcement learning. In: International Conference on Machine Learning, pp. 2107–2116. PMLR (2018)
25. Jiang, Y., Bharadwaj, S., Wu, B., Shah, R., Topcu, U., Stone, P.: Temporal-logic-based reward shaping for continuing learning tasks (2020)
26. Jothimurugan, K., Alur, R., Bastani, O.: A composable specification language for reinforcement learning tasks. In: Advances in Neural Information Processing Systems, vol. 32, pp. 13041–13051 (2019)
27. Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Compositional reinforcement learning from logical specifications. In: Advances in Neural Information Processing Systems (2021)
28. Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Specification-guided learning of Nash equilibria with high social welfare. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, CAV 2022. LNCS, vol. 13372. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_17
29. Kakade, S.M.: On the sample complexity of reinforcement learning. University of London, University College London (United Kingdom) (2003)
30. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Perform. Eval. Rev. **36**(4), 40–45 (2009)
31. Li, X., Vasile, C.I., Belta, C.: Reinforcement learning with temporal logic rewards. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3834–3839. IEEE (2017)

32. Littman, M.L., Topcu, U., Fu, J., Isbell, C., Wen, M., MacGlashan, J.: Environment-independent task specifications via GLTL. arXiv preprint arXiv:1704.04341 (2017)
33. Mandrali, E.: Weighted LTL with discounting. In: Moreira, N., Reis, R. (eds.) CIAA 2012. LNCS, vol. 7381, pp. 353–360. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31606-7_32
34. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley (2014)
35. Sadigh, D., Kim, E.S., Coogan, S., Sastry, S.S., Seshia, S.A.: A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications. In: 53rd IEEE Conference on Decision and Control, pp. 1091–1096. IEEE (2014)
36. Sickert, S., et al.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 312–332. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_17
37. Sickert, S., et al.: MoChiBA: probabilistic LTL model checking using limit-deterministic Büchi automata. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 130–137. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_9
38. Strehl, A.L., Li, L., Wiewiora, E., Langford, J., Littman, M.L.: PAC model-free reinforcement learning. In: Proceedings of the 23rd International Conference on Machine Learning, pp. 881–888 (2006)
39. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. MIT Press (2018)
40. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite state programs. In: 26th Annual Symposium on Foundations of Computer Science, SFCS 1985, pp. 327–338. IEEE (1985)
41. Wells, A.M., Lahijanian, M., Kavraki, L.E., Vardi, M.Y.: LTLf synthesis on probabilistic systems. arXiv preprint arXiv:2009.10883 (2020)
42. Xu, Z., Topcu, U.: Transfer of temporal logic formulas in reinforcement learning. In: International Joint Conference on Artificial Intelligence, pp. 4010–4018 (7 2019)
43. Yang, C., Littman, M., Carbin, M.: Reinforcement learning for general LTL objectives is intractable. arXiv preprint arXiv:2111.12679 (2021)
44. Yuan, L.Z., Hasanbeig, M., Abate, A., Kroening, D.: Modular deep reinforcement learning with temporal logic specifications. arXiv preprint arXiv:1909.11591 (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Synthesizing Permissive Winning Strategy Templates for Parity Games

Ashwani Anand, Satya Prakash Nayak(✉), and Anne-Kathrin Schmuck

Max Planck Institute for Software Systems, Kaiserslautern, Germany
{ashwani,sanayak,akschmuck}@mpi-sws.org

Abstract. We present a novel method to compute *permissive winning strategies* in two-player games over finite graphs with ω-regular winning conditions. Given a game graph G and a parity winning condition Φ, we compute a *winning strategy template* Ψ that collects an infinite number of winning strategies for objective Φ in a concise data structure. We use this new representation of sets of winning strategies to tackle two problems arising from applications of two-player games in the context of cyber-physical system design – (i) *incremental synthesis*, i.e., adapting strategies to newly arriving, *additional* ω-regular objectives Φ', and (ii) *fault-tolerant control*, i.e., adapting strategies to the occasional or persistent unavailability of actuators. The main features of our strategy templates – which we utilize for solving these challenges – are their easy computability, adaptability, and compositionality. For *incremental synthesis*, we empirically show on a large set of benchmarks that our technique vastly outperforms existing approaches if the number of added specifications increases. While our method is not complete, our prototype implementation returns the full winning region in all 1400 benchmark instances, i.e. handling a large problem class efficiently in practice.

1 Introduction

Two-player ω-regular games on finite graphs are an established modeling and solution formalism for many challenging problems in the context of correct-by-construction cyber-physical system (CPS) design [2,7,39]. Here, control software actuating a technical system "plays" against the physical environment. The winning strategy of the system player in this two-player game results in software which ensures that the controlled technical system fulfills a given temporal specification for any (possible) event or input sequence generated by the environment. Examples include warehouse robot coordination [36], reconfigurable manufacturing systems [26], and adaptive cruise control [33]. In these applications, the

S. P. Nayak and A.-K. Schmuck are supported by the DFG project 389792660 TRR 248–CPEC.

A. Anand and A.-K. Schmuck are supported by the DFG project SCHM 3541/1-1.

© The Author(s) 2023

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 436–458, 2023.
https://doi.org/10.1007/978-3-031-37706-8_22

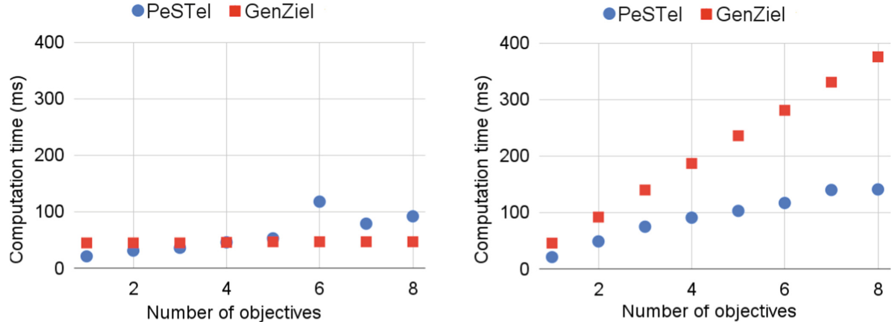


Fig. 1. Experimental results over 1400 generalized parity games comparing the performance of our tool PeSTel against the state-of-the-art generalized parity solver GenZiel [16]. Data points give the average execution time (in ms) over all instances with the same number of parity objectives. Left: all objectives are given *upfront*. Right: objectives are added *one-by-one*. See Sect. 6 for more details on those experiments.

technical system under control, as well as its requirements, are developing and changing during the design process. It is therefore desirable to allow for maintainable and adaptable control software. This, in turn, requires solution algorithms for two-player ω-regular games which allow for this adaptability.

This paper addresses this challenge by providing a new algorithm to efficiently compute *permissive winning strategy templates* in parity games which enable rich *strategy adaptations*. Given a game graph $G = (V, E)$ and an objective Φ a winning strategy template Ψ characterizes the winning region $\mathcal{W} \subseteq V$ along with three types of local edge conditions – a *safety*, a *co-live*, and a *live-group* template. The conjunction of these basic templates allows us to capture infinitely many winning strategies over G w.r.t. Φ in a simple data structure that is both (i) easy to obtain during synthesis, and (ii) easy to adapt and compose.

We showcase the usefulness of *permissive winning strategy templates* in the context of CPS design by two application scenarios: (i) *incremental synthesis*, where strategies need to be adapted to newly arriving *additional* ω-regular objectives Φ', and (ii) *fault-tolerant control*, where strategies need to be adapted to the occasional or persistent unavailability of actuators, i.e., system player edges.

We have implemented our algorithms in a prototype tool PeSTel and run it on more than 1400 benchmarks adapted from the SYNTCOMP benchmark suite [21]. These experiments show that our class of templates effectively avoids re-computations for the required strategy adaptations. For *incremental synthesis*, our experimental results are previewed in Fig. 1, where we compare PeSTel against the state-of-the-art solver GenZiel [16] for generalized parity objectives, i.e., finite conjunction of parity objectives. We see that PeSTel is as efficient as GenZiel whenever all conjuncts of the objective are given *up-front* (Fig. 1(left)) - even outperforming it in more than 90% of the instances. Whenever conjuncts of the objective arrive *one at a time*, PeSTel outperforms the existing approaches significantly if the number of objectives increases (Fig. 1(right)). This shows the potential of PeSTel towards more adaptable and maintainable control software for CPS.

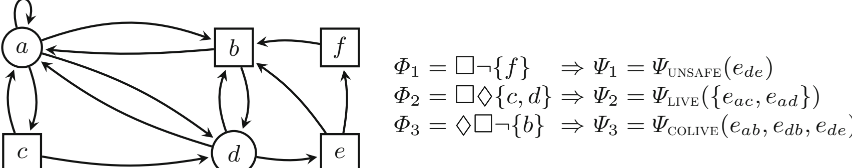


Fig. 2. A two-player game graph with Player 1 (squares) and Player 0 (circles) vertices, different winning conditions Φ_i, and corresponding winning strategy templates Ψ_i.

Illustrative Example. To appreciate the simplicity and easy adaptability of our strategy templates, consider the game graph in Fig. 2(left). The first winning condition Φ_1 requires vertex f to never be seen along a play. This can be enforced by Player 0 from vertices $\mathcal{W}_0 = \{a, b, c, d\}$ called the *winning region*. The safety template Ψ_1 ensures that the game always stays in $\mathcal{W}_0$ by forcing the edge e_{de} to never be taken. It is easy to see that every Player 0 strategy that follows this rule results in plays which are winning if they start in $\mathcal{W}_0$. Now consider the second winning condition Φ_2 which requires vertex c or d to be seen infinitely often. This induces the live-group template Ψ_2 which requires that whenever vertex a is seen infinitely often, either edge e_{ac} or edge e_{ad} needs to be taken infinitely often. It is easy to see that any strategy that complies with this edge-condition is winning for Player 0 from every vertex and there are infinitely many such compliant winning strategies. Finally, we consider condition Φ_3 requiring vertex b to be seen only finitely often. This induces the strategy template Ψ_3 which is a co-liveness template requiring that all edges from Player 0 vertices which unavoidably lead to b (i.e., e_{ab}, e_{bd}, and e_{de}) are taken only finitely often. We can now combine all templates into a new template $\Psi' = \Psi_1 \wedge \Psi_2 \wedge \Psi_3$ and observe that all strategies compliant with Ψ' are winning for $\Phi' = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$.

In addition to their compositionality, strategy templates also allow for local strategy adaptations in case of edge unavailability faults. Consider again the game in Fig. 2 with the objective Φ_2. Suppose that Player 0 follows the strategy π: $a \mapsto d$ and $d \mapsto a$, which is compliant with Ψ_2. If the edge e_{ad} becomes unavailable, we would need to re-solve the game for the modified game graph $G' = (V, E \setminus \{e_{ad}\})$. However, given the strategy template Ψ_2 we see that the strategy π': $a \mapsto c$ and $d \mapsto a$ is actually compliant with Ψ_2 over G'. This allows us to obtain a new strategy without re-solving the game.

While these examples demonstrate the potential of templates for strategy adaptation, there exist scenarios where conflicts between templates or graph modifications arise, which require re-computations. Our empirical results, however, show that such conflicts rarely appear in practical benchmarks. This suggests that our technique can handle a large problem class efficiently in practice.

Related Work. The class of templates we use was introduced in [4] and utilized to represent environment assumptions that enable a system to fulfill its specifications in a cooperative setting. Contrary to [4], this paper uses the same class of templates to represent the system's winning strategies in a zero-sum setting.

While the computation of *permissive strategies* for the control of CPS is an established concept in the field of supervisory control[1] [14,42], it has also been addressed in reactive synthesis where the considered specification class is typically more expressive, e.g., Bernet et al. [8] introduce permissive strategies that encompass all the behaviors of positional strategies and Neider et al. [31] introduce permissiveness to subsume strategies that visit losing loops at most twice. Finally, Bouyer et al. [11] take a quantitative approach to measure the permissiveness of strategies, by minimizing the penalty of not being permissive. However, all these approaches are not optimized towards strategy adaptation and thereby typically fail to preserve enough behaviors to be able to effectively satisfy subsequent objectives. A notable exception is a work by Baier et al. [23]. While their strategy templates are more complicated and more costly to compute than ours, they are *maximally* permissive (i.e., capture *all* winning strategies in the game). However, when composing multiple objectives, they restrict templates substantially which eliminates many compositional solutions that our method retains. This results in higher computation times and lower result quality for incremental synthesis compared to our approach. As no implementation of their method is available, we could not compare both approaches empirically.

Even without the incremental aspect, synthesizing winning strategies for conjunctions of ω-regular objectives is known to be a hard problem – Chatterjee et al. [16] prove that the conjunction of even *two* parity objectives makes the problem NP-complete. They provide a generalization of Zielonka's algorithm, called GenZiel for generalized parity objectives (i.e., finite conjunction of parity objectives) which is compared to our tool PeSTel in Fig. 1. While PeSTel is (in contrast to GenZiel) not complete—i.e., there exist realizable synthesis problems for which PeSTel returns no solution—our prototype implementation returns the full winning region in all 1400 benchmark instances.

Fault-tolerant control is a well-established topic in control engineering [9], with recent emphasis on the logical control layer [19,30]. While most of this work is conducted in the context of supervisory control, there are also some approaches in reactive synthesis. While [29,32] considers the *addition* of "disturbance edges" and synthesizes a strategy that tolerates as many of them as possible, we look at the complementary problem, where edges, in particular system-player edges, disappear. To the best of our knowledge, the only algorithm that is able to tackle this problem without re-computation considers Büchi games [15]. In contrast, our method is applicable to the more expressive class of Parity games.

2 Preliminaries

Notation. We use $\mathbb{N}$ to denote the set of natural numbers including zero. Given two natural numbers $a, b \in \mathbb{N}$ with $a < b$, we use $[a;b]$ to denote the set $\{n \in \mathbb{N} \mid a \leq n \leq b\}$. For any given set $[a;b]$, we write $i \in_{\text{even}} [a;b]$ and $i \in_{\text{odd}} [a;b]$ as shorthand for $i \in [a;b] \cap \{0, 2, 4, \ldots\}$ and $i \in [a;b] \cap \{1, 3, 5, \ldots\}$

[1] See [18,28,37] for connections between supervisory control and reactive synthesis.

respectively. Given two sets A and B, a relation $R \subseteq A \times B$, and an element $a \in A$, we write $R(a)$ to denote the set $\{b \in B \mid (a, b) \in R\}$.

Languages. Let Σ be a finite alphabet. The notation Σ^* and Σ^ω respectively denote the set of finite and infinite words over Σ, and Σ^∞ is equal to $\Sigma^* \cup \Sigma^\omega$. For any word $w \in \Sigma^\infty$, w_i denotes the i-th symbol in w. Given two words $u \in \Sigma^*$ and $v \in \Sigma^\infty$, the concatenation of u and v is written as the word uv.

Game Graphs. A *game graph* is a tuple $G = \big(V = V^0 \uplus V^1, E\big)$ where (V, E) is a finite directed graph with *vertices* V and *edges* E, and $V^0, V^1 \subseteq V$ form a partition of V. Without loss of generality, we assume that for every $v \in V$ there exists $v' \in V$ s.t. $(v, v') \in E$. A *play* originating at a vertex v_0 is a finite or infinite sequence of vertices $\rho = v_0 v_1 \ldots \in V^\infty$.

Winning Conditions/Objectives. Given a game graph G, we consider winning conditions/objectives specified using a formula Φ in *linear temporal logic* (LTL) over the vertex set V, that is, we consider LTL formulas whose atomic propositions are sets of vertices V. In this case the set of desired infinite plays is given by the semantics of Φ which is an ω-regular language $\mathcal{L}(\Phi) \subseteq V^\omega$. Every game graph with an arbitrary ω-regular set of desired infinite plays can be reduced to a game graph (possibly with a different set of vertices) with an LTL winning condition, as above. The standard definitions of ω-regular languages and LTL are omitted for brevity and can be found in standard textbooks [6]. To simplify notation we use $e = (u, v)$ in LTL formulas as syntactic sugar for $u \wedge \bigcirc v$, with $\bigcirc$ as the LTL *next* operator. We further use a set of edges $E' = \{e_i\}_{i \in [0;k]}$ as atomic proposition to denote $\bigvee_{i \in [0;k]} e_i$.

Games and Strategies. A *two-player (turn-based) game* is a pair $\mathcal{G} = (G, \Phi)$ where G is a game graph and Φ is a *winning condition* over G. A strategy of Player i, $i \in \{0, 1\}$, is a function $\pi^i \colon V^* V^i \to V$ such that for every $\rho v \in V^* V^i$ holds that $\pi^i(\rho v) \in E(v)$. Given a strategy π^i, we say that the play $\rho = v_0 v_1 \ldots$ is *compliant* with π^i if $v_{k-1} \in V^i$ implies $v_k = \pi^i(v_0 \ldots v_{k-1})$ for all k. We refer to a play compliant with π^i and a play compliant with both π^0 and π^1 as a π^i*-play* and a $\pi^0\pi^1$*-play*, respectively. We collect all plays originating in a set S and compliant with π^i, (and compliant with both π^0 and π^1) in the sets $\mathcal{L}(S, \pi^i)$ (and $\mathcal{L}(S, \pi^0\pi^1)$, respectively). When $S = V$, we drop the mention of the set in the previous notation, and when S is singleton $\{v\}$, we simply write $\mathcal{L}(v, \pi^i)$ (and $\mathcal{L}(v, \pi^0\pi^1)$, respectively).

Winning. Given a game $\mathcal{G} = (G, \Phi)$, a play ρ in $\mathcal{G}$ is *winning for* Player 0, if $\rho \in \mathcal{L}(\Phi)$, and it is winning for Player 1, otherwise. A strategy π^i for Player i is *winning from a vertex* $v \in V$ if all plays compliant with π^i and originating from v are winning for Player i. We say that a vertex $v \in V$ is *winning for* Player i, if there exists a winning strategy π^i from v. We collect all winning vertices of Player i in the Player i *winning region* $\mathcal{W}_i \subseteq V$. We always interpret winning w.r.t. Player 0 if not stated otherwise.

Strategy Templates. Let π^0 be a Player 0 strategy and Φ be an LTL formula. Then we say π^0 *follows* Φ, denoted $\pi^0 \Vdash \Phi$, if for all π^0-plays ρ, ρ belongs to

$\mathcal{L}(\Phi)$, i.e. $\mathcal{L}(\pi^0) \subseteq \mathcal{L}(\Phi)$. We refer to a set $\Psi = \{\Psi_1, \dots, \Psi_k\}$ of LTL formulas as *strategy templates* representing the set of strategies that follows $\Psi_1 \wedge \dots \wedge \Psi_k$. We say a strategy template Ψ is *winning from a vertex* v for a game (G, Φ) if every Player 0 strategy following the template Ψ is winning from v. Moreover, we say a strategy template Ψ is *winning* if it is winning from every vertex in $\mathcal{W}_0$. In addition, we call Ψ *maximally permissive* for $\mathcal{G}$, if every Player 0 strategy π which is winning in $\mathcal{G}$ also follows Ψ. With slight abuse of notation, we use Ψ for the set of formulas $\{\Psi_1, \dots, \Psi_k\}$, and the formula $\Psi_1 \wedge \dots \wedge \Psi_k$, interchangeably.

Set Transformers. Let $G = (V = V^0 \uplus V^1, E)$ be a game graph, $U \subseteq V$ be a subset of vertices, and $a \in \{0, 1\}$ be the player index. Then

$$\mathsf{upre}_G(U) = \{v \in V \mid \forall (v, u) \in E.\ u \in U\} \tag{1}$$

$$\mathsf{cpre}^a_G(U) = \{v \in V^a \mid \exists (v, u) \in E.\ u \in U\} \cup \{v \in V^{1-a} \mid u \in \mathsf{upre}_G(U)\} \tag{2}$$

The universal predecessor operator $\mathsf{upre}_G(U)$ computes the set of vertices with all the successors in U and the controllable predecessor operator $\mathsf{cpre}^a_G(U)$ the vertices from which Player a can force visiting U in *exactly one* step. In the following, we introduce two types of attractor operators: $\mathsf{attr}^a_G(U)$ that computes the set of vertices from which Player a can force at least a single visit to U in *finitely many* steps, and the universal attractor $\mathsf{uattr}_G(U)$ that computes the set of vertices from which both players are forced to visit U. For the following, let $\mathsf{pre} \in \{\mathsf{upre}, \mathsf{cpre}^a\}$

$$\mathsf{pre}^1_G(U) = \mathsf{pre}_G(U) \cup U \qquad \mathsf{pre}^i_G(U) = \mathsf{pre}_G(\mathsf{pre}^{i-1}_G(U)) \cup \mathsf{pre}^{i-1}_G(U) \tag{3}$$

$$\mathsf{attr}^a_G(U) = \cup_{i \geq 1}\, \mathsf{cpre}^{a,i}_G(U) \qquad \mathsf{uattr}_G(U) = \cup_{i \geq 1}\, \mathsf{upre}^i_G(U) \tag{4}$$

3 Computation of Winning Strategy Templates

Given a 2-player game $\mathcal{G}$ with an objective Φ, the goal of this section is to compute a *strategy template* that characterizes a large class of winning strategies of Player 0 from a set of vertices $U \subseteq V$ in a local, permissive, and computationally efficient way. These templates are then utilized in Sect. 5.1 for computational synthesis. In particular, this section introduces three distinct template classes—safety templates (Sect. 3.1), live-group-templates (Sect. 3.2), and co-live-templates (Sect. 3.3) along with algorithms for their computation via safety, Büchi, and co-Büchi games, respectively. We then turn to general parity objectives which can be thought of as a sophisticated combination of Büchi and co-Büchi games. We show in Sect. 3.4 how the three introduced templates can be derived for a general parity objective by a suitable combination of the previously introduced algorithms for single templates. All presented algorithms have the same worst-case computation time as the standard algorithms solving the respective game. This shows that extracting strategy *templates* instead of 'normal' strategies does not incur an additional computational cost. We prove the soundness of the algorithms and discuss the complexities in the full version [5, Appendix A].

3.1 Safety Templates

We start the construction of strategy templates by restricting ourselves to games with a safety objective—i.e., $\mathcal{G} = (G, \Phi)$ with $\Phi := \Box U$ for some $U \subseteq V$. A winning play in a safety game never leaves $U \subseteq V$. It is well known that such games allow capturing *all* winning strategies by a simple local template which essentially only allows Player 0 moves from winning vertices to other winning vertices. This is formalized in our notation as a safety template as follows,

Theorem 1 ([8, Fact 7]). *Let $\mathcal{G} = (G, \Box U)$ be a safety game with winning region $\mathcal{W}_0$ and $S = \{(u, v) \in E \mid (u \in V^0 \cap \mathcal{W}_0) \wedge (v \notin \mathcal{W}_0)\}$. Then*

$$\Psi_{\text{UNSAFE}}(S) := \Box \bigwedge_{e \in S} \neg e, \tag{5}$$

is a winning strategy template for the game $\mathcal{G}$ which is also maximally permissive.

It is easy to see that the computation of the safety template $\Psi_{\text{UNSAFE}}(S)$ reduces to computing the winning region $\mathcal{W}_0$ in the safety game $(G, \Box U)$ and extracting S. We refer to the edges in S as *unsafe edges* and we call this algorithm computing the set S as SafetyTemplate(G, U). Note that it runs in $\mathcal{O}(m)$ time, where $m = |E|$, as safety games are solvable in $\mathcal{O}(m)$ time.

3.2 Live-Group Templates

As the next step, we now move to simple liveness objectives which require a particular vertex set $I \subseteq V$ to be seen infinitely often. Here, winning strategies need to stay in the winning region (as before) but in addition always eventually need to make progress towards the vertex set I. We capture this required progress by *live-group templates*—given a group of edges $H \subseteq E$, we require that whenever a source vertex v of an edge in H is seen infinitely often, an edge $e \in H$ (not necessarily starting at v) also needs to be taken infinitely often. This template ensures that compliant strategies always eventually make progress towards I, as illustrated by the following example.

Example 1. Consider the game graph in Fig. 2 where we require visiting $\{c, d\}$ infinitely often. To satisfy this objective from vertex a, Player 0 needs to not get stuck at a, and should not visit b always (since Player 1 can force visiting a again, and stop Player 0 from satisfying the objective). Hence, Player 0 has to always eventually leave a and go to $\{c, d\}$. This can be captured by the live-group $\{e_{ac}, e_{ad}\}$. Now if the play comes to a infinitely often, Player 0 will go to either c or d infinitely often, hence satisfying the objective.

Formally, such games are called *Büchi games*, denoted by $\mathcal{G} = (G = (V, E), \Phi)$ with $\Phi := \Box\Diamond I$, for some $I \subseteq V$. In addition, a *live-group* $H = \{e_j\}_{j \geq 0}$ is a set of edges $e_j = (s_j, t_j)$ with source vertices $src(H) := \{s_j\}_{j \geq 0}$. Given a set of live-groups $\mathcal{H} = \{H_i\}_{i \geq 0}$ we define a live-group template as

$$\Psi_{\text{LIVE}}(\mathcal{H}) := \bigwedge_{i \geq 0} \Box\Diamond src(H_i) \implies \Box\Diamond H_i. \tag{6}$$

Algorithm 1. BüchiTemplate(G, I)

Input: A game graph G, and a subset of vertices I
Output: A set of unsafe edges S and a set of live-groups $\mathcal{H}$
1: $\mathcal{W}_0 \leftarrow$ Büchi(G, I); $S \leftarrow$ SafetyTemplate$(G, \mathcal{W}_0)$;
2: $G \leftarrow G|_{\mathcal{W}_0}$; $I \leftarrow I \cap \mathcal{W}_0$;
3: $\mathcal{H} \leftarrow$ ReachTemplate(G, I);
4: **return** $(S, \mathcal{H})$
5: **procedure** ReachTemplate$(G, I \subseteq V)$
6: $\quad\mathcal{H} \leftarrow \emptyset$;
7: $\quad$**while** $I \neq V$ **do**
8: $\quad\quad A \leftarrow \mathsf{uattr}_G(I)$; $B \leftarrow \mathsf{cpre}^0_G(A)$; $\mathcal{H} \leftarrow \mathcal{H} \cup \{\text{Edges}(B, A)\}$; $I \leftarrow A \cup B$;
9: $\quad$**return** $\mathcal{H}$

The live-group template says that if some vertex from the source of a live-group is visited infinitely often, then some edge from this group should be taken infinitely often by the following strategy.

Intuitively, winning strategy templates for Büchi games consist of a safety template conjuncted with a live-group template. While the former enforces all strategies to stay within the winning region $\mathcal{W}$, the latter enforces progress w.r.t. the goal set I within $\mathcal{W}$. Therefore, the computation of a winning strategy template for Büchi games reduces to the computation of the unsafe set S to define $\Psi_{\text{UNSAFE}}(S)$ in (5) and the live-group $\mathcal{H}$ to define $\Psi_{\text{LIVE}}(\mathcal{H})$ in (6). We denote by BüchiTemplate(G, I) the algorithm computing the above as detailed in Algorithm 1. The algorithm uses some new notations that we define here. Here, the function Büchi solves a Büchi game and returns the winning region (e.g., using the standard algorithm from [17]), Edges$(X, Y) = \{(u, v) \in E \mid u \in X, v \in Y\}$, is the set of edges between two subsets of vertices X and Y. $G|_U \coloneqq \big(U = U^0 \cup U^1, E'\big)$ s.t. $U^0 \coloneqq V^0 \cap U$, $U^1 \coloneqq V^1 \cap U$, and $E' \coloneqq E \cap (U \times U)$ denotes the restriction of a game graph $G \coloneqq \big(V = V^0 \cup V^1, E\big)$ to a subset of its vertices $U \subseteq V$. We have the following formal result.

Theorem 2. *Given a Büchi game $\mathcal{G} = (G, \Box\Diamond I)$ for some $I \subseteq V$, if $(S, \mathcal{H}) =$ BüchiTemplate(G, I) then $\Psi = \{\Psi_{\text{UNSAFE}}(S), \Psi_{\text{LIVE}}(\mathcal{H})\}$ is a winning strategy template for the game $\mathcal{G}$, computable in time $\mathcal{O}(nm)$, where $n = |V|$ and $m = |E|$.*

While live-group templates capture infinitely many winning strategies in Büchi games, they are *not* maximally permissive, as exemplified next.

Example 2. Consider the game graph in Fig. 2 restricted to the vertex set $\{a, b, d\}$ with the Büchi objective $\Box\Diamond d$. Our algorithm outputs the live-group template $\Psi = \Psi_{\text{LIVE}}(\{e_{ad}\})$. Now consider the winning strategy with memory that takes edge e_{da} from d, and takes e_{ab} for play suffix bda and e_{ad} for play suffix aba. This strategy does not follow the template—the play $(abd)^\omega$ is in $\mathcal{L}(\pi^0)$ but not in $\mathcal{L}(\Psi)$.

3.3 Co-live Templates

We now turn to yet another objective which is the dual of the one discussed before. The objective requires that eventually, only a particular subset of vertices I is seen. A winning strategy for this objective would try to restrict staying or going away from I after a finite amount of time. It is easy to notice that live-group templates can not ensure this, but it can be captured by *co-live templates*: given a set of edges, eventually these edges are not taken anymore. Intuitively, these are the edges that take or keep a play away from I.

Example 3. Consider the game graph in Fig. 2 where we require eventually stop visiting b, i.e. staying in $I = \{a, c, d\}$. To satisfy this objective from vertex a, Player 0 needs to stop getting out of I eventually. Hence, Player 0 has to stop taking the edges $\{e_{ab}, e_{db}, e_{de}\}$, which can be ensured by marking both edges co-live. Now since no edges are leading to b, the play eventually stays in I, satisfying the objective. We note that this can not be captured by live-groups $\{e_{aa}, e_{ac}, e_{ad}\}$ and $\{e_{da}\}$, since now the strategy that visits c and b alternatively from Player 0's vertices, does not satisfy the objective, but follows the live-group.

Formally, a co-Büchi game is a game $\mathcal{G} = (G, \Phi)$ with co-Büchi winning condition $\Phi := \Diamond\Box I$, for some goal vertices $I \subseteq V$. A play is winning for Player 0 in such a co-Büchi game if it eventually stays in I forever. The *co-live* template is defined by a set of *co-live* edges D as follows,

$$\Psi_{\text{COLIVE}}(D) := \bigwedge_{e \in D} \Diamond\Box\neg e.$$

The intuition behind the winning template is that it forces staying in the winning region using the safety template, and ensures that the play does not go away from the vertex set I infinitely often using the co-live template. We provide the procedure in Algorithm 2 and its correctness in the following theorem. Here, CoBüchi(G, I) is a standard algorithm solving the co-Büchi game with the goal vertices I, and outputs the winning regions for both players [17]. We also use the standard algorithm Safety(G, I) that solves the safety game with the objective to stay in A forever.

Theorem 3. *Given a co-Büchi game $\mathcal{G} = (G, \Diamond\Box I)$ for some $I \subseteq V$, if $(S, D) =$ coBüchiTemplate(G, I) then $\Psi = \{\Psi_{\text{UNSAFE}}(S), \Psi_{\text{COLIVE}}(D)\}$ is a winning strategy template for Player 0, computable in time $\mathcal{O}(nm)$ with $n = |V|$ and $m = |E|$.*

3.4 Parity Games

We now consider a more complex but canonical class of ω-regular objectives. Parity objectives are of central importance in the study of synthesis problems as they are general enough to model a huge class of qualitative requirements of cyber-physical systems, while enjoying the properties like positional determinacy.

Algorithm 2. coBüchiTemplate(G, I)

Input: A game graph G, and a subset of vertices I
Output: A set of unsafe edges S and a set of co-live edges D
1: $S \leftarrow \emptyset$; $D \leftarrow \emptyset$
2: $\mathcal{W}_0 \leftarrow$ CoBüchi(G, I); $S \leftarrow$ SafetyTemplate$(G, \mathcal{W}_0)$
3: $G \leftarrow G|_{\mathcal{W}_0}$; $I \leftarrow I \cap \mathcal{W}_0$;
4: **while** $V \neq \emptyset$ **do**
5: $\quad A \leftarrow$ Safety(G, I); $D \leftarrow D \cup$ Edges$(A, V \setminus A)$;
6: $\quad$ **while** $\mathsf{cpre}^0_G(A) \neq A$ **do** $\quad\quad$ ▷ Outputs $\mathsf{attr}^0_G(A)$
7: $\quad\quad B \leftarrow \mathsf{cpre}^0_G(A)$;
8: $\quad\quad D \leftarrow D \cup$ Edges$(B, V \setminus (A \cup B)) \cup$ Edges(B, B);
9: $\quad\quad A \leftarrow A \cup B$;
10: $\quad G \leftarrow G|_{V \setminus A}$; $I \leftarrow I \cap V \setminus A$;
11: **return** (S, D)

A parity game is a game $\mathcal{G} = (G, \Phi)$ with parity winning condition $\Phi = \mathit{Parity}(\mathbb{P})$, where

$$\mathit{Parity}(\mathbb{P}) := \bigwedge\nolimits_{i \in \mathrm{odd}[0;k]} \left(\Box\Diamond P_i \implies \bigvee\nolimits_{j \in \mathrm{even}[i+1;k]} \Box\Diamond P_j \right), \tag{7}$$

with $P_i = \{q \in Q \mid \mathbb{P}(q) = i\}$ for some priority function $\mathbb{P} : V \to [0; d]$ that assigns each vertex a priority. A play is winning for Player 0 in such a game if the maximum of priorities seen infinitely often is even.

Although parity objectives subsume previously described objectives, we can construct strategy templates for parity games using the combinations of previously defined templates. To this end, we give the following algorithm.

Theorem 4. *Given a parity game $\mathcal{G} = (G, \mathit{Parity}(\mathbb{P}))$ with priority function $\mathbb{P} : V \to [0; d]$, if $((\mathcal{W}_0, \mathcal{W}_1), \mathcal{H}, D) =$ ParityTemplate$(G, \mathbb{P})$, then $\Psi = \{\Psi_{\mathrm{UNSAFE}}(S), \Psi_{\mathrm{LIVE}}(\mathcal{H}), \Psi_{\mathrm{COLIVE}}(D)\}$ is a winning strategy template for the game $\mathcal{G}$, where $S =$ Edges$(\mathcal{W}_0, \mathcal{W}_1)$. Moreover, the algorithm terminates in time $\mathcal{O}(n^{d+\mathcal{O}(1)})$, which is same as that of Zielonka's algorithm.*

We refer the readers to the full version [5, Appendix A.3] for the complete proofs, and here we provide the intuition behind the algorithm and the computation of the algorithm on the parity game in Fig. 3. The algorithm follows the divide-

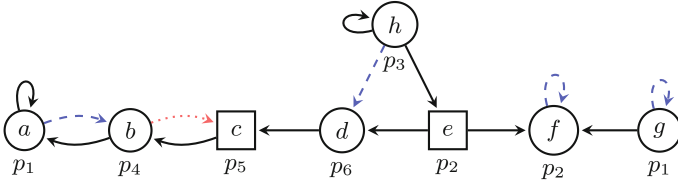


Fig. 3. A parity game, where a vertex with priority i has label p_i. The dotted edge in red is a co-live edge, while the dashed edges in blue are singleton live-groups. (Color figure online)

Algorithm 3. PARITYTEMPLATE($G, \mathbb{P}$)

Input: A game graph G, and a priority function $\mathbb{P} : V \to \{0, \ldots, d\}$
Output: Winning regions $(\mathcal{W}_0, \mathcal{W}_1)$, live-groups $\mathcal{H}$, and co-live edges D

```
 1: if d is odd then
 2:     A = attr^1_G(P_d)
 3:     if A = V then return (∅, V), ∅, ∅
 4:     else
 5:         (W_0, W_1), H, D ← ParityTemplate(G|_{V\A}, P)
 6:         if W_0 = ∅ then return (∅, V), ∅, ∅
 7:         else
 8:             B = attr^0_G(W_0)
 9:             D ← D ∪ Edges(W_0, V\W_0)
10:             H ← H ∪ ReachTemplate(G, W_0)
11:             (W'_0, W'_1), H', D' ← ParityTemplate(G|_{V\B}, P)
12:             return (W'_0 ∪ B, W'_1), H ∪ H', D ∪ D'
13: else                                                        ▷ If d is even
14:     A = attr^0_G(P_d)
15:     if A=V then return (V, ∅), ReachTemplate(G, P_d), ∅
16:     else
17:         (W_0, W_1), H, D ← ParityTemplate(G|_{V\A}, P)
18:         if W_1 = ∅ then return (V, ∅), H ∪ ReachTemplate(G|_A, P_d), D
19:         else
20:             B = attr^1_G(W_1)
21:             (W'_0, W'_1), H', D' ← ParityTemplate(G|_{V\B}, P)
22:             return (W'_0, W'_1 ∪ B), H', D'
```

and-conquer approach of Zeilonka's algorithm. Since the highest priority occurring is 6 which is even, we first find the vertices $A = \{d, h\}$ from which Player 0 can force visiting $\{d\}$ (vertices with priority 6) in line 14. Then since $A \neq V$, we find the winning strategy template in the rest of the graph $G_1 = G|_{V \setminus A}$. Then the highest priority 5 is odd, hence we compute the region $\{c\}$ from which Player 1 can ensure visiting 5. We again restrict our graph to $G_2 = G|_{\{a,b,e,f,g\}}$. Again, the highest priority is even. We further compute the region $A_2 = \{a, b\}$ from which Player 0 can ensure visiting the priority 4, giving us $G_3 = G|_{\{e,f,g\}}$. In G_3, Player 0 can ensure visiting the highest priority 2, hence satisfying the condition in line 15. Then since in this small graph, Player 0 needs to keep visiting priority 2 infinitely often, which gives us the live-groups $\{e_{gf}\}$ and $\{e_{ff}\}$ in line 15. Coming one recursive step back to G_2, since G_3 doesn't have a winning vertex for Player 1, the if condition in the line 18 is satisfied. Hence, for the vertices in A_2, it suffices to keep visiting priority 4 to win, which is ensured by the live-group $\{e_{ab}\}$ added in the line 18. Now, again going one recursive step back to G_1, we have $\mathcal{W}_0 = \{a, b, e, f, g\}$. If Player 0 can ensure reaching and staying in $\mathcal{W}_0$ from the rest of the graph G_1, it can satisfy the parity condition. Since from the vertex c, $\mathcal{W}_0$ will anyway be reached, we get a co-live edge e_{bc} in line 9 to eventually keep the play in $\mathcal{W}_0$. Coming back to the initial recursive call, since now again G_1 was winning for Player 0, they only need to be able to visit the priority 6 from every vertex in A, giving another live-group $\{e_{hd}\}$.

4 Extracting Strategies from Strategy Templates

This section discusses how a strategy that follows a computed winning strategy template can be extracted from the template. As our templates are just particular LTL formulas, one can of course use automata-theoretic techniques for this. However, as the types of templates we presented put some local restrictions on strategies, we can extract a strategy much more efficiently. For instance, the game in Fig. 2 with strategy template $\Psi = \Psi_{\text{LIVE}}(\{e_{ac}, e_{ad}\})$ allows the strategy that simply uses the edges e_{ac} and e_{ad} alternatively from vertex a.

However, strategy extraction is not as straightforward for every template, even if it only conjuncts the three template types we introduced in Sect. 3. For instance, consider again the game graph from Fig. 2 with a strategy template $\Psi = \{\Psi_{\text{UNSAFE}}(e_{ac}, e_{ad}), \Psi_{\text{COLIVE}}(e_{aa}, e_{ab})\}$. Here, non of the four choices of Player 0 (i.e., outgoing edges) from vertex a can be taken infinitely often, and, hence, the only way a play satisfies Ψ is to not visit vertex a infinitely often. On the other hand, given strategy template $\Psi' = \{\Psi_{\text{COLIVE}}(e_{ab}, e_{db}), \Psi_{\text{LIVE}}(\{e_{ab}, e_{ac}, e_{db}\})\}$, edge e_{db} is both live and co-live, which raises a conflict for vertex d. Hence, the only way a strategy can follow Ψ' is again to ensure that d is not visited infinitely often. We call such situations *conflicts*. Interestingly, the methods we presented in Sect. 3 never create such conflicts and the computed templates are therefore *conflict-free*, as formalized next and proven in the full version [5, Appendix A.4].

Definition 1. *A strategy template* $\Psi = \{\Psi_{\text{UNSAFE}}(S), \Psi_{\text{COLIVE}}(D), \Psi_{\text{LIVE}}(\mathcal{H})\}$ *in a game graph* $G = (V, E)$ *is* conflict-free *if the following are true:*

(i) or every vertex v*, there is an outgoing edge that is neither co-live nor unsafe, i.e.,* $v \times E(v) \not\subseteq D \cup S$*, and*

(ii) for every source vertex v *in a live-group* $H \in \mathcal{H}$*, there exists an outgoing edge in* H *which is neither co-live nor unsafe, i.e.,* $v \times H(v) \not\subseteq D \cup S$*.*

Proposition 1. *Algorithms 1, 2, and 3 always return conflict-free templates.*

Due to the given conflict-freeness, winning strategies are indeed easy to extract from winning strategy templates, as formalized next.

Proposition 2. *Given a game graph* $G = (V, E)$ *with* conflict-free *winning strategy template* $\Psi = \{\Psi_{\text{UNSAFE}}(S), \Psi_{\text{COLIVE}}(D), \Psi_{\text{LIVE}}(\mathcal{H})\}$*, a winning strategy* π_0 *that follows* Ψ *can be extracted in time* $\mathcal{O}(m)$*, where* m *is the number of edges.*

The proof is straightforward by constructing the winning strategy as follows. We first remove all unsafe and co-live edges from G and then construct a strategy π_0 that alternates between all remaining edges from every vertex in $\mathcal{W}_0$. This strategy is well defined as condition (i) in Definition 1 ensures that after removing all the unsafe and co-live edges a choice from every vertex remains. Moreover, if the vertex is a source of a live-group edge, condition (ii) in Definition 1 ensures that there are outgoing edges satisfying every live-group. It is easy to see that the constructed strategy indeed follows Ψ and is hence winning from vertices in $\mathcal{W}_0$, as Ψ was a winning strategy template. We call this procedure of strategy extraction ExtractStrategy(G, Ψ).

5 Applications of Strategy Templates

This section considers two concrete applications of strategy templates which utilize their structural simplicity and easy adaptability.

In the context of CPS control design problems, it is well known that the game graph of the resulting parity game used for strategy synthesis typically has a physical interpretation and results from behavioral constraints on the *existing technical system* that is subject to control. In particular, following the well-established paradigm of abstraction-based control design (ABCD) [2,7,39], an underlying (stochastic) disturbed non-linear dynamical system can be automatically abstracted into a two-player game graph using standard abstraction tools, e.g. SCOTS [35], ARCS [13], MASCOT [20], P-FACES [22], or ROCS [27].

In contrast to classical problems in reactive synthesis, it is very natural in this context to think about the game graph and the specification as two *different* objects. Here, specifications are naturally expressed via propositions that are defined over sets of states of this underlying game graph, without changing its structure. This separation is for example also present in the known LTL fragment GR(1) [10]. Arguably, this feature has contributed to the success of GR(1)-based synthesis for CPS applications, e.g. [1,3,24,25,38,40,41].

Given this insight, it is natural to define the incremental synthesis problem such that the game graph stays unchanged, while newly arriving specifications are modeled as new parity conditions over the same game graph. Formally, this results in a *generalized parity game* where the different objectives arrive *one at a time*. We show an incremental algorithm for synthesizing winning strategies for such games in Sect. 5.1. Similarly, fault-tolerant control requires the controller to adapt to unavailable actuators within the technical system under control. This naturally translates to the removal of Player 0 edges within the game graph given its physical interpretation. We show how strategy templates can be used to adapt winning strategies to these game graph modifications in Sect. 5.2.

5.1 Incremental Synthesis via Strategy Templates

In this section we consider a 2-player game $\mathcal{G}$ with a conjunction $\Phi = \bigwedge_{i=1}^{k} \Phi_i$ of multiple parity objectives Φ_i, also called a *generalized* parity objective. However, in comparison to existing work [12,16], we consider the case that different objectives Φ_i might not arrive all at the same time. The intuition of our algorithm is to solve each parity game (G, Φ_i) separately and then combine the resulting strategy templates Ψ_i to a global template $\Psi = \bigwedge_{i=1}^{k} \Psi_i$. This allows to easily incorporate newly arriving objectives Φ_{k+1}. We only need to solve the parity game (G, Φ_{k+1}) and then combine the resulting template Ψ_{k+1} with Ψ.

While Proposition 1 ensures that every individual template Ψ_i is *conflict-free*, this does unfortunately not imply that their conjunction is also *conflict-free*. Intuitively, combinations of strategy templates can cause the condition (i) and (ii) in Definition 1 to not hold anymore, resulting in a *conflict*. As already discussed in Sect. 4, this requires source vertices $U \subseteq V$ with such conflicts to

Algorithm 4. COMPOSETEMPLATE$(G, (\mathcal{W}_0', \mathcal{H}', D', (\Phi_i)_{i<\ell}), (\Phi_i)_{\ell \le i \le k})$ where $\Phi_i = Parity(\mathbb{P}_i)$

Input: A generalized parity game $G = (V, E)$ and objectives $(\Phi_i)_{i \le k}$ with $\Phi_i = Parity(\mathbb{P}_i)$ such that $\mathbb{P}_i : V \to [0; 2d_i + 1]$ along with a partial winning region, live-groups, and co-live edges $(\mathcal{W}_0, \mathcal{H}, D)$ for the generalized parity game $(G, \bigwedge_{i<\ell} \Phi_i)$.
Output: A partial winning region $\mathcal{W}_0$, live-groups $\mathcal{H}$, co-live edges D, and modified parity objectives $(\Phi_i')_{i \le k}$.
1: $(W_i, V \setminus W_i), \mathcal{H}_i, D_i \leftarrow$ PARITYTEMPLATE$(G|_{\mathcal{W}_0}, \Phi_i)$ for each $\ell \le i \le k$
2: $\mathcal{H} = \mathcal{H}' \cup \bigcup_{\ell \le i \le k} \mathcal{H}_i$; $D = D' \cup \bigcup_{\ell \le i \le k} D_i$; $\mathcal{W}_0 = \mathcal{W}_0' \cap \bigcap_{\ell \le i \le k} W_i$
3: $\mathcal{C}_1 = \{u \in \mathcal{W}_0 \mid u \times (E(u) \cap \mathcal{W}_0) \subseteq D\}$
4: $\mathcal{C}_2 = \{u \in \mathcal{W}_0 \mid u \times (H(u) \cap \mathcal{W}_0) \subseteq D,\ H \in \mathcal{H},\ H(u) \neq \emptyset\}$
5: **if** $\mathcal{C}_1 \cup \mathcal{C}_2 = \emptyset$ **then**
6: **return** $(\mathcal{W}_0, \mathcal{H}, D, (\Phi_i)_{i \le k})$
7: **else**
8: $\mathbb{P}_i'(u) \leftarrow \mathbb{P}[\mathcal{C}_1 \cup \mathcal{C}_2 \to 2d_i' + 1]$ for each $i \le k$
9: **return** COMPOSETEMPLATE$(G, (\mathcal{W}_0, \emptyset, \emptyset, \emptyset), (\Phi_i')_{i \le k})$ with $\Phi_i' = Parity(\mathbb{P}_i')$)

eventually not be visited anymore. We therefore resolve such conflicts by adding the specification $\Diamond\Box\neg U$ to every objective and recomputing the templates.

To efficiently formalize this objective change, we note that a parity objective $Parity(\mathbb{P})$ with an additional specification $\Diamond\Box\neg U$ for some $U \subseteq V$ is equivalent to another parity objective $Parity(\mathbb{P}')$, where priority function $\mathbb{P}'$ can be obtained from $\mathbb{P} : V \to [0; 2d+1]$ just by modifying the priorities of vertices in U to $2d+1$. Let us denote such a priority function by $\mathbb{P}[U \to 2d + 1]$. In particular, we have the following result:

Lemma 1. *Given a game graph G and two parity objectives $\Phi = Parity(\mathbb{P})$, $\Phi' = Parity(\mathbb{P}')$ such that $\mathbb{P} : V \to [0; 2d + 1]$ and $\mathbb{P}' = \mathbb{P}[U \to 2d + 1]$ for some vertex set $U \subseteq V$, it holds that $\mathcal{L}(\Phi') = \mathcal{L}(\Phi \wedge \Diamond\Box\neg U)$. Moreover, if a strategy template is winning from some vertex u in the game $\mathcal{G}' = (G, \Phi')$, then it is also winning from u in the game $\mathcal{G} = (G, \Phi)$.*

Using the above ideas, we present Algorithm 4 to solve generalized parity games (possibly incrementally). If no partial solution to the synthesis problem exists so far we have $\ell = 0$, otherwise the game $(G, \bigwedge_{i<\ell} \Phi_i)$ was already solved and the respective winning region and templates are known. In both cases, the algorithm starts with computing a winning strategy template for each game (G, Φ_i) for $i \in \{\ell + 1, k\}$ (line 1) and conjuncts them with the already computed ones (line 2). Then the algorithm checks for conflicts (line 3–4). If there is some conflict the algorithm modifies the objectives to ensure that the conflicted vertices are eventually not visited anymore (line 8), and then re-computes the templates in the game graph restricted to the intersection of winning regions for all objectives (line 9). If there is no conflict, then the algorithm returns the conjunction of the templates which is conflict-free, and hence, is winning from the intersection of winning regions for every objective (line 6). The latter is for-

malized in the following theorem. The proof can be found in the full version [5, Appendix B.2].

Theorem 5. *Given a generalized parity game* $\mathcal{G} = (G, \bigwedge_{i\leq k} \Phi_i)$ *with* $\Phi_i = \mathit{Parity}(\mathbb{P}_i)$ *and priority functions* $\mathbb{P}_i : V \to [0; 2d_i + 1]$*, if* $(\mathcal{W}_0, \mathcal{H}, D, (\Phi'_i)_{i\leq k}) = \textsc{ComposeTemplate}(G, \emptyset, (V, \emptyset, \emptyset), (\Phi_i)_{i\leq k})$*, then* $\Psi = \{\Psi_{\text{UNSAFE}}(S), \Psi_{\text{LIVE}}(\mathcal{H}), \Psi_{\text{COLIVE}}(D)\}$ *is an conflict-free strategy template that is winning from* $\mathcal{W}_0$ *in the game* $\mathcal{G}$*, where* $S = \textsc{Edges}(\mathcal{W}_0, V \setminus \mathcal{W}_0)$*. Further,* Ψ *is computable in time* $\mathcal{O}(kn^{2d+3})$ *time, where* $n = |V|$ *and* $d = \max_{i\leq k} d_i$*.*

Due to the conflict checks carried out within Algorithm 4 the returned modified objectives Φ'_i ensure that the *conjunction* $\Psi := \bigwedge_{i=1}^{k} \Psi'_i$ of winning strategy templates Ψ'_i for the games (G, Φ'_i) is indeed conflict-free. In particular, the conjuncted template Ψ is actually returned by the algorithm. Hence, incrementally running Algorithm 4 is actually sound. This is an immediate consequence of Theorem 5 and stated as a corollary next.

Corollary 1. *Given a generalized parity game* $\mathcal{G} = (G, \bigwedge_{i\leq k} \Phi_i)$ *with* $\Phi_i = \mathit{Parity}(\mathbb{P}_i)$ *and priority functions* $\mathbb{P}_i : V \to [0; 2d_i + 1]$*, s.t.*

$$(\mathcal{W}'_0, \mathcal{H}', D', (\Phi'_i)_{i<\ell}) := \textsc{ComposeTemplate}(G, (V, \emptyset, \emptyset, \emptyset), (\Phi_i)_{i<\ell}), \text{ and}$$
$$(\mathcal{W}_0, \mathcal{H}, D, (\Phi''_i)_{i\leq k}) := \textsc{ComposeTemplate}(G, (\mathcal{W}'_0, \mathcal{H}', D', (\Phi'_i)_{i<\ell}), (\Phi_i)_{\ell\leq i\leq k})$$

then $\Psi = \{\Psi_{\text{UNSAFE}}(S), \Psi_{\text{LIVE}}(\mathcal{H}), \Psi_{\text{COLIVE}}(D)\}$ *is an conflict-free strategy template that is winning from* $\mathcal{W}_0$ *in the game* $\mathcal{G}$*, where* $S = \textsc{Edges}(\mathcal{W}_0, V \setminus \mathcal{W}_0)$*. Further,* Ψ *is computable in time* $\mathcal{O}(kn^{2d+3})$*, where* $n = |V|$ *and* $d = \max_{i\leq k} d_i$*.*

We note that the generalized Zielonka algorithm [16] for solving generalized parity games has time complexity $\mathcal{O}(mn^{\sum 2d_i})\binom{\sum d_i}{d_1, d_2, \ldots, d_k}$ for a game with n vertices, m edges and k priority functions: $\mathbb{P}_i$ with $2d_i$ priorities for each i. Clearly, Algorithm 4 has a much better time complexity. However, it is not complete, i.e., it does not always return the complete winning region. This is due to templates being not maximally permissive and hence potentially raising conflicts which result in additional specifications that are not actually required. The next example shows such an incomplete instance for illustration. We however note that Algorithm 4 returned the *full winning region* on *all* benchmarks considered during evaluation, suggesting that such instances rarely occur in practice.

Example 4. Consider the game in Fig. 2 with objectives $\Phi_3 \wedge \Phi_4$ with $\Phi_4 = \mathit{Parity}(\mathbb{P})$, where $\mathbb{P}$ maps vertices a, b, c, d, e, f to $0, 2, 1, 1, 1, 1$, respectively. The winning strategy templates computed by ParityTemplate for objectives Φ_3 and Φ_4 are $\Psi_3 = \Psi_{\text{COLIVE}}(e_{ab}, e_{db}, e_{de})$ and $\Psi_4 = \Psi_{\text{LIVE}}(\{e_{ab}, e_{db}, e_{de}\})$, respectively. The conjunction of both templates marks all outgoing edges of vertex a and d in the live-group co-live. Hence, the algorithm would ensure that these conflicted vertices a and d are eventually not visited anymore. However, the only way to satisfy $\Phi_3 \wedge \Phi_4$ is by eventually looping on vertex a. But this solution was skipped by the strategy template Ψ_4 by putting edge e_{ab} in a live-group. Therefore, the algorithm returns the empty set as the winning region, whereas the actual winning region is the whole vertex set.

5.2 Fault-Tolerant Strategy Adaptation

In this section we consider a 2-player parity game $\mathcal{G} = (G, \mathit{Parity}(\mathbb{P}))$ and a set of faulty Player 0 edges $F \subseteq E \cap (V^0 \times V)$ which might become unavailable during runtime. Given a strategy template Ψ for $\mathcal{G}$, we can use $\Psi' = \{\Psi, \Psi_{\text{UNSAFE}}(F)\}$ for the (linear-time) extraction of a new strategy for the game, if Ψ' is conflict-free for G. In this case, no re-computation is needed. If Ψ' is not conflict-free for G, then we can remove the edges in F and compute a new winning strategy template using Algorithm 3. This is formalized in Algorithm 5, where we slightly abuse notation and assume that ParityTemplate only outputs strategy templates. The correctness of Algorithm 5 follows directly from Theorem 4.

Corollary 2. *Given a 2-player parity game $\mathcal{G} = (G, \mathit{Parity}(\mathbb{P}))$ with a strategy template $\Psi = \text{ParityTemplate}(G, \mathbb{P})$ and faulty edge set $F \subseteq E \cap (V^0 \times V)$ it holds that Ψ' obtained from Algorithm 5 is a winning strategy template for $\mathcal{G}|_{E \setminus F}$.*

Faulty edges introduce an additional safety specification for which our templates are maximally permissive. This implies that Algorithm 5 is *sound and complete* – if there exists a winning strategy for $(G|_{E \setminus F}, \mathit{Parity}(\mathbb{P}))$ Algorithm 5 finds one.

Let us now assume that F collects all edges controlling *vulnerable* actuators that *might* become unavailable. In this scenario, Algorithm 5 returns a conservative strategy that *never* uses vulnerable actuators. It might however be desirable to use actuators as long as they are available to obtain better performance. Formally, this application scenario can be defined via a time-dependent graph who's edges change over time, i.e., E_t with $E_0 = E$ are the edges available at time $t \in \mathbb{N}$ and $F := \{e \in E \mid e \notin E_i, \text{ for some } i\}$. Given the original parity game $\mathcal{G} = (G, \mathit{Parity}(\mathbb{P}))$ with a winning strategy template Ψ we can easily modify $\text{ExtractStrategy}(\mathcal{G}, \Psi)$ to obtain a time-dependent strategy π_g which reacts to the unavailability of edges, i.e., at time t, π_g takes an edge $e \in E_t \setminus (S \cup D)$ for all vertices without any live-group, and for the ones with live-groups, it alternates between the edges satisfying the live-groups whenever they are available, and an edge $e \in E_t \setminus (S \cup D)$ when no live-group edge is available.

The online strategy π_g can be implemented even without knowing when edges are available[2], i.e., without knowing the time dependent edge sequence $\{E_t\}_{t \in \mathbb{N}}$

Algorithm 5. FaultCorrection(G, Ψ, F)

Input: A parity game $\mathcal{G} = (G, \mathit{Parity}(\mathbb{P}))$, a strategy template Ψ, and a set of faulty edges F

Output: A new strategy template Ψ'

1: $\Psi' \leftarrow \{\Psi, \Psi_{\text{UNSAFE}}(F)\}$
2: **if** CheckTemplate(G, Ψ') **then return** Ψ'
3: **else**
4:     **return** ParityTemplate$(G|_{E \setminus F}, \mathbb{P}|_{E \setminus F})$

---

[2] We note that it is reasonable to assume that current actuator faults are visible to the controller at runtime, see e.g. [34] for a real water gate control example.

up front. In this case π_g is obviously winning in $\mathcal{G} = (G, Parity(\mathbb{P}))$ if Ψ is conflict-free for $\mathcal{G}|_{E \setminus F}$. If this is not the case, one needs to ensure that edges that cause conflicts are always eventually available again, as formalized next.

Definition 2. *Given a parity game $\mathcal{G} = (G, Parity(\mathbb{P}))$ we call the dynamic edge set $\{E_i\}_{i \geq 0}$ a* guaranteed availability fault (GAF) *if $\forall$ plays $\rho = v_0 v_1 \ldots$, $\forall v \in V$, if $v \in inf(\rho)$, then $\forall e = (v, w) \in F$, $\exists$ infinitely many times $t_0, t_1 \ldots$ such that $v_{t_j} = v$ and $e \in E_{t_j}$, $\forall j \geq 0$.*

Intuitively, guaranteed availability faults (GAF) ensure that a faulty edge is always eventually available when a play is in its source vertex. Under this fault, the following fault-correction result holds, which is proven in the full version [5, Appendix B.3].

Proposition 3. *Given a game graph G with a parity objective Φ, a strategy template $\Psi = \{\Psi_{\text{UNSAFE}}(S), \Psi_{\text{LIVE}}(\mathcal{H}), \Psi_{\text{COLIVE}}(D)\}$ computed by Algorithm 3 and a set $F = \{e \in E \mid e \notin E_i,$ for some $i\}$ of faulty edges, the game with the objective is realizable under GAF if for every vertex $v \in V^0$, there is an outgoing edge which is not in $S \cup D \cup F$.*

This proposition allows a simple linear-time algorithm to check if the templates computed by Algorithm 3 are GAF-tolerant: check if every vertex in the winning region has an outgoing edge which is not in $S \cup D \cup F$. If this is not the case, the recomputation is non-trivial and is out of scope of this paper. We can however collect the vertices which do not satisfy the above property and alert the system engineer that these vulnerable actuators require additional maintenance or protective hardware. Our experimental results in Sect. 6 show that conflicts arising from actuator faults are rare and very local. Our strategy templates allow to easily localize them, which supports their use for CPS applications.

6 Empirical Evaluation

We have developed a C++-based prototype tool PeSTel[3] (computing **PE**rmissive **S**trategy **TE**mp**L**ates) that implements Algorithms 1–5. We have used PeSTel to show its superior performance on the two applications considered in Sect. 5, suggesting its practical relevance. All our experiments were performed on a computer equipped with Apple M1 Pro 8-core CPU and 16 GB RAM.

Incremental Synthesis. We used PeSTel to solve generalized parity games both in one shot and incremental. We compare our algorithm with existing algorithms, i.e., GenZiel from [16] and three partial solvers[4] from [12], by executing

[3] Repository URL: https://github.com/satya2009rta/pestel.

[4] While GenZiel is sound and complete [16], we found different randomly generated games where the algorithms from [12] either return a superset or a subset of the winning region, hence compromising soundness and completeness. Since [12] lacks rigorous proof, it is not clear whether this is an implementation bug or a theoretical mishap, leaving soundness and completeness guarantees of these algorithms open.

Table 1. Aggregated experimental results on generalize parity game benchmarks with objectives given *up-front* (top) and *one-by-one* (bottom). Subrows: 1st row (mean time) – average computation time (in ms); 2nd row (incomplete) – number of examples where the corresponding tool failed to compute the complete winning region; 3rd row (faster than) – number of examples where PeSTel is faster than the respective tool; 4th row (timeouts) – number of examples where the respective tool timed out (10000 ms).

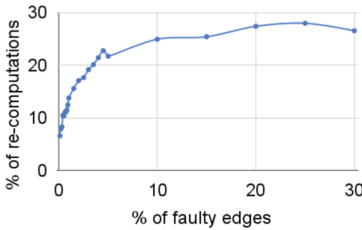
		PeSTel	GenZiel [16]	GenZiel & GenBüchi [12]	GenZiel & GenGoodEp[12]	GenZiel & GenLay[12]
Benchmark A (one shot)	mean time	343	64	68	553	1224
	incomplete	0	-	3	3	2
	faster than	-	74%	75%	96%	85%
	timeouts	0	0	0	2	20
Benchmark B (one shot)	mean time	60	47	58	112	171
	incomplete	0	-	28	27	2
	faster than	-	93%	93%	97%	95%
	timeouts	1	0	2	4	18
Overall	faster than	-	90%	90%	97%	94%
Benchmark B (incremental)	mean time	91	208	215	338	394
	incomplete	0	-	24	23	2
	faster than	-	97%	97%	98%	99%
	timeouts	2	0	0	8	23

them on a large set of benchmarks. We have generated two types of benchmarks from the games used for the Reactive Synthesis Competition (SYNTCOMP) [21]. Benchmark A was generated by converting parity games into Street games using standard methods, and as each Streett pair can be represented by a $\{0,1,2\}$-priority parity game, we represented the complete Streett objective as a conjunction of multiple $\{0,1,2\}$-priority parity objectives, resulting in a generalized parity game. Benchmark B was generated by adding randomly[5] generated parity objectives to given parity games. We considered 200 examples in Benchmark A and more than 1400 examples in Benchmark B.

We summarize the complete set of results of the experiments in[6] Table 1 and Fig. 1. We performed two kinds of experiments. First, we solved every generalized parity game in Benchmark A and B in *one shot* using the different methods. The results are shown in Table 1(top) and Fig. 1(left). Although the average time taken by PeSTel is higher than GenZiel and one partial solver, it is fastest in more than 90% of the games in both benchmarks. Thus, it shows that PeSTel is as efficient as the other methods in most cases. Moreover, for every

[5] The random generator takes three parameters: game graph "G", number of objectives "k", and maximum priority "m"; and then it generates "k" random parity objectives with maximum priority "m" as follows: 50% of the vertices in "G" are selected randomly, and those vertices are assigned priorities ranging from 0 to "m" (including 0 and m) such that 1/m-th (of those 50%) vertices are assigned priority 0 and 1/m-th are assigned priority 1 and so on. The rest 50% are assigned random priorities ranging from 0 to "m". Hence, for every priority, there are at least 1/(2m)-th vertices (i.e., 1/m-th of 50% vertices) with that priority.

[6] See the full version of this paper [5, Appendix C] for a version of Fig. 1 including all solvers considered in Table 1.



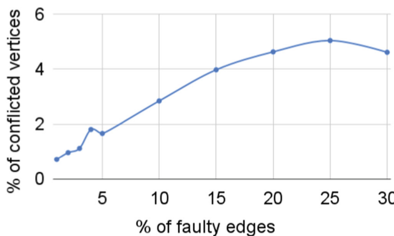


Fig. 4. Experimental results for parity games with faulty edges. Left: percentage of instances with conflicts given a certain percentage of faulty edges. Right: average percentage of vertices that created conflicts given a certain percentage of faulty edges.

game in both benchmarks, PeSTel succeeded to compute the complete winning region, whereas the partial solvers failed to do so in some cases[7]. We note that the instances which are hard for PeSTel are those where the winning region becomes empty, which is quickly detected by GenZiel but only seen by PeSTel after most objectives are (separately) considered.

Second, we solved the examples in Benchmark B by adding the objectives *one-by-one*, i.e., we solved the game with one objective, then we added one more objective and solved it again, and so on. The results are shown in Table 1(bottom) and Fig. 1(right). As PeSTel can use the pre-computed strategy templates if we add a new objective to a game, it outperforms all the other solvers significantly as they need to re-solve the game from scratch every time.

Fault-Tolerant Control. As discussed in Sect. 5.2, strategy templates can be used to implement a fault tolerant time-dependent strategy, if the set of faulty edges F does not cause conflicts with the strategy template. We have used PeSTel on over 200 examples of parity games from SYNTCOMP [21] to evaluate the relevance of such conflicts in practice. For this, we randomly selected different percentages of edges to be faulty and checked for conflicts with the given template. The results are summarized in Fig. 4. The left plot shows the number of instances for which a conflict occurs if a certain percentage of randomly selected edges is faulty. We see that the majority of the instances never faces a conflict even when 30% of the edges are faulty. Looking more closely into the instances with conflicts, Fig. 4(right) shows the average number of conflicting vertices in these benchmarks. Here we see that conflicts occur very locally at a very small number of vertices. Our strategy templates allow for a linear-time algorithm to localize them, allowing to mitigate them in practice by additional hardware.

Remark 1. We remark again that our results are directly applicable to CPS with continuous dynamics via the paradigm of abstraction-based control design (ABCD). In particular, standard abstraction tools such as SCOTS [35],

[7] Additionally, we outperform all algorithms on the benchmarks considered by Bruyère et al. [12]. We have however chosen to not include them in our analysis as many of their generalized parity games have only one objective and are therefore trivial.

ARCS [13], MASCOT [20], P-FACES [20], or ROCS [27] automatically compute a game graph from the (stochastic) continuous dynamics that can directly be used as an input to PeSTel. The winning strategy computed by PeSTel can further be refined into a correct-by-construction continuous feedback controller for the original dynamical system using standard methods from ABCD. We leave these tool integrations to future work.

References

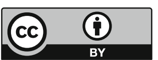
1. Maoz, S., Ringert, J.O.: Synthesizing a lego forklift controller in GR(1): a case study. In: Cerný, P., Kuncak, V., Madhusudan, P. (eds.) Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015. EPTCS, vol. 202, pp. 58–72 (2015). https://doi.org/10.4204/EPTCS.202.5
2. Alur, R.: Principles of Cyber-Physical Systems. MIT Press (2015)
3. Alur, R., Moarref, S., Topcu, U.: Counter-strategy guided refinement of GR(1) temporal logic specifications. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 26–33. IEEE (2013). https://ieeexplore.ieee.org/document/6679387/
4. Anand, A., Mallik, K., Nayak, S.P., Schmuck, A.K.: Computing adequately permissive assumptions for synthesis. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2023. Lecture Notes in Computer Science, vol. 13994, pp. 211–228. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_15
5. Anand, A., Nayak, S.P., Schmuck, A.K.: Synthesizing permissive winning strategy templates for parity games (2023)
6. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
7. Belta, C., Yordanov, B., Aydin Gol, E.: Formal Methods for Discrete-Time Dynamical Systems. SSDC, vol. 89. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-50763-7
8. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. RAIRO Theor. Inform. Appl. **36**(3), 261–275 (2002). https://doi.org/10.1051/ita:2002013
9. Blanke, M., Kinnaert, M., Lunze, J., Staroswiecki, M.: Diagnosis and Fault-Tolerant Control. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-540-35653-0
10. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**, 911–938 (2012). https://doi.org/10.1016/j.jcss.2011.08.007
11. Bouyer, P., Markey, N., Olschewski, J., Ummels, M.: Measuring permissiveness in parity games: mean-payoff parity games revisited. In: Bultan, T., Hsiung, P. (eds.) Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Taipei, Taiwan, 11–14 October 2011. LNCS, vol. 6996, pp. 135–149. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_11
12. Bruyère, V., Pérez, G.A., Raskin, J., Tamines, C.: Partial solvers for generalized parity games. In: Filiot, E., Jungers, R.M., Potapov, I. (eds.) Proceedings of the 13th International Conference on Reachability Problems, RP 2019, Brussels, Belgium, 11–13 September 2019. LNCS, vol. 11674, pp. 63–78. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-30806-3_6

13. Bulancea, O.L., Nilsson, P., Ozay, N.: Nonuniform abstractions, refinement and controller synthesis with novel BDD encodings. IFAC-PapersOnLine **51**(16), 19–24 (2018)
14. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 3rd edn. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-030-72274-6
15. Chatterjee, K., Henzinger, M.: Efficient and dynamic algorithms for alternating büchi games and maximal end-component decomposition. J. ACM **6**1(3) (2014). https://doi.org/10.1145/2597631
16. Chatterjee, K., Henzinger, T.A., Piterman, N.: Generalized parity games. In: Seidl, H. (ed.) Proceedings of the 10th International Conference on Foundations of Software Science and Computational Structures, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007. LNCS, Braga, Portugal, 24 March–1 April 2007, vol. 4423, pp. 153–167. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71389-0_12
17. Chatterjee, K., Henzinger, T.A., Piterman, N.: Algorithms for büchi games. CoRR abs/0805.2620 (2008). https://doi.org/10.48550/ARXIV.0805.2620
18. Ehlers, R., Lafortune, S., Tripakis, S., Vardi, M.Y.: Supervisory control and reactive synthesis: a comparative introduction. Discrete Event Dyn. Syst. **27**(2), 209–260 (2016). https://doi.org/10.1007/s10626-015-0223-0
19. Fritz, R., Zhang, P.: Overview of fault-tolerant control methods for discrete event systems. IFAC-PapersOnLine **51**(24), 88–95 (2018). https://doi.org/10.1016/j.ifacol.2018.09.533
20. Hsu, K., Majumdar, R., Mallik, K., Schmuck, A.K.: Multi-layered abstraction-based controller synthesis for continuous-time systems. In: HSCC 2018, pp. 120–129. ACM (2018)
21. Jacobs, S., et al.: The reactive synthesis competition (SYNTCOMP): 2018–2021. CoRR abs/2206.00251 (2022). https://doi.org/10.48550/arXiv.2206.00251
22. Khaled, M., Zamani, M.: pFaces: an acceleration ecosystem for symbolic control. In: HSCC 2019, pp. 252–257. ACM (2019)
23. Klein, J., Baier, C., Klüppelholz, S.: Compositional construction of most general controllers. Acta Informatica **52**(4–5), 443–482 (2015). https://doi.org/10.1007/s00236-015-0239-9
24. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Where's Waldo? Sensor-based temporal logic motion planning. In: 2007 IEEE International Conference on Robotics and Automation, ICRA 2007, 10–14 April 2007, Roma, Italy, pp. 3116–3121. IEEE (2007). https://doi.org/10.1109/ROBOT.2007.363946
25. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Temporal-logic-based reactive mission and motion planning. IEEE Trans. Robot. **25**(6), 1370–1381 (2009). https://doi.org/10.1109/TRO.2009.2030225
26. Lesi, V., Jakovljevic, Z., Pajic, M.: Towards plug-n-play numerical control for reconfigurable manufacturing systems. In: 21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2016, Berlin, Germany, 6–9 September 2016, pp. 1–8. IEEE (2016). https://doi.org/10.1109/ETFA.2016.7733524
27. Li, Y., Liu, J.: ROCS: a robustly complete control synthesis tool for nonlinear dynamical systems. In: HSCC 2018, pp. 130–135. ACM (2018)
28. Majumdar, R., Schmuck, A.: Supervisory controller synthesis for nonterminating processes is an obliging game. IEEE Trans. Autom. Control **68**(1), 385–392 (2023). https://doi.org/10.1109/TAC.2022.3143108

29. Meira-Góes, R., Kang, E., Lafortune, S., Tripakis, S.: On synthesizing tolerable and permissive controllers for labeled transition systems. IFAC-PapersOnLine **55**(28), 158–164 (2022). https://doi.org/10.1016/j.ifacol.2022.10.338
30. Moor, T.: A discussion of fault-tolerant supervisory control in terms of formal languages. Annu. Rev. Control. **41**, 159–169 (2016). https://doi.org/10.1016/j.arcontrol.2016.04.001
31. Neider, D., Rabinovich, R., Zimmermann, M.: Down the Borel hierarchy: solving muller games via safety games. Theor. Comput. Sci. **560**, 219–234 (2014). https://doi.org/10.1016/j.tcs.2014.01.017
32. Neider, D., Weinert, A., Zimmermann, M.: Synthesizing optimally resilient controllers. Acta Informatica **57**(1–2), 195–221 (2020). https://doi.org/10.1007/s00236-019-00345-7
33. Nilsson, P., et al.: Correct-by-construction adaptive cruise control: two approaches. IEEE Trans. Control Syst. Technol. **24**(4), 1294–1307 (2016). https://doi.org/10.1109/TCST.2015.2501351
34. Reijnen, F.F.H., Leliveld, E., van de Mortel-Fronczak, J.M., van Dinther, J., Rooda, J.E., Fokkink, W.J.: Synthesized fault-tolerant supervisory controllers, with an application to a rotating bridge. Comput. Ind. **130**, 103473 (2021). https://doi.org/10.1016/j.compind.2021.103473
35. Rungger, M., Zamani, M.: SCOTS: a tool for the synthesis of symbolic controllers. In: HSCC, pp. 99–104. ACM (2016)
36. Scher, G., Kress-Gazit, H.: Warehouse automation in a day: from model to implementation with provable guarantees. In: 16th IEEE International Conference on Automation Science and Engineering, CASE 2020, Hong Kong, 20–21 August 2020, pp. 280–287. IEEE (2020). https://doi.org/10.1109/CASE48305.2020.9217012
37. Schmuck, A.-K., Moor, T., Majumdar, R.: On the relation between reactive synthesis and supervisory control of non-terminating processes. Discrete Event Dyn. Syst. **30**(1), 81–124 (2019). https://doi.org/10.1007/s10626-019-00299-5
38. Svorenová, M., Kretínský, J., Chmelik, M., Chatterjee, K., Cerná, I., Belta, C.: Temporal logic control for stochastic linear systems using abstraction refinement of probabilistic games, pp. 259–268 (2015). https://doi.org/10.1145/2728606.2728608
39. Tabuada, P.: Verification and Control of Hybrid Systems - A Symbolic Approach. Springer, New York (2009). https://doi.org/10.1007/978-1-4419-0224-5
40. Wong, K.W., Ehlers, R., Kress-Gazit, H.: Resilient, provably-correct, and high-level robot behaviors. IEEE Trans. Robot. **34**(4), 936–952 (2018). https://doi.org/10.1109/TRO.2018.2830353
41. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Receding horizon control for temporal logic specifications. In: Johansson, K.H., Yi, W. (eds.) Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, 12–15 April 2010, pp. 101–110. ACM (2010). https://doi.org/10.1145/1755952.1755968
42. Wonham, W.M., Cai, K., et al.: Supervisory control of discrete-event systems (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Synthesizing Trajectory Queries from Examples

Stephen Mell[1(✉)], Favyen Bastani[2], Steve Zdancewic[1], and Osbert Bastani[1]

[1] University of Pennsylvania, Philadelphia, PA 19104, USA
{sm1,stevez,obastani}@cis.upenn.edu
[2] Allen Institute for AI, Seattle, WA 98104, USA
favyenb@allenai.org

Abstract. Data scientists often need to write programs to process predictions of machine learning models, such as object detections and trajectories in video data. However, writing such queries can be challenging due to the fuzzy nature of real-world data; in particular, they often include real-valued parameters that must be tuned by hand. We propose a novel framework called Quivr that synthesizes trajectory queries matching a given set of examples. To efficiently synthesize parameters, we introduce a novel technique for pruning the parameter space and a novel quantitative semantics that makes this more efficient. We evaluate Quivr on a benchmark of 17 tasks, including several from prior work, and show both that it can synthesize accurate queries for each task and that our optimizations substantially reduce synthesis time.

1 Introduction

Over the past decade, deep neural networks (DNNs) have successfully solved challenging artificial intelligence problems [47,70]. Abstractly, these models can be thought of as providing interfaces to real-world data—e.g., they can provide object classes [30,47], detections [59,60], and trajectories [10,11,83]. Then, these predictions are processed by programs, e.g., to identify driving patterns [5], events in TV broadcasts [28], or animal behaviors [67].

However, writing such programs can be challenging since they must still account for the fuzziness of real data. To do so, these programs typically include real-valued parameters that need to be manually tuned by the user. For example, consider a query over car trajectories designed to identify instances where one car turns in front of another. This query must capture the shape of the trajectory of both the turning car and the car crossing the intersection. In addition, the user must select the appropriate maximum duration from the first car changing lanes to the second car crossing the intersection. Even an expert would require significant experimentation to determine good parameter values; in our experience, it can take up to an hour to tune the parameters for a single query.

Appendices are available in the technical report [51].

© The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 459–484, 2023.
https://doi.org/10.1007/978-3-031-37706-8_23

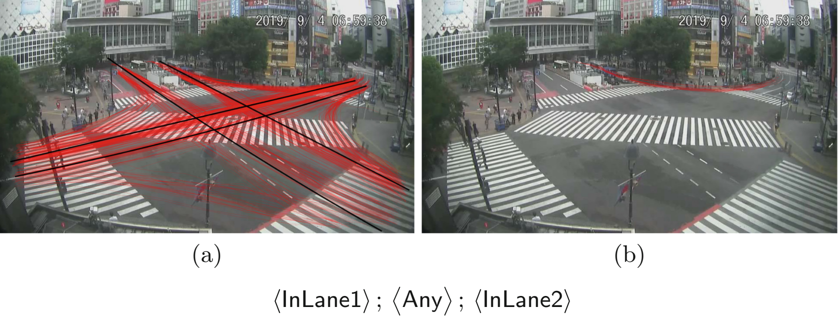
We focus on programs that query databases of trajectories output by an object tracker [5,7,8,28,40–42,54]. Given a video, the tracker predicts the positions of objects in each frame (e.g., cars, people, or mice), as well as associations between detections of the same object across successive frames. Applications often require subsequent analysis of these trajectories. For example, in autonomous driving, when a risky scenario is encountered, engineers typically search for additional examples of that driving pattern to improve their planner [63,64,66]—e.g., cars driving too close [82] or stopping in the middle of the road [6]. Object tracking has also been used to track robots [58,81], animals for behavioral analysis [12,67,75], and basketball players for sports analytics [67,85].

We propose an algorithm for synthesizing queries over object trajectories given just a handful of input-output examples. A query takes as input a representation of a trajectory as a sequence of states (e.g., position, velocity, and acceleration) in successive frames of the video, and outputs whether the trajectory matches its semantics. Our query language is based on regular expressions—in particular, a query is a composition of a user-extensible set of predicates using the sequencing, conjunction, and iteration operators. For instance, trajectories might correspond to cars in a video; Fig. 1 shows a query for identifying cars turning at an intersection. As we discuss in Sect. 6, the full query language semantics is rich enough to subsume (variants of) Kleene algebras with tests (KAT) [46] and signal temporal logic (STL) [50]; however, such generality is seldom needed, so we use a pared-down query language that works well in practice.

Our algorithm performs enumerative search over the space of possible queries to identify ones that are consistent with the given examples. A key challenge in our setting is that our predicates have real-valued parameters that must also be synthesized. Thus, our strategy enumerates *sketches*, which are partial programs that only contain holes corresponding to real-valued parameters. For each sketch, we search over the space of real-valued parameters, while using an efficient pruning strategy to reduce the search space. At a high level, we use a quantitative semantics to directly compute "boundary parameters" at which a given example switches from being labeled positive to negative. Then, depending on the target label, we can prune the entire region of the search space on one side of these boundary parameters. We prove that this synthesis strategy comes with soundness and (partial) completeness guarantees.

We implement our approach in a system called Quivr.[1] Our implementation focuses on videos from fixed-position cameras. While our language and synthesis algorithm are general, the predicates we design are tailored to specific settings. We evaluate Quivr on identifying driving patterns in traffic videos, including ones inspired by recent work on autonomous driving [63,64,66], on behavior detection in a dataset of mouse trajectories [72], and on a synthetic task from the temporal logic synthesis literature [44]. We demonstrate how both our parameter pruning strategies and our query evaluation optimizations lead to substantial reductions in the running time of our synthesizer.

[1] Quivr stands for QUery Induction for Video tRajectories.



(a) (b)

$\langle$InLane1$\rangle$; $\langle$Any$\rangle$; $\langle$InLane2$\rangle$

Fig. 1. (a) A video frame from a traffic camera, along with object trajectories (red) and manually annotated lanes (black). (b) The trajectories selected by the query (bottom), which selects cars turning at the intersection. (Color figure online)

In summary, our contributions are:

- A language for querying object trajectories (Sect. 3) and an algorithm for synthesizing such queries from examples (Sect. 4).
- An efficient parameter pruning approach based on a novel quantitative semantics (Sect. 4), yielding a 5.0× speedup over the state-of-the-art quantitative pruning technique from the temporal logic synthesis literature.
- An implementation of our approach in Quivr, and an evaluation of Quivr on identifying driving behaviors in traffic camera video and mouse behaviors in a dataset of mouse trajectories (Sect. 5), demonstrating substantially better accuracy than neural network baselines.

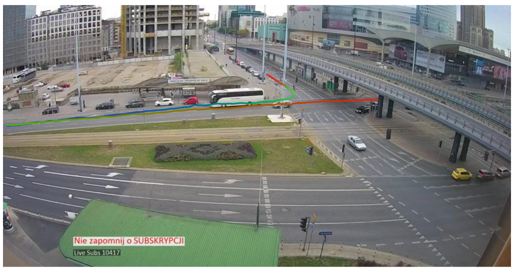
2 Overview

We consider a hypothetical scenario where an engineer is designing a control algorithm for an autonomous car and would like to identify certain driving patterns in video data. We show how they can use our framework to synthesize a query to identify car trajectories that exhibit a given behavior.

Video Data. Traffic cameras are a rich source of driving behaviors [5,13,61]; one dataset used in our evaluation is YTStreams [7], which includes video from several such cameras. Figure 1(a) shows a single frame from such a video; we have used an object tracker [83] to identify all car trajectories (in red).

Predicates. Quivr assumes it is given a set of predicates that match portions of trajectories exhibiting behaviors of interest; during synthesis, it considers queries composed of these predicates. In Fig. 1(a), the engineer has manually annotated the lanes of interest in this video (black), to specify four InLaneK predicates that select trajectories of cars driving in each lane K visible in the video. Predicates may be configured by real-valued parameters. For example,

$$\langle \mathsf{InLane1} \rangle \wedge \langle \mathsf{DispLt}_{\theta} \rangle$$



$$\left(\left\langle \mathsf{InLane1}(A)\right\rangle ; \langle \mathsf{Any}\rangle ; \left\langle \mathsf{InLane2}(A)\right\rangle\right) \wedge \left\langle \mathsf{InLane2}(B)\right\rangle$$

Fig. 2. A single match (top) for the multi-object query (bottom) which captures one car, A, turning into a lane behind another car, B, that is in that lane. The trajectories change color from red to green as a function of time. As can be seen, the car making the right turn does so just after the car going straight passes through the intersection. (Color figure online)

searches for trajectories where the car stays in lane 1 for a period of time and the car has a displacement at most θ between the beginning and end of that period. Note that atomic predicates, like $\langle \mathsf{DispLt}_\theta \rangle$, can match multiple time-steps, whereas in formalisms like regular expressions and temporal logic, atomic predicates are over single time-steps. A key feature of our framework is that the set of available predicates is highly extensible, and the user can provide their own. See Sect. 5.1 for the predicates we use in our evaluation.

Synthesis. To specify a driving pattern, the engineer provides a small number of initial positive and negative examples of trajectories; then, Quivr synthesizes a query that correctly labels these examples. In Fig. 1(b), we show the result of executing the query shown, which is synthesized to identify left turns in the data. Often, there are multiple queries consistent with the initial examples. While it may be hard for users to sift through the video for positive examples, it is usually easy for them to label a given trajectory. Thus, to disambiguate, Quivr asks the user to label additional trajectories [19,36,62].

Multi-object Queries. So far, we have focused on queries that identify trajectories by processing each trajectory in isolation. A key feature of our framework is that users can express queries over multiple trajectories—for example,

$$\left(\langle \mathsf{InLane1}(B)\rangle \wedge \langle \mathsf{ChangeLane2To1}(A)\rangle\right) ; \langle \mathsf{InFront}(A, B)\rangle.$$

This query says that car B is in lane 1 while car A changes from lane 2 to lane 1, and car A ends up in front of car B. Note that the predicates now include variables indicating which object they refer to, and the predicate $\mathsf{InFront}(A, B)$ refers to multiple objects. An example of a pair of trajectories selected by a multi-object query is shown in Fig. 2.

3 Query Language

We describe our query language for matching object trajectories in videos. Our system first preprocesses the video using an object tracker to obtain trajectories, which are sequences $z = (x_0, x_1, ..., x_{n-1})$ of states $x_i \in \mathcal{X}$. Then, a query Q in our language maps each trajectory z to a value $\mathbb{B} = \{0, 1\}$ indicating whether it matches z. Our language is similar to both STL and KAT. One key difference is that predicates are over arbitrary subsequences of z rather than single states x. In the main paper, we consider a simpler language, but in Appendix A we show how it can be extended to subsume both STL and KAT.

Trajectories. We begin by describing the input to a query in our language, which is the representation of one or more concurrent object trajectories in a video.

Consider a space $\mathcal{S}$ corresponding to a single object detection in a single video frame—e.g., $s \in \mathcal{S} \subseteq \mathbb{R}^6$ might encode the 2D position, velocity, and acceleration of s in image coordinates. When considering m concurrent objects, let the space of *states* $\mathcal{X} = \mathcal{S}^m$, and then a *trajectory* $z \in \mathcal{Z} = \mathcal{X}^*$ is a sequence $z = (x_0, x_1, ..., x_{n-1})$ of states of length $|z| = n$. We use the notation $z_{i:j} = (z_i, z_{i+1}, ..., z_{j-1})$ to denote a subtrajectory of z.

Predicates. We assume a set of predicates Φ is given, where each predicate $\varphi \in \Phi$ matches trajectories $z \in \mathcal{Z}$; we use $\mathsf{sat}_\varphi(z) \in \mathbb{B} = \{0, 1\}$ to indicate that φ matches z. As discussed below, queries in our language compose these predicates to match more complex patterns.

Next, predicates in our language may have real-valued parameters that must be specified. We denote such a predicate φ with parameter $\theta \in \mathbb{R}$ by φ_θ. To enable our synthesis algorithm to efficiently synthesize these real-valued parameters, we leverage the monotonicity in all such predicates we have used in our queries. In particular, we assume that the semantics of these predicates have the form

$$[\![\varphi_\theta]\!](z) \coloneqq \mathbb{1}(\iota_\varphi(z) \geq \theta),$$

where $\iota_\varphi : \mathcal{Z} \to \mathbb{R}$ is a scoring function. We also assume that the range of ι_φ is bounded (which can be achieved with a sigmoid function, if necessary). For example, for the predicate DispLt_θ, we have $\iota_{\mathsf{DispLt}}(z) = -\|z_0 - z_{n-1}\|$. Thus, $\iota_{\mathsf{DispLt}}(z) \geq \theta$ says the total displacement is at most $-\theta$. We describe the predicates we include in Sect. 5.1; they can easily be extended.

Syntax. The syntax of our language is

$$Q ::= \varphi \mid Q\,;\,Q \mid Q^k \mid Q \wedge Q,$$

where $Q^k = Q; Q; ...; Q$ (k times). That is, the base case is a single predicate φ, and queries can be composed using sequencing $(Q\,;\,Q)$ and conjunction $(Q \wedge Q)$. Operators for disjunction, negation, Kleene star, and STL's "until" are discussed in Appendix A.2. We describe constraints imposed on our language during synthesis in Sect. 4.7.

Semantics. The satisfaction semantics of queries have type $[\![\cdot]\!] : \mathcal{Q} \to \mathcal{Z} \to \mathbb{B}$, where $\mathcal{Q}$ is the set of all queries in our language, $\mathcal{Z}$ is the set of trajectories, and

$$\llbracket \varphi \rrbracket(z) \coloneqq \mathsf{sat}_\varphi(z)$$
$$\llbracket Q_1 \wedge Q_2 \rrbracket(z) \coloneqq \llbracket Q_1 \rrbracket(z) \wedge \llbracket Q_2 \rrbracket(z)$$
$$\llbracket Q_1 \,;\, Q_2 \rrbracket(z) \coloneqq \bigvee_{k=0}^{n} \llbracket Q_1 \rrbracket(z_{0:k}) \wedge \llbracket Q_2 \rrbracket(z_{k:n})$$

Fig. 3. Satisfaction semantics of our query language; $z \in \mathcal{Z}$ is a trajectory of length n and $\varphi \in \Phi$ are predicates. Iteration (Q^k) can be expressed as repeated sequencing.

$\mathbb{B} = \{0, 1\}$. In particular, $\llbracket Q \rrbracket(z) \in \mathbb{B}$ indicates whether the query Q matches trajectory z. The semantics are defined in Fig. 3. The base case of a single predicate φ checks whether φ matches z; conjunction $Q_1 \wedge Q_2$ checks if both conjuncts match; and sequencing $Q_1 \,;\, Q_2$ checks if z can be split into $z = z_{0:k} z_{k:n}$ in a way that Q_1 matches $z_{0:k}$ and Q_2 matches $z_{k:n}$. The semantics can be evaluated in time $O(|Q| \cdot n^2)$.

4 Synthesis Algorithm

We describe our algorithm for synthesizing queries consistent with a given set of examples. It performs a syntax-guided enumerative search over the space of possible queries [3]. In more detail, it enumerates *sketches*, which are partial programs where only parameter values are missing. For each sketch, it uses a quantitative pruning strategy to compute the subset of the input parameters for which the resulting query is consistent with the given examples. A key contribution is how our algorithm uses quantitative semantics for quantitative pruning.

4.1 Problem Formulation

Partial Queries. A *partial query* is in the grammar

$$Q ::= \,?? \mid \varphi_{??} \mid \varphi \mid Q \,;\, Q \mid Q^k \mid Q \wedge Q.$$

Note that there are two kinds of holes: (i) a *predicate hole* $h = ??$ that can be filled by a sub-query Q, and (ii) a *parameter hole* $h = \varphi_{??}$ that can be filled by a real value $\theta_h \in \mathbb{R}$. We denote the predicate holes of Q by $\mathcal{H}_\varphi(Q)$, the parameter holes by $\mathcal{H}_\theta(Q)$, and let $\mathcal{H}(Q) = \mathcal{H}_\varphi(Q) \cup \mathcal{H}_\theta(Q)$. A partial query Q is a *sketch* (denoted $Q \in \mathcal{Q}_{\text{sketch}}$) [71] if $\mathcal{H}_\varphi(Q) = \varnothing$, and is *complete* (denoted $Q \in \bar{\mathcal{Q}}$) if $\mathcal{H}(Q) = \varnothing$. For example, for $Q = \langle \mathsf{DispLt}_{??1} \rangle \wedge ??2$, we have $\mathcal{H}_\theta(Q) = \{??1\}$ and $\mathcal{H}_\varphi(Q) = \{??2\}$. (We label each hole $h = ??i$ with an identifier $i \in \mathbb{N}$ to distinguish them.)

Refinements and Completions. Given query $Q \in \mathcal{Q}$, predicate hole $h \in \mathcal{H}_\varphi(Q)$, and production $R = Q \to f(Q_1, ..., Q_k)$ we can *fill* h with R (denoted $Q' = \mathsf{fill}(Q, h, R)$) by replacing h with $f(??1, ..., ??k)$, where each $??i$ is a fresh hole,

and similarly given a parameter hole $h \in \mathcal{H}_\theta(Q)$ and a value $\theta_h \in \mathbb{R}$. We call Q' a *child* of Q (denoted $Q \rightarrow Q'$). Next, we call Q'' a *refinement* of Q (denoted $Q \xrightarrow{*} Q''$) if there exists a sequence $Q \rightarrow ... \rightarrow Q''$; if furthermore $Q'' \in \bar{\mathcal{Q}}$, we say it is a *completion* of Q. For example, we have

$$??1 \rightarrow ??2\,;\,??3 \rightarrow \langle \mathsf{InLane1} \rangle\,;\,??3 \rightarrow \ldots.$$

Here, $\langle \mathsf{InLane1} \rangle\,;\,??3$ is a child (and refinement) of $??2\,;\,??3$ obtained by filling $??2$ with $Q \rightarrow \langle \mathsf{InLane1} \rangle$—i.e.,

$$\langle \mathsf{InLane1} \rangle\,;\,??3 = \mathsf{fill}(??2\,;\,??3, ??2, Q \rightarrow \langle \mathsf{InLane1} \rangle).$$

Parameters. We let $\theta \in \mathbb{R}^{|\mathcal{H}_\theta(Q)|}$ denote a choice of parameters for each $h \in \mathcal{H}_\theta(Q)$, let $\theta_h \in \Theta_h \subseteq \mathbb{R}$ denote the parameter for hole h, and let Q_θ denote the query obtained by filling each $h \in \mathcal{H}_\theta(Q)$ with θ_h. Note that if $Q \in \mathcal{Q}_{\text{sketch}}$, then $Q_\theta \in \bar{\mathcal{Q}}$ is complete. For example, consider the sketch

$$Q = \langle \mathsf{DispLt}_{??1} \rangle \wedge \langle \mathsf{MinLength}_{??2} \rangle.$$

This query has two holes, so its parameters are $\theta \in \mathbb{R}^2$. If $\theta = (3.2, 5.0)$, then $\theta_{??1} = 3.2$ is used to fill hole $??1$ and $\theta_{??2} = 5.0$ is used to fill $??2$. In particular,

$$Q_\theta = \langle \mathsf{DispLt}_{3.2} \rangle \wedge \langle \mathsf{MinLength}_{5.0} \rangle.$$

Query Synthesis Problem. Given examples $W \subseteq \mathcal{W} = \mathcal{Z} \times \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$, our goal is to find a query $Q \in \bar{\mathcal{Q}}$ that correctly labels these examples—i.e.,

$$\psi_W(Q) \coloneqq \bigwedge_{(z,y)\in W} (\llbracket Q \rrbracket(z) = y).$$

Thus, $\psi_W(Q)$ indicates whether Q is consistent with the labeled examples W. Our goal is to devise a synthesis algorithm that is sound and complete—i.e., it finds a query that satisfies $\psi_W(Q) = 1$ if and only if one exists.

4.2 Algorithm Overview

Our algorithm enumerates sketches $Q \in \mathcal{Q}_{\text{sketch}}$; for each one, it tries to compute parameter values θ such that the completed query Q_θ is consistent with W—i.e., $\psi_W(Q_\theta) = 1$. It can either stop once it has found a consistent query, or identify additional queries that are consistent with W. Algorithm 1 shows this high-level strategy—at each iteration, it selects a sketch Q, determines a region B of the parameter space containing consistent parameters $\theta \in B$, and adds (Q, B) to a list of consistent queries that solve the synthesis problem.

The key challenge is searching over the space of continuous parameters θ for a given sketch Q such that Q_θ is consistent with W. For efficiency, we rely heavily on pruning the search space. At a high level, consider evaluating a single candidate parameter θ on a single example $(z, y) \in W$—i.e., check whether

Algorithm 1. Synthesizes consistent queries using the subroutine in Algorithm 2

1: **procedure** SynthesizeQuery(W)
2: $\mathcal{Q}_{\text{con}} \leftarrow \varnothing$
3: **for** $Q \in \mathcal{Q}_{\text{sketch}}$ **do**
4: $B \leftarrow \mathsf{SynthesizeParameters}(W, Q)$
5: $\mathcal{Q}_{\text{con}} \leftarrow \{(Q, B)\}$
6: **return** $\mathcal{Q}_{\text{con}}$

$[\![Q_\theta]\!](z) = y$. If this condition does not hold, then we can not only prune θ from the search space, but also a significant fraction of additional candidates. For instance, suppose $[\![Q_\theta]\!](z) = 1$ but $y = 0$; if $\theta' \leq \theta$ (in all components), then by a monotonicity property we prove for our semantics, we also have $[\![Q_{\theta'}]\!](z) = 1$. Thus, we can also prune θ'.

Previous work has leveraged this property to prune the search space [49,53,78]. Using a strategy based on binary search, for a given example $(z, y) \in W$, we can identify "boundary" parameters θ to accuracy ε in $O(\log(1/\varepsilon))$ steps—i.e., compute θ for which $[\![Q_{\theta-\vec{\varepsilon}}]\!](z) = 1$ and $[\![Q_{\theta+\vec{\varepsilon}}]\!](z) = 0$.

Our algorithm avoids this binary search process, which can lead to a significant speedup in practice. The key idea is to devise a quantitative semantics for queries that directly computes θ; in fact, this quantitative semantics is closely related to robust temporal logic semantics, where the conjunction and disjunction of the satisfaction semantics are replaced with minimum and maximum, respectively.

4.3 Pruning with Boundary Parameters

We begin by describing how "boundary parameters" can be used to prune a portion of the search space over parameters. First, for *any* candidate parameters θ, we can prune parameters $\theta' \leq \theta$ (if $[\![Q_\theta]\!](z) = 1$ and $y = 0$) or $\theta' \geq \theta$ (if $[\![Q_\theta]\!](z) = 0$ and $y = 1$). Pruned regions of the parameter space take the form of hyper-rectangles, which we call *boxes*. For convenience, let $\vec{\infty} := (\infty, \ldots, \infty)$.

Definition 1. Given $x, y \in \bar{\mathbb{R}}^d$, where $\bar{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$, a *box* is an axis-aligned half-open hyper-rectangle $\lfloor x, y \rceil := \{v \mid x_i < v_i \leq y_i\} \subseteq \mathbb{R}^d$.

The key property ensuring that parameters prune boxes of the search space is that the semantics are monotonically decreasing in θ.

Lemma 1. *Given sketch Q, trajectory z, and two candidate parameters $\theta, \theta' \in \mathbb{R}^d$ such that $\theta \leq \theta'$ component-wise, we have $[\![Q_\theta]\!](z) \geq [\![Q_{\theta'}]\!](z)$.*

The proof follows by structural induction on the query semantics: the base case follows since the semantics $\mathbb{1}(\iota_\varphi(z) \geq \theta_k)$ for predicates is monotonically decreasing in θ_k, and the inductive case follows since conjunction and disjunction are monotonically increasing in their inputs (so they are also monotonically decreasing in θ_k). Below, we show how monotonicity ensures that we can prune whole regions of the search space if we find boundary parameters.

As an example, suppose we have two trajectories, z_0 of a car driving quickly and then slowly, and z_1 of a car driving slowly and then quickly, and that we are trying to synthesize a query for $W = \{(z_0, 0), (z_1, 1)\}$. For simplicity, we assume both $z_0 = (0.9, 0.6)$ and $z_1 = (0.5, 0.8)$ have just two time steps each, with just a single component representing velocity. Furthermore, we assume there is just a single predicate $\langle \text{VelGt}_\theta \rangle$ matching time steps where the velocity is at least θ, where θ is a real-valued parameter. Since $\langle \text{VelGt}_\theta \rangle$ matches single time steps, the satisfaction semantics is 0 except on trajectories of length 1, so:

$$\begin{aligned} \iota_{\text{VelGt}}((z_0)_{0:1}) &= 0.9 & \iota_{\text{VelGt}}((z_0)_{1:2}) &= 0.6 & \iota_{\text{VelGt}}((z)_{i:i}) &= -\infty \\ \iota_{\text{VelGt}}((z_1)_{0:1}) &= 0.5 & \iota_{\text{VelGt}}((z_1)_{1:2}) &= 0.8 & \iota_{\text{VelGt}}((z)_{0:2}) &= -\infty \end{aligned}$$

Consider the sketch $Q = \langle \text{VelGt}_{??_1} \rangle ; \langle \text{VelGt}_{??_2} \rangle$. We can see that the candidate parameters $(0.5, 0.6)$ satisfy $[\![Q_{(0.5,0.6)}]\!](z_1) = 1$:

$$\begin{aligned} [\![Q_{(0.5,0.6)}]\!]((z_1)_{0:n}) &= \bigvee_{k=0}^{2} [\![\langle \text{VelGt}_{0.5} \rangle]\!]((z_1)_{0:k}) \wedge [\![\langle \text{VelGt}_{0.6} \rangle]\!]((z_1)_{k:n}) \\ &= [\![\langle \text{VelGt}_{0.5} \rangle]\!]((z_1)_{0:1}) \wedge [\![\langle \text{VelGt}_{0.6} \rangle]\!]((z_1)_{1:2}) \\ &= \mathbb{1}(0.5 \geq 0.5) \wedge \mathbb{1}(0.8 \geq 0.6) \\ &= 1, \end{aligned}$$

where the second equality holds because $\langle \text{VelGt}_\theta \rangle$ matches only length-1 trajectories, so the $k = 0$ and $k = 2$ cases evaluate to 0. Since the semantics are monotonically decreasing, we have $[\![Q_\theta]\!](z_1) = 1$ for any $\theta \in \lfloor (-\infty, -\infty), (0.5, 0.6) \rceil$.

Notice, however, that if we were to move any $\vec{\varepsilon} > 0$ upward, we would have $[\![Q_{(0.5+\varepsilon_1, 0.6+\varepsilon_2)}]\!](z_1) = \mathbb{1}(0.5 \geq 0.5 + \varepsilon_1) \wedge \mathbb{1}(0.8 \geq 0.6 + \varepsilon_2) = 0$. So we know $[\![Q_\theta]\!](z_1) = 0$ for any $\theta \in \lfloor (0.5, 0.6), (\infty, \infty) \rceil$. This is because $(0.5, 0.6)$ lies on the boundary between $\{\theta' \mid [\![Q_{\theta'}]\!](z) = 0\}$ and $\{\theta' \mid [\![Q_{\theta'}]\!](z) = 1\}$. This boundary plays a key role in our algorithm.

Definition 2. Given a sketch Q with d parameter holes and a trajectory z, we say $\theta \in \mathbb{R}^d \cup \{\bot, \top\}$ is a *boundary parameter* if one of the following holds:

- $\theta \in \mathbb{R}^d$ and $[\![Q_\theta]\!](z) = 1$, but $[\![Q_{\theta'}]\!](z) = 0$ for all $\theta' \in \lfloor \theta, \vec{\infty} \rceil$
- $\theta = \bot$ and $[\![Q_{\theta'}]\!](z) = 0$ for all $\theta' \in \lfloor -\vec{\infty}, \vec{\infty} \rceil$
- $\theta = \top$ and $[\![Q_{\theta'}]\!](z) = 1$ for all $\theta' \in \lfloor -\vec{\infty}, \vec{\infty} \rceil$

In the first case, by monotonicity, we also have $[\![Q_{\theta'}]\!](z) = 1$ for all $\theta' \in \lfloor -\vec{\infty}, \theta \rceil$; thus, θ lies on the boundary between parameters θ' where $Q_{\theta'}$ evaluates to 1 and those where it evaluates to 0. The second and third cases are where $Q_{\theta'}$ always evaluates to 0 and 1, respectively.

Given a boundary parameter θ for an example $(z, y) \in W$, we can prune $\lfloor \theta, \vec{\infty} \rceil$ if $y = 1$ or $\lfloor -\vec{\infty}, \theta \rceil$ if $y = 0$. Intuitively, boundary parameters provide optimal pruning along a fixed direction in the parameter space. Thus, our algorithm focuses on computing boundary parameters for pruning.

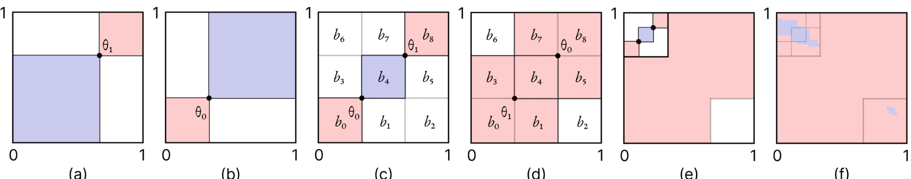


Fig. 4. (a) shows a boundary parameter, θ_1, for z_1, and a region that is inconsistent with z_1 and can be pruned (red), as well as a region that is consistent with it (blue). (b) similarly shows a boundary parameter θ_0 for z_0. (c) shows the pruning pair composed of θ_0 and θ_1, a region consistent with both (blue), and regions inconsistent with either (red). (d) is the same as (c), but if θ_0 and θ_1 swapped places. The labels b_0 through b_8 denote analogous boxes in (c) & (d). (e) shows how, if (d) were the result of the first step of search and b_6 were chosen next, search could proceed. (f) shows ground truth consistent (blue) and inconsistent (red) regions that the search process in (d) & (e) might converge toward. (Color figure online)

In Fig. 4(a), if θ_1 is a boundary parameter for z_1, we know that the blue region satisfies z_1, and thus is consistent with the label 1, while the red region dissatisfies z_1, and thus is inconsistent with the label 1. Similarly, in Fig. 4(b), if θ_0 is a boundary parameter for z_0, we know that the red satisfies z_1, and thus is inconsistent with the label 0, while the blue dissatisfies z_0, and thus is consistent with the label 0.

4.4 Pruning with Pairs of Boundary Parameters

To extend pruning to the entire dataset W, we could simply prune the union of the individual pruned regions for each $(z, y) \in W$. However, one important feature of our approach is that we can also establish regions of the parameter space where the parameters are guaranteed to be consistent with W. To formalize this idea, we introduce the concept of a "pruning pair", which is a pair of boundary parameters which might allow us to find such a consistent region.

Definition 3. Given a sketch Q and a dataset W, a pair of boundary parameters $\theta^-, \theta^+ \in \mathbb{R}^d \cup \{\bot, \top\}$ is a *pruning pair* for Q and W if all of the following hold:

- θ^+ is a boundary parameter for some $z \in W^+$ and, for all $z' \in W^+$ such that $z' \neq z$, we have $[\![Q_{\theta^+}]\!](z') = 1$.
- θ^- is a boundary parameter for some $z \in W^-$ and, for all $z' \in W^-$ such that $z' \neq z$, we have $[\![Q_{\theta^-}]\!](z') = 0$.
- $\theta^- < \theta^+$ or $\theta^- \geq \theta^+$.

If $\theta^- < \theta^+$, the pruning pair (θ^-, θ^+) is *consistent*, and *inconsistent* otherwise.

Our algorithm searches for pruning pairs along a fixed direction—i.e., it considers a curve $L \subseteq \mathbb{R}^d$ and looks for the following pruning pair along L:

$$\theta^+ = \sup \left\{ \theta \in L \;\middle|\; \bigwedge_{z \in W^+} [\![Q_\theta]\!](z) = 1 \right\}, \quad \theta^- = \inf \left\{ \theta \in L \;\middle|\; \bigwedge_{z \in W^-} [\![Q_\theta]\!](z) = 0 \right\}.$$

Intuitively, θ^+ is the largest θ that correctly classifies all positive examples, and conversely for θ^-. We restrict to curves L that are monotonically increasing in all components, in which case the supremum and infimum are well defined since L comes with a total ordering (from its smallest point to its largest) that is consistent with the standard partial order on $\mathbb{R}^d$. Then, (θ^-, θ^+) form a pruning pair: since θ^+ is a boundary parameter for z, if we take θ^+ to be any larger, then we must have $[\![Q_\theta]\!](z) = 0$ for some $z \in W^+$, and similarly for θ^-.

Given a curve L, we can compute an approximation to θ^+ and θ^- via binary search. However, our algorithm avoids the need to do so by directly computing θ^+ and θ^- using a quantitative semantics, which we describe in Sect. 4.6.

Figure 4(c) shows how the pair of boundary parameters θ_0 for z_0 and θ_1 for z_1 (where L is the diagonal line) prunes the parameter space. The blue region is guaranteed to be consistent with W, as it is the intersection of the region below θ^+, which must satisfy $[\![Q_\theta]\!](z_1) = 1$, and the region above θ^-, which must satisfy $[\![Q_\theta]\!](z_0) = 0$. Conversely, the red regions are inconsistent with either z_0 or z_1, and therefore with W. Thus, the red regions can be pruned, whereas the blue regions are solutions to our synthesis problem. Note that the red region is the union of the red regions in Fig. 4(a) and (b), whereas the blue region is the intersection of the blue regions in Fig. 4(a) and (b).

This pattern holds for any consistent pruning pair ($\theta^- < \theta^+$); if instead the pair is inconsistent ($\theta^- \geq \theta^+$), then the resulting pattern is illustrated in Fig. 4(d); in this case, we can prune the red regions as before, but there is no blue region of solutions. In general, for a d dimensional parameter space, a pruning pair divides the parameter space into 3^d boxes (i.e., for each dimension, the box can be below, in line with, or above the center box). The regions below θ^- and above θ^+ can be pruned, and the region between θ^- and θ^+ (if one exists) contains synthesis solutions. Precisely, it follows from the definitions and monotonicity that:

Lemma 2. *Every* $\theta \in \lfloor -\vec{\infty}, \theta^- \rceil$ *and* $\theta \in \lfloor \theta^+, \infty \rceil$ *is inconsistent with* W*, and every* $\theta \in \lfloor \theta^-, \theta^+ \rceil$ *is consistent with* W*.*

The remaining boxes need to be further analyzed by our algorithm.

4.5 Pruning Parameter Search Algorithm

Next, we describe Algorithm 2, which searches over the space of parameters to fill a sketch Q for a given dataset W. The algorithm uses a subroutine that takes a box and returns a pruning pair in that box, which we describe in Sect. 4.6. Given this subroutine, our algorithm maintains a work-list of "unknown" boxes (i.e., unknown whether parameters in these boxes are consistent or inconsistent with W). At each iteration, it pops a box from the work-list (in first-in-first-out order), uses the given subroutine to find a pruning pair inside that box, applies the pruning procedure described in the previous section, and then adds each new unknown box to the work-list.

For the last step, the current box b is divided into 3^d smaller boxes. The box $b_{\text{center}} := \lfloor \min\{\theta^-, \theta^+\}, \max\{\theta^-, \theta^+\} \rceil$ is pruned (added to B_{inc}) if the

Algorithm 2. Synthesizes consistent parameters for a given sketch

1: **procedure** SYNTHESIZEPARAMETERS(W, Q)
2: $B_{\text{con}} \leftarrow \varnothing$; $B_{\text{inc}} \leftarrow \varnothing$, $B_{\text{unk}} \leftarrow \{b_{\text{initial}}\}$
3: **for** $i \in \{1, ..., N\}$ **do**
4: $b \leftarrow \mathsf{Pop}(B_{\text{unk}})$
5: $\theta^-, \theta^+ \leftarrow \mathsf{ComputePruningPair}(W, Q, b)$
6: $b_{\text{center}} \leftarrow \lfloor \min\{\theta^-, \theta^+\}, \max\{\theta^-, \theta^+\} \rceil$
7: $b_{\text{lower}}, b_{\text{upper}}, B_{\text{incomp}}, B_{\text{extra}} \leftarrow \mathsf{SplitBox}(b, b_{\text{center}})$
8: **if** $\theta^- < \theta^+$ **then**
9: $B_{\text{con}} \leftarrow B_{\text{con}} \cup \{b_{\text{center}}\}$
10: $B_{\text{inc}} \leftarrow B_{\text{inc}} \cup \{b_{\text{lower}}, b_{\text{upper}}\}$
11: $B_{\text{unk}} \leftarrow B_{\text{unk}} \cup B_{\text{incomp}} \cup B_{\text{extra}}$
12: **else if** $\theta^- \geq \theta^+$ **then**
13: $B_{\text{inc}} \leftarrow B_{\text{inc}} \cup \{b_{\text{center}}, b_{\text{lower}}, b_{\text{upper}}\} \cup B_{\text{extra}}$
14: $B_{\text{unk}} \leftarrow B_{\text{unk}} \cup B_{\text{incomp}}$
15: **return** B_{con}

pair (θ^-, θ^+) is inconsistent, and contains solutions to the synthesis problem otherwise (added to B_{con}). The boxes $b_{\text{lower}} = \lfloor -\vec{\infty}, \min\{\theta^-, \theta^+\} \rceil$ and $b_{\text{upper}} = \lfloor \max\{\theta^-, \theta^+\}, \vec{\infty} \rceil$ are always pruned. The boxes $b \in B_{\text{incomp}}$ are the remaining corners of b, and always have indeterminate consistency (added to B_{unk}). The remaining boxes $b \in B_{\text{extra}}$ are indeterminate if (θ^-, θ^+) is consistent, and inconsistent otherwise. In our example, if the first step of the algorithm yielded Fig. 4(d), then the second step might pop b_6 and yield Fig. 4(e).

The following soundness result follows directly from Lemma 2.

Theorem 1. *In Algorithm 2, every $\theta \in B_{\text{con}}$ is consistent with W for Q, and every $\theta \in B_{\text{inc}}$ inconsistent.*

In addition, the algorithm is complete for almost all parameters:

Theorem 2. *The Lebesgue measure of $\{\theta \in b \mid b \in B_{\text{unk}}\} \to 0$ as $N \to \infty$.*

See Appendix D.1 for the proof. In other words, all parameters outside a subset of measure zero are eventually classified as consistent or inconsistent; intuitively, the parameters that may never be classified are the ones along the decision boundary. This result holds since at any search depth, the fraction of the parameter space pruned can be lower-bounded.

4.6 Computing Pruning Pairs via Quantitative Semantics

The pruning algorithm depends on the ability to compute, given a box b, a pruning pair (θ^-, θ^+) on the restriction of the parameter space to b. Recall that θ^+ must be a boundary parameter for some $z^+ \in W^+$ and must satisfy $[\![Q_{\theta^+}]\!](z) = 1$ for all other $z \in W^+$, and θ^- must be a boundary parameter for some $z^- \in W^-$, and must satisfy $[\![Q_{\theta^-}]\!](z) = 0$ for all other $z \in W^-$.

Given a box $b = \lfloor \theta_{\min}, \theta_{\max} \rceil$, our algorithm takes $L \subseteq \mathbb{R}^d$ to be the diagonal from $\theta_{\min}$ to $\theta_{\max}$ and computes the pruning pair along L. We can naïvely

$$\llbracket \varphi_{??i} \rrbracket^q_{v,u}(z) := \frac{\iota_\varphi(z) - v_i}{u_i}$$

$$\llbracket \varphi \rrbracket^q_{v,u}(z) := \begin{cases} \infty & \text{if } \mathsf{sat}_\varphi(z) = 1 \\ -\infty & \text{if } \mathsf{sat}_\varphi(z) = 0. \end{cases}$$

$$\llbracket Q_1 \wedge Q_2 \rrbracket^q_{v,u}(z) := \min\{\llbracket Q_1 \rrbracket^q_{v,u}(z), \llbracket Q_2 \rrbracket^q_{v,u}(z)\}$$

$$\llbracket Q_1 \,;\, Q_2 \rrbracket^q_{v,u}(z) := \max_{0 \le k \le n} \min\{\llbracket Q_1 \rrbracket^q_{v,u}(z_{0:k}), \llbracket Q_2 \rrbracket^q_{v,u}(z_{k:n})\}$$

Fig. 5. The quantitative semantics of our language, taking a sketch Q, trajectory z, parameter $v \in \mathbb{R}^d$, and positive vector $u \in \mathbb{R}^d_{>0}$. n is the length of z.

use binary search: for θ^+, we search for the parameters where $\bigwedge_{z \in W^+} \llbracket Q_\theta \rrbracket(z)$ transitions from 0 to 1, and similarly for θ^- and $\bigwedge_{z \in W^-} \neg \llbracket Q_\theta \rrbracket(z)$.

Instead, by leveraging a quantitative semantics, we can directly compute θ^+ and θ^-, thereby reducing computation time substantially. Given a sketch Q, trajectory z, parameter $v \in \mathbb{R}^d$, and positive vector $u \in \mathbb{R}^d_{>0}$, we devise a quantitative semantics $\llbracket Q \rrbracket^q_{v,u}(z) \in \bar{\mathbb{R}}$ such that the parameter $\llbracket Q \rrbracket_{v,u}(z) \cdot u + v$ is a boundary parameter. Intuitively, this semantics computes, starting at v, how many u-sized steps must be taken to reach the boundary. (For the uses in our algorithm, the number of steps is always in $[0, 1]$.) Then, for a box $b = \lfloor \theta_{\min}, \theta_{\max} \rceil$, we can take $v = \theta_{\min}$ and $u = \theta_{\max} - \theta_{\min}$, and compute

$$\theta^+ = \left(\min_{z \in W^+} \llbracket Q \rrbracket^q_{v,u}(z) \right) \cdot u + v, \qquad \theta^- = \left(\max_{z \in W^-} \llbracket Q \rrbracket^q_{v,u}(z) \right) \cdot u + v.$$

We define the quantitative semantics in Fig. 5. The base case of $\varphi_{??}$ adjusts and rescales ι_φ by v and u, and the other cases replace conjunction and disjunction in the satisfaction semantics with minimum and maximum. We have the following key result (where $\infty \cdot u := \top$, $-\infty \cdot u := \bot$, $\top + v := \top$, and $\bot + v := \bot$):

Theorem 3. *For a sketch Q, trajectory z, parameter $v \in \mathbb{R}^d$, and positive vector $u \in \mathbb{R}^d_{>0}$, we have that $\llbracket Q \rrbracket^q_{v,u}(z) \cdot u + v$ is a boundary parameter of z for Q.*

See Appendix D.2 for the full proof. For intuition, consider $\theta_{\min} = \vec{0}$ and $\theta_{\max} = \vec{1}$ (i.e., the current box $b \subseteq \mathbb{R}^d$ is the unit hypercube). Then, $v = \vec{0}$ and $u = \vec{1}$, so $\llbracket Q \rrbracket^q_{v,u}$ reduces to the standard max-min quantitative semantics for temporal logic [25].

Now, if we consider the satisfaction semantics of a base predicate $\llbracket \varphi_{\theta_i} \rrbracket = \mathbb{1}(\iota_\varphi(z) \ge \theta_i)$, then the value of θ_i where the sementics flips is just $\iota_\varphi(z)$. So any parameter with i-th component $\iota_\varphi(z)$ is a boundary parameter, and since L has the same slope in all dimensions, the boundary parameter along L is $\iota_\varphi(z) \cdot \vec{1} + \vec{0} = \llbracket \varphi_{??i} \rrbracket^q_{\vec{0},\vec{1}}(z) \cdot \vec{1} + \vec{0}$.

In the inductive cases, it suffices to show that we can replace conjunction and disjunction with minimum and maximum in the semantics. Since the satisfaction

semantics is monotonically decreasing, as we move upward along L, at some point we will transition from 1 to 0. A conjunction becomes 0 when either conjunct becomes 0, so the transition will occur when we hit the first of the conjuncts' transition points (their minimum). Dually, a disjunction becomes 0 when both disjuncts become 0, so we will transition at the last of the disjuncts' transition points (their maximum).

Finally, the intuition behind u and v is that they "preprocess" the parameters so that we evaluate along the diagonal of the current box instead of $\left\lfloor \vec{0}, \vec{1} \right\rceil$.

4.7 Implementation

We implement our approach in a system called Quivr. It begins by running Algorithm 1 on a small number of labeled examples.

Active Learning. With a small number of examples, there are typically many queries that are consistent with the labels, and yet which disagree on the labels of the remaining data. To disambiguate, we use an active learning strategy, asking the user to label specific trajectories that we choose, which are then added to our set of labeled examples. Queries that are not consistent with the new label are discarded. The labeling process continues until the set of consistent queries agrees on the labels of all unlabeled data.

When choosing the trajectory z^* to query the user for next, we select the one on which the set of consistent queries C disagrees most—i.e.,

$$z^* = \arg\min_{z \in Z} \left| J(z) - \frac{1}{2} \right|,$$

where

$$J(z) := |C|^{-1} \sum_{Q_\theta \in C} \mathbb{1}\left(\psi_{(z,y)}(Q_\theta)\right)$$

is the fraction of consistent queries that predict a positive label for trajectory z.

Search Implementation. In some cases, searching for consistent parameters may take a very long time. To improve performance, we impose a timeout: for each sketch, we pause search if either: (i) we find some consistent box of parameters or (ii) we've exceeded 25 steps. In both cases, we save the sets of consistent, inconsistent, and unknown boxes. At each step of active learning, the newly labeled example may render previously consistent parameters inconsistent, so we mark all consistent boxes as unknown. We then resume search, again until (i) we find some consistent box (which may be the same one we had before), or (ii) we again exceed 25 steps.

Note that while this timeout may cause us to query the user more often than is strictly necessary, it does not affect either the soundness or completeness of our approach, as we continue search after querying the user.

Complete Query Selection. Active learning and evaluation of F_1 scores (in Sect. 5) both require complete queries with specific parameters, rather than sketches

Table 1. The predicates used for the YTStreams dataset.

Predicate	Description
$\mathsf{InLaneK}(A)$	Whether, at every time-step in the interval, object A is sufficiently close to the annotated curve for lane K and A's movement direction is sufficiently in line with the curve for K.
$\mathsf{DurationNotShort}$	Whether the interval spans at least 5 seconds.
$\mathsf{AvgAccelGt}_\theta$	Whether the average acceleration over the interval is at least θ.
$\mathsf{DistanceLt}_\theta$	Whether, at every time-step in the interval, the distance between the two objects is less than θ.
$\mathsf{SpeedRatioGt}_\theta(A, B)$	Whether, at every time-step in the interval, the speed of A divided by the speed of B is at least θ.
$\mathsf{DispLt}_\theta(A)$	Whether the distance between the position in the first frame of the interval and the position in the last frame is less than θ.

with boxes of parameters. Since the set C of consistent queries is infinitely large, we instead we use one query for each sketch that is known to have consistent parameters (sketches where search timed-out are thus not included). For those sketches, we pick the middle of the box of known-consistent parameters.

5 Evaluation

We demonstrate how our approach can be used to synthesize queries to solve interesting tasks: in particular, we show that (i) given just a few initial examples, it can synthesize queries that achieve good performance on a held-out test set, and (ii) our optimizations significantly reduce the synthesis time.

5.1 Experimental Setup

Datasets. We evaluate on two datasets of object trajectories: YTStreams [7], consisting of video and extracted object trajectories from fixed-position traffic cameras, and MABe22 [72], consisting of trajectories of up to three mice interacting in a laboratory setting. We also evaluate on a synthetic maritime surveillance task from the STL synthesis literature [44]. On YTStreams, we use two traffic cameras, one in Tokyo and one in Warsaw, and we consider single cars or pairs of cars. On MABe22, we consider pairs of mice. For the predicates used, see Table 1 for YTStreams, Appendix Table 5 for MABe22, and Appendix Table 6 for maritime surveillance.

Tasks. On YTStreams, we manually wrote 5 ground truth queries. Several queries apply to multiple configurations (e.g., different pairs of lanes), resulting in 10 queries total (tasks H-Q in Table 2). The real-valued parameters were chosen manually, by visually examining whether they were selecting the desired trajectories. These queries cover a wide range of behaviors; for instance, they can

Table 2. Ground-truth queries for the YTStreams dataset. “IDs” indicates which tasks are instances of a given query. Multiple instantiations correspond to different lanes being used for “lane 1” and “lane 2”. The first is a one-object Shibuya query, the second is a one-object Warsaw query, and the rest are two-object Warsaw queries.

IDs	Query
H, I, J, K	$\left\langle \mathsf{InLane1}(A) \right\rangle ; \langle \mathsf{Any} \rangle ; \left\langle \mathsf{InLane2}(A) \right\rangle$
	Matches cars that turn, starting in lane 1 and ending in lane 2.
L, M	$\left\langle \mathsf{InLane1}(A) \right\rangle \wedge \left\langle \mathsf{AvgAccelGt}(A) \right\rangle_{??} \wedge \langle \mathsf{DurationNotShort} \rangle$
	Matches cars that accelerate for a significant period of time while in lane 1.
N	$\left(\left\langle \mathsf{InLane1}(A) \right\rangle ; \langle \mathsf{Any} \rangle ; \left\langle \mathsf{InLane2}(A) \right\rangle \right) \wedge \left\langle \mathsf{InLane2}(B) \right\rangle$
	Matches pairs of cars where car B is in lane 2 for the entire duration of A turning from lane 1 into lane 2.
O, P	$\left\langle \mathsf{InLane1}(A) \right\rangle \wedge \left\langle \mathsf{InLane2}(B) \right\rangle \wedge \langle \mathsf{DurationNotShort} \rangle \wedge \left\langle \mathsf{SpeedFactorGt}(A, B) \right\rangle_{??}$
	Matches pairs of cars in parallel lanes, 1 and 2, where car A is going faster than car B for a significant period of time.
Q	$\left\langle \mathsf{InLane1}(A) \right\rangle \wedge \left\langle \mathsf{InLane2}(B) \right\rangle \wedge \langle \mathsf{DurationNotShort} \rangle \wedge \left\langle \mathsf{DistanceLt}(A, B) \right\rangle_{??}$
	Matches pairs of cars in parallel lanes, 1 and 2, where the cars are close for a significant period of time

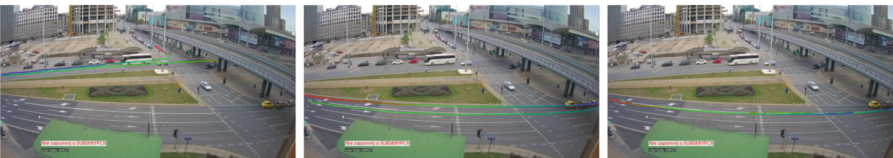


Fig. 6. Trajectories selected by multi-object queries. Each image shows two objects; the color of each one changes from red to green to denote the progression of time. Left: Unprotected right turn into lane with oncoming traffic. Middle: Bottom car drives faster than the top one and passes it. Right: One car driving closely behind the other. (Color figure online)

capture behaviors such as human drivers making unprotected turns, an important challenge for autonomous cars [64], as well as cars trying to pass [66]. We show examples of trajectories selected by three of our multi-object queries in Fig. 6. MABe22 describes 9 queries for scientifically interesting mouse behavior. We implemented the 6 most complex to use as ground truth queries (tasks A-F in Appendix Table 7). The maritime surveillance task has trajectory labels and so does not need a ground truth query (task G).

Synthesis. For each task, we divide the set Z of all trajectories into a train set Z_{train} and a test set Z_{test}, using trajectories in the first half of the video for training, and those in the second half for testing. We randomly sample a set of initial labeled examples W from Z_{train}, with 2 samples being positive and 10 being negative, and then actively label 25 additional examples from Z_{train}. For YTStreams and MABe22, labels are from the ground truth query.

Table 3. F_1 score after n steps of active learning, with our algorithm for selecting tracks to label ("Q"), an active learning ablation ("R"), an LSTM ("L"), and a transformer ("T"). For Q and R, there may be many queries consistent with the labeled data, so the median F_1 score is reported. Bold indicates best score at a given number of steps.

ID	0 Steps				5 Steps				10 Steps				25 Steps			
	Q	R	L	T	Q	R	L	T	Q	R	L	T	Q	R	L	T
A	0.69	0.69	**1.00**	0.74	1.00	**1.00**	1.00	0.74	**1.00**	1.00	1.00	0.74	**1.00**	1.00	1.00	0.74
B	**0.99**	**0.99**	0.47	0.20	**0.99**	**0.99**	0.47	0.25	**0.99**	**0.99**	0.47	0.05	**0.99**	0.98	0.47	0.06
C	**0.96**	**0.96**	0.38	0.09	**0.99**	0.96	0.38	0.08	**0.99**	0.96	0.38	0.02	**0.99**	0.98	0.38	0.01
D	**0.77**	**0.77**	0.52	0.27	**0.99**	0.96	0.52	0.28	**0.99**	**0.99**	0.52	0.32	0.99	**1.00**	0.52	0.08
E	**1.00**	**1.00**	0.44	0.29	**1.00**	**1.00**	0.44	0.14	**1.00**	**1.00**	0.44	0.13	**1.00**	**1.00**	0.44	0.07
F	**0.88**	**0.88**	0.78	0.38	**0.99**	0.96	0.78	0.39	**1.00**	0.96	0.78	0.18	**1.00**	0.96	0.78	0.27
G	0.68	0.68	0.65	**0.78**	**1.00**	**1.00**	0.65	0.77	**1.00**	**1.00**	0.65	0.77	**1.00**	**1.00**	0.65	0.77
H	**0.30**	**0.30**	0.12	0.22	0.34	**0.34**	0.13	0.23	**0.92**	**0.92**	0.13	0.22	**0.92**	**0.92**	0.13	0.37
I	**0.37**	**0.37**	0.13	0.00	**1.00**	0.37	0.13	0.00	**1.00**	**1.00**	0.13	0.00	**1.00**	**1.00**	0.13	0.31
J	0.07	0.07	0.01	**1.00**	0.41	0.07	0.01	**0.86**	**0.80**	0.09	0.04	0.75	0.80	0.09	0.04	**0.86**
K	**0.28**	**0.28**	0.15	0.00	**0.99**	0.27	0.15	0.00	**0.99**	**0.99**	0.15	0.00	**0.99**	**0.99**	0.15	0.00
L	**0.67**	**0.67**	0.07	0.37	**0.96**	0.88	0.07	0.42	**0.96**	0.88	0.07	0.08	**0.96**	0.88	0.07	0.31
M	**0.92**	**0.92**	0.10	0.37	**0.99**	0.92	0.10	0.46	**0.99**	0.92	0.10	0.00	**0.99**	0.92	0.10	0.18
N	**0.60**	**0.60**	0.02	0.00	**0.20**	0.09	0.02	0.00	0.11	**0.21**	0.02	0.00	0.18	**0.78**	0.02	0.31
O	**0.11**	**0.11**	0.01	0.04	**0.50**	0.17	0.01	0.21	**0.70**	0.17	0.01	0.21	**1.00**	0.21	0.01	0.00
P	**0.16**	**0.16**	0.04	0.04	**0.23**	0.21	0.03	0.04	**0.82**	0.21	0.03	0.14	**1.00**	0.29	0.03	0.14
Q	**0.07**	**0.07**	0.02	0.02	0.16	0.12	0.01	**0.31**	**0.92**	0.12	0.01	0.18	**1.00**	0.12	0.01	0.20

For tractability, we limit search to sketches with at most three predicates, at most two of which may have parameters. In most cases, this excludes the ground truth from the search space.

5.2 Accuracy of Synthesized Queries

We show that QUIVR synthesizes accurate queries from just a few labeled examples. We evaluate the F_1 score of the synthesized queries on Z_{test}. Recall that our algorithm returns a list C of consistent queries; we report the median F_1 score across $Q \in C$.

Baselines. We compare to (i) an ablation where we replace our active learning strategy with an approach that labels z uniformly at random from the remaining unlabeled training examples; (ii) an LSTM [16,33] neural network; and (iii) a transformer neural network [26,29,77]. Because neural networks perform poorly on such small datasets, we pretrain the LSTM on an auxiliary task, namely, trajectory forecasting [43]. Then, we freeze the hidden representation of the learned LSTM, and use these as features to train a logistic regression model on our labeled examples. The neural network baselines do active learning by selecting among the unlabeled trajectories the one with the highest predicted probability of being positive.

Results. We show the F_1 score of each of the 17 queries in Table 3 after 0, 5, 10, and 25 steps of active learning. After just 10 steps, our approach provides F_1 score above 0.99 on 10 of 17 queries, and after 25 steps, it yields an F_1 score

Table 4. Running time (seconds) of synthesis (mean ± standard error) using binary search (B) and quantitative semantics (Q) running on CPU and GPU, with 25 steps of active learning.

ID	CPU		GPU	
	B	Q	B	Q
A	$8,460 \pm 1,517$	$3,343 \pm 202$	737 ± 36	174 ± 14
B	$3,511 \pm 549$	$2,291 \pm 237$	428 ± 37	110 ± 9
C	$3,319 \pm 505$	$2,007 \pm 359$	376 ± 6	113 ± 9
D	$2,728 \pm 476$	$2,714 \pm 334$	370 ± 8	119 ± 2
E	$1,264 \pm 176$	599 ± 54	225 ± 3	50 ± 1
F	$1,689 \pm 360$	748 ± 81	285 ± 7	60 ± 1
G	661 ± 141	133 ± 23	219 ± 77	30 ± 1
H	399 ± 70	147 ± 9	185 ± 94	32 ± 17
I	400 ± 74	84 ± 13	163 ± 85	23 ± 12
J	544 ± 120	173 ± 5	227 ± 121	36 ± 19
K	493 ± 77	125 ± 25	163 ± 83	30 ± 16
L	732 ± 47	286 ± 73	252 ± 133	57 ± 29
M	697 ± 40	253 ± 49	245 ± 128	56 ± 30
N	$5,691 \pm 272$	977 ± 176	$1,393 \pm 590$	264 ± 136
O	$8,306 \pm 521$	$2,314 \pm 476$	811 ± 12	127 ± 2
P	$11,326 \pm 673$	$4,198 \pm 1,333$	970 ± 60	167 ± 8
Q	$12,430 \pm 962$	$2,915 \pm 508$	$1,141 \pm 101$	183 ± 11

above 0.9 on all but 2 queries. Thus, QUIVR is able to synthesize accurate queries with relatively little user input. The neural networks achieve poor performance, particularly on the more difficult queries.

5.3 Synthesis Running Time

Next, we show that quantitative pruning and using a GPU each significantly reduce synthesis time, evaluating total running time for 25 steps of active learning.

Ablations. We compare to two ablations: (i) using the binary search approach of [53] to find pruning pairs, rather than using our quantitative semantics, and (ii) evaluating the matrix semantics (Appendix A.1) on a CPU rather than a GPU.

Results. In Fig. 4, we report the running time of our algorithms on a CPU (2× AMD EPYC 7402 24-Core) and a GPU (1× NVIDIA RTX A6000). For binary search, on average, the GPU is 7.6× faster than the CPU. On a GPU, using the quantitative semantics rather than binary search offers another 5.0× speed-up.

6 Related Work

Monotonicity for Parameter Pruning. We build on [49] for our parameter pruning algorithm. Their approach has been applied to synthesizing STL formulas for

sequence classification by first enumerating sketches and then using monotonicity to find parameters, similar to our binary search baseline [53]. We replace binary search with our novel strategy based on quantitative semantics, leading to 5.0× speedup. There is also work building on [49] to create logically-relevant distance metrics between trajectories by taking the Hausdorff distance between parameter satisfaction regions (which they call "validity domains"), with applications to clustering [78]. For logics like STL, our quantitative semantics could provide a speedup to their approach.

Synthesis of Temporal Logic Formulas. More broadly, there has been work synthesizing parameters in a variant of STL by discretizing the parameter space and then walking the satisfaction boundary [24]; in one dimension, their approach becomes binary search, inheriting its shortcomings. There has been work on synthesizing STL formulas that are satisfied by a closed-loop control model [38], but they assume the ability to find counterexample traces for incorrect STL formulas, which is not applicable to our setting. Another approach is to synthesize parameters in STL formulas using gradient-based optimization [35] or stochastic optimization [45], but we found these methods to be ineffective in our setting, and they do not come with either soundness or completeness guarantees. There is work using decision trees to synthesize STL formulas [1,14,44,48], but these operate on a restricted subset of STL, namely Boolean combinations of a fixed set of template formulas. This restriction prevents these approaches from synthesizing temporal structure, which is a key component of the queries in our domains. Finally, there has been work on active learning of STL formulae using decision trees [48], but it assumes the ability to query for equivalence between a particular STL formula and the ground truth, which is not possible in our setting.

Synthesizing Constants. There is work on synthesizing parameters of programs using counterexampled-guided inductive synthesis and different theory solvers, including Fourier-Motzkin variable elimination and an SMT solver [2]. Though our synthesis objective can be encoded in the theory of linear arithmetic, it is extremely large, and we have found such solvers to be ineffective in practice.

Querying Video Data. There has been recent work on querying object detections and trajectories in video data [5,7,8,28,40–42,54]. The main difference is our focus on synthesis; in addition, these approaches focus on SQL-like operators such as select, inner-join, group-by, etc., over predefined predicates, which cannot capture compositions such as the sequencing and iteration operators in our language, which are necessary for identifying more complex behaviors.

Neurosymbolic Models. There has been recent work on leveraging program synthesis in the context of machine learning. For instance, there has been work on using programs to represent high-level structure in images [21–23,74,84], for reinforcement learning [9,34,79,80], and for querying websites [18]; in contrast, we use programs to classify trajectories. The most closely related work is on synthesizing functional programs operating over lists [67,76]. Our language includes key constructs not included in their languages. Most importantly, we

include sequencing; in their functional language, such an operator would need to be represented as a nested series of if-then-else operators. In addition, their language does not support predicates that match subsequences; while such a predicate could be added, none of their operators can compose such predicates.

Quantitative Synthesis. There has been work on program synthesis with quantitative properties—e.g., on synthesis for producing optimized code [37,57,65], for approximate computing [15,52], for probabilistic programming [56], and for embedded control [17]. These approaches largely focus on search-based synthesis, either using constraint optimization [52], continuous optimization [17], enumerative search [15,57], or stochastic search [37,56,65]. While we leverage ideas from this literature, our quantitative semantics based pruning strategy is novel.

Quantitative Semantics. Our quantitative semantics is similar to the "robustness degree" [25] of a temporal logic formula. The difference is that, by adjusting the denotations of the base predicates, our quantitative semantics gives a parameter on the satisfaction boundary. More broadly, there has been work on quantitative semantics for temporal logic for robust constraint satisfaction [20,25,73], and to guide reinforcement learning [39]. There has been work on quantitative regular expressions (QREs) [4], though in general, QREs cannot be efficiently evaluated due to their nondeterminism, and our language is restricted to ensure efficient computation. There has been work on synthesizing QREs for network traffic classification [68], using binary search to compute decision thresholds. Similarly, there has been work using the Viterbi semiring to obtain quantitative semantics for Datalog programs [69], which they use in conjunction with gradient descent to learn the rules of the Datalog program. In contrast, we use our quantitative semantics to efficiently prune the parameter search space in a provably correct way. Finally, there has been work on using GPUs to evaluate regular expressions [55]; however, they focus on regular expressions over strings.

Query Languages. Our language is closely related to both signal temporal logic (STL) [50] and Kleene algebras with tests (KAT) [46]. In particular, it can straightforwardly be extended to subsume both (see Appendix A for details), and our pruning strategy applies to this extended language. The addition of Kleene star, required to subsume KAT, worsens the evaluation time. STL has been used to monitor safety requirements for autonomous vehicles [32]. Spatio-Temporal Perception Logic (SPTL) is an extension of STL to support spatial reasoning [31]. Many of its operators are monotone, and thus would benefit from our algorithm. Scenic [27] is a DSL for creating static and dynamic driving scenes, but its focus is on generating scenes rather than querying for behaviors.

7 Conclusion

We have proposed a novel framework called Quivr for synthesizing queries over video trajectory data. Our language is similar to KAT and STL, but supports conjunction and sequencing over multi-step predicates. Given only a few examples, Quivr efficiently synthesizes trajectory queries consistent with those examples. A key contribution of our approach is the use of a quantitative semantics to

prune the parameter search space, yielding a 5.0× speedup over the state-of-the-art. In our evaluation, we demonstrate that Quivr effectively synthesizes queries to identify interesting driving behaviors, and that our optimizations dramatically reduce synthesis time.

Acknowledgements. We thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF Award CCF-1910769, NSF Award CCF-1917852, and ARO Award W911NF-20-1-0080.

References

1. Aasi, E., Vasile, C.I., Bahreinian, M., Belta, C.: Classification of time-series data using boosted decision trees. In: 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 1263–1268 (2022). https://doi.org/10.1109/IROS47612.2022.9982105
2. Abate, A., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Counterexample guided inductive synthesis modulo theories. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 270–288. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_15
3. Alur, R., et al.: Syntax-guided synthesis. IEEE (2013)
4. Alur, R., Mamouras, K., Ulus, D.: Derivatives of quantitative regular expressions. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 75–95. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_4
5. Bastani, F., et al.: SkyQuery: optimizing video queries over UAVs (2019)
6. Bastani, F., et al.: SkyQuery: optimizing video queries over UAVs. In: Onward! (2021)
7. Bastani, F., et al.: MIRIS: fast object track queries in video. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 1907–1921 (2020)
8. Bastani, F., Moll, O., Madden, S.: Vaas: video analytics at scale. Proc. VLDB Endow. **13**(12), 2877–2880 (2020)
9. Bastani, O., Pu, Y., Solar-Lezama, A.: Verifiable reinforcement learning via policy extraction. In: Advances in Neural Information Processing Systems, pp. 2494–2504 (2018)
10. Bergmann, P., Meinhardt, T., Leal-Taixe, L.: Tracking without bells and whistles. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 941–951 (2019)
11. Bertinetto, L., Valmadre, J., Henriques, J.F., Vedaldi, A., Torr, P.H.S.: Fully-convolutional Siamese networks for object tracking. In: Hua, G., Jégou, H. (eds.) ECCV 2016. LNCS, vol. 9914, pp. 850–865. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48881-3_56
12. Betke, M., Hirsh, D.E., Bagchi, A., Hristov, N.I., Makris, N.C., Kunz, T.H.: Tracking large variable numbers of objects in clutter. In: 2007 IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–8. IEEE (2007)
13. Bock, J., Krajewski, R., Moers, T., Runde, S., Vater, L., Eckstein, L.: The inD dataset: a drone dataset of naturalistic road user trajectories at German intersections. arXiv preprint arXiv:1911.07602 (2019)

14. Bombara, G., Belta, C.: Offline and online learning of signal temporal logic formulae using decision trees. ACM Trans. Cyber-Phys. Syst. **5**(3) (2021). https://doi.org/10.1145/3433994
15. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 775–788 (2016)
16. Chang, M.F., et al.: Argoverse: 3D tracking and forecasting with rich maps. In: Conference on Computer Vision and Pattern Recognition (CVPR) (2019)
17. Chaudhuri, S., Clochard, M., Solar-Lezama, A.: Bridging Boolean and quantitative synthesis using smoothed proof search. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 207–220 (2014)
18. Chen, Q., Lamoreaux, A., Wang, X., Durrett, G., Bastani, O., Dillig, I.: Web question answering with neurosymbolic program synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 328–343 (2021)
19. Dasgupta, S.: Analysis of a greedy active learning strategy. In: Advances in Neural Information Processing Systems, pp. 337–344 (2005)
20. Deshmukh, J.V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., Seshia, S.A.: Robust online monitoring of signal temporal logic. Formal Meth. Syst. Des. **51**(1), 5–30 (2017). https://doi.org/10.1007/s10703-017-0286-7
21. Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., Solar-Lezama, A.: Write, execute, assess: program synthesis with a repl. In: Advances in Neural Information Processing Systems, pp. 9169–9178 (2019)
22. Ellis, K., Ritchie, D., Solar-Lezama, A., Tenenbaum, J.: Learning to infer graphics programs from hand-drawn images. In: Advances in Neural Information Processing Systems, pp. 6059–6068 (2018)
23. Ellis, K., Solar-Lezama, A., Tenenbaum, J.: Unsupervised learning by program synthesis. In: Advances in Neural Information Processing Systems, pp. 973–981 (2015)
24. Ergurtuna, M., Gol, E.A.: An efficient formula synthesis method with past signal temporal logic. IFAC-PapersOnLine **52**(11), 43–48 (2019). https://doi.org/10.1016/j.ifacol.2019.09.116. https://www.sciencedirect.com/science/article/pii/S2405896319307451
25. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoret. Comput. Sci. **410**(42), 4262–4291 (2009)
26. Franco, L., Placidi, L., Giuliari, F., Hasan, I., Cristani, M., Galasso, F.: Under the hood of transformer networks for trajectory forecasting. Pattern Recogn. **138**, 109372 (2023). https://doi.org/10.1016/j.patcog.2023.109372. https://www.sciencedirect.com/science/article/pii/S0031320323000730
27. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 63–78 (2019)
28. Fu, D.Y., et al.: Rekall: specifying video events using compositions of spatiotemporal labels. arXiv preprint arXiv:1910.02993 (2019)
29. Giuliari, F., Hasan, I., Cristani, M., Galasso, F.: Transformer networks for trajectory forecasting (2020)
30. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

31. Hekmatnejad, M., Hoxha, B., Deshmukh, J.V., Yang, Y., Fainekos, G.: Formalizing and evaluating requirements of perception systems for automated vehicles using spatio-temporal perception logic (2022)
32. Hekmatnejad, M., et al.: Encoding and monitoring responsibility sensitive safety rules for automated vehicles in signal temporal logic. In: Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2019. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3359986.3361203
33. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
34. Inala, J.P., Bastani, O., Tavares, Z., Solar-Lezama, A.: Synthesizing programmatic policies that inductively generalize. In: International Conference on Learning Representations (2019)
35. Jha, S., Tiwari, A., Seshia, S.A., Sahai, T., Shankar, N.: TeLEx: learning signal temporal logic from positive examples using tightness metric. Formal Meth. Syst. Des. **54**(3), 364–387 (2019). https://doi.org/10.1007/s10703-019-00332-1
36. Ji, R., Liang, J., Xiong, Y., Zhang, L., Hu, Z.: Question selection for interactive program synthesis. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1143–1158 (2020)
37. Jia, Z., Thomas, J., Warszawski, T., Gao, M., Zaharia, M., Aiken, A.: Optimizing DNN computation with relaxed graph substitutions. In: Proceedings of the 2nd Conference on Systems and Machine Learning, SysML 2019 (2019)
38. Jin, X., Donzé, A., Deshmukh, J.V., Seshia, S.A.: Mining requirements from closed-loop control models. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **34**(11), 1704–1717 (2015). https://doi.org/10.1109/TCAD.2015.2421907
39. Jothimurugan, K., Alur, R., Bastani, O.: A composable specification language for reinforcement learning tasks. In: Advances in Neural Information Processing Systems, pp. 13041–13051 (2019)
40. Kang, D., Bailis, P., Zaharia, M.: Blazeit: optimizing declarative aggregation and limit queries for neural network-based video analytics. Proc. VLDB Endow. **13**(4), 533–546 (2019)
41. Kang, D., Mathur, A., Veeramacheneni, T., Bailis, P., Zaharia, M.: Jointly optimizing preprocessing and inference for DNN-based visual analytics. arXiv preprint arXiv:2007.13005 (2020)
42. Kang, D., Raghavan, D., Bailis, P., Zaharia, M.: Model assertions for monitoring and improving ml model. arXiv preprint arXiv:2003.01668 (2020)
43. Kitani, K.M., Ziebart, B.D., Bagnell, J.A., Hebert, M.: Activity forecasting. In: Fitzgibbon, A., Lazebnik, S., Perona, P., Sato, Y., Schmid, C. (eds.) ECCV 2012. LNCS, vol. 7575, pp. 201–214. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33765-9_15
44. Kong, Z., Jones, A., Belta, C.: Temporal logics for learning and detection of anomalous behavior. IEEE Trans. Autom. Control **62**(3), 1210–1222 (2017). https://doi.org/10.1109/TAC.2016.2585083
45. Kong, Z., Jones, A., Medina Ayala, A., Aydin Gol, E., Belta, C.: Temporal logic inference for classification and prediction from data. In: Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control, HSCC 2014, pp. 273–282. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2562059.2562146
46. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. (TOPLAS) **19**(3), 427–443 (1997)

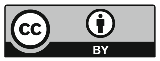
47. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017)
48. Linard, A., Tumova, J.: Active learning of signal temporal logic specifications. In: 2020 IEEE 16th International Conference on Automation Science and Engineering (CASE), pp. 779–785 (2020). https://doi.org/10.1109/CASE48305.2020.9216778
49. Maler, O.: Learning monotone partitions of partially-ordered domains (Work in Progress), July 2017. Working paper or preprint. https://hal.science/hal-01556243
50. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT - 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
51. Mell, S., Bastani, F., Zdancewic, S., Bastani, O.: Synthesizing trajectory queries from examples. Technical report, MS-CIS-23-02, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, July 2023
52. Misailovic, S., Carbin, M., Achour, S., Qi, Z., Rinard, M.C.: Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. ACM SIGPLAN Not. **49**(10), 309–328 (2014)
53. Mohammadinejad, S., Deshmukh, J.V., Puranic, A.G., Vazquez-Chanlatte, M., Donzé, A.: Interpretable classification of time-series data using efficient enumerative techniques. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control, HSCC 2020, Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3365365.3382218
54. Moll, O., Bastani, F., Madden, S., Stonebraker, M., Gadepally, V., Kraska, T.: ExSample: efficient searches on video repositories through adaptive sampling. arXiv preprint arXiv:2005.09141 (2020)
55. Naghmouchi, J., Scarpazza, D.P., Berekovic, M.: Small-ruleset regular expression matching on GPGPUs: quantitative performance analysis and optimization. In: Proceedings of the 24th ACM International Conference on Supercomputing, pp. 337–348 (2010)
56. Nori, A.V., Ozair, S., Rajamani, S.K., Vijaykeerthy, D.: Efficient synthesis of probabilistic programs. ACM SIGPLAN Not. **50**(6), 208–217 (2015)
57. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Scaling up superoptimization. In: Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 297–310 (2016)
58. Preiss, J.A., Honig, W., Sukhatme, G.S., Ayanian, N.: Crazyswarm: a large nanoquadcopter swarm. In: 2017 IEEE International Conference on Robotics and Automation (ICRA), pp. 3299–3304. IEEE (2017)
59. Redmon, J., Farhadi, A.: YOLOv3: an incremental improvement. arXiv preprint arXiv:1804.02767 (2018)
60. Ren, S., He, K., Girshick, R., Sun, J.: Fast R-CNN: towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems, pp. 91–99 (2015)
61. Robicquet, A., Sadeghian, A., Alahi, A., Savarese, S.: Learning social etiquette: human trajectory understanding in crowded scenes. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9912, pp. 549–565. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46484-8_33
62. Roy, N., McCallum, A.: Toward optimal active learning through sampling estimation of error reduction. In: International Conference on Machine Learning (2001)

63. Sadigh, D., Sastry, S.S., Seshia, S.A., Dragan, A.: Information gathering actions over human internal state. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 66–73. IEEE (2016)
64. Sadigh, D., Sastry, S., Seshia, S.A., Dragan, A.D.: Planning for autonomous cars that leverage effects on human actions. In: Robotics: Science and Systems, vol. 2. Ann Arbor, MI, USA (2016)
65. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. ACM SIGARCH Comput. Arch. News **41**(1), 305–316 (2013)
66. Schmerling, E., Leung, K., Vollprecht, W., Pavone, M.: Multimodal probabilistic model-based planning for human-robot interaction. In: 2018 IEEE International Conference on Robotics and Automation (ICRA), pp. 1–9. IEEE (2018)
67. Shah, A., Zhan, E., Sun, J., Verma, A., Yue, Y., Chaudhuri, S.: Learning differentiable programs with admissible neural heuristics. In: Advances in Neural Information Processing Systems, vol. 33 (2020)
68. Shi, L., Li, Y., Loo, B.T., Alur, R.: Network traffic classification by program synthesis. In: TACAS 2021. LNCS, vol. 12651, pp. 430–448. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_23
69. Si, X., Raghothaman, M., Heo, K., Naik, M.: Synthesizing datalog programs using numerical relaxation. In: IJCAI (2019)
70. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484–489 (2016)
71. Solar-Lezama, A.: Program Synthesis by Sketching. University of California, Berkeley (2008)
72. Sun, J.J., et al.: The MABe22 benchmarks for representation learning of multi-agent behavior. June 2022. https://doi.org/10.22002/D1.20186
73. Tabuada, P., Neider, D.: Robust linear temporal logic. In: Computer Science Logic 2016 (2016)
74. Tian, Y., et al.: Learning to infer and execute 3D shape programs. arXiv preprint arXiv:1901.02875 (2019)
75. Tweed, D., Calway, A.: Tracking multiple animals in wildlife footage. In: Object Recognition Supported by User Interaction for Service Robots, vol. 2, pp. 24–27. IEEE (2002)
76. Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C., Chaudhuri, S.: HOUDINI: lifelong learning as program synthesis. In: Advances in Neural Information Processing Systems, pp. 8687–8698 (2018)
77. Vaswani, A., et al.: Attention is all you need. In: Guyon, I., Luxburg, U.V., et al. (eds.) Advances in Neural Information Processing Systems, vol. 30. Curran Associates, Inc. (2017). https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
78. Vazquez-Chanlatte, M., Ghosh, S., Deshmukh, J.V., Sangiovanni-Vincentelli, A., Seshia, S.A.: Time-series learning using monotonic logical properties. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 389–405. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_22
79. Verma, A., Le, H., Yue, Y., Chaudhuri, S.: Imitation-projected programmatic reinforcement learning. In: Advances in Neural Information Processing Systems, pp. 15752–15763 (2019)
80. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: ICML (2018)
81. Weinstein, A., Cho, A., Loianno, G., Kumar, V.: Visual inertial odometry swarm: an autonomous swarm of vision-based quadrotors. IEEE Robot. Autom. Lett. **3**(3), 1801–1807 (2018)

82. Wishart, J., et al.: Driving safety performance assessment metrics for ads-equipped vehicles. SAE Technical Paper 2(2020-01-1206) (2020)
83. Wojke, N., Bewley, A., Paulus, D.: Simple online and realtime tracking with a deep association metric. In: 2017 IEEE International Conference on Image Processing (ICIP), pp. 3645–3649. IEEE (2017)
84. Young, H., Bastani, O., Naik, M.: Learning neurosymbolic generative models via program synthesis. In: ICML (2019)
85. Zhan, E., Tseng, A., Yue, Y., Swaminathan, A., Hausknecht, M.: Learning calibratable policies using programmatic style-consistency. In: ICML (2020)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Author Index

A

Abdulla, Parosh Aziz I-184
Akshay, S. I-266, I-367, III-86
Albert, Elvira III-176
Alistarh, Dan I-156
Alur, Rajeev I-415
Amilon, Jesper III-281
Amir, Guy II-438
An, Jie I-62
Anand, Ashwani I-436
Andriushchenko, Roman III-113
Apicelli, Andrew I-27
Arcaini, Paolo I-62
Asada, Kazuyuki III-40
Ascari, Flavio II-41
Atig, Mohamed Faouzi I-184

B

Badings, Thom III-62
Barrett, Clark II-163, III-154
Bastani, Favyen I-459
Bastani, Osbert I-415, I-459
Bayless, Sam I-27
Becchi, Anna II-288
Beutner, Raven II-309
Bisping, Benjamin I-85
Blicha, Martin II-209
Bonchi, Filippo II-41
Bork, Alexander III-113
Braught, Katherine I-351
Britikov, Konstantin II-209
Brown, Fraser III-154
Bruni, Roberto II-41
Bucev, Mario III-398

C

Calinescu, Radu I-289
Češka, Milan III-113
Chakraborty, Supratik I-367
Chatterjee, Krishnendu III-16, III-86
Chaudhuri, Swarat III-213
Chechik, Marsha III-374
Chen, Hanyue I-40
Chen, Taolue III-255
Chen, Yu-Fang III-139
Choi, Sung Woo II-397
Chung, Kai-Min III-139
Cimatti, Alessandro II-288
Cosler, Matthias II-383
Couillard, Eszter III-437
Czerner, Philipp III-437

D

Dardik, Ian I-326
Das, Ankush I-27
David, Cristina III-459
Dongol, Brijesh I-206
Dreossi, Tommaso I-253
Dutertre, Bruno II-187

E

Eberhart, Clovis III-40
Esen, Zafer III-281
Esparza, Javier III-437

F

Farzan, Azadeh I-109
Fedorov, Alexander I-156
Feng, Nick III-374
Finkbeiner, Bernd II-309
Fremont, Daniel J. I-253
Frenkel, Hadar II-309
Fu, Hongfei III-16
Fu, Yu-Fu II-227, III-329

G

Gacek, Andrew I-27
Garcia-Contreras, Isabel II-64

© The Editor(s) (if applicable) and The Author(s) 2023
C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 485–488, 2023.
https://doi.org/10.1007/978-3-031-37706-8

Gastin, Paul I-266
Genaim, Samir III-176
Getir Yaman, Sinem I-289
Ghosh, Shromona I-253
Godbole, Adwait I-184
Goel, Amit II-187
Goharshady, Amir Kafshdar III-16
Goldberg, Eugene II-110
Gopinath, Divya I-289
Gori, Roberta II-41
Govind, R. I-266
Govind, V. K. Hari II-64
Griggio, Alberto II-288, III-423
Guilloud, Simon III-398
Gurfinkel, Arie II-64
Gurov, Dilian III-281

H
Hahn, Christopher II-383
Hasuo, Ichiro I-62, II-41, III-40
Henzinger, Thomas A. II-358
Hofman, Piotr I-132
Hovland, Paul D. II-265
Hückelheim, Jan II-265

I
Imrie, Calum I-289

J
Jaganathan, Dhiva I-27
Jain, Sahil I-367
Jansen, Nils III-62
Jeż, Artur II-18
Johannsen, Chris III-483
Johnson, Taylor T. II-397
Jonáš, Martin III-423
Jones, Phillip III-483
Joshi, Aniruddha R. I-266
Jothimurugan, Kishor I-415
Junges, Sebastian III-62, III-113

K
Kang, Eunsuk I-326
Karimi, Mahyar II-358
Kashiwa, Shun I-253
Katoen, Joost-Pieter III-113
Katz, Guy II-438
Kempa, Brian III-483
Kiesl-Reiter, Benjamin II-187
Kim, Edward I-253
Kirchner, Daniel III-176
Kokologiannakis, Michalis I-230
Kong, Soonho II-187
Kori, Mayuko II-41
Koval, Nikita I-156
Kremer, Gereon II-163
Křetínský, Jan I-390
Krishna, Shankaranarayanan I-184
Kueffner, Konstantin II-358
Kunčak, Viktor III-398

L
Lafortune, Stéphane I-326
Lahav, Ori I-206
Lengál, Ondřej III-139
Lette, Danya I-109
Li, Elaine III-350
Li, Haokun II-87
Li, Jianwen II-288
Li, Yangge I-351
Li, Yannan II-335
Lidström, Christian III-281
Lin, Anthony W. II-18
Lin, Jyun-Ao III-139
Liu, Jiaxiang II-227, III-329
Liu, Mingyang III-255
Liu, Zhiming I-40
Lopez, Diego Manzanas II-397
Lotz, Kevin II-187
Luo, Ziqing II-265

M
Maayan, Osher II-438
Macák, Filip III-113
Majumdar, Rupak II-187, III-3, III-437
Mallik, Kaushik II-358, III-3
Mangal, Ravi I-289
Marandi, Ahmadreza III-62
Markgraf, Oliver II-18
Marmanis, Iason I-230
Marsso, Lina III-374
Martin-Martin, Enrique III-176
Mazowiecki, Filip I-132
Meel, Kuldeep S. II-132
Meggendorfer, Tobias I-390, III-86
Meira-Góes, Rômulo I-326
Mell, Stephen I-459
Mendoza, Daniel II-383

Metzger, Niklas II-309
Meyer, Roland I-170
Mi, Junri I-40
Milovančević, Dragana III-398
Mitra, Sayan I-351

N
Nagarakatte, Santosh III-226
Narayana, Srinivas III-226
Nayak, Satya Prakash I-436
Niemetz, Aina II-3
Nowotka, Dirk II-187

O
Offtermatt, Philip I-132
Opaterny, Anton I-170
Ozdemir, Alex II-163, III-154

P
Padhi, Saswat I-27
Păsăreanu, Corina S. I-289
Peng, Chao I-304
Perez, Mateo I-415
Preiner, Mathias II-3
Prokop, Maximilian I-390
Pu, Geguang II-288

R
Reps, Thomas III-213
Rhea, Matthew I-253
Rieder, Sabine I-390
Rodríguez, Andoni III-305
Roy, Subhajit III-190
Rozier, Kristin Yvonne III-483
Rümmer, Philipp II-18, III-281
Rychlicki, Mateusz III-3

S
Sabetzadeh, Mehrdad III-374
Sánchez, César III-305
Sangiovanni-Vincentelli, Alberto L. I-253
Schapira, Michael II-438
Schmitt, Frederik II-383
Schmuck, Anne-Kathrin I-436, III-3
Seshia, Sanjit A. I-253
Shachnai, Matan III-226
Sharma, Vaibhav I-27
Sharygina, Natasha II-209
Shen, Keyi I-351
Shi, Xiaomu II-227, III-329
Shoham, Sharon II-64
Siegel, Stephen F. II-265
Sistla, Meghana III-213
Sokolova, Maria I-156
Somenzi, Fabio I-415
Song, Fu II-413, III-255
Soudjani, Sadegh III-3
Srivathsan, B. I-266
Stanford, Caleb II-241
Stutz, Felix III-350
Su, Yu I-40
Sun, Jun II-413
Sun, Yican III-16

T
Takhar, Gourav III-190
Tang, Xiaochao I-304
Tinelli, Cesare II-163
Topcu, Ufuk III-62
Tran, Hoang-Dung II-397
Tripakis, Stavros I-326
Trippel, Caroline II-383
Trivedi, Ashutosh I-415
Tsai, Ming-Hsien II-227, III-329
Tsai, Wei-Lun III-139
Tsitelov, Dmitry I-156

V
Vafeiadis, Viktor I-230
Vahanwala, Mihir I-184
Veanes, Margus II-241
Vin, Eric I-253
Vishwanathan, Harishankar III-226

W
Waga, Masaki I-3
Wahby, Riad S. III-154
Wang, Bow-Yaw II-227, III-329
Wang, Chao II-335
Wang, Jingbo II-335
Wang, Meng III-459
Watanabe, Kazuki III-40
Wehrheim, Heike I-206
Whalen, Michael W. I-27
Wies, Thomas I-170, III-350

Wolff, Sebastian I-170
Wu, Wenhao II-265

X
Xia, Bican II-87
Xia, Yechuan II-288

Y
Yadav, Raveesh I-27
Yang, Bo-Yin II-227, III-329
Yang, Jiong II-132
Yang, Zhengfeng I-304
Yu, Huafeng I-289
Yu, Yijun III-459
Yue, Xiangyu I-253

Z
Zdancewic, Steve I-459
Zelazny, Tom II-438
Zeng, Xia I-304
Zeng, Zhenbing I-304
Zhang, Hanliang III-459
Zhang, Li I-304
Zhang, Miaomiao I-40
Zhang, Pei III-483
Zhang, Yedi II-413
Zhang, Zhenya I-62
Zhao, Tianqi II-87
Zhu, Haoqing I-351
Žikelić, Đorđe III-86
Zufferey, Damien III-350