ARCoSS

LNCS 12078

Armin Biere
David Parker (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

26th International Conference, TACAS 2020
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020
Dublin, Ireland, April 25–30, 2020, Proceedings, Part I

Part I

ETAPS
EUROPEAN JOINT CONFERENCES ON THEORY & PRACTICE OF SOFTWARE

Springer Open

Armin Biere · David Parker (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

26th International Conference, TACAS 2020
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020
Dublin, Ireland, April 25–30, 2020
Proceedings, Part I

Springer Open

Editors
Armin Biere
Johannes Kepler University
Linz, Austria

David Parker
University of Birmingham
Birmingham, UK

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-45189-9 ISBN 978-3-030-45190-5 (eBook)
https://doi.org/10.1007/978-3-030-45190-5

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

ETAPS Foreword

Welcome to the 23rd ETAPS! This is the first time that ETAPS took place in Ireland in its beautiful capital Dublin.

ETAPS 2020 was the 23rd instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of four conferences: ESOP, FASE, FoSSaCS, and TACAS. Each conference has its own Program Committee (PC) and its own Steering Committee (SC). The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations of programming language developments, analysis tools, and formal approaches to software engineering. Organizing these conferences in a coherent, highly synchronized conference program enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences. On the weekend before the main conference, numerous satellite workshops took place that attracted many researchers from all over the globe. Also, for the second time, an ETAPS Mentoring Workshop was organized. This workshop is intended to help students early in the program with advice on research, career, and life in the fields of computing that are covered by the ETAPS conference.

ETAPS 2020 received 424 submissions in total, 129 of which were accepted, yielding an overall acceptance rate of 30.4%. I thank all the authors for their interest in ETAPS, all the reviewers for their reviewing efforts, the PC members for their contributions, and in particular the PC (co-)chairs for their hard work in running this entire intensive process. Last but not least, my congratulations to all authors of the accepted papers!

ETAPS 2020 featured the unifying invited speakers Scott Smolka (Stony Brook University) and Jane Hillston (University of Edinburgh) and the conference-specific invited speakers (ESOP) Işıl Dillig (University of Texas at Austin) and (FASE) Willem Visser (Stellenbosch University). Invited tutorials were provided by Erika Ábrahám (RWTH Aachen University) on the analysis of hybrid systems and Madhusudan Parthasarathy (University of Illinois at Urbana-Champaign) on combining Machine Learning and Formal Methods. On behalf of the ETAPS 2020 attendants, I thank all the speakers for their inspiring and interesting talks!

ETAPS 2020 took place in Dublin, Ireland, and was organized by the University of Limerick and Lero. ETAPS 2020 is further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology). The local organization team consisted of Tiziana Margaria (general chair, UL and Lero), Vasileios Koutavas (Lero@UCD), Anila Mjeda (Lero@UL), Anthony Ventresque (Lero@UCD), and Petros Stratis (Easy Conferences).

The ETAPS Steering Committee (SC) consists of an Executive Board, and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST. The Executive Board consists of Holger Hermanns (Saarbrücken), Marieke Huisman (chair, Twente), Joost-Pieter Katoen (Aachen and Twente), Jan Kofron (Prague), Gerald Lüttgen (Bamberg), Tarmo Uustalu (Reykjavik and Tallinn), Caterina Urban (Inria, Paris), and Lenore Zuck (Chicago).

Other members of the SC are: Armin Biere (Linz), Jordi Cabot (Barcelona), Jean Goubault-Larrecq (Cachan), Jan-Friso Groote (Eindhoven), Esther Guerra (Madrid), Jurriaan Hage (Utrecht), Reiko Heckel (Leicester), Panagiotis Katsaros (Thessaloniki), Stefan Kiefer (Oxford), Barbara König (Duisburg), Fabrice Kordon (Paris), Jan Kretinsky (Munich), Kim G. Larsen (Aalborg), Tiziana Margaria (Limerick), Peter Müller (Zurich), Catuscia Palamidessi (Palaiseau), Dave Parker (Birmingham), Andrew M. Pitts (Cambridge), Peter Ryan (Luxembourg), Don Sannella (Edinburgh), Bernhard Steffen (Dortmund), Mariëlle Stoelinga (Twente), Gabriele Taentzer (Marburg), Christine Tasson (Paris), Peter Thiemann (Freiburg), Jan Vitek (Prague), Heike Wehrheim (Paderborn), Anton Wijs (Eindhoven), and Nobuko Yoshida (London).

I would like to take this opportunity to thank all speakers, attendants, organizers of the satellite workshops, and Springer for their support. I hope you all enjoyed ETAPS 2020. Finally, a big thanks to Tiziana and her local organization team for all their enormous efforts enabling a fantastic ETAPS in Dublin!

February 2020

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

TACAS 2020 was the 26th edition of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems conference series. TACAS 2020 was part of the 23rd European Joint Conferences on Theory and Practice of Software (ETAPS 2020). The conference was held at the Royal Marine Hotel in Dublin, Ireland, during April 25–30, 2020.

TACAS is a forum for researchers, developers, and users interested in rigorously based tools and algorithms for the construction and analysis of systems. The conference aims to bridge the gaps between different communities with this common interest and to support them in their quest to improve the utility, reliability, flexibility, and efficiency of tools and algorithms for building systems. TACAS solicited four types of submissions:

– *Research papers* advancing the theoretical foundations for the construction and analysis of systems
– *Case study papers* with an emphasis on a real-world setting
– *Regular tool papers* presenting a new tool, a new tool component, or novel extensions to an existing tool and requiring an artifact submission
– *Tool demonstration papers* focusing on the usage aspects of tools, also subject to the artifact submission requirement

This year 155 papers were submitted to TACAS, consisting of 111 research papers, 8 case study papers, 19 regular tool papers, and 17 tool demo papers. Individual authors were limited to a maximum of three submissions. Each paper was reviewed by at least three Program Committee (PC) members, who also provided feedback whether certain papers should go through a rebuttal process.

The chairs asked for 59 rebuttals, usually following such rebuttal recommendations by PC members. In parallel to PC reviewing, the Artifact Evaluation Committee (AEC) reviewed the artifacts. A formal summary review of this evaluation was made available to the PC members and taken into account in the discussion phase. The case study chair and the tools chair made sure that identical reviewing and selection criteria were applied within their respective class of papers. After this thorough reviewing, rebuttal and discussion phase, a total of 48 papers were accepted, including 31 research papers, 4 case study papers, 5 regular tool papers and 8 tool demo papers.

As in 2019, TACAS 2020 included an artifact evaluation (AE) for all types of papers. There were two rounds of the AE: for regular tool papers and tool demonstration papers AE was compulsory and artifacts had to be submitted to the first round. For research and case study papers, it was voluntary, and artifacts could be submitted to either the first or the second round. The results of the first round were communicated to the TACAS PC before their discussion phase so that the quality of the artifact could be considered prior to the TACAS decision making. Each artifact was evaluated independently by at least three reviewers. All accepted papers with accepted artifacts received a badge which is added to the title page of the respective paper if desired by the authors.

The AEC used a two-phase reviewing process: reviewers first performed an initial check to see whether the artifact was technically usable and whether the accompanying instructions were consistent, followed by a full evaluation of the artifact. The main criteria for artifact acceptance was consistency with the paper, with completeness, and documentation being handled in a more lenient manner as long as the artifact was useful overall.

In the first round, out of 44 artifact submissions, 29 were accepted and 15 were rejected. This corresponds to an acceptance rate of 66%. Out of the 36 artifacts for regular tool papers and tool demonstration papers, 25 artifacts were accepted and 11 artifacts were rejected resulting in an acceptance rate of 69%. In all but five cases, tool papers whose artifacts did not pass the evaluation were rejected. Those 5 artifacts were invited for submission in the second evaluation round and 3 of these artifacts were resubmitted and successfully evaluated. Overall, out of the 20 artifacts submitted to the second evaluation round, 17 were accepted and 3 were rejected resulting in an acceptance rate of 85%.

TACAS 2020 also hosted the 9th International Competition on Software Verification (SV-COMP 2020), chaired and organized by Dirk Beyer. The competition had again a high participation: 28 verification systems with developers from 11 countries were submitted for the systematic comparative evaluation, including 3 submissions from industry. Six teams contributed validators for verification witnesses. The TACAS proceedings includes the competition report and short papers describing 11 of the participating verification systems. These papers were reviewed by a separate SV-COMP program committee; each of the papers was assessed by at least three reviewers. Two sessions in the TACAS program were reserved for the presentation of the results: the summary by the SV-COMP chair and the participating tools by the developer teams in the first session, and the open community meeting in the second session.

We are grateful to everyone who helped to make TACAS 2020 a success. In particular, we would like to thank all PC members, external reviewers, and the members of the AEC for their detailed and informed reviews and for their discussions during the virtual PC and AEC meetings. The collection and selection of papers was organized through the EasyChair Conference System and the proceedings volumes were published with the help of Springer; we thank them all for their assistance. We also thank the SC for their advice, the Organizing Committee of ETAPS 2020 and its general chair (Tiziana Margaria) and the chair of the ETAPS Executive Board (Marieke Huisman).

March 2020

Armin Biere
David Parker
PC Chairs

Marijn Heule
Case Study Chair

Falk Howar
Tools Chair

Dirk Beyer
Competition Chair

Arnd Hartmanns
Martina Seidl
AEC Chairs

Organization

Program Committee

Christel Baier	TU Dresden, Germany
Ezio Bartocci	Vienna University of Technology, Austria
Dirk Beyer	LMU Munich, Germany
Armin Biere (Chair)	Johannes Kepler University Linz
Jasmin Blanchette	Vrije Universiteit Amsterdam, The Netherlands
Roderick Bloem	TU Graz, Austria
Hana Chockler	King’s College London, UK
Alessandro Cimatti	FBK-irst, Italy
Rance Cleaveland	University of Maryland, USA
Goran Frehse	Université Grenoble Alpes, France
Martin Fränzle	Carl von Ossietzky Univ. Oldenburg, Germany
Orna Grumberg	Technion - Israel Institute of Technology
Kim Guldstrand Larsen	Aalborg University, Denmark
Holger Hermanns	Universität des Saarlandes, Germany
Marijn Heule	Carnegie Mellon University, USA
Falk Howar	TU Clausthal, IPSSE, Germany
Benjamin Kiesl	CISPA Helmholtz Center for Inf. Security, Germany
Laura Kovacs	Vienna University of Technology, Austria
Jan Kretinsky	TU Munich, Germany
Wenchao Li	Boston University, USA
Ken McMillan	Microsoft, USA
Aina Niemetz	Stanford University, USA
Gethin Norman	University of Glasgow, UK
David Parker (Chair)	University of Birmingham, UK
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Nir Piterman	University of Gothenburg, Sweden
Kristin Yvonne Rozier	Iowa State University, USA
Philipp Ruemmer	Uppsala University, Sweden
Natasha Sharygina	Università della Svizzera italiana, Switzerland
Bernhard Steffen	TU Dortmund, Germany
Jan Strejček	Masaryk University, Czech Republic
Michael Tautschnig	Queen Mary University of London, UK
Jaco van de Pol	Aarhus University, Denmark
Tom van Dijk	University of Twente, The Netherlands
Christoph Wintersteiger	Microsoft, UK

Artifact Evaluation Committee

Pranav Ashok	TU Munich, Germany
Peter Backeman	Uppsala University, Sweden
Ismail Lahkim Bennani	Inria, France
Carlos E. Budde	University of Twente, The Netherlands
Karlheinz Friedberger	LMU Munich, Germany
Arnd Hartmanns (Chair)	University of Twente, The Netherlands
Jannik Hüls	Westfälische Wilhelms-Univ. Münster, Germany
Ahmed Irfan	Stanford University, USA
Martin Jonas	Masaryk University, Czech Republic
William Kavanagh	University of Glasgow, UK
Brian Kempa	Iowa State University, USA
Michaela Klauck	Universität des Saarlandes, Germany
Sascha Klüppelholz	TU Dresden, Germany
Bettina Könighofer	TU Graz, Austria
Sophie Lathouwers	University of Twente, The Netherlands
Florian Lonsing	Stanford University, USA
Juraj Major	Masaryk University, Czech Republic
Tobias Meggendorfer	TU Munich, Germany
Vince Molnar	Budapest Univ. of Tech. and Economics, Hungary
Alnis Murtovi	TU Dortmund, Germany
Chris Novakovic	University of Birmingham, UK
Nicola Paoletti	Royal Holloway University of London, UK
Tim Quatmann	RWTH Aachen University, Germany
Martina Seidl (Chair)	Johannes Kepler University Linz, Austria
Leander Tentrup	Universität des Saarlandes, Germany
Freark van der Berg	University of Twente, The Netherlands
Marcell Vazquez-Chanlatte	University of California at Berkeley, USA
Matthias Volk	RWTH Aachen University, Germany
Petar Vukmirovic	Vrije Universiteit Amsterdam, The Netherlands
Maximilian Weininger	TU Munich, Germany

SV-COMP – Program Committee and Jury

Dirk Beyer (Chair)	LMU Munich, Germany
Viktor Malík (2LS)	BUT Brno, Czech Republic
Lei Bu (BRICK)	Nanjing University, China
Michael Tautschnig (CBMC)	Amazon Web Services, UK
Willem Visser (COASTAL)	Stellenbosch University, South Africa
Vadim Mutilin (CPA-BAM-BnB)	ISP RAS, Russia
Martin Spiessl (CPA-Seq)	LMU Munich, Germany
Pavel Andrianov (CPALockator)	ISP RAS, Russia

Hernán Ponce de León (Dartagnan)	Bundeswehr University Munich, Germany
Henrich Lauko (DIVINE)	Masaryk University, Czechia
Felipe R. Monteiro (ESBMC)	Fed. Univ. of Amazonas, Brazil
Benjamin Quiring (GACAL)	Northeastern University, USA
Vaibhav Sharma (Java-Ranger)	University of Minnesota, USA
Philipp Ruemmer (JayHorn)	Uppsala University, Sweden
Peter Schrammel (JBMC)	University of Sussex, UK
Falk Howar (JDart)	TU Dortmund, Germany
Omar Inverso (Lazy-CSeq)	Gran Sasso Science Institute, Italy
Herbert Rocha (Map2Check)	Universidade Federal do Amazonas, Brazil
Philipp Berger (NITWIT)	RWTH Aachen, Germany
Cedric Richter (PeSCo)	Paderborn University, Germany
Saurabh Joshi (Pinaka)	IIT Hyderabad, India
Veronika Šoková (PredatorHP)	BUT Brno, Czech Republic
Willem Visser (SPF)	Amazon Web Services, USA
Marek Chalupa (Symbiotic)	Masaryk University, Czech Republic
Matthias Heizmann (UAutomizer)	University of Freiburg, Germany
Alexander Nutz (UKojak)	University of Freiburg, Germany
Daniel Dietsch (UTaipan)	University of Freiburg, Germany
Priyanka Darke (VeriAbs)	Tata Consultancy Services, India
Raveendra K. Medicherla (VeriFuzz)	Tata Consultancy Services, India
Liangze Yin (Yogar-CBMC)	Nat. Univ. of Defense Technology, China

Steering Committee

Bernhard Steffen (Chair)	TU Dortmund, Germany
Dirk Beyer	LMU Munich, Germany
Rance Cleaveland	University of Maryland, USA
Holger Hermanns	Universität des Saarlandes, Germany
Kim G. Larsen	Aalborg University, Denmark

Additional Reviewers

Alexandre Dit Sandretto, Julien
Asadi, Sepideh
Ashok, Pranav
Avigad, Jeremy
Baanen, Tim
Bacci, Giorgio
Bacci, Giovanni
Backeman, Peter
Bae, Kyungmin
Barbosa, Haniel
Bentkamp, Alexander
Berani Abdelwahab, Erzana
Biewer, Sebastian
Blahoudek, Fanda
Blicha, Martin
Bozga, Marius
Bozzano, Marco
Bønneland, Frederik M.
Cerna, David
Ceska, Milan
Chalupa, Marek
Chapoutot, Alexandre
Dierl, Simon
Dureja, Rohit
Ebrahimi, Masoud
Eisentraut, Julia
Endrullis, Jörg
Ernst, Gidon
Esen, Zafer
Fan, Jiameng
Fazekas, Katalin
Fedyukovich, Grigory
Fleury, Mathias
Fokkink, Wan
Forets, Marcelo
Freiberger, Felix
Frenkel, Hadar
Friedberger, Karlheinz
Frohme, Markus
Fu, Feisi
Fürnkranz, Johannes
Giacobbe, Mirco
Gjøl Jensen, Peter
Gossen, Frederik
Goudsmid, Ohad
Griggio, Alberto
Grover, Kush
Gutiérrez, Elena
Haaswijk, Winston
Hadžić, Vedad
Hahn, Ernst Moritz
Hansen, Mikkel
Hartmanns, Arnd
Hecking-Harbusch, Jesko
Hofmann, Jana
Holzner, Stephan
Hugunin, Jasper
Humenberger, Andreas
Hupel, Lars
Hyvärinen, Antti
Irfan, Ahmed
Jasper, Marc
Jaulin, Luc
Jensen, Mathias Claus
Jensen, Peter Gjøl
Jonas, Martin
Jonsson, Bengt
Jonáš, Martin
Kacianka, Severin
Kaminski, Benjamin Lucien
Kanav, Sudeep
Kempa, Brian
Khalimov, Ayrat
Kiourti, Panagiota
Klauck, Michaela
Klüppelholz, Sascha
Koenighofer, Bettina
Kopetzki, Dawid
Krcal, Pavel
Kröger, Paul
Kupferman, Orna
Köhl, Maximilian
Lahkim Bennani, Ismail
Legay, Axel
Lemberger, Thomas
Liang, Chencheng

Lorber, Florian
Ma, Meiyi
Major, Juraj
Mann, Makai
Marcovich, Ron
Marescotti, Matteo
Martins, Ruben
Meggendorfer, Tobias
Mikučionis, Marius
Mitsch, Stefan
Mover, Sergio
Mues, Malte
Murtovi, Alnis
Möhlmann, Eike
Mömke, Tobias
Müller, David
Narváez, David
Naujokat, Stefan
Oliveira da Costa, Ana
Otoni, Rodrigo
Pagel, Jens
Parlato, Gennaro
Paskevich, Andrei
Peppelman, Marijn
Perelli, Giuseppe
Pivoluska, Matej
Popescu, Andrei
Puch, Stefan
Putot, Sylvie
Rebola-Pardo, Adrián
Reynolds, Andrew
Rothenberg, Bat-Chen
Roveri, Marco
Rowe, Reuben
Rüthing, Oliver
Schilling, Christian
Shoukry, Yasser
Spießl, Martin
Srba, Jiri
Stankovic, Miroslav
Stierand, Ingo
Štill, Vladimír
Stjerna, Albin
Stock, Gregory
Stojic, Ivan
Theel, Oliver
Tian, Chun
Tonetta, Stefano
Trtík, Marek
van der Ploeg, Atze
Vom Dorff, Sebastian
Wardega, Kacper
Weininger, Maximilian
Wendler, Philipp
Wimmer, Simon
Winkels, Jan
Yolcu, Emre
Zeljić, Aleksandar
Zhou, Weichao

Contents – Part I

Contents – Part II

Program Verification

Software Verification with PDR: An Implementation of the State of the Art

Dirk Beyer[1] and Matthias Dangl[1]

LMU Munich, Germany

Abstract. Property-directed reachability (PDR) is a SAT/SMT-based reachability algorithm that incrementally constructs inductive invariants. After it was successfully applied to hardware model checking, several adaptations to software model checking have been proposed. We contribute a replicable and thorough comparative evaluation of the state of the art: We (1) implemented a standalone PDR algorithm and, as improvement, a PDR-based auxiliary-invariant generator for k-induction, and (2) performed an experimental study on the largest publicly available benchmark set of C verification tasks, in which we explore the effectiveness and efficiency of software verification with PDR. The main contribution of our work is to establish a reproducible baseline for ongoing research in the area by providing a well-engineered reference implementation and an experimental evaluation of the existing techniques.

Keywords: Software verification · Program analysis · Invariant generation · Property-directed reachability (PDR) · IC3 · k-Induction· VVT · CPAchecker

1 Introduction

Automatic software verification [24] is a broad research area with many success stories and large impact on technology that is applied in industry [2, 14, 27]. It complements other general approaches to ensure functional correctness, like software testing [31] and interactive software verification [3]. One large sub-area of automatic software verification includes algorithms and approaches that are based on SMT technology. Classic approaches like bounded model checking [10], predicate abstraction [1, 19], and k-induction [5, 26, 32] are well understood and evaluated; a recent survey [6] provides a uniform overview and sheds light on the differences of the algorithms. Property-directed reachability (PDR) [12] is a relatively recent (2011) approach that is not yet included in comparative evaluations that go beyond applying different implementations of the same or different techniques to a set of benchmark tasks, but additionally pair such experiments with a discussion of how the concepts can be expressed in a common formalism. The approach was originally applied to transition systems from hardware designs, but was also adapted to software verification [11, 12, 13, 15, 16, 25, 28, 29].

An extended version of this article is available as technical report [8].
A replication package is available on Zenodo [9].

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 3–21, 2020.
https://doi.org/10.1007/978-3-030-45190-5_1

While in theory, given the aforementioned body of work on the topic, the advantages and disadvantages of using PDR seem clear, we are interested in understanding the effect of applying PDR to a large set of verification tasks that were collected from academia and also from industrial software, such as the Linux kernel. To achieve this goal, we implemented one PDR adaptation for software verification, and another approach that integrates a PDR-like invariant-generation module into a k-induction approach.

PDR Adaptation for Software Verification. PDR is a model-checking algorithm that tries to construct an inductive safety invariant by incrementally learning clauses that are inductive relative to previously learned clauses. The clause-learning strategy is guided by counterexamples to induction, i.e., each time a proof of inductiveness fails, the algorithm attempts to learn a new clause to avoid the same counterexample to induction in the future. Originally, this algorithm was designed as a SAT-based technique for Boolean finite-state systems. Every adaptation of PDR to software verification therefore needs to consider how to effectively and efficiently handle the infinite state space and how to transfer the algorithm from SAT to SMT. Furthermore, the adaptation to software has to deal with the program counter.

PDR-like Invariant Generation. Whenever an induction-proof attempt fails with a counterexample, the counterexample describes a state s that can transition into a bad state (that violates the safety property), which means that in order to make the proof succeed, s must be removed from consideration by an auxiliary invariant. From this bad-state predecessor s, the clause-learning strategy of PDR proceeds to generate such an auxiliary invariant by applying the following two steps: (1) s is first generalized to a set of states C that all transition into a bad state; (2) an invariant is constructed that is (a) inductive relative to previously found invariants[1] and (b) at least strong enough to eliminate all states in C. If it fails to construct such an invariant and prove its inductiveness, then the steps are recursively re-applied to the counterexample obtained from the failed induction attempt.

We experimentally investigate two implementations of adaptations of PDR to software verification (CPAchecker-CTIGAR and Vvt-CTIGAR), as well as several combinations that use the PDR-like invariant-generation module that we designed and implemented for this study.

Example. Figure 1 shows an example C program (`eq2.c`) that contains four unsigned integer variables `w`, `x`, `y`, and `z`. In line `10`, the variable `w` is initialized to an unknown value via the input function `__VERIFIER_nondet_uint()`; then, its value is copied to `x` in line `11`. In line `12`, variable `y` is initialized with the value of `w + 1`, and in line `13`, variable `z` is initialized with the value of `x + 1`, such

[1] An assertion F is said to be inductive relative to an invariant Inv if Inv can be used as an auxiliary invariant for the proof of inductiveness $\forall s_j, s_{j+1} : F(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow F(s_{j+1})$ by conjoining Inv to the induction hypothesis $F(s_j)$, such that the modified induction query $\forall s_j, s_{j+1} : F(s_j) \boldsymbol{\wedge} \boldsymbol{Inv}(\mathbf{s_j}) \wedge T(s_j, s_{j+1}) \Rightarrow F(s_{j+1})$ allows a proof by induction to succeed. [12]

```
 1  extern void __VERIFIER_error() __attribute__
 ↪  ((__noreturn__));
 2  extern unsigned int __VERIFIER_nondet_uint(void);
 3  void __VERIFIER_assert(int cond) {
 4    if (!(cond)) {
 5      ERROR: __VERIFIER_error();
 6    }
 7    return;
 8  }
 9  int main(void) {
10    unsigned int w = __VERIFIER_nondet_uint();
11    unsigned int x = w;
12    unsigned int y = w + 1;
13    unsigned int z = x + 1;
14    while (__VERIFIER_nondet_uint()) {
15      y++;
16      z++;
17    }
18    __VERIFIER_assert(y == z);
19    return 0;
20  }
```

Fig. 1: Example C program eq2.c

that at this point, w and x are equal to each other, and y and z are also equal to each other. Then, from line 14 to line 17, a loop with a nondeterministic exit condition (and therefore an unknown number of iterations) increments in each iteration both variables y and z. Lastly, line 18 asserts that after the loop, y and z are (still) equal to each other. Since y and z are equal before the loop, and are always incremented together within the loop, the invariant $y = z$ is inductive. However, since there is no direct connection between y and z but only an indirect one via their shared dependency on w, naïve data-flow-based techniques may fail to find this invariant. In fact, we tried several configurations of the verification framework CPAchecker, and found that many of them fail to prove this program:

- Plain k-induction without auxiliary-invariant generation fails, because it never checks if $y = z$ is a loop invariant and instead only checks the reachability of the assertion failure (located after loop). The reachability of the assertion failure, in turn, depends on the nondeterministic loop-exit condition. Therefore we cannot conclude from "the assertion failure was not reached in k previous iterations" that "the assertion failure cannot be reached in the next iteration": In the absence of auxiliary invariants, a valid counterexample to this induction hypothesis would always be that in the previous iterations the assertion *condition* was in fact violated and an assertion *failure* was not reached only *because the loop was not exited.*
- A data-flow analysis based on the abstract domain of Boxes [21] fails, because it is not able to track variable equalities.
- A data-flow analysis based on a template Eq for tracking the equality of pairs of variables fails, because while it detects the invariant $w = x$, it is unable to

make the step to $y = z$ due to the inequalities between `w` and `y`, and `x` and `z`, respectively.
- For consistency with our evaluation, we also applied a data-flow analysis based on a template for tracking whether a variable is even or odd; obviously this is not useful for this program, and thus, this configuration also fails.
- Even combining the previous three techniques into a compound invariant generator that computes auxiliary invariants for k-induction does not yield a successful configuration for this verification task.
- The invariant generator KIPDR (the above-mentioned adaptation of PDR to k-induction, which we present in more detail in Sect. 3), however, detects the invariant $y = z$ and is therefore able to construct a proof by induction for this verification task.

We will now briefly sketch how KIPDR detects the invariant $y = z$ for the example verification task. At first, KIPDR attempts to prove by induction that when line `18` is reached, the assertion condition holds, which fails as discussed previously. However, this failed induction attempt yields a counterexample to induction where the values of `y` and `z` differ from each other, e.g., $y = 0 \wedge z = 1$, which is then generalized to $y \neq z$, i.e., a set of states that includes the concrete predecessor of a bad state from the counterexample, as well as many other states that would violate the assertion, if they were reachable themselves. Then, KIPDR attempts to find an inductive invariant that eliminates all of these states, and the attempt succeeds with the invariant $y = z$. Afterwards, KIPDR re-attempts its original induction proof to show that the assertion is never violated, which now succeeds due to the auxiliary invariant $y = z$.

Contributions. We present the following contributions:

- We implement one adaptation of PDR to software verification (based on [11, 20]) in the open-source verification framework CPAchecker, in order to establish a baseline for comparison with new ideas for improvement.
- We design and implement the algorithm KIPDR, as a new module for invariant generation that is based on ideas from PDR and use this module as an extension to a state-of-the-art approach to k-induction [5].
- We conduct a large experimental study to compare several tools and approaches to software verification using PDR as a component, to highlight strengths and weaknesses of PDR in the domain of software verification.
- We contribute a set of small examples that need invariants that are more difficult to obtain for standard data-flow-based approaches than the invariants necessary for programs in the large benchmark set.

Related Work. While PDR (also known as IC3 for its first implementation [12]) was introduced as a SAT-based algorithm for model checking finite-state Boolean transition systems [13], several approaches have since then been presented to extend it to SMT and to apply it to the verification of software models: PDR has been suggested as an interpolation engine for Impact, but experiments have shown that it is too expensive in the general case, and is most effective if only applied as a fall-back engine for cases where a cheaper interpolation engine fails to produce useful interpolants [15]. It also has been proposed to improve this approach by tracking control-flow locations explicitly instead of symbolically [28], thereby avoiding the problem that many iterations of the algorithm are spent only to learn the control flow, and this idea has later been extended by several improvements to the generalization step of PDR [29]. Another approach is to model the program using a Boolean abstraction, which has the advantage that it requires only few changes to the original algorithm, but the disadvantage that a refinement procedure is necessary to handle the spurious paths introduced by the abstraction: One such approach uses infeasible error paths (i.e., counterexample-guided abstraction refinement (CEGAR) [17]) to refine the abstraction [16], while another (CTIGAR) uses counterexamples to induction [11]; both of these refinement techniques use interpolation to obtain abstraction predicates; the latter of the two techniques is used in two of the configurations we compare in our evaluation (CPAchecker-CTIGAR and Vvt-CTIGAR [20]). A different extension of PDR to verify infinite-state systems that does not require abstraction refinement is property-directed k-induction [25], which increases the power of the induction checks used in PDR by applying k-induction instead of 1-induction, and which uses model-based generalization in addition to interpolation to reason about potentially-infinite sets of states. Unfortunately, support for effective model-based generalization is rare in SMT solvers [2], making this approach impractical. In contrast, our KIPDR algorithm presented in Sect. 3 only requires support for interpolation, which is available in several SMT solvers.

Despite this multitude of adaptations of PDR to infinite-state systems, most implementations in practice require their input to be encoded as transition systems. The only available software verifiers applicable to actual C programs and implement PDR-based techniques are CPAchecker [7], SeaHorn [23], and Vvt [20].

2 Background

In this section, we briefly introduce the algorithms PDR and k-induction, which provide the core concepts on which we base our ideas. In the following description of PDR and k-induction, we use the following notation: given the state variables s and s' within a state-transition system T that represents the program, predicate $I(s)$ denotes that s is an initial state, $T(s, s')$ that a transition from s to s' exists, and $P(s)$ that the safety property P holds for state s.

2.1 PDR

PDR maintains a list of k frames, where a frame F_i is a predicate that represents an overapproximation of all states reachable within at most $0 \leq i \leq k$ steps, and a queue of proof obligations, which guide invariant discovery towards invariants

[2] The implementation of the approach of property-directed k-induction combines two SMT solvers, because neither of them supports all features required by the technique.

relevant to prove the correctness of a safety property P. For a given state s, the notation $F_i(s)$ means that the predicate F_i holds for state s. The index i of a frame F_i is called its *level*, and the frame F_k is called the *frontier*, because it represents the largest overapproximation of reachable states computed by the algorithm [12]. The algorithm maintains the following invariants:

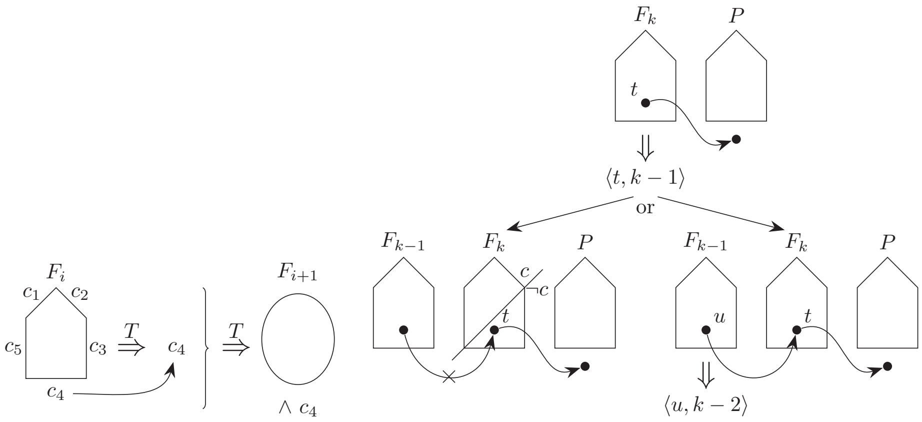
1. $F_0(s) = I(s)$, i.e., the first frame represents precisely the initial states.
2. $\forall i \in \{0, \ldots, k\} : F_i(s) \Rightarrow P(s)$, i.e., every frame contains only states that satisfy the safety property.
3. $\forall i \in \{0, \ldots, k-1\} : F_i(s) \Rightarrow F_{i+1}(s)$, i.e., a frame F_{i+1} represents in addition such states that are reachable with $i+1$ steps.
4. $\forall i \in \{0, \ldots, k-1\} : F_i(s) \wedge T(s, s') \Rightarrow F_{i+1}(s')$, i.e., each frame is inductive relative to its predecessor.

Using these data structures and algorithm invariants, the algorithm attempts to find either a counterexample to P or a 1-inductive invariant F_i such that $F_i(s) \Leftrightarrow F_{i+1}(s)$ for some level $i \in \{0, \ldots, k-1\}$. Until either of these potential outcomes is reached, PDR shifts back and forth between the following two phases:

1. If the set of states represented by the frontier F_k does not contain any predecessor states of $\neg P$-states (i.e., $\forall s_j, s_{j+1} : F_k(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow P(s_{j+1})$, called frontier-incrementation check), a new frontier F_{k+1} is created and initialized to P. Subsequently, the algorithm attempts to push forward [3] each predicate c of each frame F_i with $0 \leq i \leq k$ for which the consecution check $F_i(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow c(s_{j+1})$ holds (see Fig. 2a). If, on the other hand, the frontier-incrementation check fails, PDR extracts a $\neg P$-predecessor t in F_k, which represents a counterexample to induction (CTI), from the failed query as proof obligation $\langle t, k-1 \rangle$ (see Fig. 2b, top).
2. While the queue of proof obligations is not empty, PDR processes the queue by trying to prove for each proof obligation $\langle t, i \rangle$ that the CTI-state t is itself not reachable from F_i and therefore does not need to be considered as a relevant $\neg P$-predecessor. For this proof, PDR chooses some predicate $c \Rightarrow \neg t$ with $\forall s : F_i(s) \Rightarrow c(s)$. PDR then checks if c is inductive relative to F_i by performing the consecution check $F_i(s_j) \wedge c(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow c(s_{j+1})$. If the consecution check succeeds, the frames $F_1, \ldots, F_{i+1}$ can be strengthened by adding c, thus ruling out the CTI t in these frames for the future (see Fig. 2b, left). Also, unless $i = k$, we add a new proof obligation $\langle t, i+1 \rangle$ to the queue as an optimization to initiate forward propagation, because we expect that the CTI-state s would otherwise be rediscovered later at a higher level [11]. Otherwise, i.e., the consecution check does *not* succeed for clause c, the algorithm extracts a predecessor u of t from the failed consecution check, which is added as a new proof obligation $\langle u, i-1 \rangle$ if $i > 0$ and $t \wedge I$ is unsatisfiable (see Fig. 2b, right). Otherwise, u represents the initial state of a real counterexample to P.

An example of this algorithm is presented in a technical report [8, pp. 7–8]. A more detailed presentation of PDR can be found in the literature [12].

[3] By "push forward", we mean to add a predicate c from frame F_i to frame F_{i+1} [12].



(a) Consecution check makes sure to only conjoin to frame F_{i+1} such c_i from F_i that are inductive relative to F_i w.r.t. transition relation T

(b) If phase 1 results in a proof obligation $\langle t, k-1 \rangle$ (top), then phase 2 resolves either by strengthening F_k with c (left), or by creating a new (backwards) proof obligation $\langle u, k-2 \rangle$ (right); if the chain of proof obligations propagates back to the initial states, then a feasible error path is found

Fig. 2: Visualization of (a) the consecution check and (b) the handling of proof-obligations.

2.2 *k*-Induction

Like PDR, k-induction attempts to prove a safety property P by applying induction. However, while PDR strengthens its induction hypothesis by using clauses extracted from specific counterexamples to induction after failed induction attempts, k-induction strengthens its induction hypothesis by increasing the length of the unrolling of the transition relation.

Starting with an initial value for the bound k (usually 1), the k-induction algorithm increases the value of k iteratively after each unsuccessful attempt at finding a specification violation (base case), proving correctness via complete loop unrolling (forward condition), or inductively proving correctness of the program (inductive-step case).

Base Case. The base case of k-induction consists of running BMC with the current bound k.[4] This means that starting from all initial program states, all

[4] We define the loop bound as the number of visits of the loop head, that is, with loop bound $k = 1$, the loop head is visited once, but there was not yet any unwinding of the loop body. This nicely matches the intuition for k-induction: 1-inductiveness means that if the invariant holds for one state (without loop unrolling), then it holds again after one loop unrolling in the successor state; k-inductiveness means that if the invariant holds for k states ($k-1$ loop unrollings), then it holds again after one more loop unrolling in the successor state.

states of the program reachable within at most $k-1$ unwindings of the transition relation are explored. If a $\neg P$-state is found, the algorithm terminates.

Forward Condition. If no $\neg P$-state is found by the BMC in the base case, the algorithm continues by performing the forward-condition check, which attempts to prove that BMC fully explored the state space of the program by checking that no state with distance $k' > k-1$ to the initial state is reachable. If this check is successful, the algorithm terminates.

Inductive-Step Case. The forward-condition check, however, can only prove safety for programs with finite (and, in practice, short) loops. To prove safety beyond the bound k, the algorithm applies induction: The inductive-step case attempts to prove that after every sequence of k unrollings of the transition relation that did not reach a $\neg P$-state, there can also be no subsequent transition into a $\neg P$-state by unwinding the transition relation once more. In the realm of model checking of software, however, the safety property P is often not directly k-inductive for any value of k, thus causing the inductive-step-case check to fail. It is therefore state-of-the-art practice to add auxiliary invariants to this check to further strengthen the induction hypothesis and make it more likely to succeed. Thus, the inductive-step case proves a program safe if the following condition is unsatisfiable:

$$Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} \left(P(s_i) \wedge T(s_i, s_{i+1})\right) \wedge \neg P(s_{n+k})$$

where Inv is an auxiliary invariant, and $s_n, \ldots, s_{n+k}$ is any sequence of states. If this check fails, the induction attempt is inconclusive, and the program is neither proved safe nor unsafe yet with the current value of k and the given auxiliary invariant. In this case, the algorithm increases the value of k and starts over.

A detailed presentation of k-induction can be found in the literature [5, 6].

3 Combining k-Induction with PDR

Algorithm 1 shows an extension of k-induction with continuously-refined invariants [5] that applies PDR's aspect of learning from counterexamples to induction and that can be applied both as a main proof engine as well as an invariant generator. This allows us to apply this extension of k-induction as an invariant generator to a main k-induction procedure, similar to the KI←↻–KI approach [5].

Inputs. The algorithm takes the following inputs: The value k_{init} is used to initialize the unrolling bound k, whereas the function inc is used to increase k in line 33 after each major iteration of the algorithm, up to an upper limit of k defined by the value k_{max} enforced in line 3. The set of initial program states is described by the predicate I, the possible state transitions are described

Algorithm 1 Iterative-Deepening k-Induction with Property Direction

```
Input: the initial value $k_{init} \geq 1$ for the bound $k$,
    an upper limit $k_{max}$ for the bound $k$,
    a function $\mathsf{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \mathsf{inc}(n) > n$,
    the initial states defined by the predicate $I$,
    the transfer relation defined by the predicate $T$,
    a safety property $P$,
    a function get_currently_known_invariant to obtain auxiliary invariants,
    a Boolean $pd$ that enables or disables property direction,
    a function $\mathsf{lift} : \mathbb{N} \times (S \to \mathbb{B}) \times (S \to \mathbb{B}) \times S \to (S \to \mathbb{B})$, and
    a function $\mathsf{strengthen} : \mathbb{N} \times (S \to \mathbb{B}) \times (S \to \mathbb{B}) \to (S \to \mathbb{B})$,
    where $S$ is the set of program states.
Output: true if $P$ holds, false otherwise
Variables: the current bound $k := k_{init}$,
    the invariant $InternalInv := true$ computed by this algorithm internally, and
    the set $O := \{\}$ of current proof obligations.
 1: while $k \leq k_{max}$ do
 2:   $O_{prev} := O$
 3:   $O := \{\}$
 4:   $base\_case := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left( \bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg P(s_n) \right)$
 5:   if sat($base\_case$) then
 6:     return false
 7:   $forward\_condition := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$
 8:   if ¬ sat($forward\_condition$) then
 9:     return true
10:   if $pd$ then
11:     for each $o \in O_{prev}$ do
12:       $base\_case_o := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left( \bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg o(s_n) \right)$
13:       if sat($base\_case_o$) then
14:         return false
15:       else
16:         $step\_case_{o_n} := \bigwedge_{i=n}^{n+k-1} (o(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg o(s_{n+k})$
17:         $ExternalInv :=$ get_currently_known_invariant()
18:         $Inv := InternalInv \wedge ExternalInv$
19:         if sat($Inv(s_n) \wedge step\_case_{o_n}$) then
20:           $s_o :=$ satisfying predecessor state
21:           $O := O \cup \{\neg \mathsf{lift}(k, Inv, o, s_o)\}$
22:         else
23:           $InternalInv := InternalInv \wedge \mathsf{strengthen}(k, Inv, o)$
24:   $step\_case_n := \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$
25:   $ExternalInv :=$ get_currently_known_invariant()
26:   $Inv := InternalInv \wedge ExternalInv$
27:   if sat($Inv(s_n) \wedge step\_case_n$) then
28:     if $pd$ then
29:       $s :=$ satisfying predecessor state
30:       $O := O \cup \{\neg \mathsf{lift}(k, Inv, P, s)\}$
31:   else
32:     return true
33:   $k := \mathsf{inc}(k)$
34: return unknown
```

by the transition relation T, and the set of safe states is described by the safety property P. The accessor get_currently_known_invariant is used to obtain the strongest invariant currently available via a concurrently running (external) auxiliary-invariant generator. A Boolean flag *pd* (reminding of "property-directed") is used to control whether or not failed induction checks are used to guide the algorithm towards a sufficient strengthening of the safety property P to prove correctness; if *pd* is set to *false*, the algorithm behaves exactly like standard k-induction. Given a failed attempt to prove some candidate invariant Q[5] by induction, the function lift is used to obtain from a concrete counterexample-to-induction (CTI) state a set of CTI states described by a state predicate C. An implementation of the function lift needs to satisfy the condition that for a CTI $s \in S$ where S is the set of program states, $k \in \mathbb{N}$, $Inv \in (S \to \mathbb{B})$, $Q \in (S \to \mathbb{B})$, and $C = \mathsf{lift}(k, Inv, Q, s)$, the following holds:

$$C(s) \wedge \left(\forall s_n \in S : C(s_n) \quad \Rightarrow \quad Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (Q(s_i) \wedge T(s_i, s_{i+1})) \Rightarrow \neg Q(s_{n+k}) \right),$$

which means that the CTI s must be an element of the set of states described by the resulting predicate C and that all states in this set must be CTIs, i.e., they need to be k-predecessors of $\neg Q$-states, or in other words, each state in the set of states described by the predicate C must reach some $\neg Q$-state via k unrollings of the transition relation T. We can implement lift using Craig interpolation [18, 30] between $A : s = s_n$ and $B : Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (Q(s_i) \wedge T(s_i, s_{i+1})) \Rightarrow \neg Q(s_{n+k})$, because s is a CTI, and therefore we know that $A \Rightarrow B$ holds.[6] Hence, the resulting interpolant satisfies the criteria for C to be a valid lifting of s according to the requirements towards the function lift as outlined above. The function strengthen is used to obtain for a k-inductive invariant a stronger k-inductive invariant, i.e., its result needs to imply the input invariant, and, just like the input invariant, it must not be violated within k loop iterations and must be k-inductive.

Algorithm. Lines 4 to 6 show the base-case check (BMC) and lines 7 to 9 show the forward-condition check, both as described in Sect. 2. If *pd* is set to *true*, lines 10 to 23 attempt to prove each proof obligation using k-induction: Lines 12 to 14 check the base case for a proof obligation o. If any violations of the proof obligation o are found, this means that a predecessor state of a $\neg P$-state, and thus, transitively, a $\neg P$-state, is reachable, so we return *false*. If, otherwise, no violation was found, lines 16 to 23 check the inductive-step case to prove o.[7] We strengthen the induction hypothesis of the step-case check by

[5] Depending on the step the algorithm is in, Q may be either the safety property P or a proof obligation o.

[6] The formula C is called Craig interpolant for two formulas A and B with $A \Rightarrow B$, if $A \Rightarrow C$, $C \Rightarrow B$, and all variables in C occur in both A and B.

[7] Note that we do not need to check the forward condition for proof obligations, because the forward condition is unrelated to the safety property and the proof obligations, and therefore only needs to be checked once in each major iteration (i.e., once after each increment of k).

conjoining auxiliary invariants from an external invariant generator (via a call to `get_currently_known_invariant`) and the auxiliary invariant computed internally from proof obligations that we successfully proved previously. If the step-case check for o is unsuccessful, we extract the resulting CTI state, `lift` it to a set of CTI states, and construct a new proof obligation so that we can later attempt to prove that these CTI states are unreachable. If, on the other hand, the step-case check for o is successful, we no longer track o in the set O of unproven proof obligations (this case corresponds to line 22). We could now directly use the proof obligation as an invariant, but instead, in line 23 we first try to `strengthen` it into a stronger invariant that removes even more unreachable states from future consideration before conjoining it to our internally computed auxiliary invariant. In our implementation, we implement `strengthen` by attempting to drop components from a (disjunctive) invariant and checking if the remaining clause is still inductive. In lines 24 to 32, we check the inductive-step case for the safety property P. This check is mostly analogous to the inductive-step case check for the proof obligations described above, except that if the check is successful, we immediately return *true*.

Note that Alg. 1 eagerly increases k, even if the set O of proof obligations is not empty. This heuristic prevents the PDR part from iterating through long chains of proof obligations, it rather delegates the unrolling to the k-induction part.

An in-depth discussion of a practical example of Alg. 1 is presented in a technical report [8, pp. 12–14].

4 Evaluation

In this section, we present an extensive experimental study on the effectiveness and efficiency of adaptations of PDR to software verification.

4.1 Compared Approaches

We use the following abbreviations to distinguish between the different techniques that we evaluated:

CTIGAR: CTIGAR [11] is an adaptation of PDR to software verification. Our evaluation compares two implementations of CTIGAR, namely Vvt-CTIGAR from the tool Vvt and our own implementation CPAchecker-CTIGAR. Vvt [20] also provides a configuration that runs a parallel portfolio combination of Vvt-CTIGAR and bounded model checking, which we call Vvt-Portfolio.

KI: KI [5] denotes the plain k-induction algorithm without property direction and without auxiliary invariants, i.e., we configure Alg. 1 such that $pd = false$ and `get_currently_known_invariant()` always returns *true*.

KIPDR: KIPDR denotes a configuration of Alg. 1 such that $pd = true$ and `get_currently_known_invariant()` always returns *true*, i.e., k-induction with property direction but without additional auxiliary-invariant generation. KIPDR is, like CTIGAR, an adaptation of PDR to software verification.

KI↫DF: KI↫DF [5] denotes a parallel combination of k-induction (without property direction) with a data-flow-based auxiliary-invariant generator that continuously supplies the k-induction procedure with invariants. Here, we configure Alg. 1 such that $pd = \mathit{false}$ and `get_currently_known_invariant()` always returns the most recent (strongest) invariant computed by the data-flow-based auxiliary-invariant generator.

KI↫KIPDR: Similarly to KI↫DF, KI↫KIPDR denotes a parallel combination of k-induction with an auxiliary-invariant generator — in this case, KIPDR — that continuously supplies invariants to the k-induction procedure. Here, we configure one instance of Alg. 1 such that $pd = \mathit{false}$ and `get_currently_known_invariant()` always returns the most recent (strongest) invariant computed by KIPDR (a second instance of Alg. 1 that is configured such that $pd = \mathit{true}$ and `get_currently_known_invariant()` always returns *true*).

KI↫DF;KIPDR KI↫DF;KIPDR denotes a parallel combination of k-induction with an auxiliary-invariant generator that uses a sequential combination of a data-flow-based invariant generator and KIPDR to continuously supply k-induction with auxiliary invariants. We configure one instance of Alg. 1 such that $pd = \mathit{false}$ and `get_currently_known_invariant()` always returns the most recent (strongest) invariant computed by a sequential combination of the data-flow-based invariant generator and KIPDR (a second instance of Alg. 1 that runs after the invariant generator finishes and is configured such that $pd = \mathit{true}$ and `get_currently_known_invariant()` always returns *true*).

We do not evaluate the used invariant generators as standalone approaches, as they are designed specifically to be used as auxiliary components and do not perform well enough in isolation. For example, data-flow based invariant-generation approaches are often too imprecise to verify tasks, whereas more precise techniques like KIPDR might run into too many timeouts to be competitive. Instead, we use the framework of k-induction with continuously refined invariant generation, which has been shown to be able to combine quick and precise techniques [5].

4.2 Experimental Setup

Details about the experimental setup can be found in the technical report [8], which describes in Sect. 4.2 which tool versions and SMT theory we used, in Sect. 4.3 which benchmark sets we used and why, in Sect. 4.4 which existing verifiers we compared to and which versions we took, in Sect. 4.5 which computing resources and execution environment were used, in Sect. 4.6 the scoring schema, and in Sect. 4.12 which threats to the validity of the evaluation we identified and how we mitigated them.

4.3 Results

In the following, we pick a few highlights from the results of our experimental evaluation, in order to illustrate the potential of the approaches. A complete and more detailed report of the results is available in the extended version of this article [8].

Table 1: Results for all 5 591 verification tasks, 1 457 of which contain bugs, while the other 4 134 are considered to be safe, for the two CTIGAR implementations CPAchecker-CTIGAR and Vvt-CTIGAR, for a theoretical “virtual best” combination of both CTIGAR implementations where an oracle selects the best implementation for each task, for k-induction without auxiliary invariants (KI), and for the best configurations of each tool: CPAchecker’s KI←↻DF;KIPDR, SeaHorn, and Vvt as a portfolio verifier.

Verifier	CTIGAR		KI	Best of each tool		
	CPAchecker	Vvt		KI←↻DF; KIPDR	SeaHorn	Vvt - Portfolio
Score	1 903	879	3 282	**5 398**	2 848	727
Correct results	1 087	739	2 075	3 095	3 468	839
Correct proofs	832	524	1 239	2 335	2 724	528
Correct alarms	255	215	836	760	744	311
Wrong proofs	**0**	5	**0**	**0**	46	9
Wrong alarms	1	14	2	2	117	22
Timeouts	3 982	110	2 764	2 006	1 476	524
Out of memory	23	28	315	243	231	22
Other inconclusive	498	4 695	435	245	253	4 175
Times for correct results						
Total CPU Time (h)	9.0	3.2	30	54	29	5.7
Mean CPU Time (s)	30	16	52	63	31	25
Median CPU Time (s)	4.9	0.24	9.8	10	0.89	0.45

Suitability of CPAchecker for PDR. The first set of experiments showed that our implementation is at least as good as (and even better than) the only available implementation of PDR for software model checking. Columns two and three of Table 1 compare the results obtained by running the two implementations of CTIGAR on the whole benchmark set, and the last column of the table shows the results achieved with the standard configuration of Vvt, which runs not only CTIGAR, but a portfolio analysis of CTIGAR and bounded model checking. The quantile plot in Fig. 3 shows the CPU times that the two tool configurations spent on their correct results.

KIPDR versus Data-Flow Techniques. Data-flow-based techniques are usually more efficient than KIPDR. The higher efficiency of data-flow-based techniques is most likely due to the simple form of the invariants needed to prove the programs correct. In order to experiment with progams that have some more interesting invariants, we created a few programs by hand and tried to verify those. Table 2 shows the results we obtained for these tasks. Our experiments support the hypothesis that KIPDR can be very strong and efficient on tasks that other approaches can not solve. It is important to note that this is an ‘exists’ statement and can not be generalized, as shown by the results that KIPDR is often outperformed by simpler, data-flow-based invariant-generation techniques.

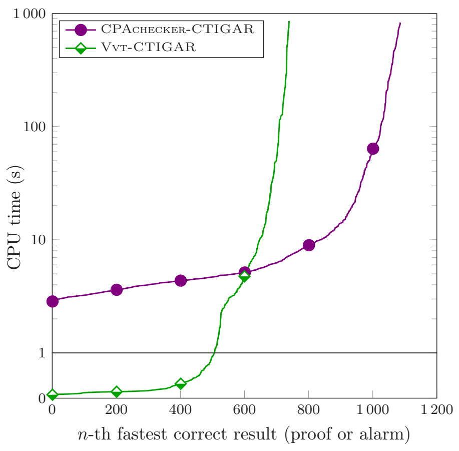


Fig. 3: Comparing two implementations of CTIGAR; quantile plot for accumulated number of solved tasks (proofs and alarms) showing the CPU time (linear scale below 1 s, logarithmic above) for the successful results of CPAchecker-CTIGAR and Vvt-CTIGAR

Table 2: Results of four k-induction-based configurations in CPAchecker with different approaches for generating auxiliary invariants for seven manually crafted verification tasks that do not contain bugs and are not solved by k-induction without auxiliary invariants; an entry "T" means that the CPU-time limit was exceeded, an entry "M" means that the memory limit was exceeded, and all other entries represent the CPU time a configuration spent to correctly solve the task

Task	KI←DF			KI←⟲-KIPDR
	Boxes	Boxes, Eq	Boxes, Eq, Mod2	
`const.c`	3.3 s	3.3 s	**3.2 s**	3.8 s
`eq1.c`	T	**3.2 s**	3.3 s	4.9 s
`eq2.c`	M	M	M	**3.9 s**
`even.c`	T	T	**3.5 s**	3.9 s
`odd.c`	T	T	**3.4 s**	4.1 s
`mod4.c`	T	T	T	**3.6 s**
`bin-suffix-5.c`	M	M	M	**3.6 s**

Comparison with Non-PDR Approaches. The seven example programs[8] were added to the benchmark collection that was also used for SV-COMP 2019, and thus, results are available for all verifiers that participated in the competition[9]. Table 3 summarizes the results of the best six verifiers in comparison with the KI←⟲-KIPDR approach that we created for the study in this paper. Those verifiers are, in alphabetical order, Skink, Ultimate Automizer, Ultimate Kojak,

[8] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp19/c/loop-invariants/

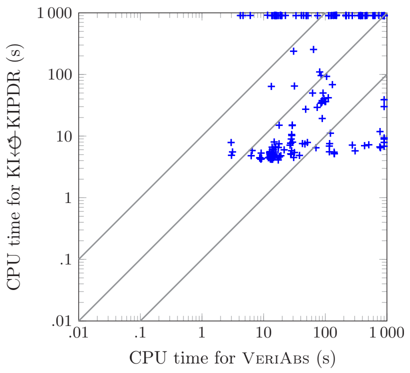
[9] See the last seven rows in this table: https://sv-comp.sosy-lab.org/2019/results/results-verified/ReachSafety-Loops.table.html

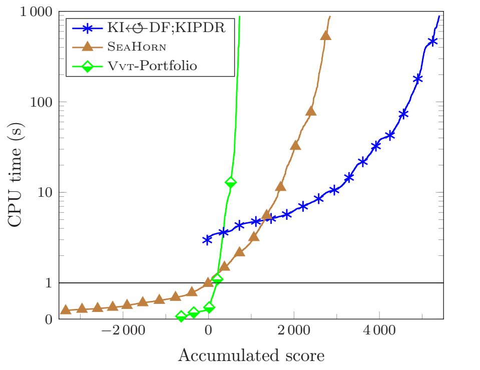
Table 3: Results of SV-COMP 2019 for the six verifiers that performed best on our seven manually crafted verification tasks, compared to the results of KI⟲KIPDR approach previously shown in Table 2; an entry "T" means that the CPU-time limit was exceeded, an entry "M" means that the memory limit was exceeded, an entry "O" means that the verifier gave up deliberately for other reasons, and all other entries represent the CPU time a verifier configuration spent to correctly solve the task; note that SV-COMP 2019 used Ubuntu 18.04 based on Linux 4.15, whereas our evaluation of KI⟲KIPDR used Ubuntu 16.04 based on Linux 4.4; otherwise, the evaluation environment was the same

Task	SV-COMP 2019						KI⟲KIPDR
	Skink	UAutomizer	UKojak	UTaipan	VeriAbs	VIAP	
`const.c`	4.2 s	8.7 s	9.1 s	8.2 s	13 s	110 s	**3.8 s**
`eq1.c`	290 s	7.8 s	7.6 s	8.3 s	14 s	57 s	**4.9 s**
`eq2.c`	4.1 s	8.1 s	8.6 s	7.6 s	14 s	4.7 s	**3.9 s**
`even.c`	**3.7 s**	7.4 s	8.2 s	8.6 s	140 s	4.5 s	3.9 s
`odd.c`	O	9.6 s	T	11 s	140 s	4.6 s	**4.1 s**
`mod4.c`	4.0 s	8.4 s	8.4 s	7.7 s	140 s	4.5 s	**3.6 s**
`bin-suffix-5.c`	O	14 s	T	13 s	13 s	4.7 s	**3.6 s**

Ultimate Taipan, VeriAbs, and VIAP. Fig. 4a directly compares the CPU times spent on tasks of in the subcategory *ReachSafety-Loops*, which is known to contain many tasks that require effort to be spent on generating loop invariants, by both VeriAbs, which was the best verifier in that subcategory, and KI⟲KIPDR. We observe that for the majority of tasks that were solved by both verifiers, KI⟲KIPDR is faster than VeriAbs, often by more than an order of magnitude. This shows that the invariant generator KIPDR can be significantly faster than other approaches, depending on the benchmark set. As before, a more in-depth discussion can be found in the technical report [8].

Comparison against PDR-Based Verification Tools. The last three columns of Table 1 give an overview over the best configurations of three software verifiers that use adaptations of PDR: For CPAchecker, we selected KI⟲DF;KIPDR. For SeaHorn, we used the same configuration as submitted by the developers to the 2016 Competition on Software Verification (SV-COMP 2016) [22]. For Vvt, we used the portfolio configuration. We observe that SeaHorn achieves the highest number of correct proofs, but also has a significant amount of incorrect proofs. CPAchecker is the slowest of the three tools and finds fewer proofs than SeaHorn, but CPAchecker has no wrong proofs, and also closely leads in the amount of found bugs. The score-based quantile plot of these results displayed in Fig. 4b visualizes the effects of incorrect results on the computed score. While the graph for SeaHorn is longer, i.e., shows that it solved the most tasks, it is offset to the left by a total penalty of −3 344 points, such that in the end, KI⟲DF;KIPDR accumulates the highest score because it has a smaller penalty of only −32 points.





(a) Scatter plot comparing the CPU times spent on tasks by VERIABS and KI⇜⟲-KIPDR

(b) Quantile plot for accumulated score of solved tasks (offset to the left by total penalty from wrong results) showing the CPU time (linear scale below 1 s, logarithmic above) for the successful results of KI⇜⟲-DF;KIPDR, SEAHORN, and VVT-Portfolio

Fig. 4: Plots that support the claim that the conclusions of the evaluation are relevant

These results confirm our hypothesis that our previous conclusions are relevant, because they are supported by an implementation that is competitive when compared to the best available PDR-based tool implementations.

5 Conclusion

Property-directed reachability (a.k.a. IC3) is a verification approach that is popular and successful in some fields of formal verification (e.g., hardware designs, Horn clauses). Unfortunately, there is a large gap between this success story and the applicability in practical software verification. We are closing this gap by (a) providing a well-engineered implementation of one published adaptation of PDR to software verification, (b) designing and implementing an invariant generator based on the ideas of PDR, and (c) providing an evaluation of all applicable tools and approaches on the largest available benchmark set of C verification tasks. This provides a good foundation as baseline for ongoing research in this area.

The results of our comparative evaluation extend the knowledge about PDR for software verification in the following ways: (1) Our implementation outperforms the existing implementation of PDR (VVT) and is more precise than the other software verifier that uses PDR (SEAHORN). Thus, our implementation can serve as a reference implementation for further research on PDR for software verification. (2) On most of the programs in the widely used *sv-benchmarks* collection of verification tasks, other techniques are more effective (solve more problems) and more efficient (solve the problems faster). (3) PDR can be an effective and efficient technique for computing invariants that are difficult to obtain: there are programs for which our PDR-based approach is more efficient than the best invariant generator from SV-COMP in the subcategory *ReachSafety-Loops*.

5.1 Data Availability Statement

A replication package for this article including all evaluated implementations and BenchExec is available at Zenodo [9]. Current versions of CPAchecker are available at https://github.com/sosy-lab/cpachecker. The benchmark set of SV-COMP 2018 used in Sect. 4 is available online at https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp18 and the dataset from SV-COMP 2019 [4] that we analyzed is available at https://sv-comp.sosy-lab.org/2019/results/results-verified/All-Raw.zip.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Proc. TACAS. pp. 268–283. LNCS 2031, Springer (2001). https://doi.org/10.1007/3-540-45319-9_19
2. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). https://doi.org/10.1145/503272.503274
3. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. IEEE Intelligent Systems **29**(1), 20–29 (2014). https://doi.org/10.1109/MIS.2014.3
4. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
5. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
6. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
7. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
8. Beyer, D., Dangl, M.: Software verification with PDR: Implementation and empirical evaluation of the state of the art (August 2019), http://arxiv.org/abs/1908.06271
9. Beyer, D., Dangl, M.: Replication package for article 'Software verification with PDR: An implementation of the state of the art'. Zenodo (2020). https://doi.org/10.5281/zenodo.3678766
10. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
11. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Proc. CAV. pp. 831–848. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_55
12. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI. pp. 70–87. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
13. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. **20**(4-5), 379–405 (2008). https://doi.org/10.1007/s00165-008-0080-9

14. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
15. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Proc. CAV. pp. 277–293. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_23
16. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. FMSD **49**(3), 190–218 (2016). https://doi.org/10.1007/s10703-016-0257-4
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
18. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
19. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
20. Günther, H., Laarman, A., Weissenbacher, G.: Vienna Verification Tool: IC3 for parallel software (competition contribution). In: Proc. TACAS. pp. 954–957. LNCS 9636, Springer (2016)
21. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: Proc. SAS. pp. 287–303 (2010). https://doi.org/10.1007/978-3-642-15769-1_18
22. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: A framework for verifying C programs (competition contribution). In: Proc. TACAS. pp. 447–450. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_41
23. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
24. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
25. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Proc. FMCAD. pp. 85–92. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886665
26. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Proc. Int. Workshop on Parallel and Distributed Methods in Verification. pp. 55–62. EPTCS 72 (2011). https://doi.org/10.4204/EPTCS.72
27. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
28. Lange, T., Neuhäußer, M.R., Noll, T.: IC3 software model checking on control flow automata. In: Proc. FMCAD. pp. 97–104 (2015)
29. Lange, T., Prinz, F., Neuhäußer, M.R., Noll, T., Katoen, J.: Improving generalization in software IC3. In: Proc. SPIN'18. pp. 85–102. LNCS 10869, Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_5
30. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
31. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley Publishing, 3rd edn. (2011)
32. Wahl, T.: The k-induction principle (2013), available at `http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf`

Verifying Array Manipulating Programs with Full-Program Induction

Supratik Chakraborty[1], Ashutosh Gupta[1], and Divyesh Unadkat[1,2]

[1] Indian Institute of Technology Bombay, Mumbai, India
{supratik,akg}@cse.iitb.ac.in
[2] TCS Research, Pune, India
divyesh.unadkat@tcs.com

Abstract. We present a full-program induction technique for proving (a sub-class of) quantified as well as quantifier-free properties of programs manipulating arrays of parametric size N. Instead of inducting over individual loops, our technique inducts over the entire program (possibly containing multiple loops) directly via the program parameter N. Significantly, this does not require generation or use of loop-specific invariants. We have developed a prototype tool Vajra to assess the efficacy of our technique. We demonstrate the performance of Vajra vis-a-vis several state-of-the-art tools on a set of array manipulating benchmarks.

1 Introduction

Programs with loops manipulating arrays are common in a variety of applications. Unfortunately, assertion checking in such programs is undecidable. Existing tools therefore use a combination of techniques that work well for certain classes of programs and assertions, and yield conservative results otherwise. In this paper, we present a new technique to add to this arsenal of techniques. Specifically, we focus on programs with loops manipulating arrays, where the size of each array is a symbolic integer parameter N (> 0). We allow (a sub-class of) quantified and quantifier-free pre- and post-conditions that may depend on the symbolic parameter N. Thus, the problem we wish to solve can be viewed as checking the validity of a parameterized Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all values of N (> 0), where the program P_N computes with arrays of size N, and N is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$. Fig. 1(a) shows an example of one such Hoare triple, written using `assume` and `assert`. This triple effectively verifies that $\sum_{j=0}^{i-1}\left(1+\sum_{k=0}^{j-1} 6\cdot(k+1)\right) = i^3$ for all $i \in \{0 \ldots N-1\}$, and for all $N > 0$. Although each loop in Fig. 1(a) is simple, their sequential composition makes it difficult even for state-of-the-art tools like VIAP [26], VeriAbs [8], FreqHorn [10], Tiler [4], Vaphor [24], or Booster [1] to prove the post-condition correct. In fact, none of the above tools succeed in automatically proving the post-condition in Fig. 1(a). In contrast, the technique presented in this paper, called *full-program induction*, proves the post-condition in Fig. 1(a) correct within a few seconds.

Like several earlier approaches [29], full-program induction relies on mathematical induction to reason about programs with loops. However, the way in

https://doi.org/10.1007/978-3-030-45190-5_2

```
// assume(true)
1. for (int t1=0; t1<N; t1=t1+1) {
2.   if (t1==0) { A[t1] = 6; }
3.   else { A[t1] = A[t1-1]+6; }
4. }
5. for (int t2=0; t2<N; t2=t2+1) {
6.   if (t2==0) { B[t2] = 1; }
7.   else { B[t2] = B[t2-1]+A[t2-1]; }
8. }
9. for (int t3=0; t3<N; t3=t3+1) {
10.  if (t3==0) { C[t3] = 0; }
11.  else { C[t3] = C[t3-1]+B[t3-1]; }
12.}
// assert(forall i in 0..N-1, C[i]= i^3)
```

(a)

```
// assume(true)
1.  A[0] = 6;
2.  B[0] = 1;
3.  C[0] = 0;
// assert((C[0] = 0^3) and (B[0] = 1^3 - 0^3) and
//         (A[0] = 2^3 - 2*1^3 + 0^3))
```

(b)

```
// assume((N > 1) and (C_Nm1[N-2] = (N-2)^3) and
//         (B_Nm1[N-2] = (N-1)^3 - (N-2)^3) and
//         (A_Nm1[N-2] = N^3 - 2*(N-1)^3 + (N-2)^3))
1.  A[N-1] = A_Nm1[N-2] + 6;
2.  B[N-1] = B_Nm1[N-2] + A_Nm1[N-2];
3.  C[N-1] = C_Nm1[N-2] + B_Nm1[N-2];
// assert((C[N-1] = (N-1)^3) and
//         (B[N-1] = N^3 - (N-1)^3) and
//         (A[N-1] = (N+1)^3 - 2*N^3 + (N-1)^3))
```

(c)

Fig. 1. Original and simplified Hoare triples

which the inductive claim is formulated and proved differs significantly. Specifically, (i) we *do not require explicit or implicit loop-specific invariants* to be provided by the user or generated by a solver (viz. by constrained Horn clause solvers [21,15,10] or recurrence solvers [26,17]), (ii) we *induct on the full program* (possibly containing multiple loops) with parameter N and not on iterations of individual loops in the program, and (iii) we perform *non-trivial correct-by-construction code transformations*, whenever feasible, to simplify the inductive step of reasoning. The combination of these factors often reduces reasoning about a program with multiple loops to reasoning about one with fewer (sometimes even none) and "simpler" loops, thereby simplifying proof goals. In this paper, we demonstrate this, focusing on programs with sequentially composed, but non-nested loops.

As an illustration of simplifications that can result from application of full-program induction, consider the problem in Fig. 1(a) again. Full-program induction reduces checking the validity of the Hoare triple in Fig. 1(a) to checking the validity of two "simpler" Hoare triples, represented in Figs. 1(b) and 1(c). Note that the programs in Figs. 1(b) and 1(c) are loop-free. In addition, their pre- and post-conditions are quantifier-free. The validity of these Hoare triples (Figs. 1(b) and 1(c)) can therefore be easily proved, e.g. by bounded model checking [6] with a back-end SMT solver like Z3 [25]. Note that the value computed in each iteration of each loop in Fig. 1(a) is data-dependent on previous iterations of the respective loops. Hence, none of these loops can be trivially translated to a set of parallel assignments.

Invariant-based techniques, viz. [13,16,23,7,14,30,2,19], are popularly used to reason about array manipulating programs. If we were to prove the assertion in Fig. 1(a) using such techniques, it would be necessary to use appropriate loop-specific invariants for each of the three loops in Fig. 1(a). The weakest loop invariants needed to prove the post-condition in this example are: $\forall i \in [0...t1-1]\ (A[i] = 6i + 6)$ for the first loop (lines 1-4), $\forall j \in [0...t2-1]\ (B[j] = 3j^2 + 3j + 1) \wedge (A[j] = 6j + 6)$ for the second loop (lines 5-8), and $\forall k \in [0...t3-1]\ (C[k] = k^3) \wedge (B[k] = 3k^2 + 3k + 1)$ for the third loop (lines

9-12). Unfortunately, automatically deriving such quantified non-linear loop invariants is far from trivial. Template-based invariant generators, viz. [12,9], are among the best-performers when generating such complex invariants. However, their abilities are fundamentally limited by the set of templates from which they choose. We therefore choose not to depend on invariants for individual loops in our work at all. Instead of inducting over the iterations of each individual loop, we propose to reason about the entire program (containing one or more loops) directly, while inducting on the parameter N. Needless to say, each approach has its own strengths and limitations, and the right choice always depends on the problem at hand. Our experiments show that full-program induction is able to solve several difficult problem instances with an off-the-shelf SMT solver (Z3) at the back-end, which other techniques either fail to solve these instances, or rely on sophisticated recurrence solvers.

The primary contributions of our work can be summarized as follows.

- We introduce the notion of *full-program induction* for reasoning about assertions in programs with loops manipulating arrays.
- We present practical algorithms for full-program induction.
- We describe a prototype tool Vajra that implements the algorithms, using an off-the-shelf SMT solver, viz. Z3, at the back-end to discharge verification conditions. Vajra outperforms several state-of-the-art tools on a suite of array-manipulating benchmark programs.

Related Work. Earlier work on inductive techniques can be broadly categorized into those that require loop-specific invariants to be provided or automatically generated, and those that work without them. Requiring a "good" inductive invariant for every loop in a program effectively shifts the onus of assertion checking to that of invariant generation. Among techniques that do not require explicit inductive invariants or mid-conditions for each loop, there are some that require loop invariants to be implicitly generated by a constraint solver. These include techniques based on constrained Horn clause solving [21,15,10,24], acceleration and lazy interpolation for arrays [1] and those that use inductively defined predicates and recurrence solving [26,17], among others. Thanks to the impressive capabilities of modern constraint solvers and the effectiveness of carefully tuned heuristics for stringing together multiple solvers, this approach has shown a lot of promise in recent years. However, at a fundamental level, these formulations rely on solving implicitly specified loop invariants garbed as constraint solving problems. There are yet other techniques, such as that in [28], that truly do not depend on loop invariants being generated. In fact, the technique of [28] comes closest to our work in principle. However, [28] imposes severe restrictions on the input programs, and the example in Fig. 1 does not meet these restrictions. Therefore, the technique of [28] is applicable only to a small part of the program-assertion space over which our technique works. Techniques such as tiling [4] reason one loop at a time and apply only when loops have simple data dependencies across iterations (called *non-interference* of tiles in [4]). It effectively uses a slice of the post-condition of a loop as an inductive invariant, and also requires strong enough mid-conditions to be generated in the case of sequentially composed loops. We circumvent all of these requirements in the current work. For some other techniques for analyzing array manipulating programs, please see [7,19,18].

2 Overview of Full-program Induction

Recall that our objective is to check the validity of the parameterized Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all $N > 0$. At a high level, our approach works like any other inductive technique. Thus, we have a base case, where we verify that the parameterized Hoare triple holds for some small values of N, say $0 < N \leq M$. We then hypothesize that $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds for some $N > M$, and try to show that this implies $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$. While this sounds simple in principle, there are several technical difficulties en route. Our contribution lies in overcoming these difficulties algorithmically for a large class of programs and assertions, thereby making *full-program induction* a viable and competitive technique for proving properties of array manipulating programs.

We rely on an important, yet reasonable, assumption that can be stated as follows: *For every value of N (> 0), every loop in P_N can be statically unrolled a fixed number (say $f(N)$) of times to yield a loop-free program $\widehat{\mathsf{P}_N}$ that is semantically equivalent to P_N.* Note that this does not imply that reasoning about loops can be translated into loop-free reasoning. In general, $f(N)$ is a non-constant function, and hence, the number of unrollings of loops in P_N may strongly depend on N. In our experience, loops in a vast majority of array manipulating programs (including Fig. 1(a)) satisfy the above assumption. Consequently, the base case of our induction reduces to checking a Hoare triple for a loop-free program. Checking such a Hoare triple is easily achieved by compiling the pre-condition, program and post-condition into an SMT formula, whose (un)satisfiability can be checked with an off-the-shelf back-end SMT solver.

The inductive step is the most complex one, and is the focus of the rest of the paper. Recall that the inductive hypothesis asserts that $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ is valid. To make use of this hypothesis in the inductive step, we must relate the validity of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ to that of $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$. We propose doing this, whenever possible, via two key notions – that of "difference" program and "difference" pre-condition. Given a parameterized program P_N, intuitively the "difference" program $\partial\mathsf{P}_N$ is one such that $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$ is semantically equivalent to P_N, where ";" denotes sequential composition. It turns out that for our purposes, the semantic equivalence alluded to above is not really necessary; it suffices to have $\partial\mathsf{P}_N$ such that $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ is valid iff $\{\varphi(N)\}\ \mathsf{P}_{N-1}; \partial\mathsf{P}_N\ \{\psi(N)\}$ is valid. We will henceforth use this interpretation of a "difference" program. The "difference" pre-condition $\partial\varphi(N)$ is a formula such that (i) $\varphi(N) \rightarrow (\varphi(N-1) \wedge \partial\varphi(N))$ and (ii) the execution of P_{N-1} doesn't affect the truth of $\partial\varphi(N)$. Computing $\partial\mathsf{P}_N$ and $\partial\varphi(N)$ is not easy in general, and we discuss this in detail in the rest of the paper.

Assuming we have $\partial\mathsf{P}_N$ and $\partial\varphi(N)$ with the properties stated above, the proof obligation $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ can now be reduced to proving $\{\varphi(N -$

$1)\}$ P_{N-1} $\{\psi(N-1)\}$ and $\{\psi(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N)\}$. The first triple follows from the inductive hypothesis. Proving the second triple may require strengthening the pre-condition, say by a formula $\mathsf{Pre}(N-1)$, in general. Recalling that we are in the inductive step of mathematical induction, we formulate the new proof sub-goal in such a case as $\{(\psi(N-1) \wedge \mathsf{Pre}(N-1)) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N) \wedge \mathsf{Pre}(N)\}$. While this is somewhat reminiscent of loop invariants, observe that $\mathsf{Pre}(N)$ is *not* really a loop-specific invariant. Instead, it is analogous to computing an invariant for the entire program, possibly containing multiple loops. Specifically, the above process strengthens both the pre- and post-condition of $\{\psi(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N)\}$ simultaneously using $\mathsf{Pre}(N-1)$ and $\mathsf{Pre}(N)$, respectively. The strengthened post-condition of the resulting Hoare triple may, in turn, require a new pre-condition $\mathsf{Pre}'(N-1)$ to be satisfied. This process of strengthening the pre- and post-conditions of the Hoare triple involving $\partial\mathsf{P}_N$ can be iterated until a fix-point is reached, i.e. no further pre-conditions are needed for the parameterized Hoare triple to hold. While the fix-point was quickly reached for all benchmarks we experimented with, we also discuss how to handle cases where the above process may not converge easily. Note that since we effectively strengthen the pre-condition of the Hoare triple in the inductive step, for the overall induction to go through, it is also necessary to check that the strengthened assertions hold at the end of each base case check. The technique described above is called *full-program induction*, and the following theorem guarantees its soundness.

Theorem 1. *Given* $\{\varphi(N)\}$ P_N $\{\psi(N)\}$*, suppose the following are true:*

1. *For* $N > 1$*,* $\{\varphi(N)\}$ $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$ $\{\psi(N)\}$ *holds iff* $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ *holds.*
2. *For* $N > 1$*, there exists a formula* $\partial\varphi(N)$ *such that (a)* $\partial\varphi(N)$ *doesn't refer to any program variable or array element modified in* P_{N-1}*, and (b)* $\varphi(N) \rightarrow \varphi(N-1) \wedge \partial\varphi(N)$*.*
3. *There exists an integer* $M \geq 1$ *and a parameterized formula* $\mathsf{Pre}(M)$ *such that (a)* $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ *holds for* $0 < N \leq M$*, (b)* $\{\varphi(M)\}$ P_M $\{\psi(M) \wedge \mathsf{Pre}(M)\}$ *holds, and (c)* $\{\psi(N-1) \wedge \mathsf{Pre}(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N) \wedge \mathsf{Pre}(N)\}$ *holds for* $N > M$*.*

Then $\{\varphi_N\}$ P_N $\{\psi_N\}$ *holds for all* $N \geq 1$*.*

Proof. For $0 < N \leq M$, condition 3(a) ensures that $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds. For $N > M$, note that by virtue of condition 1 and 2(b), $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ holds if $\{\varphi(N-1) \wedge \partial\varphi(N)\}$ $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$ $\{\psi(N) \wedge \mathsf{Pre}(N)\}$ holds. With $\psi(N-1) \wedge \mathsf{Pre}(N-1)$ as a mid-condition, and by virtue of condition 2(a), the latter Hoare triple holds for $N > M$ if $\{\varphi(M)\}$ P_M $\{\psi(M) \wedge \mathsf{Pre}(M)\}$ holds and $\{\psi(N-1) \wedge \mathsf{Pre}(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{\psi(N) \wedge \mathsf{Pre}(N)\}$ holds for all $N > M$. Both these triples are seen to hold by virtue of conditions 3(b) and (c). $\square$

3 Algorithms for Full-program Induction

We now discuss the *full-program induction* algorithm, focusing on generation of three crucial components: *difference program* $\partial\mathsf{P}_N$, *difference pre-condition* $\partial\varphi(N)$, and the formula $\mathsf{Pre}(N)$ for strengthening pre- and post-conditions.

3.1 Preliminaries

We consider array manipulating programs generated by the grammar shown below (adapted from [4]).

$$
\begin{aligned}
\mathsf{PB} &::= \mathsf{St}\\
\mathsf{St} &::= v := \mathsf{E} \mid\ A[\mathsf{E}] := \mathsf{E} \mid\ \textbf{if}(\mathsf{BoolE})\ \textbf{then}\ \mathsf{St}\ \textbf{else}\ \mathsf{St} \mid\ \mathsf{St}\ ;\ \mathsf{St} \mid\\
&\quad\ \textbf{for}\ (\ell := 0;\ \ell < \mathsf{E};\ \ell := \ell{+}1)\ \ \{\mathsf{St1}\}\\
\mathsf{St1} &::= v := \mathsf{E} \mid\ A[\mathsf{E}] := \mathsf{E} \mid\ \textbf{if}(\mathsf{BoolE})\ \textbf{then}\ \mathsf{St1}\ \textbf{else}\ \mathsf{St1} \mid\ \mathsf{St1}\ ;\ \mathsf{St1}\\
\mathsf{E} &::= \mathsf{E}\ \mathsf{op}\ \ \mathsf{E} \mid\ A[\mathsf{E}]\ \mid\ v\ \mid\ \ell\ \mid\ \mathsf{c}\ \mid\ N\\
\mathsf{op} &::= +\ \mid\ \text{-}\ \mid\ *\ \mid\ /\\
\mathsf{BoolE} &::= \mathsf{E}\ \mathsf{relop}\ \mathsf{E} \mid\ \mathsf{BoolE\ AND\ BoolE}\ \mid\ \mathsf{NOT\ BoolE}\ \mid\ \mathsf{BoolE\ OR\ BoolE}
\end{aligned}
$$

This grammar restricts programs to have non-nested loops. While this limits the set of programs to which our technique currently applies, there is a large class of useful programs, with possibly long sequences of loops, that are included in the scope of our work. In reality, our technique also applies to a subclass of programs with nested loops. However, characterizing this class of programs through a grammar is a bit unwieldy, and we avoid doing so for reasons of clarity. A program P_N is a tuple $(\mathcal{V}, \mathcal{L}, \mathcal{A}, \mathsf{PB}, N)$, where $\mathcal{V}$ is a set of scalar variables, $\mathcal{L} \subseteq \mathcal{V}$ is a set of scalar loop counter variables, $\mathcal{A}$ is a set of array variables, PB is the program body, and N is a special symbol denoting a positive integer parameter. In the grammar shown above, we assume $A \in \mathcal{A}$, $v \in \mathcal{V} \backslash \mathcal{L}$, $\ell \in \mathcal{L}$ and $\mathsf{c} \in \mathbb{Z}$. Furthermore, "$\mathsf{relop}$" is assumed to be one of the relational operators and "op" is an arithmetic operator from the set $\{+, \text{-}, *, /\}$. We also assume that each loop L has a unique loop counter variable ℓ which is initialized at the beginning of L and is incremented by 1 at the end of each iteration. Assignments in the body of L are assumed not to update ℓ. Finally, for each loop with termination condition $\ell < \mathsf{E}$, we assume that E is an expression in terms of N. We denote by $k_L(N)$ the number of times loop L iterates in the program with parameter N. We verify Hoare triples of the form $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$, where $\varphi(N)$ and $\psi(N)$ are either universally quantified formulas of the form $\forall I\ (\Phi(I, N) \implies \Psi(\mathcal{A}, \mathcal{V}, I, N))$ or quantifier-free formulas of the form $\Xi(\mathcal{A}, \mathcal{V}, N)$. In the above, I is a sequence of array index variables, Φ is a quantifier-free formula in the theory of arithmetic over integers, and Ψ and Ξ are quantifier-free formulas in the combined theory of arrays and arithmetic over integers.

Static single assignment (SSA) [27] is a well-known technique for renaming scalar variables such that a variable is written at most once in a program. For our purposes, we also wish to rename arrays so that each loop updates its own version of an array and multiple writes to an array element within the same loop happen on different versions of the array. Array SSA [20] renaming has been studied earlier in the context of compilers to achieve this goal. We propose using SSA renaming for both scalars and arrays as a pre-processing step of our analysis. Therefore, we assume henceforth that the input program is SSA renamed (for both scalars and arrays). We also assume that the post-condition is expressed in terms of these SSA renamed scalar and array variables.

We represent a program using a *control flow graph* $G = (Locs, Edges, \mu)$, where *Locs* denotes the set of control locations (nodes) of the program, $Edges \subseteq Locs \times Locs \times \{\mathbf{tt}, \mathbf{ff}, \mathsf{U}\}$ represents the flow of control and $\mu : Locs \rightarrow \mathsf{AssignSt} \cup \mathsf{BoolE}$ annotates every node in *Locs* with either an assignment statement (of the form $v := \mathsf{E}$ or $A[\mathsf{E}] := \mathsf{E}$) from the set of assignment statements $\mathsf{AssignSt}$, or a Boolean condition. Two distinguished control locations, called Start and End in *Locs*, represent the entry and exit points of the program. An edge $(n_1, n_2, label)$ represents flow of control from n_1 to n_2 without any other intervening node. It is labeled $\mathbf{tt}$ or $\mathbf{ff}$ if $\mu(n_1)$ is a Boolean condition, and is labeled U otherwise. If $\mu(n_1)$ is a Boolean condition, there are two outgoing edges from n_1, labeled $\mathbf{tt}$ and $\mathbf{ff}$ respectively, and control flows from n_1 to n_2 along $(n_1, n_2, label)$ only if $\mu(n_1)$ evaluates to *label*. If $\mu(n_1)$ is an assignment statement, there is a single outgoing edge from n_1, and it is labeled U. Henceforth, we use CFG to refer to the control flow graph.

A CFG may have cycles due to the presence of loops in the program. A *back-edge* of a loop is an edge from the node corresponding to the last statement in the loop body to the node representing the loop head. An *exit-edge* is an edge from the loop head to a node outside the loop body. An *incoming-edge* is an edge to the loop head from a node outside the loop body. We assume that every loop has exactly one *back-edge*, one *incoming-edge* and one *exit-edge*. For technical reasons, and without loss of generality, we also assume that the *exit-edge* of a loop always goes to a "nop" node (say, having a statement `x = x;`).

Given a program, the program dependence graph (or PDG) $G = (V, DE, CE)$ represents data and control dependencies among program statements. Here, V denotes vertices representing assignment statements and boolean expressions, $DE \subseteq V \times V$ denotes data dependence edges and $CE \subseteq V \times V$ denotes control dependence edges. Standard dataflow analysis identifies dependencies between program variables and thereby among statements. Dependence between statements updating array elements requires a more careful analysis. Let S_1 and S_2 be two statements in loops L_1 and L_2 where there is a control-flow path from S_1 to S_2 in the CFG. Suppose S_1 is of the form $A[f(i_1, N)] = F(\ldots)$; where f is an array index expression, i_1 is the loop counter variable of L_1, and F is an arbitrary expression. Suppose S_2 is of the form $X = G(A[g(i_2, N)])$;, where X is a variable or array element, G is an arbitrary expression, and g is an array index expression.

Definition 1. *We say that S_2 in L_2* ***depends*** *on S_1 in L_1 if there exists i_1, i_2 such that $0 \leq i_1 < k_{L_1}(N)$ and $0 \leq i_2 < k_{L_2}(N)$ and $f(i_1, N) = g(i_2, N)$.*

The routine ComputeRefinedPDG shown in Algorithm 1 constructs and refines the program dependence graph $G = (V, DE, CE)$ for the input program P_N. It uses the function ConstructPDG (line 1) based on the technique of [11] to create an initial graph. For a node n in G, let $def(n)$ and $uses(n)$ refer to the set of variables/array elements defined and used, respectively, in the statement/boolean expression corresponding to n. Similarly, let $subscript(v, n)$ refer to the index expression of the array element v referred to at node n. Predicate $is\text{-}array(v)$ evaluates to true if v is an array element and false if v is

Algorithm 1 ComputeRefinedPDG(P_N : Program)

```
1: $G(V, DE, CE)$ := ConstructPDG($\mathsf{P}_N$);
2: if $\exists v, n, n'.\ (n, n') \in DE \wedge \textit{is-array}(v) \wedge v \in \textit{def}(n) \wedge v \in \textit{uses}(n')$ then
3:     if $n$ is part of a loop $L$ then
4:         $\ell$ := loop counter of $L$;
5:         Let $\phi(n)$ be the constraint $(0 \leq \ell < k_L)$;
6:     else
7:         Let $\phi(n)$ be $true$;
8:     if $n'$ is part of a loop $L'$ then
9:         $\ell'$ := loop counter of $L'$;
10:        Let $\phi'(n')$ be the constraint $(0 \leq \ell' < k_{L'})$;
11:    else
12:        Let $\phi'(n')$ be $true$;
13:    if $\phi(n) \wedge \phi(n') \wedge \big(\textit{subscript}(v, n) = \textit{subscript}(v, n')\big)$ is unsatisfiable then
14:        $DE = DE \setminus \{(n, n')\}$;    ▷ Remove dependence edges with non-overlapping subscripts
15: return $G(V, DE, CE)$;
```

Algorithm 2 PeelAllLoops(($\textit{Locs}, \textit{Edges}, \mu$) : CFG of P_N)

```
1: $\mathsf{P}^p_N := (\textit{Locs}^p, \textit{Edges}^p, \mu^p)$, where $L^p = \textit{Locs}$, $\textit{Edges}^p = \textit{Edges}$, $\mu^p = \mu$;    ▷ Copy of $\mathsf{P}_N$
2: $\textit{peelNodes} := \varnothing$;
3: for each loop $L \in$ Loops($\mathsf{P}^p_N$) do
4:     Let $k_L(N)$ be the expression for iteration count of $L$ in $\mathsf{P}^p_N$;
5:     $\textit{peelCount}$ := Simplify($k_L(N) - k_L(N-1)$);
6:     if $\textit{peelCount}$ is non-constant then throw "Failed to peel non-constant number of iterations";
7:     $\langle \mathsf{P}^p_N, \textit{Locs}' \rangle$ := PeelSingleLoop($\mathsf{P}^p_N, L, k_L(N-1), \textit{peelCount}$);
   ▷ Transforms loop $L$ so that last $\textit{peelCount}$ iterations of $L$ are peeled/unrolled. Updated
   CFG and newly created CFG nodes for the peeled iterations are returned by PeelSingleLoop.
8:     $\textit{peelNodes} := \textit{peelNodes} \cup \textit{Locs}'$;
9: return $\langle \mathsf{P}^p_N, \textit{peelNodes} \rangle$;
```

a scalar variable. Note that lines 2-14 of ComputeRefinedPDG removes data dependence edges between nodes of G that do not satisfy Definition 1.

3.2 Core Modules in the Technique

Peeling the Loops. To relate P_N to P_{N-1}, we first ensure that the corresponding loops in both programs iterate the same number of times by *peeling* extra iterations from the loops in P_N. This is done by routine PeelAllLoops shown in Algorithm 2. The algorithm first makes a copy, viz. P^p_N, of the input CFG P_N. Let Loops(P^p_N) denote the set of loops of P^p_N, and let $k_L(N)$ and $k_L(N-1)$ denote the number of times loop L iterates in P^p_N and P^p_{N-1} respectively. The difference $k_L(N) - k_L(N-1)$, computed in line 5, gives the extra iterations of loop L in P^p_N. If this difference is not a constant, we currently report a failure of our technique (line 6). Otherwise, routine PeelSingleLoop transforms loop L of P^p_N as follows: it replaces the termination condition $(\ell < k_L(N))$ of L by $(\ell < k_L(N-1))$. It also peels (or unrolls) the last $(k_L(N) - k_L(N-1))$ iterations of L and adds control flow edges such that the the peeled iterations are executed immediately after the loop body is iterated $k_L(N-1)$ times. Effectively, PeelSingleLoop unrolls/peels the last $(k_L(N) - k_L(N-1))$ iterations of loop L in P^p_N. The transformed CFG is returned as the updated P^p_N in line 7. In addition, PeelSingleLoop also returns the set $\textit{Locs}'$ of all CFG nodes newly added while peeling the loop L. The overall updated CFG and the set of all peeled nodes obtained after peeling all loops in P^p_N is returned in line 9.

Lemma 1. $\{\varphi_N\}\ \mathsf{P}_N\ \{\psi_N\}$ *holds iff* $\{\varphi_N\}\ \mathsf{P}^p_N\ \{\psi_N\}$ *holds.*

Algorithm 3 COMPUTEAFFECTED(P_N : Program, *peelNodes* : Peeled Statements)

```
1: $G(V, DE, CE)$ := COMPUTEREFINEDPDG($\mathsf{P}_N$);
2: AffectedVars := $\{N\}$;                                   ▷ $N$ is in the affected set
3: repeat
4:     WorkList := $V \setminus peelNodes$;                  ▷ all non-peeled nodes in $G$
5:     while WorkList $\neq \{\}$ do
6:         Remove a node $n$ from WorkList;
7:         if $\exists v.\ is\text{-}array(v) \wedge (\exists u.\ u \in subscript(v, n) \wedge u \in$ AffectedVars$)$ then
8:             AffectedVars := AffectedVars $\cup\ v$;
9:         if $\exists v.\ v \in uses(n)$ then
10:            if $\exists m.\ m \in reaching\text{-}def(v, n) \wedge m \in peelNodes$ then
11:                AffectedVars := AffectedVars $\cup\ def(n)$;
12:            if $\exists m.\ m \in reaching\text{-}def(v, n) \wedge def(m) \in$ AffectedVars then
13:                AffectedVars := AffectedVars $\cup\ def(n)$;
14:            if $v \in$ AffectedVars $\wedge\ n$ is a assignment node then
15:                AffectedVars := AffectedVars $\cup\ def(n)$;
16:            if $v \in$ AffectedVars $\wedge\ n$ is a predicate node then
17:                for each edge $(n, n') \in CE$ do
18:                    AffectedVars := AffectedVars $\cup\ def(n')$;
19: until AffectedVars does not change
20: return AffectedVars;
```

Affected Variable Analysis. Before we discuss the generation of $\partial\mathsf{P}_N$, we present an analysis that identifies variables/array elements that may take different values in P_N and P_{N-1}. For example, the first $k_L(N-1)$ iterations of L in P_N may not be semantically equivalent to the (entire) $k_L(N-1)$ iterations of L in P_{N-1}. This is because the semantics of statements in L may depend on the value of N either directly or indirectly. We call variables/array elements updated in such statements as *affected* variables. For every loop with statements having potentially different semantics in P_N and P_{N-1}, the difference program $\partial\mathsf{P}_N$ must have a version of the loop with statements that restore the effect of the first $k_L(N-1)$ iterations of L in P_N after the (entire) $k_L(N-1)$ iterations of L in P_{N-1} have been executed. Furthermore, for statements in P_N that are not enclosed within loops but have potentially different semantics from the corresponding statements in P_{N-1}, $\partial\mathsf{P}_N$ must also rectify the values of variables/array elements updated in such statements.

Subroutine COMPUTEAFFECTED, shown in Algorithm 3, computes the set of *affected* variables P_N. We first construct the program dependence graph by calling the function COMPUTEREFINEDPDG (line 1) defined in Algorithm 1. Let AffectedVars represent the set of *affected* variables/array elements. We initialize it (line 2) with variable N since its value is different in P_N and P_{N-1}. For a node n in the PDG G, we use *reaching-def*(v, n) to refer to the set of nodes where the variable/array element v is defined and the definition reaches its use at node n. In line 4, we collect nodes in the graph that are not the ones peeled from loops in P_N. The loop in lines 5-18 iterates over the collected nodes to identify affected variables. If a variable in the index expression of an array access is affected then that array element is considered affected (lines 7-8). A definition at a node n is affected (marked in line 11) if any variable v used in the statement (checked in line 9) is defined in a *peeled* node (line 10). Similarly if the reaching definition of v is affected (line 12) the definition at n is affected (line 13). A variable defined in terms of an affected variable is also deemed to be affected (lines 14-

15). Finally, a variable definition that is control dependent on an affected variable is also considered affected (lines 16-18). The computation of affected variables is iterated until the set AffectedVars saturates.

Lemma 2. *Variables/Array elements not present in* AffectedVars *have the same value after $k_L(N-1)$ iterations of its enclosing loop (if any) in P_{N-1} as in P_N.*

Generating the Difference Program $\partial \mathsf{P}_\mathbf{N}$. The routine ProgramDiff in Algorithm 4 shows how the difference program is computed. We peel each loop in the program and collect the list of peeled nodes (line 1) using Algorithm 2. We then compute the set of *affected* variables (line 2) using Algorithm 3. The difference program $\partial \mathsf{P}_N$ inherits the skeletal structure of the program P_N after peeling each loop (line 4). The algorithm then traverses the CFG of each loop in P_N and removes the loops (lines 16-17) that do not update any *affected* variables from $\partial \mathsf{P}_N$. For every CFG node in other loops, it determines the corresponding node type (assignment or branch) and acts accordingly (lines 7-14). To explain the intuition behind the steps of this algorithm, we use the convention that all variables and arrays of P_{N-1} have the suffix `_Nm1` (for N-minus-1), while those of P_N have the suffix `_N`. This allows us to express variables/array elements of P_N in terms of the corresponding variables/array elements of P_{N-1} in a systematic way in $\partial \mathsf{P}_N$, given that the intended composition is $\mathsf{P}_{N-1}; \partial \mathsf{P}_N$.

For assignment statements using simple arithmetic operators (`+,-,*,/`), the sub-routine AssignmentDiff in Algorithm 4 computes a "difference" statement as follows. We assume that Nodes(L) returns the set of CFG nodes in loop L. For every assignment statement of the form `v = E;` in L, a corresponding statement is generated in $\partial \mathsf{P}_N$ that expresses `v_N` in terms of `v_Nm1` and the difference (or ratio) between versions of variables/arrays that appear as sub-expressions in `E` in P_{N-1} and P_N. For example, the statement `A_N[i] = B_N[i] + v_N;` in P_N gives rise to the "difference" statement `A_N[i] = A_Nm1[i] + (B_N[i] - B_Nm1[i]) + (v_N - v_Nm1);` in $\partial \mathsf{P}_N$. Similarly, the statement `A_N[i] = B_N[i] * v_N;` in P_N gives rise to the "difference" statement `A_N[i] = A_Nm1[i] * (B_N[i] / B_Nm1[i]) * (v_N / v_Nm1);` under the assumption `B_Nm1[i] * v_Nm1` $\neq 0$.

There are additional kinds of statements that need special processing when generating $\partial \mathsf{P}_N$. These relate to accumulation of differences (or ratios). For example, if P_N has a loop `for(i = 0; i < N; i++) sum_N = sum_N + A_N[i];` then the difference `A_N[i] - A_Nm1[i]` is aggregated over all indices from 0 through $N-2$. In this case, the corresponding "difference" loop in $\partial \mathsf{P}_N$ has the following form: `sum_N = sum_Nm1; for(i = 0; i < N-1; i++) sum_N = sum_N + (A_N[i] - A_Nm1[i]);`. A similar aggregation for multiplicative ratios can also be defined. Sub-routine AggregateAssignmentDiff in Algorithm 4 generates these "difference" statements.

Note that expressions like `(B_N[i] - B_Nm1[i])` or `(v_N/v_Nm1)` can often be simplified from the already generated part of $\partial \mathsf{P}_N$. For example, if the already generated part has a statement of the form `B_N[i] = B_Nm1[i] + expr1;` or `v_N = expr2*v_Nm1;`, and if `expr1` and `expr2` are constants or functions of N and loop counters, then we can use `expr1` for `B_N[i] - B_Nm1[i]` and `expr2` for

Algorithm 4 ProgramDiff(P_N: program)

```
1: $\langle \mathsf{P}_N, peelNodes\rangle$ := PeelAllLoops($\mathsf{P}_N$);
2: $\mathsf{AffectedVars}$ := ComputeAffected($\mathsf{P}_N, peelNodes$);
3: Let the CFG of $\mathsf{P}_N$ be $(Locs, E, \mu)$;
4: $\partial\mathsf{P}_N := (Locs', E', \mu')$, where $Locs' := Locs$, $E' := E$, and $\mu' := \emptyset$;
5: for each loop $L \in$ Loops($\mathsf{P}_N$) do
6:     if $\exists v$ such that $v$ is updated in $L$ and $v \in \mathsf{AffectedVars}$ then
7:         for each node $n \in$ Nodes($L$) do
8:             $st_N := \mu(n)$;
9:             if $st_N$ is of the form $w_N := r_N^1$ op $r_N^2$ then
10:                $\mu'(n)$ := AssignmentDiff( $w_N := r_N^1$ op $r_N^2$ );
11:            else if $st_N$ is of the form $w_N := w_N$ op $r_N^1$ wherein $w_N$ is a scalar then
12:                $\mu'(n)$ := AggregateAssignmentDiff( $L$, $w_N := w_N$ op $r_N^1$ );
13:            else                                   ▷ $st_N$ is a conditional statement
14:                $\mu'(n)$ := BranchDiff( $st_N$, $\mathsf{AffectedVars}$ );
15:    else                                       ▷ Remove loop $L$ from CFG of $\partial\mathsf{P}_N$
16:        $(n_1, n, \mathsf{U})$ := IncomingEdge($L$); $(n, n_2, \mathbf{ff})$ := ExitEdge($L$);
17:        $E' := E' \setminus \{(n_1, n, \mathsf{U}), (n, n_2, \mathbf{ff})\} \cup \{(n_1, n_2, \mathsf{U})\}$; $Locs' := Locs' \setminus \mathsf{Nodes}(L)$;
18: return $\partial\mathsf{P}_N$;
```

AssignmentDiff($w_N := r_N^1$ op r_N^2)

```
1: Let invop be the arithmetic inverse operator of op;
   ▷ + and − are inverse operators of each other, and so are × and ÷
2: if op $\in \{+, \times\}$ then
3:     return $w_N := w_{Nm1}$ op (Simplify($r_N^1$ invop $r_{Nm1}^1$) op Simplify($r_N^2$ invop $r_{Nm1}^2$));
4: else if op $\in \{-, \div\}$ then
5:     return $w_N := w_{Nm1}$ invop (Simplify($r_N^1$ op $r_{Nm1}^1$) op Simplify($r_N^2$ op $r_{Nm1}^2$));
6: else
7:     throw "Specified operator not handled";
```

AggregateAssignmentDiff(L: loop, $w_N := w_N$ op r_N^1)

```
1: $n_{fresh}$ := FreshNode(); $\mu'(n_{fresh}) := (w_N := w_{Nm1})$; $Locs' := Locs' \cup \{n_{fresh}\}$;
2: $(n', n'', \mathsf{U})$ := IncomingEdge($L$);
3: $E' := E' \setminus \{(n', n'', \mathsf{U})\} \cup \{(n', n_{fresh}, \mathsf{U}), (n_{fresh}, n'', \mathsf{U})\}$;
4: if op $\in \{+, *\}$ then
5:     return $w_N := w_N$ op Simplify($r_N^1$ invop $r_{Nm1}^1$);
6: else if op $\in \{-, \div\}$ then
7:     return $w_N := w_N$ op Simplify($r_N^1$ op $r_{Nm1}^1$);
8: else
9:     throw "Specified operator not handled";
```

BranchDiff(st_N: branch condition, $\mathsf{AffectedVars}$: set of affected variables)

```
1: Let $n$ be CFG node corresponding to $st_N$;
2: if ($\exists v$ such that v is read in $st_N$ and $v \in \mathsf{AffectedVars}$) $\vee$ ($st_N \neq st_{N-1}$ is satisfiable) then
3:     throw "Branch conditions in $\mathsf{P}_N$ and $\mathsf{P}_{N-1}$ may not evaluate to same value";
4: else
5:     return $st_{N-1}$;
```

`v_N/v_Nm1` respectively. We use these optimizations aggressively in the function Simplify used in AssignmentDiff and AggregateAssignmentDiff.

For every CFG node representing a conditional branch in P_N, Algorithm BranchDiff is used to determine if the result of the condition check can differ in P_N and P_{N-1}. If not, the conditional statement can be retained as such in the "difference" program. Otherwise, our current technique cannot compute $\partial\mathsf{P}_N$ and we report a failure of our technique (see body of BranchDiff). For example, the conditional statement `if (t3 == 0)` in line 10 of Fig. 1(a) behaves identically in P_{N-1} and P_N, and therefore can be used as is in the loop in the difference program.

Lemma 3. $\partial\mathsf{P}_N$ *generated by* ProgramDiff *is such that, for all* $N > 1$, $\{\varphi(N)\}\ \mathsf{P}_{N-1}; \partial\mathsf{P}_N\ \{\psi(N)\}$ *holds iff* $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *holds.*

Algorithm 5 SimplifyDiff($\partial\mathsf{P}_N$: difference program)

```
1: $\partial\mathsf{P}_N := (Locs, E, \mu)$
2: $\partial\mathsf{P}'_N := (Locs', E', \mu')$, where $Locs' := Locs$, $E' := E$, and $\mu' := \mu$;
3: for each loop $L \in$ Loops($\partial\mathsf{P}_N$) do
4:     $(n_1, n, \mathsf{U}) :=$ IncomingEdge($L$); $(n, n_2, \mathbf{ff}) :=$ ExitEdge($L$);
5:     if Loop body of $L$ is of the form $w_N := w_N$ op $expr$, wherein $w_N$ is a scalar variable then
6:         $n_{acc}$ = FreshNode();
7:         if op $\in \{+, -\}$ then
8:             $\mu'(n_{acc}) := (w_N := w_N$ op Simplify$(k_L(N-1) * expr))$;
9:         else if op $\in \{*, \div\}$ then
10:            $\mu'(n_{acc}) := (w_N := w_N$ op Simplify$(expr^{k_L(N-1)}))$;
11:        else throw "Specified operator not handled";
12:        $E' := E' - \{(n_1, n, \mathsf{U}), (n, n_2, \mathbf{ff})\} \cup \{(n_1, n_{acc}, \mathsf{U}), (n_{acc}, n_2, \mathsf{U})\}$;
13:        $Locs' := Locs' -$ Nodes($L$) $\cup \{n_{acc}\}$ ;
14:    if Loop body of $L$ is of the form $w_N := w_{Nm1}$ or $w_N := w_N$ then
15:        $E' := E' - \{(n_1, n, \mathsf{U}), (n, n_2, \mathbf{ff})\} \cup \{(n_1, n_2, \mathsf{U})\}$; $Locs' := Locs' -$ Nodes($L$);
16: return $\partial\mathsf{P}'_N$
```

Simplifying the Difference Program. While we have described a simple strategy to generate $\partial\mathsf{P}_N$ above, this may lead to redundant statements in the naively generated "difference" code. For example, we may have a loop like `for (i=0; i < N-1; i++)  A_N[i] = A_Nm1[i];`. Our implementation aggressively optimizes and removes such redundant code, renaming variables/arrays as needed (see routine SimplifyDiff in Algorithm 5). The program $\partial\mathsf{P}_N$ may also contain loops that compute values of variables that can be accelerated. For example, we may have a loop `for(i=0; i < N-1; i++)  sum = sum + 1;`. Algorithm SimplifyDiff removes this loop and introduces the statement `sum = sum + (N-1);`. This helps in $\partial\mathsf{P}_N$ having fewer and simpler loops in a lot of cases.

Lemma 4. *Program $\partial\mathsf{P}'_N$ generated by* SimplifyDiff *is such that, for all $N > 1$, $\{\varphi(N)\}\ \mathsf{P}_{N-1}; \partial\mathsf{P}'_N\ \{\psi(N)\}$ holds iff $\{\varphi(N)\}\ \mathsf{P}_{N-1}; \partial\mathsf{P}_N\ \{\psi(N)\}$ holds.*

Generating the Difference Pre-condition $\partial\varphi(\mathbf{N})$. We now present a simple syntactic algorithm, called SyntacticDiff, for generation of the difference pre-condition $\partial\varphi(N)$. Although this suffices for all our experiments, for the sake of completeness, we present later a more sophisticated algorithm for generating $\partial\varphi(N)$ simultaneously with $\mathsf{Pre}(N)$.

Formally, given $\varphi(N)$, algorithm SyntacticDiff generates a formula $\partial\varphi(N)$ such that $\varphi(N) \rightarrow (\varphi(N-1) \wedge \partial\varphi(N))$. Observe that if such a $\partial\varphi(N)$ exists, then $\varphi(N) \rightarrow \varphi(N-1)$ holds as well. Therefore, we can use the validity of $\varphi(N) \rightarrow \varphi(N-1)$ as a test to decide the existence of $\partial\varphi(N)$.

If $\varphi(N)$ is of the syntactic form $\forall i \in \{0 \ldots N\}\ \widehat{\varphi}(i)$, then $\partial\varphi(N)$ is easily seen to be $\hat{\varphi}(N)$. If $\varphi(N)$ is of the syntactic form $\varphi^1(N) \wedge \cdots \wedge \varphi^k(N)$, then $\partial\varphi(N)$ can be computed as $\partial\varphi^1(N) \wedge \cdots \wedge \partial\varphi^k(N)$. Finally, if $\varphi(N)$ doesn't belong to any of these syntactic forms or if condition 2(a) of Theorem 1 is violated by the heuristically computed $\partial\varphi(N)$, then we over-approximate $\partial\varphi_N$ by True. For a large fraction of our benchmarks, the pre-condition $\varphi(N)$ was True, and hence $\partial\varphi(N)$ was also True.

Generating the Formula $\mathsf{Pre}(\mathbf{N-1})$. We use Dijsktra's weakest pre-condition computation to obtain $\mathsf{Pre}(N-1)$ after the "difference" pre-condition $\partial\varphi(N)$ and the "difference" program $\partial\mathsf{P}_N$ have been generated. The weakest pre-condition

Algorithm 6 FPIVerify(P_N: program, $\varphi(N)$: pre-condn, $\psi(N)$: post-condn)

```
1: if Base case check $\{\varphi(1)\}$ $\mathsf{P}_1$ $\{\psi(1)\}$ fails then
2:     return "Counterexample found!";
3: $\partial\varphi(N)$ := SyntacticDiff($\varphi(N)$);
4: $\partial\mathsf{P}_N$ := ProgramDiff($\mathsf{P}_N$);
5: $\partial\mathsf{P}_N$ := SimplifyDiff($\partial\mathsf{P}_N$);                ▷ Simplify and Accelerate loops
6: $i := 0$;
7: $\mathsf{Pre}_i(N) := \psi(N)$;
8: $c\_\mathsf{Pre}_i(N)$ := True;                              ▷ Cumulative conjoined pre-condition
9: do
10:     if $\{c\_\mathsf{Pre}_i(N-1) \wedge \psi(N-1) \wedge \partial\varphi(N)\}$ $\partial\mathsf{P}_N$ $\{c\_\mathsf{Pre}_i(N) \wedge \psi(N)\}$ then
11:         return True;                                         ▷ Assertion verified
12:     $i := i + 1$;
13:     $\mathsf{Pre}_i(N-1)$ := LoopFreeWP($\mathsf{Pre}_{i-1}(N), \partial\mathsf{P}_N$);   ▷ Dijkstra's WP sans WP-for-loops
14:     if no new $\mathsf{Pre}_i(N-1)$ obtained then            ▷ Can happen if $\partial\mathsf{P}_N$ has a loop
15:         return FPIVerify($\partial\mathsf{P}_N$, $c\_\mathsf{Pre}_{i-1}(N-1) \wedge \psi(N-1) \wedge \partial\varphi(N)$, $c\_\mathsf{Pre}_{i-1}(N) \wedge \psi(N)$);
16:     else
17:         $c\_\mathsf{Pre}_i(N) := c\_\mathsf{Pre}_{i-1}(N) \wedge \mathsf{Pre}_i(N)$;
18: while Base case check $\{\varphi(1)\}$ $\mathsf{P}_1$ $\{c\_\mathsf{Pre}_i(1)\}$ passes;
19: return False;                                  ▷ Failed to prove by full-program induction
```

can always be computed using quantifier elimination engines in state-of-the-art SMT solvers like Z3 if $\partial\mathsf{P}_N$ is loop-free. In such cases, we use a set of heuristics to simplify the calculation of the weakest pre-condition before harnessing the power of the quantifier elimination engine. If $\partial\mathsf{P}_N$ contains a loop, it may still be possible to obtain the weakest pre-condition if the loop doesn't affect the post-condition. Otherwise, we compute as much of the weakest pre-condition as can be computed from the non-loopy parts of $\partial\mathsf{P}_N$, and then try to recursively solve the problem by invoking full-program induction on $\partial\mathsf{P}_N$ with appropriate pre- and post-conditions.

Verification by Full-program Induction. The basic full-program induction algorithm is presented as routine FPIVerify in Algorithm 6. The main steps of this algorithm are: checking conditions 3(a), 3(b) and 3(c) of Theorem 1 (lines 1, 18 and 10), calculating the weakest pre-condition of the relevant part of the post-condition (line 13), and strengthening the pre-condition and post-condition with the weakest pre-condition thus calculated (line 17). Since the weakest pre-condition computed in every iteration of the loop ($Pre_i(N-1)$ in line 13) is conjoined to strengthen the inductive pre-condition ($c\_Pre_i(N)$ in line 17), it suffices to compute the weakest pre-condition of $Pre_{i-1}(N)$ (instead of $c\_Pre_i(N) \wedge \psi(N)$) in line 13. The possibly multiple iterations of strengthening of pre- and post-conditions is effected by the loop in lines 9-18. In case the loop terminates via the `return` statement in line 11, the inductive claim has been successfully proved. If the loop terminates by a violation of the condition in line 18, we report that verification by full-program induction failed. In case $\partial\mathsf{P}_N$ has loops and no further weakest pre-conditions can be generated, we recursively invoke FPIVerify on $\partial\mathsf{P}_N$ in line 15. This situation arises if, for example, we modify the example in Fig. 1(a) by having the statement `C[t3] = N;` (instead of `C[t3] = 0;`) in line 10. In this case, $\partial\mathsf{P}_N$ has a single loop corresponding to the third loop in Fig. 1(a). The difference program of $\partial\mathsf{P}_N$ is, however, loop-free, and hence the recursive invocation of full-program induction on $\partial\mathsf{P}_N$ easily succeeds.

Algorithm 7 FPIDecomposeVerify(i : integer)

```
1:  do
2:      $\langle \mathsf{Pre}'_i(N-1), \partial\varphi'_i(N)\rangle$ := NextDecomposition($\mathsf{Pre}_i(N-1)$);
3:      Check if (a) $\partial\varphi'_i(N) \wedge \mathsf{Pre}'_i(N-1) \rightarrow \mathsf{Pre}_i(N-1)$,
4:                (b) $\varphi(N) \rightarrow \varphi(N-1) \wedge \big(\partial\varphi'_i(N) \wedge \partial\varphi(N)\big)$,
5:                (c) $\mathsf{P}_{N-1}$ does not update any variable or array element in $\partial\varphi'_i(N)$
6:      if any check in lines 3-5 fails then
7:          if HasNextDecomposition($\mathsf{Pre}_i(N-1)$) then
8:              continue;
9:          else
10:             return False;
11:     if $\{c\_\mathsf{Pre}_{i-1}(N-1) \wedge \psi(N-1) \wedge \mathsf{Pre}_i(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{c\_\mathsf{Pre}_{i-1}(N) \wedge \psi(N) \wedge \mathsf{Pre}'_i(N)\}$
    then
12:         return True;                                                    ▷ Assertion verified
13:     else
14:         $c\_\mathsf{Pre}_i(N) := c\_\mathsf{Pre}_{i-1}(N) \wedge \mathsf{Pre}'_i(N)$;
15:         $i := i + 1$;
16:         $\mathsf{Pre}_i(N-1)$ := LoopFreeWP($\mathsf{Pre}'_{i-1}(N), \partial\mathsf{P}_N$);    ▷ Dijkstra's WP sans WP-for-loops
17:         if $\{\varphi(1)\}\ \mathsf{P}_1\ \{c\_\mathsf{Pre}_{i-1}(1) \wedge \mathsf{Pre}_i(1)\}$ does not hold then
18:             $i := i - 1$;
19:         else
20:             $prev\_\partial\varphi(N) := \partial\varphi(N)$;
21:             $\partial\varphi(N) := \partial\varphi'_{i-1}(N) \wedge \partial\varphi(N)$;
22:             if FPIDecomposeVerify($i$) returns False then
23:                 $i := i - 1$; $\partial\varphi(N) := prev\_\partial\varphi(N)$;
24:             else
25:                 return True;
26: while HasNextDecomposition($\mathsf{Pre}_i(N-1)$);
27: return False;
```

Generalized FPI Algorithm. While algorithm FPIVerify suffices for all of our experiments, we may not always be so lucky. Specifically, even if $\partial\mathsf{P}_N$ is loop-free, the analysis may exit the loop in lines 9-18 of FPIVerify by violating the base case check in line 18. To handle (at least partly) such cases, we propose the following strategy. Whenever a (weakest) pre-condition $\mathsf{Pre}_i(N-1)$ is generated, instead of using it directly to strengthen the current pre- and post-conditions, we "decompose" it into two formulas $\mathsf{Pre}'_i(N-1)$ and $\partial\varphi'_i(N)$ with a two-fold intent: (a) potentially weaken $\mathsf{Pre}_i(N-1)$ to $\mathsf{Pre}'_i(N-1)$, and (b) potentially strengthen the difference formula $\partial\varphi(N)$ to $\partial\varphi'_i(N) \wedge \partial\varphi(N)$. The checks for these intended usages of $\mathsf{Pre}'_i(N-1)$ and $\partial\varphi'_i(N)$ are implemented in lines 3, 4, 5, 11 and 17 of routine FPIDecomposeVerify, shown as Algorithm 7. This routine is meant to be invoked as FPIDecomposeVerify(i) after each iteration of the loop in lines 9-18 of routine FPIVerify (so that $\mathsf{Pre}_i(N)$, $c\_\mathsf{Pre}_i(N)$ etc. are initialized properly). In general, several "decompositions" of $\mathsf{Pre}_i(N)$ may be possible, and some of them may work better than others. FPIDecompseVerify permits multiple decompositions to be tried through the use of the NextDecomposition and HasNextDecomposition functions. Lines 22-25 of FPIDecomposeVerify implement a simple back-tracking strategy, allowing a search of the space of decompositions of $\mathsf{Pre}_i(N-1)$. Observe that when we use FPIDecomposeVerify, we simultaneously compute a difference formula $(\partial\varphi'_i(N) \wedge \partial\varphi(N))$ and an inductive pre-condition $(c\_\mathsf{Pre}_{i-1}(N) \wedge \mathsf{Pre}'_i(N))$.

Lemma 5. *Algorithms* FPIVerify *and* FPIDecomposeVerify *ensure conditions 2 and 3 of Theorem 1 upon successful termination.*

While we have presented our technique focusing on a single symbolic parameter N, a straightforward extension works for multiple independent parameters, multiple independent array sizes, different induction directions, and non-uniform loop termination conditions. For more details, please refer to the long version of our paper at [3].

Limitations. There are several scenarios under which full-program induction may not produce a conclusive result. Currently, we only analyze programs with non-nested loops with $+, -, \times, \div$ expressions in assignments. We also do not handle branch conditions that are dependent on the parameter N (this doesn't include loop conditions, which are handled by unrolling the loop). The technique also remains inconclusive when the difference program $\partial\mathsf{P}_N$ does not have fewer loops than the original program. Reduction in verification complexity of the program, in terms of the number of loops and assignment statements dependent on N, is crucial to the success of full-program induction. Finally, our technique may fail to verify a correct program if the heuristics used for weakest pre-condition either fail or return a pre-condition that causes violation of the base case check in line 18 of FPIVerify. Despite these limitations, our experiments show that full-program induction performs remarkably well on a large suite of benchmarks.

4 Implementation and Experiments

We have implemented our technique in a prototype tool called Vajra, available at [5]. It takes a C program in SVCOMP format as input. The tool, written in `C++`, is built on top of the LLVM/CLANG [22] 6.0.0 compiler infrastructure and uses Z3 [25] v4.8.7 as the SMT solver to prove Hoare triples for loop-free programs.

We have evaluated Vajra on a test-suite of 42 safe benchmarks inspired from different algebraic functions that compute polynomials as well as a standard array operations such as copy, min, max and compare. Our programs take a symbolic parameter N which specifies the size of each array as well as the number of times each loop executes. Assertions, possibly quantified, are (in-)equalities over array elements, scalars and (non-)linear polynomial terms over N.

All experiments were performed on a Ubuntu 18.04 machine with 16GB RAM and running at 2.5 GHz. We have compared Vajra against VIAP(v1.0) [26], VeriAbs(v1.3.10) [8], Booster(v0.2) [1], Vaphor(v1.2) [24] and FreqHorn(v3) [10]. C programs were manually converted to mini-Java as required by Vaphor and CHC's as required by FreqHorn. Our results are shown in Table 1. Vajra verified 36 benchmarks, compared to 23 verified by VIAP, 12 by VeriAbs, 8 by Booster, 5 each by Vaphor and FreqHorn. Vajra was unable to compute the difference program for 5 benchmarks and was inconclusive on 1 benchmark.

Vajra verified 17 benchmarks on which VIAP diverged, primarily due to the inability of VIAP's heuristics to get closed form expressions. VIAP verified 4 benchmarks that could not be verified by the current version of Vajra due to syntactic limiations. Vajra, however, is two orders of magnitude faster than VIAP on programs that were verified by both. Vajra proved 28 benchmarks on which VeriAbs diverged. VeriAbs ran out of time on programs where loop shrinking and merging abstractions were not strong enough

Name	#L	T1	T2	T3	T4	T5	T6
pcomp	3	✓0.68	TO	TO	?0.23	TO	?0.58
ncomp	3	✓0.68	TO	TO	?0.41	TO	?0.68
eqnm2	2	✓0.52	TO	TO	?0.07	TO	?0.59
eqnm3	2	✓0.53	TO	TO	?0.07	TO	?0.56
eqnm4	2	✓0.51	TO	TO	?0.07	TO	?0.60
eqnm5	2	✓0.55	TO	TO	?0.07	TO	?0.58
sqm	2	✓0.51	✓69.7	TO	?0.11	TO	?0.57
res1	4	✓0.17	TO	TO	TO	TO	TO
res1o	4	✓0.18	TO	TO	TO	TO	TO
res2	6	✓0.20	TO	TO	TO	TO	TO
res2o	6	✓0.22	TO	TO	TO	TO	TO
ss1	4	✓0.40	TO	TO	✗0.13	?19.2	?1.7
ss2	6	✓0.46	TO	TO	✗0.13	TO	?9.7
ss3	5	✓0.35	TO	TO	✗0.13	TO	?2.1
ss4	4	✓0.29	TO	TO	✗0.13	TO	?1.6
ssina	5	✓0.41	✓72.5	TO	TO	TO	?2.0
sina1	2	✓0.56	✓65.4	TO	TO	TO	TO
sina2	3	✓0.69	✓66.5	TO	TO	TO	TO
sina3	4	✓0.83	TO	TO	TO	TO	TO
sina4	4	✓0.85	TO	TO	TO	TO	TO
sina5	5	✓0.93	TO	TO	TO	TO	TO

Name	#L	T1	T2	T3	T4	T5	T6
zerosum1	2	✓0.33	✓62.0	✓11	✓0.77	✗0.29	TO
zerosum2	4	✓0.46	✓75.8	✓18	TO	✗1.64	TO
zerosum3	6	✓0.59	✓73.1	✓39	TO	✗3.13	TO
zerosum4	8	✓0.76	✓76.1	TO	?18.2	✗6.85	TO
zerosum5	10	✓0.97	✓80.6	TO	?16.5	✗10.4	TO
zerosumm2	4	✓0.46	✓71.5	✓24	TO	✗1.22	TO
zerosumm3	6	✓0.59	✓70.9	TO	TO	✗5.22	TO
zerosumm4	8	✓0.77	✓76.4	TO	?16.7	✗12.39	TO
zerosumm5	10	✓0.98	✓81.7	TO	?18.7	✗22.8	TO
zerosumm6	12	✓1.29	✓86.8	TO	?16.1	TO	TO
copy9	9	✓0.69	✓86.8	✓3.91	✓18.8	TO	✓0.67
min	1	✓0.48	✓23.6	✓3.82	✓0.52	✓0.14	✓0.13
max	1	✓0.46	✓25.4	✓4.70	✓1.0	✓0.28	✓0.18
compare	1	✓0.82	✓18.8	✓17.9	✓0.06	✓0.84	✓0.31
conda	3	✓0.72	✓13.9	TO	✓0.07	✓0.09	TO
condn	1	?0.51	✓14.7	✓18.9	✓0.02	✓0.15	✓0.20
condm	2	?0.59	✓20.5	✓16.7	✓0.04	TO	-
condg	3	?0.52	TO	TO	TO	TO	TO
modn	2	?0.63	✓22.6	TO	-	TO	TO
mods	4	?0.61	TO	✓18.2	-	-	-
modp	2	?0.71	✓17.3	✓40	-	?32	-

Table 1. First column is the benchmark name. Second column indicates the number loops in the benchmark (excluding the assertion loop). Successive columns indicate the results generated by tools and the time taken where T1 is Vajra, T2 is VIAP, T3 is VeriAbs, T4 is Booster, T5 is Vaphor, T6 is FreqHorn. ✓indicates assertion safety, ✗indicates assertion violation, ? indicates unknown result, and - indicates an abrupt stop. All the times are in seconds. TO is time-out of 100 secs.

to prove the assertions. VeriAbs reported 1 program as unsafe due to the imprecision of its abstractions and it proved 4 benchmarks that Vajra could not. Vajra verified 30 benchmarks that Booster could not. Booster reported 4 benchmarks as unsafe due to imprecise abstractions, its fixed-point computation engine reported unknown result on 12 benchmarks and it ended abruptly on 3 benchmarks. Booster also proved 2 benchmarks that couldn't be handled by the current version of Vajra due to syntactic limitations. Vajra verified 32 benchmarks on which Vaphor was inconclusive. Distinguished cell abstraction in Vaphor is unable to prove safety of programs, when the value at each array index needs to be tracked. Vaphor reported 9 programs unsafe due to imprecise abstraction, returned unknown on 2 programs and ended abruptly on 1 program. Vaphor proved a benchmark that Vajra could not. Vajra verified 32 programs on which FreqHorn diverged, especially when constants and terms that appear in the inductive invariant are not syntactically present in the program. FreqHorn ran out of time on 22 programs, reported unknown result on 12 and ended abruptly on 3 benchmarks. FreqHorn verified a benchmark with a single loop that Vajra could not. On an extended set of 231 benchmarks, Vajra verified 110 programs out of 121 safe programs, falsified 108 out of 110 unsafe programs, and was inconclusive on the remaining 13 programs.

5 Conclusion

We presented a novel property-driven verification method that performs induction over the entire program via parameter N. Significantly, this obviates the need for loop-specific invariants. Experiments show that full-program induction performs remarkably well vis-a-vis state-of-the-art tools for analyzing array manipulating programs. Further improvements in the algorithms for computing difference programs and for strengthening of pre- and post-conditions are envisaged as part of future work.

Data Availability Statement

The datasets generated and analyzed during the current study are available in the figshare repository: https://doi.org/10.6084/m9.figshare.11875428.v1

References

1. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: An acceleration-based verification framework for array programs. In: Proc. of ATVA. pp. 18–23 (2014)
2. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Proc. of VMCAI. pp. 378–394 (2007)
3. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction, https://www.cse.iitb.ac.in/~supratik/publications/papers/FPI_longversion.html
4. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying Array Manipulating Programs by Tiling. In: Proc. of SAS. pp. 428–449 (2017)
5. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying Array Manipulating Programs with Full-program Induction - Artifacts TACAS 2020. Figshare (2020). https://doi.org/10.6084/m9.figshare.11875428.v1
6. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. FMSD **19**(1), 7–34 (2001)
7. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proc. of POPL. pp. 105–118 (2011)
8. Darke, P., Prabhu, S., Chimdyalwar, B., Chauhan, A., Kumar, S., Basakchowdhury, A., Venkatesh, R., Datar, A., Medicherla, R.K.: VeriAbs: Verification by abstraction and test generation. In: TACAS (Competition Contribution). pp. 457–462 (2018)
9. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3), 35–45 (2007)
10. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided-synthesis. In: Proc. of CAV. pp. 259–277 (2019)
11. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. TOPLAS **9**(3), 319–349 (1987)
12. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Proc. of FME. pp. 500–517 (2001)
13. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: Proc. of POPL. pp. 338–350 (2005)
14. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proc. of POPL. pp. 235–246 (2008)
15. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Proc. of ATVA. pp. 248–266 (2018)
16. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proc. of PLDI. pp. 339–348 (2008)
17. Henzinger, T.A., Hottelier, T., Kovács, L., Rybalchenko, A.: Aligators for arrays (tool paper). In: Proc. of LPAR. pp. 348–356 (2010)
18. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. of NFM. pp. 41–55 (2011)

19. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Proc. of CAV. pp. 193–206 (2007)
20. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Proc. of POPL. pp. 107–120 (1998)
21. Komuravelli, A., Bjorner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using Horn clauses over integers and arrays. In: Proc. of FMCAD. pp. 89–96 (2015)
22. Lattner, C.: LLVM and Clang: Next generation compiler technology. In: The BSD Conference. pp. 1–2 (2008)
23. Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: Proc. of VMCAI. pp. 282–299 (2015)
24. Monniaux, D., Gonnord, L.: Cell Morphing: From array programs to array-free horn clauses. In: Proc. of SAS. pp. 361–382 (2016)
25. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. of TACAS. pp. 337–340 (2008)
26. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Proc. of VSTTE. pp. 38–49 (2018)
27. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proc. of POPL. pp. 12–27 (1988)
28. Seghir, M.N., Brain, M.: Simplifying the verification of quantified array assertions via code transformation. In: Proc. of LOPSTR. pp. 194–212 (2012)
29. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. of FMCAD. pp. 127–144 (2000)
30. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. ACM Sigplan Notices **44**(6), 223–234 (2009)

Interpretation-Based Violation Witness Validation for C: NITWIT

Jan Švejda, Philipp Berger, and Joost-Pieter Katoen

RWTH Aachen University, Germany
{berger, katoen}@cs.rwth-aachen.de
jan.svejda@rwth-aachen.de

Abstract. As software verification is gaining traction in academia and industry the number and complexity of verification tools is growing constantly. This initiated research and interest into exchangeable verification witnesses as well as tools for automated witness validation. Initial witness validators used model checkers that were amended to benefit from guidance information provided by the witness. This approach comes with substantial overhead. Second-generation execution-based validators traded speed for reduced strength in case of incomplete and non-exact witnesses. This was done by extracting test harnesses and compiling them with the original program. We present the NITWIT tool, a new interpretation-based violation witness validator for C programs that is trimmed to be fast and memory efficient. It verifies a record number of witnesses of SV-COMP'20 in the ReachSafety category. Our novel tool exchanges initial compilation overhead and optimized execution for rapid startup performance. NITWIT borrows C semantics from the compiler used for compilation. This offloads this hard-to-get-right task and enables using several compilers in parallel to inspect possible semantic differences.

1 Introduction

The importance of witnesses. Model checking is a very successful automated verification technique with many applications. Its usage is rapidly increasing and one may fairly argue that model checking has penetrated various industries. This is true as well for software model checkers that, as opposed to first generation model checkers, directly verify program code. Model checking is in particular a very effective *bug hunting technique*: in case a property is violated, a counterexample is provided witnessing the property's violation. This is why they are often named witnesses. As phrased by Clarke *et al.* [16] "It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems. Some people use model checking just for this feature."

Witness validation. Early model checkers provided witnesses for safety properties such as "certain bad states should always be avoided" as finite paths that

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 40–57, 2020.
https://doi.org/10.1007/978-3-030-45190-5_3

end in a bad state. A simple witness-steered simulation could reveal the flaw. Modern model checkers heavily use abstraction, and witnesses are no longer concrete, but rather phrased in terms of some abstract model. This is in particular true for software model checkers. Witnesses are in fact finite paths through an abstracted program representing sets of paths in the concrete program that is to be verified. These sets may contain spurious concrete paths. This raises the question whether witnesses are correct. Witness validation is the process of checking whether a witness produced by a software model checker is indeed a witness showing that the concrete program violates the property. Software model checkers such as CBMC, CPAchecker and so on, that generate witnesses are called *producers*, while software tools that perform the witness validation are named *validators*. With a single exception [12], existing validators are incorporated or directly built on top of the existing software model checkers CPAchecker [13] or Ultimate Automizer [19,18,17].

A format for witnesses. In order to facilitate the validation of witnesses by various different tools, a witness format has been developed that nowadays is used by many software model checkers. For safety properties as above, this format prescribes how to represent a witness for reaching a bad state. Due to this format, witnesses are exchangeable and witness validation can be done using different techniques and tools. This format allows (i) a cross-platform exchange of information that enables "drop-in" replacement of tools such as visualization and reviews of results [10], (ii) validation of witnesses which strengthens trust in verification results, especially if the verifier and validator use different techniques and (iii) a significant amount of false bug alarms to be caught by failed validation.

Witness validation in software verification competitions. Since a few years, the use of witnesses has become an important part in software competitions such as the annual TACAS Competition on Software Verification (SV-COMP) [2,3,4,5,6]. SV-COMP is a competition in automatic software verification, in which academic, but also some industrial, software verifiers participate. In the 2019 edition [6], 31 verifiers participated in verifying 10 522 verification tasks for C programs (and 368 for Java programs). SV-COMP has different categories, such as reachability, memory and concurrency safety, absence of overflows, and termination. SV-COMP adopted violation witnesses as part of its benchmark scoring schema since 2015 [3] and adhered to it also in the following editions [4,5,6]. This means that a verifier does not receive a point for a violated property unless the produced violation witness could be validated by at least one validator. This applies to all categories. To reflect that violation witnesses contain sufficient information for validation, the validators are granted only limited resources (e.g., only 10% of the amount of time available for verification, and 7 GB memory). Correctness witnesses were incorporated into the score evaluation in 2017 [5] – since this competition, validated correctness witnesses yield a bonus point for the producer.

Contributions of this paper. This paper presents the interpretation-based witness validator NITWIT. It validates violation witnesses for safety reachability properties as above. It does so for C programs. In contrast to most other validators (a) it does not rely on an existing software model checker, and (b) exploits an interpretation-based approach. NITWIT uses a home-made extension of the *PicoC* interpreter which feeds a witness automaton with steering information during a step-by-step interpretation of the C program, see Figure 1. NITWIT was evaluated on 11 533 violation witnesses in the ReachSafety category during SV-COMP 2020 and we compared its outcomes to another five witness validators that participated. NITWIT was able to validate more witnesses in this category (8 526 in total) than all its competitors, and did so substantially faster. In addition, NITWIT was able to validate 399 witnesses that could not be validated with any of the five competitors.

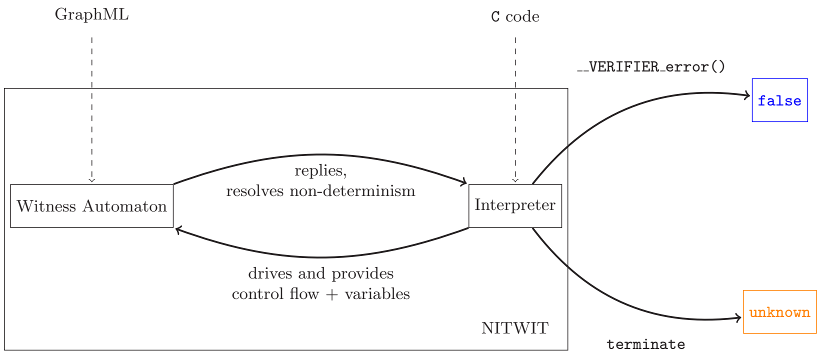


Fig. 1: High-level architecture of the NITWIT Validator.

2 Background

The need for achieving portability of counterexamples and proofs between tools gave rise to a type of non-deterministic finite automaton (NFA) called a *witness automaton*, or simply a witness [11]. Two types of witnesses exist – a violation and a correctness witness. In this paper, we focus on *violation witnesses.*

The concepts defined in this section follow the definitions of [22,11]. We represent programs by control-flow graphs (CFGs).

Definition 1 (Control-flow graph). *A* control-flow graph $\mathcal{C} = (L, l_0, G, V)$ *is a finite set of locations L, initial location $l_0 \in L$, $G \subseteq L \times \mathrm{Op} \times L$ a set of edges where* $\mathrm{Op} = \{skip, assume(\varphi), assign(x, E)\}$ *with $x \in V, \varphi$ a predicate over the program variables V and E an expression over V.*

In a CFG over $V = \{x, y\}$, e.g., an assignment is of the form $x := x + y$. The interpretation of a CFG is given by a (possibly countably infinite) transition system where states are of the form (l, v) where $l \in L$ and v is a variable assignment over V. For the sake of brevity, we refrain from a formal definition.

For predicate φ over V, let $v \models \varphi$ denote that φ holds in valuation v. A witness automaton (WA) is a finite-state automaton (NFA) used by the validator to run in parallel to the CFG such that a program run violating the specification is accepted.

Definition 2 (Witness automaton). *A* witness automaton *(WA)* $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_E)$ *for a CFG* $\mathcal{C} = (L, l_0, G, V)$ *is an NFA with states* Q*, initial state* $q_0 \in Q$ *and* $\delta : Q \times \Sigma \to 2^Q$ *as usual,* q_E *the accepting state and* $\Sigma \subseteq 2^G \times \Phi$*, where* Φ *is the set of predicates over* V*.*

The transitions of $\mathcal{A}$ have source code and guards [11] that identify program edges and place constraints on variable assignments respectively. They correspond to pairs (D_i, φ_i), where $D_i \subseteq G$ and φ_i is a predicate over variables.

Definition 3 (Simulation). *Let* $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_E)$ *be a WA for a CFG* $\mathcal{C} = (L, l_0, G, V)$ *and* $\rho = l_0 \xrightarrow{g_1} \dots \xrightarrow{g_n} l_n$ *a path in* $\mathcal{C}$*. The run* $q_0 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} q_n$ *in* $\mathcal{A}$ simulates ρ *iff* $\sigma_{i+1} = (D_{i+1}, \varphi_{i+1})$ *with* $(l_i, g_{i+1}, l_{i+1}) \in D_{i+1}$ *and* $v_{i+1} \models \varphi_{i+1}$ *for some state* (l_{i+1}, v_{i+1})*. The run is* accepted *if* $q_n = q_E$ *and* $\mathcal{L}(\mathcal{A})$ *is the set of words* $\sigma_1 \dots \sigma_n$ *for which* $\mathcal{A}$ *has an accepting run.*

The path $l_0 \xrightarrow{g_1} \dots \xrightarrow{g_n} l_n$ represents a set of concrete program executions $(l_0, v_0) \to \dots (l_n, v_n)$ in which variable x has value $v_i(x)$. The state conditions φ_{i+1} restrict the set of concrete program executions to those for which $v_{i+1} \models \varphi_{i+1}$, for all $i < n$. Thus, a predicate φ_{i+1} constrains the concrete values in $\mathcal{C}$.

When a verifier checks a property, its output should not only be *yes* or *no*, but preferably also a program execution that leads to the property violation. It is not always easy to construct a precise program execution path, as various verification techniques apply abstractions. This is taken into consideration in the witness format, for they represent a part of the state space that contains a property violation. The "narrower" the space they represent is, the easier it is to re-verify that a property is truly violated. A trivial witness automaton, e.g., which consists of only an (accepting) state with a self-loop, does not restrict the program's execution at all. Witness validation essentially then requires a verification from scratch. On the other hand, a precise witness permits only program executions leading to an error state, thereby making the validation as direct as possible.

Definition 4 (Exact Witness). *Let* $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_E)$ *be a WA for a CFG* $\mathcal{C} = (L, l_0, G, V)$ *and* $L_E \subseteq L$ *be a set of error locations. A WA* $\mathcal{A}$ *is* exact *iff for all* $(D_1, \varphi_1) \dots (D_n, \varphi_n) \in \mathcal{L}(\mathcal{A})$ *it holds for all path* $l_0 \xrightarrow{g_1} \dots \xrightarrow{g_n} l_n$ *of* $\mathcal{C}$*: if* $(l_i, g_{i+1}, l_{i+1}) \in D_{i+1}$ *and* $v_{i+1} \models \varphi_{i+1}$ *in state* (l_{i+1}, v_{i+1}) *for all* $0 \leq i < n$*, then* $l_n \in L_E$*.*

3 Validators for Violation Witnesses

Apart from a new format for exchanging verification results, [11] also presents a feasibility study with implementing both a witness producer and a validator in two well-established tools – CPAchecker and Ultimate Automizer.

Subsequently, [12] reports on two more validators that extract test harnesses from violation witnesses to perform validation. A test harness is compiled with the program to supply input values during runtime and provide definitions for necessary external functions. This approach differs from tools using formal verification/model-checking techniques by offloading semantics to a compiler and only investigating a single path through the program. Validators that explore a single path through compilation/execution are called *execution-based* validators. In addition, a new validator MetaVal[1] was introduced in SV-COMP 2020 – we refrain from describing it as it is yet to be published though we do include it in the benchmark evaluation. All five validators participated in SV-COMP.

CPAchecker This tool employs a so-called Configurable Program Analysis (CPA), which allows selecting the desired level of precision to control the trade-off between performance gain and spurious counterexamples [13]. When witness validation is enabled, it matches a witness automaton against the program's CFG. Afterwards, as part of the CPA, it strengthens the exploration with state-space guards from the witness at matched locations. [11] reports that e.g. their value analysis and predicate analysis are capable of using this strengthening [15,14].

Ultimate Automizer This tool uses an automata-based approach to verification [19,18,17]. Prior to the analysis, it transforms programs into a variant of CFGs over an alphabet of program statements. Such a CFG, say $\mathcal{C}_{error}$, recognizes control-flow traces – sequences of statements – that lead to a property violation. A control-flow trace is *feasible* if it is a run of $\mathcal{C}_{error}$ and ends in an accepting error state. For validation, the tool creates a new CFG $\mathcal{C}_w$ from the Cartesian product of the $\mathcal{C}_{error}$ and a witness automaton. Subsequently, the tool runs the same analysis over the CFG $\mathcal{C}_w$ as for a usual verification run and validates the witness if an error trace is found. State-space guards, such as φ_{i+1} in Definition 3 over control edges and source code guards that characterize branching are ignored.

CPA-witness2test This tool exploits the verifier of CPAchecker. It constructs and matches a CFG with the witness, but does not perform a CPA analysis. It collects the input and initialization values from matched assumptions and assembles an ordered vector of values for every used nondeterministic function, which it then transforms into a `switch` statement supplied as function implementation. For uninitialized variables, which in C are also nondeterministic, no values are injected.

In automatic software verification, programs are usually decorated with an external function `__VERIFIER_error` to identify a point which should never be reached, i.e., an error location. CPA-witness2test implements the function as a call to `exit(107)`, which immediately terminates an execution with return

[1] https://gitlab.com/sosy-lab/software/metaval

code 107. This signals the successful validation of a witness, because the error was reached.

FShell-witness2test This tool does not rely on an existing software model checker. It begins with reading the specification and parses the program with `pycparser`[2] – a Python library for C, which constructs an abstract syntax tree (AST). This AST is traversed to find uninitialized variables and uses of non-deterministic functions. This yields *watch points*, indicating where variable(s) need to be resolved in order to find the right concrete path. Once watch points are established, the tool reads the provided witness and obtains a sequence of control states from program start to the error state. Further on, states of the sequence are matched to the found watch points. For any such match, the tool tries to determine the watch point value from a corresponding assumption in the witness. Finally, these values are added to a test vector, which is transformed into a test harness prepared for compilation. If the function `__VERIFIER_error` is called during execution, then the witness is accepted.

4 Interpretation-based Witness Validation

This section presents a new interpretation-based validator for violation witnesses of C programs with an embedded[3] reachability safety property. The validator is named Nitwit Validator (or nitwit for short) as a shorthand for iNterpretation-based vIolaTion WITness Validator. The programs *must* designate the error location by a function call to `__VERIFIER_error` in order for nitwit to recognize that a program violates the invariant "begin in `main` and never call `__VERIFIER_error`". nitwit is restricted to these programs.

A bird's eye view on nitwit*.* Our implementation approach consists of combining an existing C interpreter with a witness automaton that provides witness assumptions used for resolving variables according to the current position (l_i, v_i) in the program execution. The WA is fed with information from the interpreter, which executes the C program step by step. For validations both source code and state-space guards are taken into account. When a state-space guard (an assumption) does not hold for the current variable values, then the WA does not proceed. To illustrate, suppose an integer variable x initiated to one and incremented on every line (numbered from one). A witness control edge consisting of an assumption $x = 7$ matches only on line seven and will block the WA until then if no other edge is satisfied. If, however, the assumption concerns nondeterministic variables, then we extract a value from it and resolve the nondeterminism in the interpreter. E.g., if x is not initialized at all, then assumption $x = 7$ assigns it the value 7 already on line one.

[2] https://github.com/eliben/pycparser

[3] The program is enhanced with error location(s) `__VERIFIER_error`, assume statements `__VERIFIER_assume` with conditions and calls to `__VERIFIER_nondet` functions, which return nondeterministic values.

As the program executes, the WA progresses through its states until either the execution ends or the error function is called. The latter we consider a testament to the property violation, accepting the witness.

Implementing NITWIT. An *interpreter* is a program that takes as input a program, parses it and executes commands as part of its own runtime instead of producing machine code like a compiler. Interpreters translate programs directly into the behavior they represent; they keep track of all variable values and execute statements based on results of expressions and control flow [21,1,20].

NITWIT's input is a C program. The choice of C interpreters is limited — moreover, compiled C often widely outperforms interpreters in terms of speed, due to extensive compiler optimizations and the unavoidable overhead in parsing and program state management. Nonetheless, in a witness validation setting, when a program only needs to be executed once, the advantage of machine code speed can fade away, because compilation-based validators spend effort on optimizations and translation, which is part of validation time. Furthermore, we wanted to control the simulated program during runtime to alter variables and track the position in source code, which is difficult after compilation.

Our requirements on an interpreter in the order of relevance were: (i) an open-source license permitting free use and distribution of the source code, (ii) a moderate learning curve because of the limited time for implementation, (iii) flexibility so that we can easily modify it, (iv) good coverage of C and (v) tested with realistic C programs. We have chosen *PicoC*[4], a portable interpreter written in C with a very small code base originally built as a scripting language interpreter for unmanned aerial vehicles (UAVs). In its original form, PicoC supports the basics of ANSI C, but misses some important features like function pointers or an implementation of `const` variables. For being able to execute C99-compliant C code, which is common in the benchmarks of SV-COMP, we extended it with new functionalities, such as `goto` constructs, function pointers, the `double`, `long long` and `const` types, better parsing for numerical constants, variable shadowing, `struct` initialization and bit fields.

By using an interpreter, NITWIT has full control over the simulation of a program. For our purposes, we have supplemented PicoC with function callbacks at locations corresponding to places from which a verifier might extract control-flow edges. During execution, the interpreter returns control to our witness automaton whenever it reaches a callback. The callbacks carry all of the necessary information like the current position, variable values, presence of non-determinism or the selected branch in if-statements, loops and ternary operators.

The validator's managing component stores the witness automaton and starts the program's simulation in PicoC. It also stores the current control state in the witness and tries to progress to the error state whenever it receives a callback and the source code and state-space guards match. If a state-space guard involves a nondeterministic variable, NITWIT attempts to extract a value from the given assumption. Upon success, the value is stored in the variable management system.

[4] https://gitlab.com/zsaleeba/picoc

For the assumption evaluation we execute assumptions (recall a WA-transition may have multiple of them) as conditions in the program context and if any one of them fails, then the control edge is considered as non-matching. If an assumption resolves a nondeterministic variable (e.g. the assumption `x` = `2` resolves the nondeterministic variable x), then we automatically accept it and store the given variable value. A variable becomes nondeterministic if it has no initialization or if it is assigned a nondeterministic value (for example from a `__VERIFIER_nondet` function). Analogously, it becomes deterministic when a deterministic value is assigned to it, e.g., as a result from an expression involving only deterministic variables and constants. Moreover, if in the assumption evaluator an assumption involving a nondeterministic variable occurs and is resolved, then the variable gets assigned the new value and is registered as being deterministic.

5 Evaluation

5.1 Benchmarks

Primarily, we have tested NITWIT on witnesses produced during SV-COMP 2019 [6], however, as data from the current edition were already available to us, we present the results attained during SV-COMP 2020. The set of all witnesses produced is available at [9]. It consists of the witnesses and index files that contain information about the witness producer, date of creation, corresponding program file and its hash value (that can be used to find the program in the SV-COMP program repository), the programming language, specification, type of witness and so on. The witnesses and programs cover a large spectrum of possible language features in a variety of applications and settings. We used the dataset of the previous edition [7] to evaluate NITWIT extensively and prepare it for competing in 2020.

During the competition NITWIT was executed only on witnesses in the category *ReachSafety* with a known specification violation as our validator targets only reachability safety violations. This amounts to a set of 11 533 violation witnesses produced by 17 different verifiers.

The witnesses were not manually reviewed to check for each if the language of the WA indeed contains a violating path. This would be a laborious task — doing it automatically is a better fit, which in fact is precisely what validators are designed for. Nevertheless, this means that we cannot claim that our or other validators are incorrect when they do not find a violation, because the witness may steer them inappropriately. As the dataset does not exclusively contain exact witnesses, some witnesses might not resolve enough nondeterminism for NITWIT to find a violation based on the selected single execution.

Witnesses show a lot of heterogeneity based on their producer. Whilst some are very detailed, like in the case of PINAKA and MAP2CHECK with approximately 23 and 13 thousand nodes on average respectively, others tend to keep the WA more succinct or even minimal. For example, tools like BRICK or DIVINE usually provide the least verbose witnesses. The average number of edges typically lies near the average number of nodes due to the fact that witness producers output automata that lead directly to the error location. Not many specify information about function *enter* and *return*. Except for VeriFuzz, Map2Check and Symbiotic, tools usually put assumptions on edges selectively, though there are also some that do not use them – DIVINE and PredatorHP. Assumptions are an important part of witness automata, they restrict the exploration of state space and potentially save the most work during validation. Nevertheless, having to check a large number of them may prove difficult. On the whole, an average witness has around 2 000 nodes, 2 200 transitions between them, 1 300 state-space guards in form of assumptions, 360 controls for branching conditions, 15 function calls and return guards. The largest witness was produced by Pinaka and contains 2.1 million nodes and transitions with assumptions on almost half of them.

5.2 Evaluation Setting

The runtime was limited to 90 s, while memory was limited to 7 GB [6]. Based on recorded data and extracted results, we distinguish six different outcomes of a validator:

False Validator found that an error location is reachable in the program. This is the desired result, nevertheless, be aware that not all witnesses in the available dataset necessarily describe valid violation paths.
Unknown The validator could not find a definite answer.
True The validator claims the program does not reach an error location in the state-space restricted by the witness.
Timeout The validator exceeded the granted CPU time before reaching an answer.
Error An error occurred in the validator during computation (*not* in the program under inspection). Includes errors due to malformed witnesses.
Out of memory The validator exceeded the allowed amount of memory.

5.3 Experimental Results

Figure 2 presents the results on validating 11 533 witnesses by the five violation witness validators. Note that sometimes validator names in tables or plots are abbreviated for readability. The colors discern the possible outcomes described above. The validators are sorted in ascending order by the number of `False` results (blue). nitwit and CPAchecker manage to find the most violations (8 526 and 7 642 respectively), closely followed by FShell-witness2test (7 005). CPA-witness2test is able to validate 6 104, Ultimate Automizer finds 4 393 and MetaVal 1 681.

All validators except for nitwit output `True` (green) in some cases, which means the validator rejected the witness. Ultimate Automizer rejects the majority of witnesses during validation. CPA-witness2test shows the highest ratio of `Unknown` results, whereas FShell-witness2test exhibits the largest amount of unaccepted witnesses due to malformation (`Bad witness`). MetaVal exceeds the alloted time in most cases. The results are detailed in Table 2 on page 13.

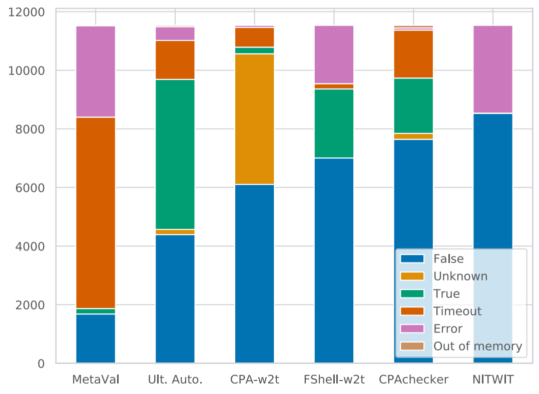


Fig. 2: Validator outcomes on 11 533 witnesses from SV-COMP 2020.

Producing output false. With the result `False`, validators indicate they have found a property violation, i.e., a *reachable error location*. These results are of particular interest, as the dataset used for our evaluation contains witnesses only for programs deemed incorrect.

For 10 933 witnesses at least one validator validated the verification result. Figure 3 presents a Venn diagram that displays the partitioning of these witnesses between validators based on shared successful validations. The shape as a whole stands for all of the validated witnesses and each validator is represented by a distinctly colored enclosure. Circles group intersecting results and the bigger numbers inside describe their cardinality. The smaller numbers underneath are for making clear which validators belong to the group (ordering is from top to bottom, so CPAchecker is number one and so on). The diagram reveals that only about 226 witnesses are approved by all verifiers, though the largest shared subset has 2 010 of them – it corresponds to results shared by all of the validators with exception of Ultimate Automizer and MetaVal. In total, 1 411 instances are validated only once, 1 878 twice, 2 290 thrice, 3 682 four times and 1 446 five times. NITWIT validates 399 witnesses that no other tool validates. Interestingly, none of the validators subsume each other in terms of `False` results, each has some not negligible amount of witnesses validated uniquely.

Concerning resource usage, Figure 4(a) depicts the reached number of successfully validated witnesses plotted against the required CPU time (in log-scale). Data points are sorted by the required CPU time and the black line at the top marks the timeout. NITWIT finds violations systematically faster than any other tool. Its mean runtime amounts to 0.63 seconds, the median was noticeably smaller at 0.02 seconds, standard deviation was 4.74. The runtime for NITWIT is skewed towards zero with most results achieved under half a second. We also see that running NITWIT more than 10 seconds scarcely produces any new results. That is not the case for CPAchecker, Ultimate Automizer, MetaVal and CPA-witness2test, which frequently need more than that, even though they rarely finish without a considerable headroom until the limit of 90 seconds. On average, NITWIT is about 4.2 times faster than the runner up FShell-witness2test, 17.8 times than CPA-witness2test, 22.0 times than CPAchecker, 35.3 times than Ultimate Automizer and 39.1 times than MetaVal.

Validator	**Result**	**Witnesses**	**Mean**	**Median**	**Std.dev.**	**Total**
NITWIT	False	8526	0.63	0.02	4.74	5393
	All	11533	0.64	0.02	4.99	7386
FShell-witness2test	False	7005	2.67	1.30	5.90	18734
	All	11533	3.93	1.40	12.69	45337
CPA-witness2test	False	6104	11.21	8.40	8.90	68407
	All	11533	13.62	8.60	15.59	157037
CPAchecker	False	7642	13.87	11.00	11.29	105968
	All	11533	26.02	12.00	30.88	300145
Ultimate Automizer	False	4393	22.23	16.00	16.03	97640
	All	11533	28.57	16.00	28.01	329488
MetaVal	False	1681	24.66	17.00	17.81	41453
	All	11533	63.12	96.00	39.82	728008

Table 1: Runtime statistics for validators (in seconds).

Figure 4(b) shows the memory usage in successful validations plotted on a log-scale with data sorted again in ascending order. NITWIT needed the least (5 MB on average; maximum 1 GB) RAM, closely followed by FShell-witness2test. The validators were only rarely approaching the limit of 7 GB (black line at the top); the largest value slightly above 4 GB during a successful validation was exhibited by CPA-witness2test. The tools do not suffer from a lack of available memory, which is also demonstrated by the low rate of `Out of memory` results in Figure 2.

All validations. Figures 4(c) and 4(d) demonstrate the resource consumption of all validations. Until about the 10 500th witness, NITWIT remains consistently faster than all other validators, usually finishing under one second. Then, it struggles to find the answers as some witnesses do not resolve enough non-determinism or contain very long or even infinite paths.

Compilation-based FShell- and CPA-witness2test avoid the overhead of an interpreter, so are mostly able to finish before the 90 second mark, because even if the harness they extract is incomplete (still contains nondeterminism), then after compilation the execution ends quicker than if it were interpreted. In terms of absolute numbers, NITWIT takes an average 0.64 seconds per witness on the whole dataset with a median of 0.02 and standard deviation 4.99. The runtime difference on average is 3.3 seconds in favor of Nitwit compared to FShell-witness2test and 13.0 seconds to CPA-witness2test. More interesting is the median though, this was 0.02 seconds, 1.4 seconds, 8.6 seconds,

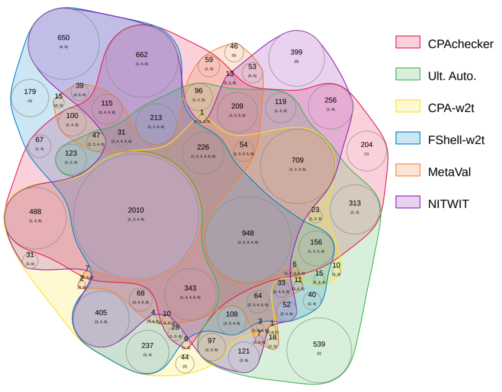
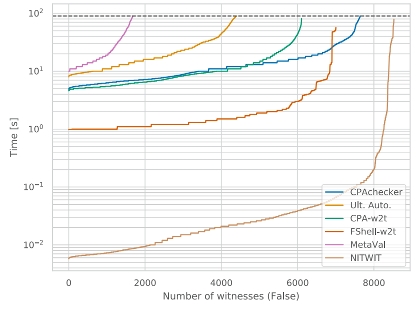
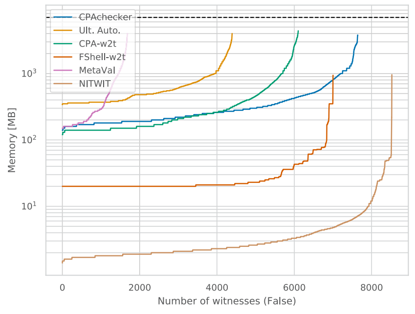


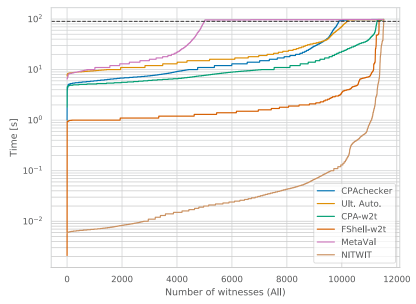
Fig. 3: A Venn diagram showing the coverage of `False` validation results by various validators.



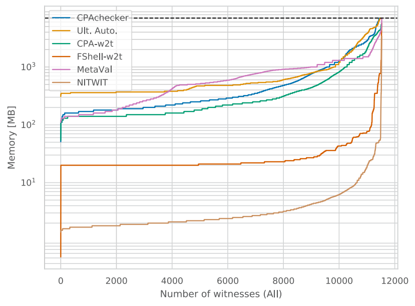
(a) CPU time (seconds) in validations for result `False`.



(b) Memory consumption (in MB) during validations for result `False`.



(c) CPU time (seconds) in validations for all results.



(d) Memory consumption (in MB) in validations for all results.

Fig. 4: Resources for results `False` (first row) and all together (second row).

12.0 seconds, 16.0 seconds, 96.0 seconds for Nitwit, FShell-witness2test, CPA-witness2test, CPAchecker, Ultimate Automizer and MetaVal respectively.

Figure 5 shows how nitwit compares to the other four validators in terms of time and successful validation results. In each plot, a validator is compared against nitwit. Witnesses validated by both have a blue color, validated only by nitwit yellow, by the other tool green and any other are depicted in red. The diagonal line is supplemented by two other lines representing a ±30% difference in CPU time. The result, if not `false`, is plotted on one of six lines at the end of its axis. These lines correspond to a `Timeout` (abbreviated by *to*), `Unknown` (*uk*), `True` (*tu*), `Error` (*er*) and `Out of memory` (*om*). Every point represents a witness (identical for both validators).

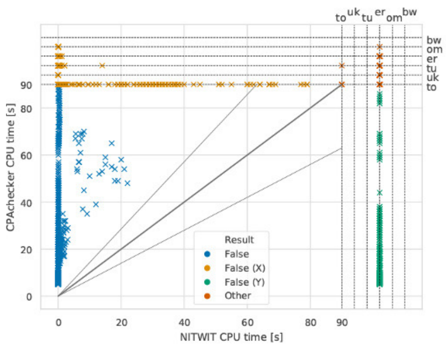
Figure 5 shows that in instances of agreed `False` results, nitwit is always faster than other validators. FShell-witness2test has 1 114 validations within less than one second difference. This is 0 for all of the others.

Verifier	**CPAchecker**	**Ult. Auto.**	**CPA-w2t**	**FS-w2t**	**MetaVal**	**NITWIT**	**Virt. best**	**Total**
2LS	114	164	166	358	91	332	477	563
BRICK	20	11	38	36	17	38	42	43
CBMC	171	423	456	358	114	488	768	905
CPAchecker	1091	713	927	491	0	1070	1171	1189
DIVINE	400	46	110	280	181	237	448	460
ESBMC	572	131	605	843	249	730	955	1022
GACAL	0	10	0	10	7	15	15	15
Map2Check	80	32	106	129	120	137	211	264
PeSCo	1030	625	845	606	0	988	1064	1081
Pinaka	531	518	541	454	54	440	616	629
PredatorHP	55	35	18	44	53	20	69	70
Symbiotic	1033	12	866	894	134	1047	1103	1106
UAutomizer	391	574	55	189	159	233	630	662
UKojak	291	310	38	151	135	179	348	348
UTaipan	370	379	53	189	150	205	427	452
VeriAbs	501	366	326	892	10	1139	1298	1427
VeriFuzz	992	44	954	1081	207	1228	1291	1297
Total	7642	4393	6104	7005	1681	8526	10933	11533

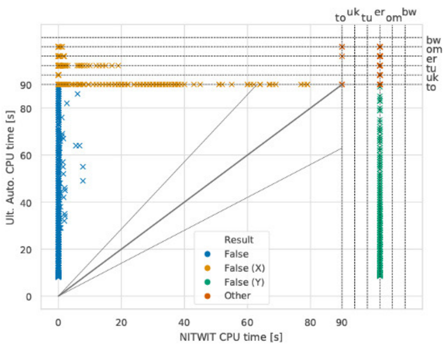
Table 2: Results on successful validations of violation witnesses generated by the various verifiers. Column *Virtual best* aggregates witnesses that are validated at least once.

5.4 Discussion

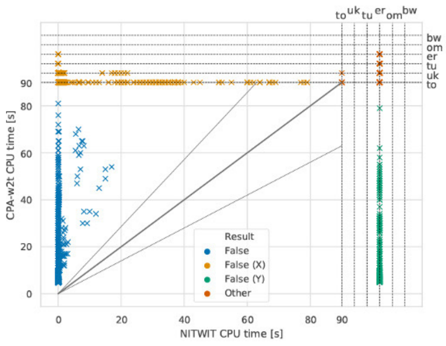
Nondeterminism in programs. nitwit is not designed for proving a program correct with respect to some specification, because the validator explores only a single path. Nevertheless, to prove a program incorrect it may suffice to look at a single path and although the program may contain nondeterministic choices (e.g., if a condition depends on a nondeterministic variable) – if these are resolved using a witness, then the execution becomes deterministic. This is the main idea behind execution- and interpretation-based validators, because after resolving



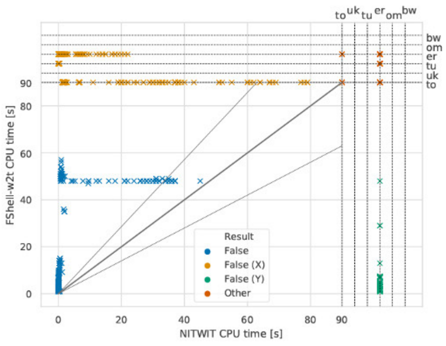
(a) CPAchecker



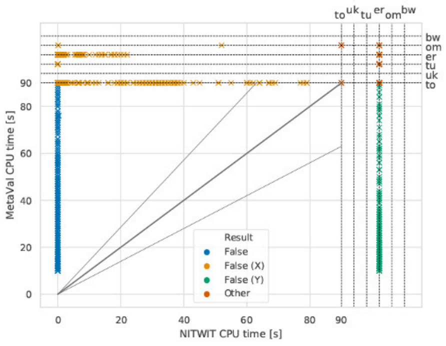
(b) Ultimate Automizer



(c) CPA-witness2test



(d) FShell-witness2test



(e) MetaVal

Fig. 5: Comparing NITWIT (x-axis) with the other five validators (y-axis) in terms of speed and outcome. Each point represents a witness.

nondeterminism, there exists only a single path through the program. If this leads to an error location, then the validator may confidently claim that the provided program and witness constitute a specification violation.

NITWIT guarantees (except for implementation bugs, supported syntax and available stack- and heap size) a validated violation witness iff it allows only such abstract paths that end in an error location. Thus, given a well-specified exact witness, NITWIT should always find a violation, because it has the program state space restricted to only such paths which reach an error location. If a witness allows inexact abstract paths, then NITWIT (and in fact also an execution-based validator) may select the wrong path and see no error state. Results in Section 5 demonstrate that even without the guarantee of exact witnesses, interpretation-based validators can find a substantial amount of violations.

Finding violations. Results clearly show that NITWIT is a competitive validator of witnesses for C programs and invariant properties. Our validators implemented independently of any verification platform can efficiently reestablish violations from witnesses. We outperform other tools especially on the less time intensive instances as NITWIT works well in validating witnesses that restrict the state space sufficiently. For these witnesses, it is the fastest among state-of-the-art validators and has the smallest memory footprint.

We attribute the good outcomes in speed and memory to the choice of employing *an interpretation-based approach.* As NITWIT explores only one path, it is obviously faster than full fledged model-checking validators that explore many paths. Interestingly, an interpreter-based execution analysis is often much faster than compiled. This difference might be attributed to the fact that execution-based tools build the whole AST and CFG, whereas PicoC saves a lot of time by not having to construct them. Moreover, a compiler translates the program into machine code, a non-trivial task which PicoC circumvents.

Weaknesses. One of NITWIT's limitations is inherent to exploring only a single execution. Suppose a non-terminating program P, a trivial witness without assumptions and a property violation, whose reachability depends on a nondeterministic variable being zero. NITWIT, if it cannot resolve a nondeterministic variable, *assumes it has value one.* In such a setting, the simulated program diverges and so does NITWIT, because it cannot recognize an infinite execution. A similar situation may occur even if the witness is non-trivial. If its transitions are not matched to the right operations (which can be a fault in both the witness producer or validator), then P will diverge due to unresolved nondeterminism.

Secondly, as we employ an interpreter, there is a noticeable overhead compared to compiled programs in terms of CPU instructions per operation. Therefore, even if an execution is finite or reaches a violation in finitely many steps, it might simply be too computationally intensive for NITWIT to provide an answer within time. Combined with unresolved nondeterminism, this explained a relatively high amount of `Timeout` results in an early version of NITWIT benchmarked on SV-COMP 2019.

To combat the timeouts, we decided to implement a simple check in the witness automaton. After a certain number of unsuccessful transitions to a different state, we deliberately stop the validation and output `Unknown`. We experimented with the threshold and concluded that 1 million attempts is appropriate. By enabling this threshold, we went from 784 to 123 killed validations and lost only 25 witnesses that would otherwise have been validated, which is an acceptable trade-off. An analysis showed that 573 of the 784 timeouts were validations of possibly non-terminating programs, 18 for terminating and the 193 remaining validations without specified termination[5]. The check for the threshold can be disabled.

Processing witnesses. In some cases, software verifiers do not always produce witnesses in exactly the correct format. For example, in GraphML it is necessary to define attributes for the graph, nodes and edges. If a witness happens to contain no such definitions, we supply a basic configuration that allows for its successful parsing. By default, we also do not extensively check for correctness of all of the graph attributes like the program hash.

Furthermore, we consider a reached error location as a proof of violation even if the witness automaton itself does not finish in an error state. This behavior can be changed by a compilation flag to rejection. Nevertheless, if a witness resolves enough determinism for one execution to find an error, we think it is sufficiently "good" for it to be a viable witness. For some programs, the variable resolving at the start suffices to reach a violation. However, we output a special exit code to make it clear that the witness did not in fact accept this path.

6 Conclusion

We presented the new interpretation-based violation witness validator NITWIT, that was able to validate 8 526 witnesses from a dataset of 11 533 witnesses [9] that were produced in the ReachSafety category of the 2020 edition of SV-COMP. NITWIT was able to validate 399 witnesses that have not been validated by any other participating tool. In addition, NITWIT has a small memory footprint and is mostly significantly faster than its competitors.

Data Availability Statement and Acknowledgments. NITWIT is available for free at https://github.com/moves-rwth/nitwit-validator and is licensed under the New BSD license. The replication artifact can be found at the Zenodo repository https://doi.org/10.5281/zenodo.3518139 [23] and the datasets analyzed during the current study at https://doi.org/10.5281/zenodo.3630205 [8]. We thank Dirk Beyer for very useful feedback on an earlier version of the paper and assistance with configuring NITWIT for SV-COMP 2020.

[5] We know whether these programs are (non-)terminating, as they were reviewed in SV-COMP before including them in the competition on termination analysis.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley series in computer science / World student series edition, Addison-Wesley (1986)
2. Beyer, D.: Competition on software verification - (SV-COMP). In: TACAS. Lecture Notes in Computer Science, vol. 7214, pp. 504–524. Springer (2012)
3. Beyer, D.: Software verification and verifiable witnesses - (report on SV-COMP 2015). In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 401–416. Springer (2015)
4. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In: TACAS. Lecture Notes in Computer Science, vol. 9636, pp. 887–904. Springer (2016)
5. Beyer, D.: Software verification with validation of results - (report on SV-COMP 2017). In: TACAS (2). Lecture Notes in Computer Science, vol. 10206, pp. 331–349 (2017)
6. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: TACAS (3). Lecture Notes in Computer Science, vol. 11429, pp. 133–155. Springer (2019)
7. Beyer, D.: Verification Witnesses from SV-COMP 2019 Verification Tools (Feb 2019). https://doi.org/10.5281/zenodo.2559175
8. Beyer, D.: Results of the 9th International Competition on Software Verification (SV-COMP 2020) (Jan 2020). https://doi.org/10.5281/zenodo.3630205
9. Beyer, D.: Verification Witnesses from SV-COMP 2020 Verification Tools (Jan 2020). https://doi.org/10.5281/zenodo.3630188
10. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: CAV (2). Lecture Notes in Computer Science, vol. 9780, pp. 502–509. Springer (2016)
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: ESEC/SIGSOFT FSE. pp. 721–733. ACM (2015)
12. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses - execution-based validation of verification results. In: TAP. Lecture Notes in Computer Science, vol. 10889, pp. 3–23. Springer (2018)
13. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: CAV. Lecture Notes in Computer Science, vol. 4590, pp. 504–518. Springer (2007)
14. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: FMCAD. pp. 189–197. IEEE (2010)
15. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: FASE. Lecture Notes in Computer Science, vol. 7793, pp. 146–162. Springer (2013)
16. Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking. Lecture Notes in Computer Science, vol. 5000, pp. 1–26. Springer (2008)
17. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate automizer with SMTInterpol - (competition contribution). In: TACAS. Lecture Notes in Computer Science, vol. 7795, pp. 641–643. Springer (2013)
18. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: Ultimate automizer with array interpolation - (competition contribution). In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 455–457. Springer (2015)

19. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 36–52. Springer (2013)
20. Holub, A.I.: Compiler Design in C. Prentice Hall (1990)
21. Mak, R.: Writing Compilers and Interpreters: A Software Engineering Approach. Wiley Publishing, 3rd edn. (2009)
22. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
23. Švejda, J., Berger, P., Katoen, J.P.: Replication artifact for the NITWIT Validator submitted to TACAS20 (Oct 2019). https://doi.org/10.5281/zenodo.3518139

A Calculus for Modular Loop Acceleration*

Florian Frohn

Max Planck Institute for Informatics
Saarland Informatics Campus, Saarbrücken, Germany

Abstract. Loop acceleration can be used to prove safety, reachability, runtime bounds, and (non-)termination of programs operating on integers. To this end, a variety of acceleration techniques has been proposed. However, all of them are monolithic: Either they accelerate a loop successfully or they fail completely. In contrast, we present a calculus that allows for combining acceleration techniques in a modular way and we show how to integrate many existing acceleration techniques into our calculus. Moreover, we propose two novel acceleration techniques that can be incorporated into our calculus seamlessly. An empirical evaluation demonstrates the applicability of our approach.

1 Introduction

In the last years, loop acceleration techniques have successfully been used to build static analyses for programs operating on integers [2, 8, 11, 16–18, 28]. Essentially, such techniques extract a quantifier-free first-order formula ψ from a single-path loop $\mathcal{T}$, i.e., a loop without branching in its body, such that ψ under-approximates (resp. is equivalent to) $\mathcal{T}$. More specifically, each model of the resulting formula ψ corresponds to an execution of $\mathcal{T}$ (and vice versa). By integrating such techniques into a suitable program-analysis framework [3, 11, 16–18, 23], whole programs can be transformed into first-order formulas which can then be analyzed by off-the-shelf solvers. Applications include proving safety [23] or reachability [23, 28], deducing bounds on the runtime complexity [16, 17], and proving (non-) termination [8, 11].

However, existing acceleration techniques only apply if certain prerequisites are in place. So the power of static analyses built upon loop acceleration depends on the applicability of the underlying acceleration technique.

In this paper, we introduce a calculus which allows for combining several acceleration techniques modularly in order to accelerate a single loop. Consequently, it can handle classes of loops where all standalone techniques fail. Moreover, we present two novel acceleration techniques and integrate them into our calculus.

In the following, we introduce preliminaries in Sec. 2. Then, we discuss existing acceleration techniques in Sec. 3. In Sec. 4, we present our calculus to combine acceleration techniques. Sec. 5 shows how existing acceleration techniques can be

* This work has been funded by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

integrated into our framework. Next, we present two novel acceleration techniques and incorporate them into our calculus in Sec. 6. After discussing related work in Sec. 7, we demonstrate the applicability of our approach via an empirical evaluation in Sec. 8 and conclude in Sec. 9. All proofs can be found in [13].

2 Preliminaries

We use bold letters $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$, ... for vectors. Let $\mathscr{C}(\boldsymbol{z})$ be the set of *closed-form expressions* over the variables $\boldsymbol{z}$ containing, e.g., all arithmetic expressions built from $\boldsymbol{z}$, integer constants, addition, subtraction, multiplication, division, and exponentiation.[1] We consider loops of the form

$$\textbf{while } \varphi \textbf{ do } \boldsymbol{x} \leftarrow \boldsymbol{a} \tag{$\mathcal{T}_{loop}$}$$

where $\boldsymbol{x}$ is a vector of d pairwise different variables that range over the integers, the loop condition $\varphi \in \mathit{Prop}(\mathscr{C}(\boldsymbol{x}))$ is a finite propositional formula over the atoms $\{p > 0 \mid p \in \mathscr{C}(\boldsymbol{x})\}$, and $\boldsymbol{a} \in \mathscr{C}(\boldsymbol{x})^d$ such that the function[2] $\boldsymbol{x} \mapsto \boldsymbol{a}$ maps integers to integers. *Loop* denotes the set of all such loops.

We identify $\mathcal{T}_{loop}$ and the pair $\langle \varphi, \boldsymbol{a} \rangle$. Moreover, we identify $\boldsymbol{a}$ and the function $\boldsymbol{x} \mapsto \boldsymbol{a}$ where we sometimes write $\boldsymbol{a}(\boldsymbol{x})$ to make the variables $\boldsymbol{x}$ explicit and we use the same convention for other (vectors of) expressions. Similarly, we identify the formula φ resp. $\varphi(\boldsymbol{x})$ and the predicate $\boldsymbol{x} \mapsto \varphi$.

Throughout this paper, let n be a designated variable and let:

$$\boldsymbol{a} := \begin{pmatrix} a_1 \\ \cdots \\ a_d \end{pmatrix} \qquad \boldsymbol{x} := \begin{pmatrix} x_1 \\ \cdots \\ x_d \end{pmatrix} \qquad \boldsymbol{x}' := \begin{pmatrix} x_1' \\ \cdots \\ x_d' \end{pmatrix} \qquad \boldsymbol{y} := \begin{pmatrix} \boldsymbol{x} \\ n \\ \boldsymbol{x}' \end{pmatrix}$$

Intuitively, the variable n represents the number of loop iterations and $\boldsymbol{x}'$ corresponds to the values of the program variables $\boldsymbol{x}$ after n iterations.

$\mathcal{T}_{loop}$ induces a relation $\longrightarrow_{\mathcal{T}_{loop}}$ on $\mathbb{Z}^d$:

$$\varphi(\boldsymbol{x}) \land \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}) \iff \boldsymbol{x} \longrightarrow_{\mathcal{T}_{loop}} \boldsymbol{x}'$$

Our goal is to find a formula $\psi \in \mathit{Prop}(\mathscr{C}(\boldsymbol{y}))$ such that

$$\psi \iff \boldsymbol{x} \longrightarrow^n_{\mathcal{T}_{loop}} \boldsymbol{x}' \qquad \text{for all } n > 0. \tag{equiv}$$

To see why we use $\mathscr{C}(\boldsymbol{y})$ instead of, e.g., polynomials, consider the loop

$$\textbf{while } x_1 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1 - 1 \\ 2 \cdot x_2 \end{smallmatrix}\right). \tag{$\mathcal{T}_{exp}$}$$

Here, an acceleration technique synthesizes, e.g., the formula

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \left(\begin{smallmatrix} x_1 - n \\ 2^n \cdot x_2 \end{smallmatrix}\right) \land x_1 - n + 1 > 0 \tag{ψ_{exp}}$$

[1] Note that there is no widely accepted definition of "closed forms" and the results of the current paper are independent of the precise definition of $\mathscr{C}(\boldsymbol{z})$.

[2] i.e., the (anonymous) function that maps $\boldsymbol{x}$ to $\boldsymbol{a}$

where $\left(\begin{smallmatrix} x_1 - n \\ 2^n \cdot x_2 \end{smallmatrix}\right)$ is equivalent to the value of $\left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right)$ after n iterations and the inequation $x_1 - n + 1 > 0$ ensures that $\mathcal{T}_{exp}$ can be executed at least n times. Clearly, the growth of x_2 cannot be captured by a polynomial, i.e., even the behavior of quite simple loops is beyond the expressiveness of polynomial arithmetic.

In practice, one can restrict our approach to weaker classes of expressions to ease automation, but the presented results are independent of such considerations.

Some acceleration techniques cannot guarantee (equiv), but the resulting formula is an under-approximation of $\mathcal{T}_{loop}$, i.e., we have

$$\psi \implies \boldsymbol{x} \longrightarrow^n_{\mathcal{T}_{loop}} \boldsymbol{x}' \qquad \text{for all } n > 0. \tag{approx}$$

If (equiv) resp. (approx) holds, then ψ is *equivalent* to resp. *approximates* $\mathcal{T}_{loop}$.

Definition 1 (Acceleration Technique). *An* acceleration technique *is a partial function*

$$accel : Loop \rightharpoonup Prop(\mathscr{C}(\boldsymbol{y})).$$

It is sound *if $accel(\mathcal{T})$ approximates $\mathcal{T}$ for all $\mathcal{T} \in \mathrm{dom}(accel)$. It is* exact *if $accel(\mathcal{T})$ is equivalent to $\mathcal{T}$ for all $\mathcal{T} \in \mathrm{dom}(accel)$.*

3 Existing Acceleration Techniques

We now recall several existing acceleration techniques. In Sec. 4 we will see how these techniques can be combined in a modular way. All of them first compute a *closed form* $\boldsymbol{c} \in \mathscr{C}(\boldsymbol{x}, n)^d$ for the values of the program variables after n iterations.

Definition 2 (Closed Form). *We call $\boldsymbol{c} \in \mathscr{C}(\boldsymbol{x}, n)^d$ a* closed form *of $\mathcal{T}_{loop}$ if*

$$\forall \boldsymbol{x} \in \mathbb{Z}^d, n \in \mathbb{N}.\ \boldsymbol{c} = \boldsymbol{a}^n(\boldsymbol{x}).$$

Here, $\boldsymbol{a}^n$ is the n-fold application of $\boldsymbol{a}$, i.e., $\boldsymbol{a}^0(\boldsymbol{x}) = \boldsymbol{x}$ and $\boldsymbol{a}^{n+1}(\boldsymbol{x}) = \boldsymbol{a}(\boldsymbol{a}^n(\boldsymbol{x}))$. To find closed forms, one tries to solve the system of recurrence equations $\boldsymbol{x}^{(n)} = \boldsymbol{a}(\boldsymbol{x}^{(n-1)})$ with the initial condition $\boldsymbol{x}^{(0)} = \boldsymbol{x}$. In the sequel, we assume that we can represent $\boldsymbol{a}^n(\boldsymbol{x})$ in closed form. Note that one can always do so if $\boldsymbol{a}(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ with $A \in \mathbb{Z}^{d \times d}$ and $\boldsymbol{b} \in \mathbb{Z}^d$, i.e., if $\boldsymbol{a}$ is affine. To this end, one considers the matrix $B := \left(\begin{smallmatrix} A & \boldsymbol{b} \\ \boldsymbol{0}^T & 1 \end{smallmatrix}\right)$ and computes its Jordan normal form $B = T^{-1}JT$ where J is a block diagonal matrix (which has complex entries if B has complex eigenvalues). Then the closed form for J^n can be given directly (see, e.g., [31]) and $\boldsymbol{a}^n(\boldsymbol{x}) = T^{-1}J^nT \left(\begin{smallmatrix} \boldsymbol{x} \\ 1 \end{smallmatrix}\right)$. Moreover, one can compute a closed form if $\boldsymbol{a} = \left(\begin{smallmatrix} c_1 \cdot x_1 + p_1 \\ \cdots \\ c_d \cdot x_d + p_d \end{smallmatrix}\right)$ where $c_i \in \mathbb{N}$ and each p_i is a polynomial over $x_1, \ldots, x_{i-1}$ [15].

3.1 Acceleration via Decrease *or* Increase

The first acceleration technique discussed in this section exploits the following observation: If $\varphi(\boldsymbol{a}(\boldsymbol{x}))$ implies $\varphi(\boldsymbol{x})$ and $\varphi(\boldsymbol{a}^{n-1}(\boldsymbol{x}))$ holds, then $\mathcal{T}_{loop}$ is applicable at least n times. So in other words, it requires that the indicator function (or characteristic function) $I_\varphi : \mathbb{Z}^d \to \{0, 1\}$ of φ with $I_\varphi(\boldsymbol{x}) = 1 \iff \varphi(\boldsymbol{x})$ is monotonically decreasing w.r.t. $\boldsymbol{a}$, i.e., $I_\varphi(\boldsymbol{x}) \geq I_\varphi(\boldsymbol{a}(\boldsymbol{x}))$.

Theorem 1 (Acceleration via Monotonic Decrease [28]). *If*

$$\varphi(\boldsymbol{a}(\boldsymbol{x})) \implies \varphi(\boldsymbol{x}),$$

then the following acceleration technique is exact:

$$\mathcal{T}_{loop} \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land \varphi(\boldsymbol{a}^{n-1}(\boldsymbol{x}))$$

So for example, Thm. 1 accelerates $\mathcal{T}_{exp}$ to ψ_{exp}. However, the requirement $\varphi(\boldsymbol{a}(\boldsymbol{x})) \implies \varphi(\boldsymbol{x})$ is often violated in practice. To see this, consider the loop

$$\textbf{while } x_1 > 0 \land x_2 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1 - 1 \\ x_2 + 1 \end{smallmatrix}\right). \qquad (\mathcal{T}_{non\text{-}dec})$$

It cannot be accelerated with Thm. 1 as

$$x_1 - 1 > 0 \land x_2 + 1 > 0 \centernot\implies x_1 > 0 \land x_2 > 0.$$

A dual acceleration technique is obtained by "reversing" the implication in the prerequisites of Thm. 1. Then I_φ is monotonically increasing w.r.t. $\boldsymbol{a}$. So φ is an invariant and thus $\{\boldsymbol{x} \in \mathbb{Z}^d \mid \varphi(\boldsymbol{x})\}$ is a *recurrent set* [22] of $\mathcal{T}_{loop}$.

Theorem 2 (Acceleration via Monotonic Increase). *If*

$$\varphi(\boldsymbol{x}) \implies \varphi(\boldsymbol{a}(\boldsymbol{x})),$$

then the following acceleration technique is exact:

$$\mathcal{T}_{loop} \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land \varphi(\boldsymbol{x})$$

As a minimal example, Thm. 2 accelerates

$$\textbf{while } x > 0 \textbf{ do } x \leftarrow x + 1$$

to $x' = x + n \land x > 0$.

3.2 Acceleration via Decrease *and* Increase

Both acceleration techniques presented so far have been generalized in [11].

Theorem 3 (Acceleration via Monotonicity [11]). *If*

$$\begin{aligned} \varphi(\boldsymbol{x}) &\iff \varphi_1(\boldsymbol{x}) \land \varphi_2(\boldsymbol{x}) \land \varphi_3(\boldsymbol{x}), \\ \varphi_1(\boldsymbol{x}) &\implies \varphi_1(\boldsymbol{a}(\boldsymbol{x})), \\ \varphi_1(\boldsymbol{x}) \land \varphi_2(\boldsymbol{a}(\boldsymbol{x})) &\implies \varphi_2(\boldsymbol{x}), \qquad \textit{and} \\ \varphi_1(\boldsymbol{x}) \land \varphi_2(\boldsymbol{x}) \land \varphi_3(\boldsymbol{x}) &\implies \varphi_3(\boldsymbol{a}(\boldsymbol{x})), \end{aligned}$$

then the following acceleration technique is exact:

$$\mathcal{T}_{loop} \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land \varphi_1(\boldsymbol{x}) \land \varphi_2(\boldsymbol{a}^{n-1}(\boldsymbol{x})) \land \varphi_3(\boldsymbol{x})$$

Here, φ_1 and φ_3 are again invariants of the loop. Thus, as in Thm. 2 it suffices to require that they hold before entering the loop. On the other hand, φ_2 needs to satisfy a similar condition as in Thm. 1 and thus it suffices to require that φ_2 holds before the last iteration. We also say that φ_2 is a *converse invariant* (w.r.t. φ_1). It is easy to see that Thm. 3 is equivalent to Thm. 1 if $\varphi_1 \equiv \varphi_3 \equiv \top$ (where $\top$ denotes logical truth) and it is equivalent to Thm. 2 if $\varphi_2 \equiv \varphi_3 \equiv \top$.

With this approach, $\mathcal{T}_{\mathit{non\text{-}dec}}$ can be accelerated to

$$\left(\begin{smallmatrix} x_1' \\ x_2' \end{smallmatrix}\right) = \left(\begin{smallmatrix} x_1 - n \\ x_2 + n \end{smallmatrix}\right) \land x_2 > 0 \land x_1 - n + 1 > 0 \tag{$\psi_{\mathit{non\text{-}dec}}$}$$

by choosing $\varphi_1 := x_2 > 0$, $\varphi_2 := x_1 > 0$, and $\varphi_3 := \top$.

Thm. 3 naturally raises the question: Why do we need *two* invariants? To see this, consider a restriction of Thm. 3 where $\varphi_3 := \top$. It would fail for a loop like

$$\textbf{while } x_1 > 0 \land x_2 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1 + x_2 \\ x_2 - 1 \end{smallmatrix}\right) \tag{$\mathcal{T}_{2\text{-}\mathit{invs}}$}$$

which can easily be handled by Thm. 3 (by choosing $\varphi_1 := \top$, $\varphi_2 := x_2 > 0$, and $\varphi_3 := x_1 > 0$). The problem is that the converse invariant $x_2 > 0$ is needed to prove invariance of $x_1 > 0$. Similarly, a restriction of Thm. 3 where $\varphi_1 := \top$ would fail for the following variant of $\mathcal{T}_{2\text{-}\mathit{invs}}$:

$$\textbf{while } x_1 > 0 \land x_2 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1 - x_2 \\ x_2 + 1 \end{smallmatrix}\right)$$

Here, the problem is that the invariant $x_2 > 0$ is needed to prove converse invariance of $x_1 > 0$.

3.3 Acceleration via Metering Functions

Another approach for loop acceleration uses *metering functions*, a variation of classical *ranking functions* from termination and complexity analysis [17]. While ranking functions give rise to *upper* bounds on the runtime of loops, metering functions provide *lower* runtime bounds, i.e., the definition of a metering function $\mathit{mf} : \mathbb{Z}^d \to \mathbb{Q}$ ensures that for each $\boldsymbol{x} \in \mathbb{Z}^d$, the loop under consideration can be applied at least $\lceil \mathit{mf}(\boldsymbol{x}) \rceil$ times.

Theorem 4 (Acceleration via Metering Functions [17]). *Let mf be a metering function for $\mathcal{T}_{\mathit{loop}}$. Then the following acceleration technique is sound:*

$$\mathcal{T}_{\mathit{loop}} \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land \varphi(\boldsymbol{x}) \land n < \mathit{mf}(\boldsymbol{x}) + 1$$

So using the metering function x, Thm. 4 accelerates $\mathcal{T}_{\mathit{exp}}$ to

$$\left(\begin{smallmatrix} x_1' \\ x_2' \end{smallmatrix}\right) = \left(\begin{smallmatrix} x_1 - n \\ 2^n \cdot x_2 \end{smallmatrix}\right) \land x_1 > 0 \land n < x_1 + 1 \quad \equiv \quad \psi_{\mathit{exp}}.$$

However, synthesizing non-trivial (i.e., non-constant) metering functions is challenging. Moreover, unless the number of iterations of $\mathcal{T}_{\mathit{loop}}$ equals $\lceil \mathit{mf}(\boldsymbol{x}) \rceil$ for all $\boldsymbol{x} \in \mathbb{Z}^d$, *acceleration via metering functions* is not exact.

Linear metering functions can be synthesized via Farkas' Lemma and SMT solving [17]. However, many loops do not have non-trivial linear metering functions. To see this, reconsider $\mathcal{T}_{non\text{-}dec}$. Here, $(x_1, x_2) \mapsto x_1$ is not a metering function as $\mathcal{T}_{non\text{-}dec}$ cannot be iterated at least x_1 times if $x_2 \leq 0$. Thus, [16] proposes a refinement of [17] based on metering functions of the form $\boldsymbol{x} \mapsto I_\xi(\boldsymbol{x}) \cdot f(\boldsymbol{x})$ where $\xi \in \mathit{Prop}(\mathscr{C}(\boldsymbol{x}))$ and f is linear. With this improvement, the metering function $(x_1, x_2) \mapsto I_{x_2>0}(x_2) \cdot x_1$ can be used to accelerate $\mathcal{T}_{non\text{-}dec}$ to

$$\left(\begin{smallmatrix} x_1' \\ x_2' \end{smallmatrix}\right) = \left(\begin{smallmatrix} x_1 - n \\ x_2 + n \end{smallmatrix}\right) \land x_1 > 0 \land x_2 > 0 \land n < x_1 + 1.$$

4 A Calculus for Modular Loop Acceleration

All acceleration techniques presented so far are monolithic: Either they accelerate a loop successfully or they fail completely. In other words, we cannot *combine* several techniques to accelerate a single loop. To this end, we now present a calculus that repeatedly applies acceleration techniques to simplify an *acceleration problem* resulting from a loop $\mathcal{T}_{loop}$ until it is *solved* and hence gives rise to a suitable $\psi \in \mathit{Prop}(\mathscr{C}(\boldsymbol{y}))$ which approximates resp. is equivalent to $\mathcal{T}_{loop}$.

Definition 3 (Acceleration Problem). *A tuple*

$$[\![\psi \mid \check{\varphi} \mid \widehat{\varphi} \mid \boldsymbol{a}]\!]$$

where $\psi \in \mathit{Prop}(\mathscr{C}(\boldsymbol{y}))$*,* $\check{\varphi}, \widehat{\varphi} \in \mathit{Prop}(\mathscr{C}(\boldsymbol{x}))$*, and* $\boldsymbol{a} : \mathbb{Z}^d \to \mathbb{Z}^d$ *is an* acceleration problem*. It is* consistent *if* ψ *approximates* $\langle \check{\varphi}, \boldsymbol{a} \rangle$*,* exact *if* ψ *is equivalent to* $\langle \check{\varphi}, \boldsymbol{a} \rangle$*, and* solved *if it is consistent and* $\widehat{\varphi} \equiv \top$*. The* canonical acceleration problem *of a loop* $\mathcal{T}_{loop}$ *is*

$$[\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \mid \top \mid \varphi(\boldsymbol{x}) \mid \boldsymbol{a}(\boldsymbol{x})]\!].$$

Example 1. The canonical acceleration problem of $\mathcal{T}_{non\text{-}dec}$ is

$$\left[\!\!\left[\left(\begin{smallmatrix} x_1' \\ x_2' \end{smallmatrix}\right) = \left(\begin{smallmatrix} x_1 - n \\ x_2 + n \end{smallmatrix}\right) \;\middle|\; \top \;\middle|\; x_1 > 0 \land x_2 > 0 \;\middle|\; \left(\begin{smallmatrix} x_1 - 1 \\ x_2 + 1 \end{smallmatrix}\right) \right]\!\!\right].$$

The first component ψ of an acceleration problem $[\![\psi \mid \check{\varphi} \mid \widehat{\varphi} \mid \boldsymbol{a}]\!]$ is the partial result that has been computed so far. The second component $\check{\varphi}$ corresponds to the part of the loop condition that has already been processed successfully. As our calculus preserves consistency, ψ always approximates $\langle \check{\varphi}, \boldsymbol{a} \rangle$. The third component is the part of the loop condition that remains to be processed, i.e., the loop $\langle \widehat{\varphi}, \boldsymbol{a} \rangle$ still needs to be accelerated. The goal of our calculus is to transform a canonical into a solved acceleration problem.

More specifically, when we have simplified a canonical acceleration problem $[\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \mid \top \mid \varphi(\boldsymbol{x}) \mid \boldsymbol{a}(\boldsymbol{x})]\!]$ to $[\![\psi_1(\boldsymbol{y}) \mid \check{\varphi}(\boldsymbol{x}) \mid \widehat{\varphi}(\boldsymbol{x}) \mid \boldsymbol{a}(\boldsymbol{x})]\!]$, then $\varphi \equiv \check{\varphi} \land \widehat{\varphi}$ and

$$\psi_1 \implies \boldsymbol{x} \longrightarrow^n_{\langle \check{\varphi}, \boldsymbol{a} \rangle} \boldsymbol{x}'.$$

Thus, it then suffices to find some $\psi_2 \in \mathit{Prop}(\mathscr{C}(\boldsymbol{y}))$ such that

$$\boldsymbol{x} \longrightarrow^n_{\langle \breve{\varphi}, \boldsymbol{a} \rangle} \boldsymbol{x}' \wedge \psi_2 \implies \boldsymbol{x} \longrightarrow^n_{\langle \widehat{\varphi}, \boldsymbol{a} \rangle} \boldsymbol{x}'. \tag{1}$$

The reason is that we have $\longrightarrow_{\langle \breve{\varphi}, \boldsymbol{a} \rangle} \cap \longrightarrow_{\langle \widehat{\varphi}, \boldsymbol{a} \rangle} = \longrightarrow_{\langle \breve{\varphi} \wedge \widehat{\varphi}, \boldsymbol{a} \rangle} = \longrightarrow_{\langle \varphi, \boldsymbol{a} \rangle}$ and thus

$$\psi_1 \wedge \psi_2 \implies \boldsymbol{x} \longrightarrow^n_{\langle \varphi, \boldsymbol{a} \rangle} \boldsymbol{x}',$$

i.e., $\psi_1 \wedge \psi_2$ approximates $\mathcal{T}_{loop}$.

Note that the acceleration techniques presented so far would map $\langle \widehat{\varphi}, \boldsymbol{a} \rangle$ to some $\psi_2 \in \mathit{Prop}(\mathscr{C}(\boldsymbol{y}))$ such that

$$\psi_2 \implies \boldsymbol{x} \longrightarrow^n_{\langle \widehat{\varphi}, \boldsymbol{a} \rangle} \boldsymbol{x}', \tag{2}$$

which is more restrictive than (1). In Sec. 5, we will adapt all acceleration techniques from Sec. 3 to search for some $\psi_2 \in \mathit{Prop}(\mathscr{C}(\boldsymbol{y}))$ that satisfies (1) instead of (2), i.e., we will turn them into *conditional acceleration techniques.*

Definition 4 (Conditional Acceleration). *We call a partial function*

$$\mathit{accel} : \mathit{Loop} \times \mathit{Prop}(\mathscr{C}(\boldsymbol{x})) \rightharpoonup \mathit{Prop}(\mathscr{C}(\boldsymbol{y})).$$

a conditional acceleration technique*. It is* sound *if*

$$\boldsymbol{x} \longrightarrow^n_{\langle \breve{\varphi}, \boldsymbol{a} \rangle} \boldsymbol{x}' \wedge \mathit{accel}(\langle \chi, \boldsymbol{a} \rangle, \breve{\varphi}) \quad \textit{implies} \quad \boldsymbol{x} \longrightarrow^n_{\langle \chi, \boldsymbol{a} \rangle} \boldsymbol{x}'$$

for all $(\langle \chi, \boldsymbol{a} \rangle, \breve{\varphi}) \in \mathrm{dom}(\mathit{accel})$, $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{Z}^d$, and $n > 0$. It is exact *if additionally*

$$\boldsymbol{x} \longrightarrow^n_{\langle \chi \wedge \breve{\varphi}, \boldsymbol{a} \rangle} \boldsymbol{x}' \quad \textit{implies} \quad \mathit{accel}(\langle \chi, \boldsymbol{a} \rangle, \breve{\varphi})$$

for all $(\langle \chi, \boldsymbol{a} \rangle, \breve{\varphi}) \in \mathrm{dom}(\mathit{accel})$, $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{Z}^d$, and $n > 0$.

We are now ready to present our *acceleration calculus*, which combines loop acceleration techniques in a modular way. In the following, w.l.o.g. we assume that propositional formulas are in CNF and we identify the formula $\bigwedge_{i=1}^k C_i$ with the set of clauses $\{C_i \mid 1 \leq i \leq k\}$.

Definition 5 (Acceleration Calculus). *The relation $\rightsquigarrow$ on acceleration problems is defined by the following rule:*

$$\frac{\emptyset \neq \chi \subseteq \widehat{\varphi} \quad \mathit{accel}(\langle \chi, \boldsymbol{a} \rangle, \breve{\varphi}) = \psi_2}{[\![\psi_1 \mid \breve{\varphi} \mid \widehat{\varphi} \mid \boldsymbol{a}]\!] \rightsquigarrow_{(e)} [\![\psi_1 \cup \psi_2 \mid \breve{\varphi} \cup \chi \mid \widehat{\varphi} \setminus \chi \mid \boldsymbol{a}]\!]} \quad \textit{accel is a sound conditional acceleration technique}$$

A $\rightsquigarrow$-step is exact *(written $\rightsquigarrow_e$) if accel is exact.*

So our calculus allows us to pick a subset χ (of clauses) from the yet unprocessed condition $\widehat{\varphi}$ and "move" it to $\breve{\varphi}$, which has already been processed successfully. To this end, $\langle \chi, \boldsymbol{a} \rangle$ needs to be accelerated by a conditional acceleration technique, i.e., when accelerating $\langle \chi, \boldsymbol{a} \rangle$ we may assume $\boldsymbol{x} \longrightarrow^n_{\langle \breve{\varphi}, \boldsymbol{a} \rangle} \boldsymbol{x}'$.

Note that every acceleration technique trivially gives rise to a conditional acceleration technique (by disregarding the second argument $\breve{\varphi}$ of *accel* in Def. 4). Thus, our calculus allows for combining arbitrary existing acceleration techniques without adapting them. However, many acceleration techniques can easily be turned into more sophisticated conditional acceleration techniques (cf. Sec. 5), which increases the power of our approach.

Example 2. We continue Ex. 1 and fix $\chi := x_1 > 0$. Thus, we need to accelerate the loop $\left\langle x_1 > 0, \left(\begin{smallmatrix} x_1-1 \\ x_2+1 \end{smallmatrix}\right)\right\rangle$ to enable a $\rightsquigarrow$-step. We obtain

$$\begin{aligned}
&\left[\!\!\left[\psi^{init}_{non\text{-}dec} := \begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \left(\begin{smallmatrix} x_1-n \\ x_2+n \end{smallmatrix}\right) \,\middle|\, \top \,\middle|\, x_1 > 0 \wedge x_2 > 0 \,\middle|\, \left(\begin{smallmatrix} x_1-1 \\ x_2+1 \end{smallmatrix}\right) \right]\!\!\right] \\
\overset{Thm.\ 1}{\rightsquigarrow_e}\ & \left[\!\left[\psi^{init}_{non\text{-}dec} \wedge x_1 - n + 1 > 0 \,\middle|\, x_1 > 0 \,\middle|\, x_2 > 0 \,\middle|\, \left(\begin{smallmatrix} x_1-1 \\ x_2+1 \end{smallmatrix}\right) \right]\!\right] \\
\overset{Thm.\ 2}{\rightsquigarrow_e}\ & \left[\!\left[\psi^{init}_{non\text{-}dec} \wedge x_1 - n + 1 > 0 \wedge x_2 > 0 \,\middle|\, x_1 > 0 \wedge x_2 > 0 \,\middle|\, \top \,\middle|\, \left(\begin{smallmatrix} x_1-1 \\ x_2+1 \end{smallmatrix}\right) \right]\!\right] \\
=\ & \left[\!\left[\psi_{non\text{-}dec} \,\middle|\, x_1 > 0 \wedge x_2 > 0 \,\middle|\, \top \,\middle|\, \left(\begin{smallmatrix} x_1-1 \\ x_2+1 \end{smallmatrix}\right) \right]\!\right]
\end{aligned}$$

where Thm. 2 was applied to the loop $\left\langle x_2 > 0, \left(\begin{smallmatrix} x_1-1 \\ x_2+1 \end{smallmatrix}\right)\right\rangle$ in the second step. Thus, we successfully constructed the formula $\psi_{non\text{-}dec}$, which is equivalent to $\mathcal{T}_{non\text{-}dec}$.

The crucial property of our calculus is the following.

Lemma 1. *$\rightsquigarrow$ preserves consistency and $\rightsquigarrow_e$ preserves exactness.*

Then the correctness of our calculus follows immediately. The reason is that $[\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \mid \top \mid \varphi(\boldsymbol{x}) \mid \boldsymbol{a}(\boldsymbol{x})]\!] \rightsquigarrow^*_{(e)} [\![\psi(\boldsymbol{y}) \mid \check{\varphi}(\boldsymbol{x}) \mid \top \mid \boldsymbol{a}(\boldsymbol{x})]\!]$ implies $\varphi \equiv \check{\varphi}$.

Theorem 5 (Correctness of $\rightsquigarrow$). *If*

$$[\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \mid \top \mid \varphi(\boldsymbol{x}) \mid \boldsymbol{a}(\boldsymbol{x})]\!] \rightsquigarrow^* [\![\psi(\boldsymbol{y}) \mid \check{\varphi}(\boldsymbol{x}) \mid \top \mid \boldsymbol{a}(\boldsymbol{x})]\!],$$

then ψ approximates $\mathcal{T}_{loop}$. If

$$[\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \mid \top \mid \varphi(\boldsymbol{x}) \mid \boldsymbol{a}(\boldsymbol{x})]\!] \rightsquigarrow^*_e [\![\psi(\boldsymbol{y}) \mid \check{\varphi}(\boldsymbol{x}) \mid \top \mid \boldsymbol{a}(\boldsymbol{x})]\!],$$

then ψ is equivalent to $\mathcal{T}_{loop}$.

Termination of our calculus is trivial, as the size of the third component $\widehat{\varphi}$ of the acceleration problem is decreasing.

Theorem 6 (Termination of $\rightsquigarrow$). *$\rightsquigarrow$ terminates.*

5 Conditional Acceleration Techniques

We now show how to turn the acceleration techniques from Sec. 3 into conditional acceleration techniques, starting with *acceleration via monotonic decrease.*

Theorem 7 (Conditional Acceleration via Monotonic Decrease). *If*

$$\check{\varphi}(\boldsymbol{x}) \wedge \chi(\boldsymbol{a}(\boldsymbol{x})) \implies \chi(\boldsymbol{x}),$$

then the following conditional acceleration technique is exact:

$$(\langle \chi, \boldsymbol{a} \rangle, \check{\varphi}) \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \wedge \chi(\boldsymbol{a}^{n-1}(\boldsymbol{x}))$$

So we just add $\check{\varphi}$ to the premise of the implication that needs to be checked to apply *acceleration via monotonic decrease.* Thm. 2 can be adapted analogously.

Theorem 8 (Conditional Acceleration via Monotonic Increase). *If*

$$\check{\varphi}(\boldsymbol{x}) \wedge \chi(\boldsymbol{x}) \implies \chi(\boldsymbol{a}(\boldsymbol{x})),$$

then the following conditional acceleration technique is exact:

$$(\langle \chi, \boldsymbol{a} \rangle, \check{\varphi}) \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \wedge \chi(\boldsymbol{x})$$

Example 3. For the canonical acceleration problem of $\mathcal{T}_{\textit{2-invs}}$, we obtain:

$$\begin{aligned}
&\left[\!\left[\boldsymbol{x}' = \boldsymbol{a}^n_{\textit{2-invs}}(\boldsymbol{x}) \;\middle|\; \top \;\middle|\; x_1 > 0 \wedge x_2 > 0 \;\middle|\; \boldsymbol{a}_{\textit{2-invs}} := \left(\begin{smallmatrix} x_1+x_2 \\ x_2-1 \end{smallmatrix}\right) \right]\!\right] \\
\overset{\textit{Thm. } 7}{\rightsquigarrow_e}\; &[\![\boldsymbol{x}' = \boldsymbol{a}^n_{\textit{2-invs}}(\boldsymbol{x}) \wedge x_2 - n + 1 > 0 \mid x_2 > 0 \mid x_1 > 0 \mid \boldsymbol{a}_{\textit{2-invs}}]\!] \\
\overset{\textit{Thm. } 8}{\rightsquigarrow_e}\; &[\![\boldsymbol{x}' = \boldsymbol{a}^n_{\textit{2-invs}}(\boldsymbol{x}) \wedge x_2 - n + 1 > 0 \wedge x_1 > 0 \mid x_2 > 0 \wedge x_1 > 0 \mid \top \mid \boldsymbol{a}_{\textit{2-invs}}]\!]
\end{aligned}$$

While we could also use Thm. 1 for the first step, Thm. 2 is inapplicable in the second step. The reason is that we need the converse invariant $x_2 > 0$ to prove invariance of $x_1 > 0$.

It is not a coincidence that $\mathcal{T}_{\textit{2-invs}}$, which could also be accelerated with *acceleration via monotonicity* (cf. Thm. 3) directly, can be handled by applying our novel calculus with Theorems 7 and 8.

Remark 1. If applying *acceleration via monotonicity* to $\mathcal{T}_{\textit{loop}}$ yields ψ, then

$$[\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \mid \top \mid \varphi(\boldsymbol{x}) \mid \boldsymbol{a}(\boldsymbol{x})]\!] \rightsquigarrow_e^{\leq 3} [\![\psi(\boldsymbol{y}) \mid \varphi(\boldsymbol{x}) \mid \top \mid \boldsymbol{a}(\boldsymbol{x})]\!]$$

where either Thm. 7 or Thm. 8 is applied in each $\rightsquigarrow_e$-step.

Thus, there is no need for a conditional variant of *acceleration via monotonicity*. Note that combining Theorems 7 and 8 with our calculus is also useful for loops where *acceleration via monotonicity* is inapplicable.

Example 4. Consider the following loop, which can be accelerated by splitting its guard into one invariant and two converse invariants.

$$\textbf{while } x_1 > 0 \wedge x_2 > 0 \wedge x_3 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \\ x_3 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1-1 \\ x_2+x_1 \\ x_3-x_2 \end{smallmatrix}\right) \qquad (\mathcal{T}_{\textit{2-c-invs}})$$

Let

$$\begin{aligned}
\varphi_{\textit{2-c-invs}} &:= x_1 > 0 \wedge x_2 > 0 \wedge x_3 > 0, \\
\boldsymbol{a}_{\textit{2-c-invs}} &:= \left(\begin{smallmatrix} x_1-1 \\ x_2+x_1 \\ x_3-x_2 \end{smallmatrix}\right), \\
\psi^{\textit{init}}_{\textit{2-c-invs}} &:= \boldsymbol{x}' = \boldsymbol{a}^n_{\textit{2-c-invs}}(\boldsymbol{x}),
\end{aligned}$$

and let $x_i^{(m)}$ be the i^{th} component of $\boldsymbol{a}^m_{\textit{2-c-invs}}(\boldsymbol{x})$. Starting with the canonical acceleration problem of $\mathcal{T}_{\textit{2-c-invs}}$, we obtain:

$$\begin{aligned}
&\left[\!\left[\psi^{\textit{init}}_{\textit{2-c-invs}} \;\middle|\; \top \;\middle|\; \varphi_{\textit{2-c-invs}} \;\middle|\; \boldsymbol{a}_{\textit{2-c-invs}} \right]\!\right] \\
\overset{\textit{Thm. } 7}{\rightsquigarrow_e}\; &\left[\!\left[\psi^{\textit{init}}_{\textit{2-c-invs}} \wedge x_1^{(n-1)} > 0 \;\middle|\; x_1 > 0 \;\middle|\; x_2 > 0 \wedge x_3 > 0 \;\middle|\; \boldsymbol{a}_{\textit{2-c-invs}} \right]\!\right] \\
\overset{\textit{Thm. } 8}{\rightsquigarrow_e}\; &\left[\!\left[\psi^{\textit{init}}_{\textit{2-c-invs}} \wedge x_1^{(n-1)} > 0 \wedge x_2 > 0 \;\middle|\; x_1 > 0 \wedge x_2 > 0 \;\middle|\; x_3 > 0 \;\middle|\; \boldsymbol{a}_{\textit{2-c-invs}} \right]\!\right] \\
\overset{\textit{Thm. } 7}{\rightsquigarrow_e}\; &\left[\!\left[\psi^{\textit{init}}_{\textit{2-c-invs}} \wedge x_1^{(n-1)} > 0 \wedge x_2 > 0 \wedge x_3^{(n-1)} > 0 \;\middle|\; \varphi_{\textit{2-c-invs}} \;\middle|\; \top \;\middle|\; \boldsymbol{a}_{\textit{2-c-invs}} \right]\!\right]
\end{aligned}$$

Finally, we present a variant of Thm. 4 for conditional acceleration. The idea is similar to the approach for deducing metering functions of the form $\boldsymbol{x} \mapsto I_{\check{\varphi}}(\boldsymbol{x}) \cdot f(\boldsymbol{x})$ from [16] (see Sec. 3.3 for details). But in contrast to [16], in our setting the "conditional" part $\check{\varphi}$ does not need to be an invariant of the loop.

Theorem 9 (Conditional Acceleration via Metering Functions). *Let $mf : \mathbb{Z}^d \to \mathbb{Q}$. If*

$$\begin{aligned} \check{\varphi}(\boldsymbol{x}) \wedge \chi(\boldsymbol{x}) &\implies mf(\boldsymbol{x}) - mf(\boldsymbol{a}(\boldsymbol{x})) \leq 1 \qquad\qquad \textit{and} \\ \check{\varphi}(\boldsymbol{x}) \wedge \neg\chi(\boldsymbol{x}) &\implies mf(\boldsymbol{x}) \leq 0, \end{aligned}$$

then the following conditional acceleration technique is sound:

$$(\langle \chi, \boldsymbol{a} \rangle, \check{\varphi}) \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \wedge \chi(\boldsymbol{x}) \wedge n < mf(\boldsymbol{x}) + 1$$

6 Acceleration via Eventual Monotonicity

The combination of the calculus from Sec. 4 and the conditional acceleration techniques from Sec. 5 still fails to handle certain interesting classes of loops. Thus, to improve the applicability of our approach we now present two new acceleration techniques based on *eventual* monotonicity.

6.1 Acceleration via Eventual Decrease

All (combinations of) techniques presented so far fail for the following example.

$$\textbf{while } x_1 > 0 \textbf{ do } \left(\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right) \leftarrow \left(\begin{smallmatrix} x_1 + x_2 \\ x_2 - 1 \end{smallmatrix}\right) \qquad (\mathcal{T}_{ev\text{-}dec})$$

The reason is that x_1 does not behave monotonically, i.e., $x_1 > 0$ is neither an invariant nor a converse invariant. Essentially, $\mathcal{T}_{ev\text{-}dec}$ proceeds in two phases: In the first (optional) phase, x_2 is positive and hence the value of x_1 is monotonically increasing. In the second phase, x_2 is non-positive and consequently the value of x_1 decreases (weakly) monotonically. The crucial observation is that once the value of x_1 decreases, it can never increase again. Thus, despite the non-monotonic behavior of x_1, it suffices to require that $x_1 > 0$ holds before the first and before the n^{th} loop iteration to ensure that the loop can be iterated at least n times.

Theorem 10 (Acceleration via Eventual Decrease). *If $\varphi(\boldsymbol{x}) \equiv \bigwedge_{i=1}^{k} C_i$ where each C_i contains an inequation $expr_i(\boldsymbol{x}) > 0$ such that*

$$expr_i(\boldsymbol{x}) \geq expr_i(\boldsymbol{a}(\boldsymbol{x})) \implies expr_i(\boldsymbol{a}(\boldsymbol{x})) \geq expr_i(\boldsymbol{a}^2(\boldsymbol{x})),$$

then the following acceleration technique is sound:

$$\mathcal{T}_{loop} \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \wedge \bigwedge_{i=1}^{k} \left(expr_i(\boldsymbol{x}) > 0 \wedge expr_i(\boldsymbol{a}^{n-1}(\boldsymbol{x})) > 0 \right)$$

If $C_i \equiv expr_i > 0$ for all $i \in [1, k]$, then it is exact.

With Thm. 10, we can accelerate $\mathcal{T}_{ev\text{-}dec}$ to

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \frac{n-n^2}{2}+x_2 \cdot n + x_1 \\ x_2 - n \end{pmatrix} \wedge x_1 > 0 \wedge \frac{n-1-(n-1)^2}{2} + x_2 \cdot (n-1) + x_1 > 0$$

as we have

$$(x_1 \geq x_1 + x_2) \equiv (0 \geq x_2) \implies (0 \geq x_2 - 1) \equiv (x_1 + x_2 \geq x_1 + x_2 + x_2 - 1).$$

Turning Thm. 10 into a conditional acceleration technique is straightforward.

Theorem 11 (Conditional Acceleration via Eventual Decrease). *If we have $\chi(\boldsymbol{x}) \equiv \bigwedge_{i=1}^k C_i$ where each C_i contains an inequation $expr_i(\boldsymbol{x}) > 0$ such that*

$$\breve{\varphi}(\boldsymbol{x}) \wedge expr_i(\boldsymbol{x}) \geq expr_i(\boldsymbol{a}(\boldsymbol{x})) \implies expr_i(\boldsymbol{a}(\boldsymbol{x})) \geq expr_i(\boldsymbol{a}^2(\boldsymbol{x})), \tag{3}$$

then the following conditional acceleration technique is sound:

$$(\langle \chi, \boldsymbol{a} \rangle, \breve{\varphi}) \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \wedge \bigwedge_{i=1}^{k} \left(expr_i(\boldsymbol{x}) > 0 \wedge expr_i(\boldsymbol{a}^{n-1}(\boldsymbol{x})) > 0 \right)$$

If $C_i \equiv expr_i > 0$ for all $i \in [1,k]$, then it is exact.

Example 5. Consider the following variant of $\mathcal{T}_{ev\text{-}dec}$.

$$\textbf{while } x_1 > 0 \wedge x_3 > 0 \textbf{ do } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_1 + x_2 \\ x_2 - x_3 \\ x_3 \end{pmatrix}$$

Starting with its canonical acceleration problem, we get

$$\begin{aligned} & \left[\!\!\left[\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \;\middle|\; \top \;\middle|\; x_1 > 0 \wedge x_3 > 0 \;\middle|\; \boldsymbol{a} := \begin{pmatrix} x_1 + x_2 \\ x_2 - x_3 \\ x_3 \end{pmatrix} \right]\!\!\right] \\ \overset{Thm.\ 8}{\leadsto_e}\ & [\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \wedge x_3 > 0 \mid x_3 > 0 \mid x_1 > 0 \mid \boldsymbol{a}]\!] \\ \overset{Thm.\ 11}{\leadsto_e}\ & \left[\!\!\left[\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \wedge x_3 > 0 \wedge x_1 > 0 \wedge x_1^{(n-1)} > 0 \;\middle|\; x_3 > 0 \wedge x_1 > 0 \;\middle|\; \top \;\middle|\; \boldsymbol{a} \right]\!\!\right] \end{aligned}$$

where the second step can be performed via Thm. 11 as

$$(\breve{\varphi}(\boldsymbol{x}) \wedge expr(\boldsymbol{x}) \geq expr(\boldsymbol{a}(\boldsymbol{x}))) \equiv (x_3 > 0 \wedge x_1 \geq x_1 + x_2) \equiv (x_3 > 0 \wedge 0 \geq x_2)$$

implies

$$(0 \geq x_2 - x_3) \equiv (x_1 + x_2 \geq x_1 + x_2 + x_2 - x_3) \equiv (expr(\boldsymbol{a}(\boldsymbol{x})) \geq expr(\boldsymbol{a}^2(\boldsymbol{x}))).$$

6.2 Acceleration via Eventual Increase

Still, all (combinations of) techniques presented so far fail for

$$\textbf{while } x_1 > 0 \textbf{ do } \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} x_1 + x_2 \\ x_2 + 1 \end{pmatrix}. \tag{$\mathcal{T}_{ev\text{-}inc}$}$$

As in the case of $\mathcal{T}_{ev\text{-}dec}$, the value of x_1 does not behave monotonically, i.e., $x_1 > 0$ is neither an invariant nor a converse invariant. However, this time x_1 is eventually *increasing*, i.e., once x_1 starts to grow, it never decreases again. Thus, in this case it suffices to require that x_1 is positive and (weakly) increasing.

Theorem 12 (Acceleration via Eventual Increase). *If* $\varphi(\boldsymbol{x}) \equiv \bigwedge_{i=1}^{k} C_i$ *where each* C_i *contains an inequation* $\mathit{expr}_i(\boldsymbol{x}) > 0$ *such that*

$$\mathit{expr}_i(\boldsymbol{x}) \leq \mathit{expr}_i(\boldsymbol{a}(\boldsymbol{x})) \implies \mathit{expr}_i(\boldsymbol{a}(\boldsymbol{x})) \leq \mathit{expr}_i(\boldsymbol{a}^2(\boldsymbol{x})),$$

then the following acceleration technique is sound:

$$\mathcal{T}_{loop} \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land \bigwedge_{i=1}^{k} 0 < \mathit{expr}_i(\boldsymbol{x}) \leq \mathit{expr}_i(\boldsymbol{a}(\boldsymbol{x}))$$

With Thm. 12, we can accelerate $\mathcal{T}_{\mathit{ev\text{-}inc}}$ to

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \frac{n^2-n}{2} + x_2 \cdot n + x_1 \\ x_2 + n \end{pmatrix} \land 0 < x_1 \leq x_1 + x_2 \qquad (\psi_{\mathit{ev\text{-}inc}})$$

as we have

$$(x_1 \leq x_1 + x_2) \equiv (0 \leq x_2) \implies (0 \leq x_2 + 1) \equiv (x_1 + x_2 \leq x_1 + x_2 + x_2 + 1).$$

However, Thm. 12 is *not* exact, as the resulting formula only covers program runs where each expr_i behaves monotonically. So $\psi_{\mathit{ev\text{-}inc}}$ only covers those runs of $\mathcal{T}_{\mathit{ev\text{-}inc}}$ where the initial value of x_2 is non-negative. Again, turning Thm. 12 into a conditional acceleration technique is straightforward.

Theorem 13 (Conditional Acceleration via Eventual Increase). *If we have* $\chi(\boldsymbol{x}) \equiv \bigwedge_{i=1}^{k} C_i$ *where each* C_i *contains an inequation* $\mathit{expr}_i(\boldsymbol{x}) > 0$ *such that*

$$\breve{\varphi}(\boldsymbol{x}) \land \mathit{expr}_i(\boldsymbol{x}) \leq \mathit{expr}_i(\boldsymbol{a}(\boldsymbol{x})) \implies \mathit{expr}_i(\boldsymbol{a}(\boldsymbol{x})) \leq \mathit{expr}_i(\boldsymbol{a}^2(\boldsymbol{x})), \qquad (4)$$

then the following conditional acceleration technique is sound:

$$(\langle \chi, \boldsymbol{a} \rangle, \breve{\varphi}) \mapsto \boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land \bigwedge_{i=1}^{k} 0 < \mathit{expr}_i(\boldsymbol{x}) \leq \mathit{expr}_i(\boldsymbol{a}(\boldsymbol{x}))$$

Example 6. Consider the following variant of $\mathcal{T}_{\mathit{ev\text{-}inc}}$.

$$\textbf{while } x_1 > 0 \land x_3 > 0 \textbf{ do } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_1 + x_2 \\ x_2 + x_3 \\ x_3 \end{pmatrix}$$

Starting with its canonical acceleration problem, we get

$$\begin{aligned}
& \left[\!\!\left[\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \;\middle|\; \top \;\middle|\; x_1 > 0 \land x_3 > 0 \;\middle|\; \boldsymbol{a} := \begin{pmatrix} x_1 + x_2 \\ x_2 + x_3 \\ x_3 \end{pmatrix} \right]\!\!\right] \\
\overset{Thm.\ 8}{\leadsto_e}\ & [\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land x_3 > 0 \mid x_3 > 0 \mid x_1 > 0 \mid \boldsymbol{a}]\!] \\
\overset{Thm.\ 13}{\leadsto}\ & [\![\boldsymbol{x}' = \boldsymbol{a}^n(\boldsymbol{x}) \land x_3 > 0 \land 0 < x_1 \leq x_1 + x_2 \mid x_3 > 0 \land x_1 > 0 \mid \top \mid \boldsymbol{a}]\!]
\end{aligned}$$

where the second step can be performed via Thm. 13 as

$$(\breve{\varphi}(\boldsymbol{x}) \land \mathit{expr}(\boldsymbol{x}) \leq \mathit{expr}(\boldsymbol{a}(\boldsymbol{x}))) \equiv (x_3 > 0 \land x_1 \leq x_1 + x_2) \equiv (x_3 > 0 \land 0 \leq x_2)$$

implies

$$(0 \leq x_2 + x_3) \equiv (x_1 + x_2 \leq x_1 + x_2 + x_2 + x_3) \equiv (\mathit{expr}(\boldsymbol{a}(\boldsymbol{x})) \leq \mathit{expr}(\boldsymbol{a}^2(\boldsymbol{x}))).$$

We also considered versions of Theorems 11 and 13 where the inequations in (3) resp. (4) are strict, but this did not lead to an improvement in our experiments. Moreover, we experimented with a variant of Thm. 13 that splits the loop under consideration into two consecutive loops, accelerates them independently, and composes the results. While such an approach can accelerate loops like $\psi_{ev\text{-}inc}$ exactly, the impact on our experimental results was minimal. Thus, we postpone an in-depth investigation of this idea to future work.

7 Related Work

Acceleration-like techniques are also used in *over-approximating* settings (see, e.g., [10, 20, 21, 25, 26, 29, 32, 33]), whereas we consider *exact* and *under-approximating* loop acceleration techniques. As many related approaches have already been discussed in Sec. 3, we only mention two more techniques here.

First, [4, 7] presents an exact acceleration technique for *finite monoid affine transformations* (FMATs), i.e., loops with linear arithmetic whose body is of the form $\boldsymbol{x} \leftarrow A\boldsymbol{x} + \boldsymbol{b}$ where $\{A^i \mid i \in \mathbb{N}\}$ is finite. For such loops, Presburger-Arithmetic is sufficient to construct an equivalent formula ψ, i.e., it can be expressed in a decidable logic. In general, this is clearly not the case for the techniques presented in the current paper (which may even synthesize non-polynomial closed forms, see $\mathcal{T}_{exp}$). As a consequence and in contrast to our technique, this approach cannot handle loops where the values of variables grow super-linearly (i.e., it cannot handle examples like $\mathcal{T}_{2\text{-}invs}$). Implementations are available in the tools FAST [2] and Flata [24]. Further theoretical results on linear transformations whose n-fold closure is definable in (extensions of) Presburger-Arithmetic can be found in [5].

Second, [6] shows that *octagonal relations* can be accelerated exactly and in [27], it is proven that such relations can even be accelerated in polynomial time. This generalizes earlier results for *difference bound constraints* [9]. As in the case of FMATs, the resulting formula can be expressed in Presburger-Arithmetic. Octagonal relations are defined by a finite conjunction ξ of inequations of the form $\pm x \pm y \leq c$, $x, y \in \boldsymbol{x} \cup \boldsymbol{x}'$, $c \in \mathbb{Z}$. Then ξ induces the relation $\boldsymbol{x} \longrightarrow_{\xi} \boldsymbol{x}' \iff \xi(\boldsymbol{x}, \boldsymbol{x}')$. So in contrast to the loops considered in the current paper where $\boldsymbol{x}'$ is uniquely determined by $\boldsymbol{x}$, octagonal relations can represent non-deterministic programs. Therefore and due to the restricted form of octagonal relations, the work from [6, 27] is orthogonal to ours.

8 Implementation and Experiments

We prototypically implemented our approach in our open-source Loop Acceleration Tool LoAT [11, 16, 17]:

https://github.com/aprove-developers/LoAT/tree/tacas20

It uses Z3 [30] to check implications and PURRS [1] to compute closed forms.

For technical reasons, the closed forms computed by LoAT are valid only if $n > 0$, whereas Def. 2 requires them to be valid for all $n \in \mathbb{N}$. The reason is that PURRS has only limited support for initial conditions. In the future, we plan to use a different recurrence solver to circumvent this problem. Thus, LoAT's results are only correct for all $n > 1$ (instead of all $n > 0$). Moreover, LoAT can currently compute closed forms only if the loop body is *triangular*, meaning that each a_i is an expression over $x_1, \ldots, x_i$. The reason is that PURRS cannot solve *systems* of recurrence equations, but only a single recurrence equation at a time. However, LoAT failed to compute closed forms for just 26 out of 1511 loops in our experiments, i.e., this appears to be a minor restriction in practice. Furthermore, *conditional acceleration via metering functions* has not yet been integrated into the implementation of our calculus. While LoAT can synthesize formulas with non-polynomial arithmetic, it cannot yet parse them, i.e., the input is restricted to polynomials. Finally, LoAT does not yet support disjunctive loop conditions.

Apart from these differences, our implementation closely follows the current paper. It repeatedly applies the conditional acceleration techniques from Sections 5 and 6 with the following priorities: *Thm.* 8 $>$ *Thm.* 7 $>$ *Thm.* 11 $>$ *Thm.* 13.

To evaluate our approach, we extracted 1511 loops with conjunctive guards from the category *Termination of Integer Transition Systems* of the *Termination Problems Database* [35], the benchmark collection which is used at the annual *Termination and Complexity Competition* [19], as follows:

1. We parsed all examples with LoAT and extracted each single-path loop with conjunctive guard (resulting in 3829 benchmarks).
2. We removed duplicates by checking syntactic equality (resulting in 2825 benchmarks).
3. We removed loops whose runtime is trivially constant using an incomplete check (resulting in 1733 benchmarks).
4. We removed loops which do not admit any terminating runs, i.e., loops where Thm. 2 applies (resulting in 1511 benchmarks).

We compared our implementation with LoAT's implementation of *acceleration via monotonicity* (Thm. 3, [11]) and its implementation of *acceleration via metering functions* (Thm. 4, [17]), which also incorporates the improvements proposed in [16]. We did not include the techniques from Theorems 1 and 2 in our evaluation, as they are subsumed by *acceleration via monotonicity*. Furthermore, we compared with Flata [24], which implements the techniques to accelerate FMATs and octagonal relations discussed in Sec. 7. Note that our benchmark collection contains 16 loops with non-linear arithmetic where Flata is bound to fail, since it only supports linear arithmetic. We did not compare with FAST [2], which uses a similar approach as the more recent tool Flata.

All tests have been run on StarExec [34]. The results can be seen in Table 1. They show that our novel calculus was superior to the competing techniques in our experiments. In all but 7 cases where our calculus successfully accelerated the given loop, the resulting formula was polynomial. Thus, integrating our approach into existing acceleration-based verification techniques should not present major obstacles w.r.t. automation.

	LoAT	Monot.	Meter	Flata
exact	1444	845	0[3]	1231
approx	38	0	733	0
fail	29	666	778	280
avg rt	0.16s	0.11s	0.09s	0.47s

Table 1.

	~~Ev-Inc~~	~~Ev-Dec~~	~~Ev-Mon~~
exact	1444	845	845
approx	0	493	0
fail	67	173	666
avg rt	0.15s	0.14s	0.09s

Table 2.

LoAT:	Acceleration calculus + Theorems 7, 8, 11 and 13
Monot.:	*Acceleration via Monotonicity*, Thm. 3
Meter:	*Acceleration via Metering Functions*, Thm. 4
Flata:	The tool Flata, see http://nts.imag.fr/index.php/Flata
~~Ev-Inc~~:	Acceleration calculus + Theorems 7, 8 and 11
~~Ev-Dec~~:	Acceleration calculus + Theorems 7, 8 and 13
~~Ev-Mon~~:	Acceleration calculus + Theorems 7 and 8
exact:	Number of examples that were accelerated *exactly*
approx:	Number of examples that were accelerated *approximately*
fail:	Number of examples that could not be accelerated
avg rt:	Average runtime per example

Furthermore, we evaluated the impact of our new acceleration techniques from Sec. 6 independently. To this end, we once disabled *acceleration via eventual increase*, *acceleration via eventual decrease*, and both of them. The results can be seen in Table 2. They show that our calculus does not improve over *acceleration via monotonicity* if both *acceleration via eventual increase* and *acceleration via eventual decrease* are disabled (i.e., our benchmark collection does not contain examples like $\mathcal{T}_{2\text{-}c\text{-}invs}$). However, enabling either *acceleration via eventual decrease* or *acceleration via eventual increase* resulted in a significant improvement. Interestingly, there are many examples that can be accelerated with either of these two techniques: When both of them were enabled, LoAT (exactly or approximately) accelerated 1482 loops. When one of them was enabled, it accelerated 1444 resp. 1338 loops. But when none of them was enabled, it only accelerated 845 loops. We believe that this is due to examples like

$$\textbf{while } x_1 > 0 \wedge \ldots \textbf{ do } \begin{pmatrix} x_1 \\ x_2 \\ \ldots \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_2 \\ \ldots \end{pmatrix}$$

where Thm. 11 *and* Thm. 13 are applicable (since $x_1 \leq x_2$ implies $x_2 \leq x_2$ and $x_1 \geq x_2$ implies $x_2 \geq x_2$).

Flata exactly accelerated 49 loops where LoAT failed or approximated and LoAT exactly accelerated 262 loops where Flata failed. So there were only 18 loops where both Flata and the full version of our calculus failed to compute an exact result. Among them were the only 3 examples where our implementation found a closed form, but failed anyway. One of them was[4]

[3] While acceleration via metering functions may be exact in some cases (see the discussion after Thm. 4), our implementation cannot check whether this is the case.

[4] The other two are structurally similar, but more complex.

$$\textbf{while } x_3 > 0 \textbf{ do } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_1+1 \\ x_2-x_1 \\ x_3+x_2 \end{pmatrix}.$$

Here, the updated value of x_1 depends on x_1, the update of x_2 depends on x_1 and x_2, and the update of x_3 depends on x_2 and x_3. Hence, the closed form of x_1 is linear, the closed form of x_2 is quadratic, and the closed form of x_3 is cubic:

$$x_3^{(n)} = -\frac{1}{6} \cdot n^3 + \frac{1-x_1}{2} \cdot n^2 + \left(\frac{x_1}{2} + x_2 - \frac{1}{3}\right) \cdot n + x_3$$

So when fixing x_1, x_2, and x_3, $x_3^{(n)}$ has up to 2 extrema, i.e., its monotonicity may change twice. However, our techniques based on eventual monotonicity require that the respective expressions behave monotonically once they start to de- or increase, so these techniques only allow one change of monotonicity.

This raises the question if our approach can accelerate *every* loop with conjunctive guard and linear arithmetic whose closed form is a vector of (at most) quadratic polynomials with rational coefficients. We leave this to future work.

For our benchmark collection, links to the StarExec-jobs of our evaluation, and a pre-compiled binary (Linux, 64 bit) we refer to [14].

9 Conclusion and Future Work

After discussing existing acceleration techniques (Sec. 3), we presented a calculus to combine acceleration techniques modularly (Sec. 4). Then we showed how to combine existing (Sec. 5) and two novel (Sec. 6) acceleration techniques with our calculus. This improves over prior approaches, where acceleration techniques were used independently, and may thus improve acceleration-based verification techniques [6,7,11,16–18,28] in the future. An empirical evaluation (Sec. 8) shows that our approach is more powerful than state-of-the-art acceleration techniques. Moreover, if it is able to accelerate a loop, then the result is exact (instead of just an under-approximation) in most cases. Thus, our calculus can be used for under-approximating techniques (e.g., to find bugs or counterexamples) as well as in over-approximating settings (e.g., to prove safety or termination).

In the future, we plan to implement the missing features mentioned in Sec. 8 and integrate our novel calculus into our own acceleration-based program analyses to prove lower bounds on the runtime complexity [16,17] and non-termination [11] of integer programs. Furthermore, our experiments indicate that integrating specialized techniques for FMATs (cf. Sec. 7) would improve the power of our approach, as Flata exactly accelerated 49 loops where LoAT failed to do so (cf. Sec. 8). Moreover, we plan to design a *loop acceleration library*, such that our technique can easily be incorporated by other verification tools.

Data Availability Statement and Acknowledgments The tools and datasets used for the current study are available in the Zenodo repository [12].

I thank Carsten Fuhs, Marcel Hark, Sophie Tourret, and the anonymous reviewers for helpful feedback and comments. Moreover, I thank Radu Iosif and Filip Konecný for their help with Flata.

References

1. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis (2005), arXiv:cs/0512056 [cs.MS]
2. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Acceleration from theory to practice. STTT **10**(5), 401–424 (2008). https://doi.org/10.1007/s10009-008-0064-3
3. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: ATVA '05. pp. 474–488. LNCS 3707 (2005). https://doi.org/10.1007/11562948_35
4. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. Ph.D. thesis, Université de Liège (1999), https://orbi.uliege.be/bitstream/2268/74874/1/Boigelot98.pdf
5. Boigelot, B.: On iterating linear transformations over recognizable sets of integers. Theoretical Computer Science **309**(1-3), 413–468 (2003). https://doi.org/10.1016/S0304-3975(03)00314-1
6. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: TACAS '09. pp. 337–351. LNCS 5505 (2009). https://doi.org/10.1007/978-3-642-00768-2_29
7. Bozga, M., Iosif, R., Konecný, F.: Fast acceleration of ultimately periodic relations. In: CAV '10. pp. 227–242. LNCS 6174 (2010). https://doi.org/10.1007/978-3-642-14295-6_23
8. Bozga, M., Iosif, R., Konecný, F.: Deciding conditional termination. Logical Methods in Computer Science **10**(3) (2014). https://doi.org/10.2168/LMCS-10(3:8)2014
9. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: CAV '98. pp. 268–279. LNCS 1427 (1998). https://doi.org/10.1007/BFb0028751
10. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD '15. pp. 57–64 (2015). https://doi.org/10.1109/FMCAD.2015.7542253
11. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: FMCAD '19. pp. 221–230 (2019). https://doi.org/10.23919/FMCAD.2019.8894271
12. Frohn, F.: A calculus for modular loop acceleration – artifact evaluation (2020). https://doi.org/10.5281/zenodo.3676348
13. Frohn, F.: A calculus for modular loop acceleration (2020), extended version, arXiv:2001.01516 [cs.LO]
14. Frohn, F.: Empirical evaluation of "A calculus for modular loop acceleration" (2020), https://ffrohn.github.io/acceleration-calculus
15. Frohn, F., Hark, M., Giesl, J.: On the decidability of termination for polynomial loops (2019), arXiv:1910.11588 [cs.LO]
16. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs (2019), arXiv:1911.01077 [cs.LO]
17. Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower runtime bounds for integer programs. In: IJCAR '16. pp. 550–567. LNCS 9706 (2016). https://doi.org/10.1007/978-3-319-40229-1_37
18. Ganty, P., Iosif, R., Konecný, F.: Underapproximation of procedure summaries for integer programs. STTT **19**(5), 565–584 (2017). https://doi.org/10.1007/s10009-016-0420-7
19. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: TACAS '19. pp. 156–166. LNCS 11429 (2019). https://doi.org/10.1007/978-3-030-17502-3_10

20. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: SAS '06. pp. 144–160. LNCS 4134 (2006). https://doi.org/10.1007/11823230_10
21. Gonnord, L., Schrammel, P.: Abstract acceleration in linear relation analysis. Science of Computer Programming **93**, 125–153 (2014). https://doi.org/10.1016/j.scico.2013.09.016
22. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.: Proving non-termination. In: POPL '08. pp. 147–158 (2008). https://doi.org/10.1145/1328438.1328459
23. Hojjat, H., Iosif, R., Konecný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: ATVA '12. pp. 187–202. LNCS 7561 (2012). https://doi.org/10.1007/978-3-642-33386-6_16
24. Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: FM '12. pp. 247–251. LNCS 7436 (2012). https://doi.org/10.1007/978-3-642-32759-9_21
25. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: POPL '14. pp. 529–540 (2014). https://doi.org/10.1145/2535838.2535843
26. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.W.: Compositional recurrence analysis revisited. In: PLDI '17. pp. 248–262 (2017). https://doi.org/10.1145/3062341.3062373
27. Konecný, F.: PTIME computation of transitive closures of octagonal relations. In: TACAS '16. pp. 645–661. LNCS 9636 (2016). https://doi.org/10.1007/978-3-662-49674-9_42
28. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. FMSD **47**(1), 75–92 (2015). https://doi.org/10.1007/s10703-015-0228-1
29. Madhukar, K., Wachter, B., Kroening, D., Lewis, M., Srivas, M.K.: Accelerating invariant generation. In: FMCAD '15. pp. 105–111 (2015). https://doi.org/10.1109/FMCAD.2015.7542259
30. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS '08. pp. 337–340. LNCS 4963 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
31. Ouaknine, J., Pinto, J.S., Worrell, J.: On termination of integer linear loops. In: SODA '15. pp. 957–969 (2015). https://doi.org/10.1137/1.9781611973730.65
32. Silverman, J., Kincaid, Z.: Loop summarization with rational vector addition systems. In: CAV '19. pp. 97–115. LNCS 11562 (2019). https://doi.org/10.1007/978-3-030-25543-5_7
33. Strejcek, J., Trtík, M.: Abstracting path conditions. In: ISSTA '12. pp. 155–165 (2012). https://doi.org/10.1145/2338965.2336772
34. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: IJCAR '14. pp. 367–373. LNCS 8562 (2014). https://doi.org/10.1007/978-3-319-08587-6_28
35. Termination problems data base (TPDB), http://termination-portal.org/wiki/TPDB

SAT and SMT

Mind the Gap: Bit-vector Interpolation recast over Linear Integer Arithmetic

Takamasa Okudono[1,2] and Andy King[3]

[1] National Institute of Informatics, Tokyo, Japan
[2] The Graduate University for Advanced Studies (SOKENDAI), Tokyo, Japan
[3] University of Kent, Canterbury, UK

Abstract. Much of an interpolation engine for bit-vector (BV) arithmetic can be constructed by observing that BV arithmetic can be modeled with linear integer arithmetic (LIA). Two BV formulae can thus be translated into two LIA formulae and then an interpolation engine for LIA used to derive an interpolant, albeit one expressed in LIA. The construction is completed by back-translating the LIA interpolant into a BV formula whose models coincide with those of the LIA interpolant. This paper develops a back-translation algorithm showing, for the first time, how back-translation can be universally applied, whatever the LIA interpolant. This avoids the need for deriving a BV interpolant by bit-blasting the BV formulae, as a backup process when back-translation fails. The new back-translation process relies on a novel geometric technique, called gapping, the correctness and practicality of which are demonstrated.

1 Introduction

Given two formulae A and B which are inconsistent, an interpolant for the ordered pair $\langle A, B\rangle$ is a formula I over the variables common to both A and B which is a relaxation of A that is still inconsistent with B. For example, when working over the theory of linear inequalities, if $A = (x = y + 1) \wedge (y = 0)$ and $B = (x = z + 2) \wedge (1 \leq z)$ then interpolants for $\langle A, B\rangle$ are $I_1 = (x = 1)$, $I_2 = (x \leq 1)$ and $I_3 = (x < 3)$, ordering by increasing generality. The intuition behind I_1, I_2 and I_3 is that they are abstractions of A which concisely explain the inconsistency between A and B. Interpolation has attracted growing attention over the last decade [26], because of the crucial role it plays in model checking in lazy [18] predicate abstraction [15] and lazy abstraction with interpolants [25], as exemplified in BLAST [5] and IMPACT [25] respectively. In lazy predicate abstraction [25], interpolation is used to synthesise predicates which describe program state. Predicates are added, on demand, to explain why a path through a program cannot reach an error state. In lazy abstraction with interpolants [25], program state is described with unrestricted formulae, rather than merely using predicates, and interpolation is applied to relax sequences of formulae that describe the states down paths which do not error. Interpolation simplify these formulae but increasing the likelihood of covering, again accelerating path exploration. In effect, interpolation is the key abstraction mechanism.

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 79–96, 2020.
https://doi.org/10.1007/978-3-030-45190-5_5

Context As solvers for richer theories have evolved so have interpolation engines for these theories, with a notable flurry of activity around one decade ago [10, 11, 19, 20, 23, 24, 30]. However, progress on the important theory of bit-vectors (BV) has been surprisingly slow, the two key works [2, 16] taking opposing approaches. One takes advantage of existing interpolation engines [16] and the another develops a bespoke interpolation engine around lazy reduction [2], which supports bit-vector operations by expanding them, on demand, to Presburger arithmetic [2]. This paper develops the former approach, aiming to use an LIA solver as is.

The central problem in bit-vector interpolation is to construct an interpolant which is compact (one might even say beautiful [1]). Although a pair of inconsistent BV formulae can always be bit-blasted (unfolded) into a pair of inconsistent propositional formulae, it is not always obvious how the resulting propositional interpolant can be folded back into a compact bit-vector (BV) formula to derive a BV interpolant. Interpolation engines over linear integer arithmetic (LIA) have thus been repurposed for BV interpolation [16]. First, operations on bit-vectors are reformulated as LIA formulae. An interpolant over LIA is then reinterpreted as a candidate interpolant for a pair of BV formulae. Because of wrap-around, LIA does not necessarily align with BV arithmetic, hence the LIA interpolant is adopted as a BV interpolant only if it passes a (unsatisfiability) check over bit-vectors. This checks that the interpolant relaxes the first BV formula of the pair and yet is still inconsistent with the second. If the candidate fails the check, then the two BV formulae are bit-blasted to recover a propositional interpolant, albeit one which looses the high-level structure of bit-vectors, and therefore is not compact. This approach is promising: it exploits robust off-the-shelf LIA interpolation [17] yet is compromised by the quality of the interpolants which follow from bit-blasting.

Contribution This paper plugs this gap, addressing the issue of interpolant quality by developing a new, principled encoding LIA formulae into BV formulae which does not enlarge the bit-width of the BV formulae. This ensures that the interpolant is still drawn from the language used to define BV formulae. We show that a naïve encoding of an LIA inequality as a BV inequality can give a formula which has a completely different meaning from LIA inequality: the BV inequality can have solutions not admitted by the LIA inequality and vice versa. Moreover, we illustrate how a straightforward encoding of a single LIA inequality can require many BV inequalities, which compromises the quality of a BV interpolant. We therefore propose a technique, which we call gapping, which adds range constraints to LIA inequality which reduces the LIA inequality into two or three LIA systems the solutions of which are amenable to compact BV representation. The term gapping reflects a geometric interpretation of this transformation which introduces a gap[4] between the solutions of the two LIA systems. We demonstrate the value of this approach with a BV interpolation engine which side-steps bit-blasting (and the complexity of providing bit-level

[4] The title of the paper alludes to both this geometric technique, the conceptual gap in previous work, and collaboration which entailed traveling through London.

circuits for arithmetic) and show that the approach usually gives a modest slow-down relative to LIA. We also prove the validity of the BV encoding, and the correctness of the reductions the encoding relies on, though the proofs themselves are omitted here for brevity. To summarise, the contributions of this paper are as follows:

– We show how interpolating theorem provers for LIA can be used to interpolate BV formulae, without recourse to bit-blasting.
– We develop a rigorous theory which explains gapping and proves that the resulting BV interpolant has exactly the same set of models as the LIA interpolant of the two BV formulae.
– We provide evaluation, within the established [6] framework of lazy abstraction with interpolants [25] which demonstrates the practicality of the approach for BV interpolation.

Use case Since BV formulae are converted to LIA one might wonder why one cannot work with LIA throughout and avoid BV interpolation all together. First, such an approach would not fit with a layered approach to interpolation [17] where one uses one lightweight theory (eg. uninterpreted functors) and then, if necessary, a more complicated one (eg. LIA) to construct a BV interpolant. BV formulae provide a uniform way expressing interpolants, no matter how they are derived. Second, computing LIA interpolants is complex and it is not surprising that these engines contain subtle[5] bugs. Translating a LIA interpolant back into a BV formula enables interpolants to be validated using a BV solver [3, 7, 27], using the reference (BV) semantics of a program. Moreover, validation need make no assumption on the correctness of a translation between theories. Validation can be performed on-the-fly, as the unwinding tree [25] is constructed, or by translating the complete, stable unwinding tree into its BV counterpart. The BV version can then be validated as a form of post-processing, akin to post-fixpoint validation in abstract interpretation [4, 14].

Road map This paper is structured as follows: Section 2 gives the intuition behind boxing and gapping whereas Section 3 argues for the correctness of the approach. Section 4 presents the experimental work. Section 5 presents the related work and section 6 concludes.

2 Boxing and Gapping in Pictures

Given a linear inequality ℓ, we seek to find a bit-vector formula f such that $[\![f]\!]_{\mathsf{BV}} = [\![\ell]\!]_{\mathsf{LIA}}$ where $[\![f]\!]_{\mathsf{BV}}$ and $[\![\ell]\!]_{\mathsf{LIA}}$ are respectively the sets of solutions (models) of f and ℓ in the linear integer arithmetic (LIA) and bit-vector (BV) semantics. Ideally f should be compact where we measure size by the number of binary logical connectives in f. This section gives the intuition behind two

[5] We refrain from mentioning specific solvers because we do not want to embarrass any particular research team to whom we are grateful.

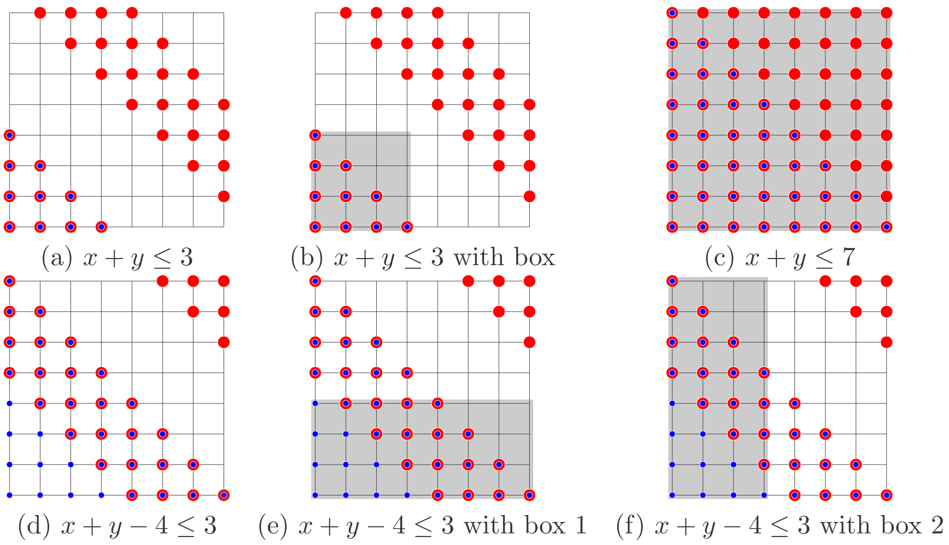


Fig. 1. Gapping and boxing for $x + y \leq 3$ and $x + y \leq 7$

techniques, boxing and gapping, and demonstrate how they are used together to construct such an f; the sequel provides a more formal development.

To illustrate boxing and gapping, first consider the set of solutions to the inequality $x + y \leq 3$, when interpreted with both the LIA semantics and BV semantics. Figure 1(a) gives the LIA solutions in blue and the BV solutions in red over the non-negative integer grid $\{(x, y) \mid 0 \leq x < 8 \wedge 0 \leq y < 8\}$ using a modulo of 8 for bit-vectors. The solution sets differ on, for instance, $(5, 6)$ since $(5 + 6) \pmod 8 = 3 \leq 3$ but $5 + 6 = 11 \not\leq 3$. It does not generally follow that $[\![f]\!]_{\mathsf{LIA}} \subseteq [\![f]\!]_{\mathsf{BV}}$ as Figure 1(d) illustrates for $f = x + y - 4 \leq 3$. Then $(1, 2) \in [\![x + y - 4 \leq 3]\!]_{\mathsf{LIA}}$ since $1+2-4 = -1 \leq 3$ but $(1, 2) \notin [\![x + y - 4 \leq 3]\!]_{\mathsf{BV}}$ since $(1 + 2 - 4) \pmod 8 = 7 \not\leq 3$.

Enumeration A naive approach to finding a formula f such that $[\![f]\!]_{\mathsf{BV}} = [\![\ell]\!]_{\mathsf{LIA}}$ is to enumerate all solutions of $[\![\ell]\!]_{\mathsf{LIA}}$ to then summarise them in a single BV formula. Figure 1(a) illustrates the $4 + 3 + 2 + 1 = 10$ LIA solutions for $\ell = (x + y \leq 3)$ which are summarised in the following BV formula:

$$f_1 = (x = 0 \wedge y = 0) \vee \ldots \vee (x = 0 \wedge y = 3) \vee \ldots \vee (x = 3 \wedge y = 0)$$

This formula has 9 binary disjuncts and 10 binary conjuncts, hence 19 logical connectives in total. A more compact formulation is to cover the blue triangular region of Figure 1(a) with columns as realised with the following BV formula:

$$f_2 = (x = 0 \wedge y \leq 3) \vee \ldots \vee (x = 3 \wedge y \leq 0)$$

Only non-negative solutions on the grid are considered so there is no need to additionally assert $0 \leq y$. This formula has 3 binary disjuncts and 4 binary conjuncts giving and 7 connectives in total.

Boxing Observe from Figure 1(a) that the extra solutions of $[\![x+y\leq 3]\!]_{\mathsf{BV}}$ over $[\![x+y\leq 3]\!]_{\mathsf{LIA}}$ stem from overflow. Overflow can be avoided by constraining BV solutions with $x\leq 3$ and $y\leq 3$ which amounts to placing a box (in general a hyper-rectangle) around the LIA solutions, as illustrated in Figure 1(b). This tactic, henceforth called boxing, leads to the following formula:

$$f_3 = (x+y\leq 3\wedge x\leq 3\wedge y\leq 3)$$

which requires 2 binary conjuncts.

Gapping Figure 1(c) illustrates that in general boxing cannot be applied in isolation because a box around the LIA solutions would not eliminate any extraneous BV solutions. Boxing is successful for Figure 1(b) because of the absence of solutions (a gap) between the LIA solutions inside the box and the BV solutions outside the box. No such gap exists for the box of Figure 1(c). Yet boxing can still be applied by decomposing the inequality $x+y\leq 7$ into two inequalities both of which are amenable to boxing. The construction is based on $[\![x+y\leq 7]\!]_{\mathsf{LIA}} = [\![x+y\leq 3]\!]_{\mathsf{LIA}} \cup [\![4\leq x+y\wedge x+y\leq 7]\!]_{\mathsf{LIA}} = [\![x+y\leq 3]\!]_{\mathsf{LIA}} \cup [\![0\leq x+y-4\wedge x+y-4\leq 3]\!]_{\mathsf{LIA}}$. Recall that boxing alone allows the LIA solutions of $x+y\leq 3$ to be expressed as a BV formula of 2 binary connectives. Thus consider the compound formula $\ell' = (0\leq x+y-4\wedge x+y-4\leq 3)$ whose LIA solutions are illustrated in Figure 1(d). Observe that the BV solutions of ℓ' can be covered with two rectangles without including the extraneous 6 BV solutions in top right. Then $[\![\ell']\!]_{\mathsf{LIA}} = [\![x+y-4\leq 3\wedge(x\leq 3\vee y\leq 3)]\!]_{\mathsf{BV}}$ which leads to the complete formula

$$f_3 = (x+y\leq 3\wedge x\leq 3\wedge y\leq 3)\vee(x+y-4\leq 3\wedge(x\leq 3\vee y\leq 3))$$

such that $[\![f_3]\!]_{\mathsf{BV}} = [\![x+y\leq 7]\!]_{\mathsf{LIA}}$. This tactic of artificially introducing a gap, henceforth called gapping, is equally applicable for larger grids too. For instance, working over a modulo of 32 $[\![x+y\leq 31]\!]_{\mathsf{LIA}} = [\![f_4]\!]_{\mathsf{BV}}$ where

$$f_4 = (x+y\leq 15\wedge x\leq 15\wedge y\leq 15)\vee(x+y-16\leq 15\wedge(x\leq 15\vee y\leq 15))$$

3 Formal correctness of boxing and gapping

In what follows we consider LIA and BV formulae over an ordered set of variables $\{x_1,\ldots,x_d\}$ for some $d>1$. We consider bit-vectors of fixed width $w>1$ and interpret LIA and BV formulae over the product space $\mathbb{M}^d$ where $\mathbb{M} = \{0,1,2,\ldots,m-1\}$ and $m=2^w$ as follows:

Definition 1. Let $\boldsymbol{c},\boldsymbol{c'}\in\mathbb{Z}^d$ and $b,b'\in\mathbb{Z}$. If $\ell\equiv(\sum_{i=1}^d c_i x_i)+b\leq(\sum_{i=1}^d c'_i x_i)+b'$ then

$$[\![\ell]\!]_{\mathsf{LIA}} = \left\{\boldsymbol{x}\in\mathbb{M}^d \,\middle|\, \textstyle\sum_{i=1}^d c_i x_i + b \leq \sum_{i=1}^d c'_i x_i + b'\right\}$$

$$[\![\ell]\!]_{\mathsf{BV}} = \left\{\boldsymbol{x}\in\mathbb{M}^d \,\middle|\, (\textstyle\sum_{i=1}^d c_i x_i + b) \bmod m \leq (\sum_{i=1}^d c'_i x_i + b') \bmod m\right\}$$

Furthermore, the LIA semantics can be lifted from inequalities to LIA formulae by: $[\![f_1 \vee f_2]\!]_{\mathsf{LIA}} = [\![f_1]\!]_{\mathsf{LIA}} \cup [\![f_2]\!]_{\mathsf{LIA}}$, $[\![f_1 \wedge f_2]\!]_{\mathsf{LIA}} = [\![f_1]\!]_{\mathsf{LIA}} \cap [\![f_2]\!]_{\mathsf{LIA}}$ and $[\![\neg f]\!]_{\mathsf{LIA}} = \mathbb{M}^d \setminus [\![f]\!]_{\mathsf{LIA}}$. Likewise for BV formulae.

In the sequel, $\mathbb{N}$ denotes the set of (strictly) positive integers, $\mathbb{R}$ the set of real numbers, and $\mathbb{R}_{\geq 0}$ the set of non-negative real numbers. We extend the floor and ceiling function for the sequences in $\mathbb{R}^d$ in a component-wise manner: $\lfloor \boldsymbol{x} \rfloor_i = \lfloor x_i \rfloor$ and $\lceil \boldsymbol{x} \rceil_i = \lceil x_i \rceil$. If $\boldsymbol{x} \in \mathbb{R}^d$ then $|\boldsymbol{x}| = d$. The partial order $\leq$ on $\mathbb{R}^d$ is defined by $\boldsymbol{x} \leq \boldsymbol{y}$ if and only if $x_i \leq y_i$ for all $i = 1, \ldots, d$.

3.1 Boxing

Boxing is founded on the following result and its corollary in which sets of solutions to inequalities which describe hyper-rectangles are pinched, above and below, by inclusions to systems of inequalities with positive, unary coefficients:

Lemma 1. *Let $d > 1$ and $L \in \mathbb{N}$. Then:*

$$\begin{aligned}
&\left\{\boldsymbol{x} \in \mathbb{R}^d_{\geq 0} \middle| \textstyle\sum_{i=1}^d x_i \leq L \cdot (m/2) - 1\right\} \\
\subseteq &\textstyle\bigcup_{\boldsymbol{p} \in I_d((d-1)(L+1))} \bigcap_{i=1}^d \left\{\boldsymbol{x} \in \mathbb{R}^d_{\geq 0} \mid x_i < \frac{p_i \cdot (m/2)}{d-1}\right\} \\
\subseteq &\left\{\boldsymbol{x} \in \mathbb{R}^d_{\geq 0} \middle| \textstyle\sum_{i=1}^d x_i < (L+1) \cdot (m/2)\right\}
\end{aligned}$$

where $I_d(n) = \big\{(i_1, \ldots, i_d) \in \mathbb{N}^d \mid i_1 + \cdots + i_d = n\big\}$.

Corollary 1. *Let $d > 1$, $L \in \mathbb{N}$ and $\boldsymbol{c} \in \mathbb{N}^d$. Then:*

$$\begin{aligned}
&\left\{\boldsymbol{x} \in \mathbb{Z}^d_{\geq 0} | \textstyle\sum_{i=1}^d c_i x_i \leq L \cdot (m/2) - 1\right\} \\
\subseteq &\textstyle\bigcup_{\boldsymbol{p} \in I_d((d-1)(L+1))} \bigcap_{j=1}^d \left\{\boldsymbol{x} \in \mathbb{Z}^d_{\geq 0} \mid x_j \leq \lceil \frac{p_j \cdot (m/2)}{c_i(d-1)} \rceil - 1\right\} \\
\subseteq &\left\{\boldsymbol{x} \in \mathbb{Z}^d_{\geq 0} \mid \textstyle\sum_{i=1}^d c_i x_i \leq (L+1) \cdot (m/2) - 1\right\}
\end{aligned}$$

The corollary leads to two types of box constraint: one for LIA and the other, reducing boxing, for BV. Boxing formulae are purely conceptual and are used to reason about correctness; reduced boxing formulae are deployed within BV interpolants.

Definition 2. Let $\boldsymbol{c} \in \mathbb{N}^d$, $b \in \mathbb{N}$ and $L \in \mathbb{N}$ be the unique natural number such that $(L-1) \cdot (m/2) \leq b \leq L \cdot (m/2) - 1$. The *boxing* and *reduced boxing* of $\sum_{i=1}^d c_i x_i \leq b$ are formulae defined as follows:

$$\mathrm{box}_{\mathsf{LIA}}(\boldsymbol{c}; b) \equiv \bigvee_{\boldsymbol{p} \in I_d((d-1)(L+1))} \bigwedge_{j=1}^d \left(x_j \leq \lceil \frac{p_j \cdot (m/2)}{c_j(d-1)} \rceil - 1\right) \quad (1)$$

$$\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; b) \equiv \bigvee_{\boldsymbol{p} \in I_d((d-1)(L+1))} \bigwedge_{j=1}^d \left(x_j \leq \min\left(\lceil \frac{p_j \cdot (m/2)}{c_j(d-1)} \rceil - 1, m-1\right)\right) \quad (2)$$

Given m and $b \in \mathbb{N}$, it is always possible to find a unique $L \in \mathbb{N}$ which satisfies Definition 2 by putting $L = \lfloor \frac{2b}{m} \rfloor + 1$. Then $L - 1 = \lfloor \frac{2b}{m} \rfloor \leq \frac{2b}{m} < \lfloor \frac{2b}{m} \rfloor + 1 = L$ hence $(L-1)(m/2) \leq b < L(m/2)$ whence $(L-1)(m/2) \leq b \leq L(m/2) - 1$ because b and $L(m/2)$ are integral.

One might expect that the cardinality of $I_d((d-1)(L+1))$ becomes large as d or L grow large. Yet d is the number of variables occurring in the LIA interpolant, which is typically small. Furthermore, when L is large, the values of p are also large, so that many terms become equivalent because of the min operation in equation (2) of Definition 2. Thus the number of terms required to define $\text{box}_{\mathsf{BV}}(\boldsymbol{c}; b)$ does not grow excessively large in practice.

The following proposition asserts that the boxing and reduced boxing formulae share the same solution set when interpreted with, respectively, the LIA and BV semantics.

Proposition 1. $[\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b)]\!]_{\mathsf{LIA}} = [\![\text{box}_{\mathsf{BV}}(\boldsymbol{c}; b)]\!]_{\mathsf{BV}}$

Example 1. To demonstrate this equivalence, consider again $x+y \leq 3$ for $m = 8$. Then put $L = \lfloor 6/8 \rfloor + 1 = 1$ and $I_2((d-1)(L+1)) = I_2(2) = \{\langle 1, 1\rangle\}$. Observe $\text{box}_{\mathsf{LIA}}(\langle 1, 1\rangle; 3) = \text{box}_{\mathsf{BV}}(\langle 1, 1\rangle; 3)$ since

$$\text{box}_{\mathsf{LIA}}(\langle 1, 1\rangle; 3) = (x \leq \lceil 4/1 \rceil - 1 = 3) \wedge (y \leq \lceil 4/1 \rceil - 1 = 3)$$

$$\text{box}_{\mathsf{BV}}(\langle 1, 1\rangle; 3) = (x \leq \min(3, 7) = 3) \wedge (y \leq \min(3, 7) = 3)$$

Example 2. Although $[\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b)]\!]_{\mathsf{LIA}} = [\![\text{box}_{\mathsf{BV}}(\boldsymbol{c}; b)]\!]_{\mathsf{BV}}$, it does not necessarily follow that $[\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b)]\!]_{\mathsf{LIA}} = [\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b)]\!]_{\mathsf{BV}}$. To illustrate, consider $x + y \leq 7$ for $d = 2$ and $m = 4$. Thus $\boldsymbol{c} = \langle 1, 1\rangle$ and $b = 7$. Then $L = \lfloor 14/4 \rfloor + 1 = 4$ and $I_2((d-1)(L+1)) = I_2(5) = \{\langle 1, 4\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle\}$ hence

$$\begin{aligned}\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b) = {} & (x \leq 1 \wedge y \leq 7) \vee (x \leq 3 \wedge y \leq 5) \vee \\ & (x \leq 5 \wedge y \leq 3) \vee (x \leq 7 \wedge y \leq 1)\end{aligned}$$

Therefore $[\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b)]\!]_{\mathsf{LIA}} = \mathbb{M}^2$ but $(2, 2) \notin [\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b)]\!]_{\mathsf{BV}}$.

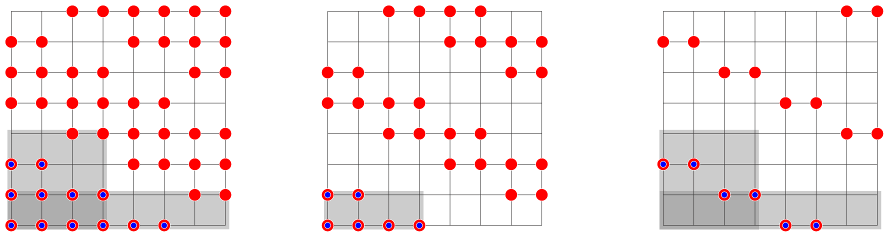
The following lemma shows that the solution sets for boxing grow monotonically as the constant of the inequality is relaxed.

Lemma 2. If $b \leq b'$ then $[\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b)]\!]_{\mathsf{LIA}} \subseteq [\![\text{box}_{\mathsf{LIA}}(\boldsymbol{c}; b')]\!]_{\mathsf{LIA}}$.

The following results explains how to augment an inequality with a box so as to align its BV semantics with its LIA semantics.

Theorem 1 (boxing without gapping). *Let $\boldsymbol{c} \in \mathbb{N}^d$ and $b \in \mathbb{N}$. If $b < m/2$ then*

$$[\![\sum_{i=1}^{d} c_i x_i \leq b]\!]_{\mathsf{LIA}} = [\![(\sum_{i=1}^{d} c_i x_i \leq b) \wedge \text{box}_{\mathsf{BV}}(\boldsymbol{c}; b)]\!]_{\mathsf{BV}}$$



(a) $x + 2y \leq 5$ with boxes (b) $x + 2y \leq 3$ with box (c) $0 \leq x + 2y - 4 \leq 1$ with boxes

Fig. 2. Gapping and boxing for $x + 2y \leq 5$

Observe that the result requires $b < m/2$. In this circumstance $L = \lfloor 2b/m \rfloor + 1 = 1$ and number of logical connectives in $\text{box}_{\textsf{BV}}(\boldsymbol{c}; b)$ is determined by the cardinality of the set $I_d((d-1)(L+1)) = I_d(2(d-1))$, which is given below:

d	$2(d-1)$	$I_d(2(d-1))$	$\lvert I_d(2(d-1))\rvert$
2	2	$\Pi(\langle 1,1\rangle)$	1
3	4	$\Pi(\langle 1,1,2\rangle)$	3
4	6	$\Pi(\langle 1,1,1,3\rangle) \cup \Pi(\langle 1,1,2,2\rangle)$	10
5	8	$\Pi(\langle 1,1,1,1,4\rangle) \cup \Pi(\langle 1,1,1,2,3\rangle) \cup \Pi(\langle 1,1,2,2,2\rangle)$	35

where $\Pi(v)$ denote the set of permutations of the vector v. For $d = 4$, $\text{box}_{\textsf{BV}}(\boldsymbol{c}; b)$ thus requires $10(d-1) = 30$ binary conjunctions and $10 - 1 = 9$ disjunctions.

3.2 Boxing and Gapping

Example 3. Consider $[\![x + 2y \leq 5]\!]_{\textsf{BV}}$ and $[\![x + 2y \leq 5]\!]_{\textsf{LIA}}$ for $m = 8$ as shown in Figure 2(a). Observe

$$\text{box}_{\textsf{BV}}(\langle 1,2\rangle; 5) = (x \leq 3 \wedge y \leq 3) \vee (x \leq 7 \wedge y \leq 1)$$

which is illustrated by the two grey rectangles. Hence $\langle 2,3\rangle \notin [\![x + 2y \leq 5]\!]_{\textsf{LIA}}$ but $\langle 2,3\rangle \in [\![x + 2y \leq 5 \wedge \text{box}_{\textsf{BV}}(\langle 1,2\rangle; 5)]\!]_{\textsf{BV}}$ therefore using boxing alone is not sufficient to encode the LIA inequality $x + 2y \leq 5$.

Example 4. Yet the LIA inequality $x + 2y \leq 5$ can be decomposed as follows:

$$\begin{aligned} [\![x + 2y \leq 5]\!]_{\textsf{LIA}} &= [\![x + 2y \leq 3]\!]_{\textsf{LIA}} \cup [\![4 \leq x + 2y \leq 5]\!]_{\textsf{LIA}} \\ &= [\![x + 2y \leq 3]\!]_{\textsf{LIA}} \cup [\![0 \leq x + 2y - 4 \leq 1]\!]_{\textsf{LIA}} \end{aligned}$$

Figures 2(b, c) illustrates boxing for $x + 2y \leq 3$ and $0 \leq x + 2y - 4 \leq 1$ where:

$$\begin{aligned} [\![x + 2y \leq 3]\!]_{\textsf{LIA}} &= [\![x + 2y \leq 3 \wedge \text{box}_{\textsf{BV}}(\langle 1,2\rangle; 3)]\!]_{\textsf{BV}} \\ &= [\![x + 2y \leq 3 \wedge (x \leq 3 \wedge y \leq 1)]\!]_{\textsf{BV}} \end{aligned}$$

Observe from Figure 2(c) that

$$\llbracket 0 \leq x + 2y - 4 \leq 1 \rrbracket_{\mathsf{LIA}} = \llbracket 0 \leq x + 2y - 4 \leq 1 \rrbracket_{\mathsf{BV}} \cap \llbracket \mathrm{box}_{\mathsf{BV}}(\langle 1, 2\rangle; 5) \rrbracket_{\mathsf{BV}}$$

and moreover $0 \bmod 8 = 0 \leq (x + 2y - 4) \bmod 8$ for all $(x, y) \in \mathbb{M}^2$ thus

$$\llbracket 0 \leq x + 2y - 4 \leq 1 \rrbracket_{\mathsf{LIA}} = \llbracket x + 2y - 4 \leq 1 \wedge \mathrm{box}_{\mathsf{BV}}(\langle 1, 2\rangle; 5) \rrbracket_{\mathsf{BV}}$$

therefore cumulatively $\llbracket x + 2y \leq 5 \rrbracket_{\mathsf{LIA}} = \llbracket \varphi_1 \vee \varphi_2 \rrbracket_{\mathsf{BV}}$ where

$$\begin{aligned} \varphi_1 &= [x + 2y \leq 3 \qquad\quad \wedge (x \leq 3 \wedge y \leq 1)] \\ \varphi_2 &= [x + 2y - 4 \leq 1 \wedge ((x \leq 3 \wedge y \leq 3) \vee (x \leq 7 \wedge y \leq 1))] \end{aligned}$$

The general rule of the separation of the given inequality and the boxing is shown in this theorem:

Theorem 2 (boxing with gapping). *Let* $\boldsymbol{c} \in \mathbb{N}^d$ *and* $b \in \mathbb{N}$. $\llbracket \sum_{i=1}^{d} c_i x_i \leq b \rrbracket_{\mathsf{LIA}} = \llbracket \phi_0 \vee \phi_1 \vee \phi_2 \rrbracket_{\mathsf{BV}}$ *where* $S = \lfloor b/(m/2) \rfloor$ *and*

$$\begin{aligned} \phi_0 &\equiv \left(\textstyle\sum_{i=1}^{d} c_i x_i - (S-2)(m/2) \leq m/2 - 1 \right) \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; (S-1)(m/2) - 1) \\ \phi_1 &\equiv \left(\textstyle\sum_{i=1}^{d} c_i x_i - (S-1)(m/2) \leq m/2 - 1 \right) \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; S(m/2) - 1) \\ \phi_2 &\equiv \left(\textstyle\sum_{i=1}^{d} c_i x_i - S(m/2) \leq b \bmod (m/2) \right) \;\; \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; b) \end{aligned}$$

Corollary 2 (boxing and gapping with simplification). *If* $\lfloor b/(m/2) \rfloor = 1$ *or* $b \bmod m = m/2 - 1$ *then* $\llbracket \sum_{i=1}^{d} c_i x_i \leq b \rrbracket_{\mathsf{LIA}} = \llbracket \phi_1 \vee \phi_2 \rrbracket_{\mathsf{BV}}$.

Example 5. Let $m = 8$ and consider again $x + 2y \leq 5$ so that $\boldsymbol{c} = \langle 1, 2\rangle$. Then $S = \lfloor 5/4 \rfloor = 1$ and, applying corollary 2, $\llbracket x + 2y \leq 5 \rrbracket_{\mathsf{LIA}} = \llbracket \phi_1 \vee \phi_2 \rrbracket_{\mathsf{BV}}$ where

$$\begin{aligned} \phi_1 &\equiv (x + 2y - 0 \cdot 4 \leq 4 - 1) \qquad \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 1 \cdot 4 - 1) = \varphi_1 \\ \phi_2 &\equiv (x + 2y - 1 \cdot 4 \leq 5 \bmod 4) \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 5) \qquad\quad = \varphi_2 \end{aligned}$$

aligning with the intuition given in example 4.

Example 6. Figure 3 illustrates Theorem 2 for $7x + 3y \leq 17$ and $m = 8$. Then $S = \lfloor 17/(8/2) \rfloor = 4$ and $\llbracket 7x + 3y \leq 17 \rrbracket_{\mathsf{LIA}} = \llbracket \phi_0 \vee \phi_1 \vee \phi_2 \rrbracket_{\mathsf{BV}}$ where

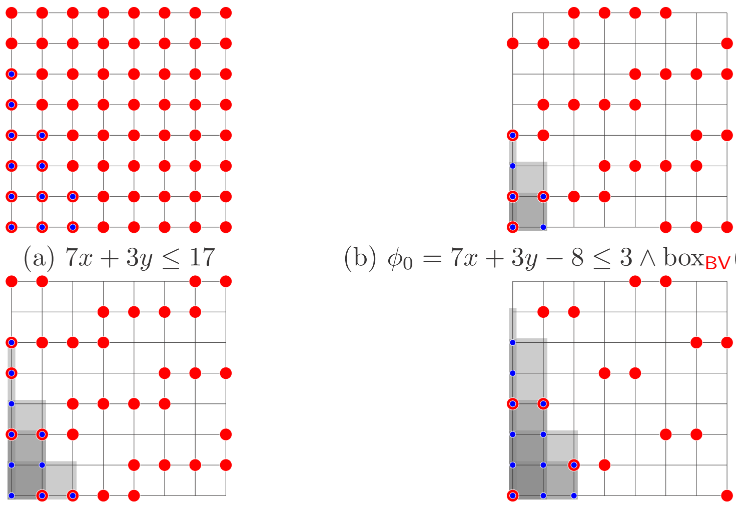
$$\begin{aligned} \phi_0 &= 7x + 3y - \;\, 8 \leq 3 \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 11) \\ \phi_1 &= 7x + 3y - 12 \leq 3 \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 15) \\ \phi_2 &= 7x + 3y - 16 \leq 1 \wedge \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 17) \end{aligned}$$

The $\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 11), \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}, 15), \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 17)$ formulae are again depicted in grey. For example,

$$\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 11) = (x \leq 0 \wedge y \leq 3) \vee (x \leq 1 \wedge y \leq 2) \vee (x \leq 1 \wedge y \leq 1)$$

because $d = 2$, $L = 3$ and $I_2((d-1)(L+1)) = \{\langle 1, 3\rangle, \langle 2, 2\rangle, \langle 3, 1\rangle\}$. From Figure 3 observe $\llbracket 7x + 3y \leq 17 \rrbracket_{\mathsf{LIA}} = \llbracket \phi_0 \rrbracket_{\mathsf{BV}} \cup \llbracket \phi_1 \rrbracket_{\mathsf{BV}} \cup \llbracket \phi_2 \rrbracket_{\mathsf{BV}}$.

Example 7. Consider again example 5 where $S = 1$. Then $\phi_0 = \mathit{false}$ because $\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; (S-1)(m/2) - 1) = \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; -1) = \mathit{false}$. This is because $L = 0$ and $I_d((d-1)(L+1)) = I_2(1) = \emptyset$. Theorem 2 then gives $\llbracket x + 2y \leq 5 \rrbracket_{\mathsf{LIA}} = \llbracket \phi_1 \vee \phi_2 \rrbracket_{\mathsf{BV}}$ which squares with Corollary 2.



(a) $7x + 3y \leq 17$ (b) $\phi_0 = 7x + 3y - 8 \leq 3 \wedge \text{box}_{\text{BV}}(\boldsymbol{c}; 11)$

(c) $\phi_1 = 7x + 3y - 12 \leq 3 \wedge \text{box}_{\text{BV}}(\boldsymbol{c}; 15)$ (d) $\phi_2 = 7x + 3y - 16 \leq 1 \wedge \text{box}_{\text{BV}}(\boldsymbol{c}; 17)$

Fig. 3. Gapping and boxing for $7x + 3y \leq 17$ where $\boldsymbol{c} = \langle 7, 3 \rangle$, $m = 8$ and $S = 4$

3.3 Boxing, Gapping and Flipping

To handle inequalities which have indeterminates with negative coefficients, boxing and gapping are augmented with a third technique, which we have informally named flipping. Flipping transforms an inequality into a syntactic form which is amenable to boxing and gapping by reflecting the solutions of the inequality. To detail the transformation, we assume without loss of generality, that an inequality takes the syntactic form $\boldsymbol{c}^+ \cdot \boldsymbol{x}^+ + \boldsymbol{c}^- \cdot \boldsymbol{x}^- \leq b$ where $\boldsymbol{c}^+ > \boldsymbol{0}$ and $\boldsymbol{c}^- < \boldsymbol{0}$. Hence $\boldsymbol{x} = \boldsymbol{x}^+ \circ \boldsymbol{x}^-$ where $\circ$ denotes vector concatenation. The act of flipping reflects the solutions of the inequality simultaneously around the axes $x_1^- = 0, \ldots, x_e^- = 0$ where $\boldsymbol{x}^- = \langle x_1^-, \ldots, x_e^- \rangle$ and e is the dimension of x^-. The development starts with the flipping transformation itself:

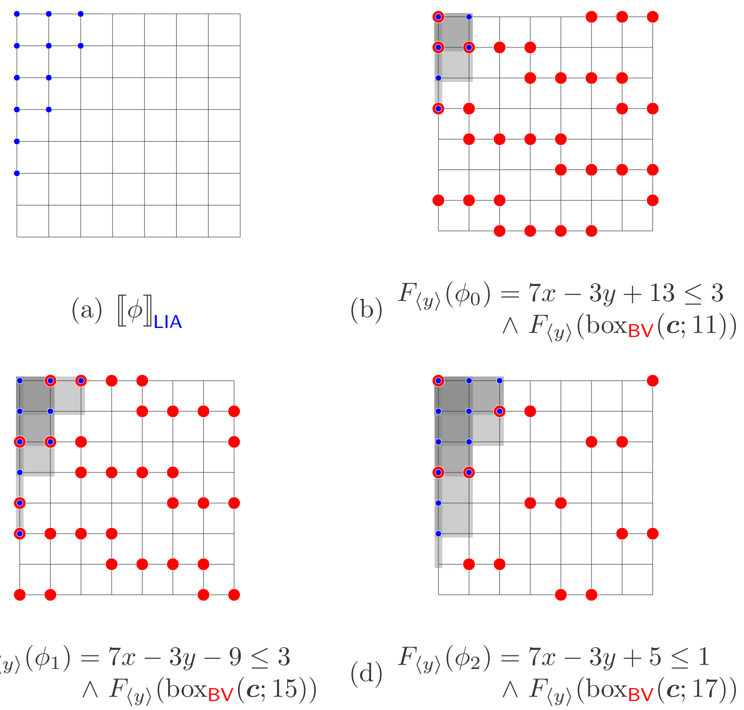
Definition 3. Given $e \in \{1, \ldots, d\}$, then the (semantic) flipping function $F_e : \mathbb{M}^d \to \mathbb{M}^d$ is defined:

$$F_e(\langle x_1^+, \ldots, x_{d-e}^+, x_1^-, \ldots, x_e^- \rangle) = \langle x_1^+, \ldots, x_{d-e}^+, m - 1 - x_1^-, \ldots, m - 1 - x_e^- \rangle.$$

Given an inequality with negative coefficients, we derive a new inequality whose solutions coincide with the flipped solutions of the given inequality. This transformation is then lifted to formulae as follows:

Definition 4. Given a partition of $\boldsymbol{x}$ into the sub-vectors $\boldsymbol{x}^+ = \langle x_1^+, \ldots, x_{d-e}^+ \rangle$ and $\boldsymbol{x}^- = \langle x_1^-, \ldots, x_e^- \rangle$, then the (syntactic) flipping function $F_{\boldsymbol{x}^-}$ is defined:

$$\begin{aligned} F_{\boldsymbol{x}^-}(\boldsymbol{c}^+ \cdot \boldsymbol{x}^+ + \boldsymbol{c}^- \cdot \boldsymbol{x}^- \leq b) &= \boldsymbol{c}^+ \cdot \boldsymbol{x}^+ - \boldsymbol{c}^- \cdot \boldsymbol{x}^- + (m-1)(\boldsymbol{c}^- \cdot \boldsymbol{1}) \leq b \\ F_{\boldsymbol{x}^-}(f_1 \vee f_2) &= F_{\boldsymbol{x}^-}(f_1) \vee F_{\boldsymbol{x}^-}(f_2) \\ F_{\boldsymbol{x}^-}(f_1 \wedge f_2) &= F_{\boldsymbol{x}^-}(f_1) \wedge F_{\boldsymbol{x}^-}(f_2) \\ F_{\boldsymbol{x}^-}(\neg f) &= \neg F_{\boldsymbol{x}^-}(f) \end{aligned}$$


(a) $[\![\phi]\!]_{\mathsf{LIA}}$ (b) $F_{\langle y \rangle}(\phi_0) = 7x - 3y + 13 \leq 3 \wedge F_{\langle y \rangle}(\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 11))$

(c) $F_{\langle y \rangle}(\phi_1) = 7x - 3y - 9 \leq 3 \wedge F_{\langle y \rangle}(\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 15))$ (d) $F_{\langle y \rangle}(\phi_2) = 7x - 3y + 5 \leq 1 \wedge F_{\langle y \rangle}(\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 17))$

Fig. 4. Flipping $\phi = 7x - 3y \leq -4$ where $m = 8$, $\boldsymbol{x} = \langle x, y \rangle$, $\boldsymbol{x}^+ = \langle x \rangle$ and $\boldsymbol{x}^- = \langle y \rangle$

The overall strategy involves applying boxing and gapping to an inequality derived by the flipping function $F_{\boldsymbol{x}^-}$. The validity of this strategy is based on the following proposition:

Proposition 2. If $|\boldsymbol{x}^-| = e$ then

- $[\![F_{\boldsymbol{x}^-}(f)]\!]_{\mathsf{LIA}} = F_e([\![f]\!]_{\mathsf{LIA}})$
- $[\![F_{\boldsymbol{x}^-}(f)]\!]_{\mathsf{BV}} = F_e([\![f]\!]_{\mathsf{BV}})$

A complete strategy for handling inequalities with negative coefficients is justified by the following corollary. The strategy entails flipping an LIA inequality, deriving a BV formula by boxing and gapping, and then flipping the BV formula.

Corollary 3. Suppose $\boldsymbol{c}^+ > \mathbf{0}$, $\boldsymbol{c}^- < \mathbf{0}$ and

$$[\![\boldsymbol{c}^+ \cdot \boldsymbol{x}^+ - \boldsymbol{c}^- \cdot \boldsymbol{x}^- \leq b + (1 - m)(\boldsymbol{c}^- \cdot \mathbf{1})]\!]_{\mathsf{LIA}} = [\![\phi_0 \vee \phi_1 \vee \phi_2]\!]_{\mathsf{BV}}$$

Then $[\![\boldsymbol{c}^+ \cdot \boldsymbol{x}^+ + \boldsymbol{c}^- \cdot \boldsymbol{x}^- \leq b]\!]_{\mathsf{LIA}} = [\![F_{\boldsymbol{x}^-}(\phi_0) \vee F_{\boldsymbol{x}^-}(\phi_1) \vee F_{\boldsymbol{x}^-}(\phi_2)]\!]_{\mathsf{BV}}$

Example 8. Consider $\phi = 7x - 3y \leq -4$ which is illustrated in Fig. 4(a). Then $\boldsymbol{x}^+ = \langle x \rangle$, $\boldsymbol{x}^- = \langle y \rangle$ and $F_{\boldsymbol{x}^-}(\phi) = F_{\langle y \rangle}(\phi) = 7x + 3y - 21 \leq -4$. Fig. 3(a) shows $[\![7x+3y-21 \leq -4]\!]_{\mathsf{LIA}} = [\![7x+3y \leq 17]\!]_{\mathsf{LIA}}$ and so building on example 6 $[\![7x+3y \leq 17]\!]_{\mathsf{LIA}} = [\![\phi_0 \vee \phi_1 \vee \phi_2]\!]_{\mathsf{BV}}$. By corollary 3 it follows $[\![\phi]\!]_{\mathsf{LIA}} = [\![F_{\langle y \rangle}(\phi_0) \vee F_{\langle y \rangle}(\phi_1) \vee F_{\langle y \rangle}(\phi_2)]\!]_{\mathsf{BV}}$ where $F_{\langle y \rangle}(\phi_0)$, $F_{\langle y \rangle}(\phi_1)$ and $F_{\langle y \rangle}(\phi_2)$ are given in Fig. 4(b), (c) and (d) respectively. Finally, to illustrate the handling of boxing, recall $\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 11)$ from example 6 and

$$\begin{aligned} \mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 11) = {} & (x \leq 0 \wedge y \leq 3) \\ \vee {} & (x \leq 1 \wedge y \leq 2) \\ \vee {} & (x \leq 1 \wedge y \leq 1) \end{aligned} \qquad \begin{aligned} F_{\langle y \rangle}(\mathrm{box}_{\mathsf{BV}}(\boldsymbol{c}; 11)) = {} & (x \leq 0 \wedge (-y+7 \leq 3)) \\ \vee {} & (x \leq 1 \wedge (-y+7 \leq 2)) \\ \vee {} & (x \leq 1 \wedge (-y+7 \leq 1)) \end{aligned}$$

Finally observe

$$\begin{aligned} [\![x \leq 0 \wedge (-y+7 \leq 3)]\!]_{\mathsf{LIA}} &= \{(0,y) \in \mathbb{M}^2 \mid 4 \leq y \leq 7\} \\ [\![x \leq 1 \wedge (-y+7 \leq 2)]\!]_{\mathsf{LIA}} &= \{(x,y) \in \mathbb{M}^2 \mid 0 \leq x \leq 1 \wedge 5 \leq y \leq 7\} \end{aligned}$$

and that the disjunct $(x \leq 1 \wedge (-y+7 \leq 1))$ is actually redundant.

3.4 Boxing, Gapping, Flipping and Demoding

Griggio [16] gives a procedure for encoding machine arithmetic in LIA, illustrating that the resulting LIA interpolants can include inequalities such as $-x_2+x_3-256\lfloor -x_2/256 \rfloor \leq 255$ [16, Example 5]. Relaxing inequalities to include ceiling (or floor) functions can reduce the size of interpolants whilst simplifying their derivation [17]. These more general forms of interpolant include inequalities of the form $\boldsymbol{c} \cdot \boldsymbol{x} + n' \lfloor \boldsymbol{c}' \cdot \boldsymbol{x}/n \rfloor \leq b$ [9] or $\boldsymbol{c} \cdot \boldsymbol{x} + n' \lceil \boldsymbol{c}' \cdot \boldsymbol{x}/n \rceil \leq b$ [17], though for our purposes it is sufficient to consider $\boldsymbol{c} \cdot \boldsymbol{x} + n'2^n \lfloor \boldsymbol{c}' \cdot \boldsymbol{x}/2^n \rfloor \leq b$ or $\boldsymbol{c} \cdot \boldsymbol{x} + n'2^n \lceil \boldsymbol{c}' \cdot \boldsymbol{x}/2^n \rceil \leq b$, where the divisors are powers of 2, stemming from the way they model wrap-around in machine arithmetic. To extend boxing to these generalised interpolants we extend the LIA and BV semantics two new types of atomic constraint (though the definitions are almost vacuous):

Definition 5. *If* $\ell \equiv \boldsymbol{c} \cdot \boldsymbol{x} + n' \lfloor \boldsymbol{c}' \cdot \boldsymbol{x}/2^n \rfloor \leq b$ *then*

$$\begin{aligned} [\![\ell]\!]_{\mathsf{LIA}} &= \left\{ \boldsymbol{x} \in \mathbb{M}^d \middle| \boldsymbol{c} \cdot \boldsymbol{x} + n' \lfloor \boldsymbol{c}' \cdot \boldsymbol{x}/2^n \rfloor \leq b \right\} \\ [\![\ell]\!]_{\mathsf{BV}} &= \left\{ \boldsymbol{x} \in \mathbb{M}^d \middle| (\boldsymbol{c} \cdot \boldsymbol{x} + n' \lfloor \boldsymbol{c}' \cdot \boldsymbol{x}/2^n \rfloor) \bmod m \leq b \bmod m \right\} \end{aligned}$$

The following proposition shows generalised LIA interpolants are not an obstacle to boxing. These inequalities are handled through a transformation scheme which exploits the property that if $n \leq w$ then $(\boldsymbol{c} \cdot \boldsymbol{x} \bmod 2^n) \bmod m = \boldsymbol{c} \cdot \boldsymbol{x} \bmod 2^n$. We informally call this transformation tactic demoding, because like gapping and flipping, it is designed to increase the general applicability of boxing.

Proposition 3. Suppose $0 \leq n \leq w$ and $[\![(\boldsymbol{c} + n'\boldsymbol{c}') \cdot \boldsymbol{x} - n'y \leq b]\!]_{\mathsf{LIA}} = [\![\phi]\!]_{\mathsf{BV}}$. If y does not occur in $\boldsymbol{x}$ then

$$[\![\boldsymbol{c} \cdot \boldsymbol{x} + n'2^n \lfloor \boldsymbol{c}' \cdot \boldsymbol{x}/2^n \rfloor \leq b]\!]_{\mathsf{LIA}} = [\![\phi[y \mapsto \boldsymbol{c}' \cdot \boldsymbol{x} \bmod 2^n]]\!]_{\mathsf{BV}}$$

Inequalities such as $\boldsymbol{c} \cdot \boldsymbol{x} + n' 2^n \lceil \boldsymbol{c}' \cdot \boldsymbol{x} / 2^n \rceil \leq b$ can be handled similarly. For completeness, we note that expansion can be applied for non-powers of 2:

Proposition 4. Suppose $n > 0$. Then

$$[\![\boldsymbol{c} \cdot \boldsymbol{x} + n' \lfloor \boldsymbol{c}' \cdot \boldsymbol{x} / n \rfloor \leq b]\!]_{\mathsf{LIA}} = [\![\bigvee_{i=\ell}^{u} (\boldsymbol{c} \cdot \boldsymbol{x} \leq b - n'i \wedge ni \leq \boldsymbol{c}' \cdot \boldsymbol{x} \leq ni - 1)]\!]_{\mathsf{LIA}}$$

where $\ell = \min\{\lfloor \boldsymbol{c}' \cdot \boldsymbol{x} / n \rfloor \mid \boldsymbol{x} \in \mathbb{M}^d\}$ and $u = \max\{\lfloor \boldsymbol{c}' \cdot \boldsymbol{x} / n \rfloor \mid \boldsymbol{x} \in \mathbb{M}^d\}$.

4 Experiments

To evaluate the performance of boxing we implemented a model checker based on the lazy abstraction (IMPACT) [25] algorithm. The model checker is implemented in Python 3.7.2 and uses MathSAT5 [9] for satisfiability checking and interpolation over LIA. The model checker parses a subset of the C language, but is rich enough to handle 312 benchmarks drawn from [2, 12]. The model checker was instantiated in one of three ways to use: (1) LIA interpolation [17]; (2) BV interpolation by covering the solutions of an LIA interpolate with columns (recall f_2 of section 2); and (3) BV interpolation by covering the solutions of an LIA interpolate using boxing, gapping and flipping. Experiments were performed using an Amazon Web Service EC2 c3.xlarge cloud architecture of 14 EC2 Computing Units [31] each equipped with 4 cores and 7.5 GB of RAM. The timeout for each run of IMPACT was set to 600 seconds.

All arithmetic is idealised in configuration (1) taking no account of integer overflow and underflow. This is not, in general, safe. In configurations (2) and (3) the model checker interprets machine arithmetic and bit operations using the LIA encoding of BV operations outlined in [16, Fig 1]. This is safe but complicates the LIA formulae, often substantially. One would expect this to enlarge the interpolants, even before boxing and gapping are deployed. We would also expect (1) to be substantially faster than (2) and (3). Due to differences in the semantics of arithmetic, we might also see differences in the number of programs proved to be safe or found to be unsafe. The experiments quantify these predictions. To discuss the experiments, (2) will be referred to as the naive encoding, even though it improves on complete enumeration (recall f_1 of section 2).

4.1 Overall Result

Table 1 summarises the outcomes of running IMPACT on all 312 programs, using the three different instances of interpolation, categorised as to whether the run proved safety (safe rows) or found a counterexample (unsafe rows). The Solved column of the left-hand table gives the total of the programs there were either shown to be safe or unsafe within 600 seconds. Time is the mean execution of a run (for all those programs which did not timeout). Size is mean

Table 1. Comparison of the theories: performance and correctness

Theory	Safety	Solved	Time (seconds)	Size (inequalities)
LIA	safe	165	15.1	440
	unsafe	41	9.0	392
	(total)	206	13.9	431
BV (naive)	safe	87	30.1	32583
	unsafe	57	24.2	49138
	(total)	144	27.8	39136
BV (boxing)	safe	99	20.0	6938
	unsafe	66	20.1	15246
	(total)	165	20.0	10261

		LIA	
		safe	unsafe
BV	safe	90	1
	unsafe	17	34

total number of atomic constraints in all interpolants encountered over a run (for those programs which did not timeout). We observe that more programs can be analysed to completion with LIA than with BV, as one would expect, but that BV (boxing) improves on BV (naive), the speedup being significant when proving safety.

The right-hand table compares a terminating run of LIA to a terminating run of BV (boxing). For 17 of these 142 runs, LIA (incorrectly) verified the program to be safe whereas BV found a counter-example. Unexpectantly for trex03_true-unreach-call.i.annot.c from [12], LIA found a counter-example but BV verified safety. This program contains three integers, `x1`, `x2` and `x3`, which can become negative in the idealised arithmetic employed in LIA, triggering an assertion. But `x1`, `x2` and `x3` are actually unsigned.

4.2 Runtime for Naive encoding and Boxing

The scatter plot of Figure 5 compares the runtime of the naive encoding against that of boxing and its allied techniques of gapping and flipping. The scatter plot excludes timeouts and depicts 151 pairs of runs. Almost all points are under the dotted line, indicating the boxing significantly improves performance. The line graph plots the ratio of the execution times, from which we observe that boxing does not accelerate the verification for almost half of the runs, but does speed it up between 2- and 256-fold for the other half.

4.3 Interpolant Size for Naive encoding and Boxing

The line graph on Figure 6 compares the relative size of interpolants for boxing versus the naive encoding. Size is the sum of the sizes of all the interpolants generated during a run, where the size of an interpolant is itself defined as the number of atomic constraints that occur within it. We observe that for most problems the size ratio is around one, but a second peak occurs at 1/32, giving an overall size reduction. The scatter plot explores how interpolant size correlates with runtime, showing how the relative size of interpolants varies with relative runtimes. We observe that reducing the size of interpolants improves runtime, and that two peaks of the line graph manifesting themselves as two clusters of points in the scatter plot.

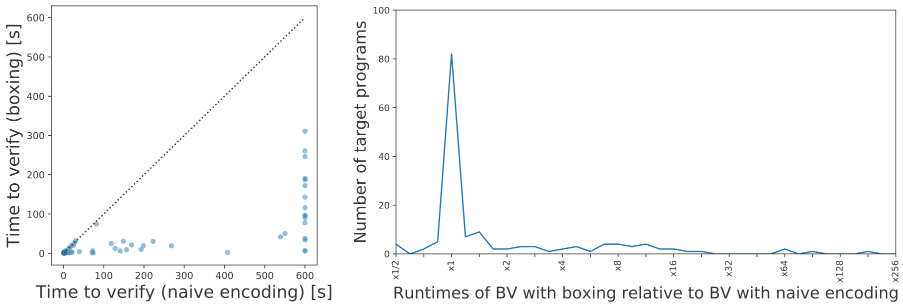


Fig. 5. Runtime of boxing versus naive: scatter plot and ratio plot

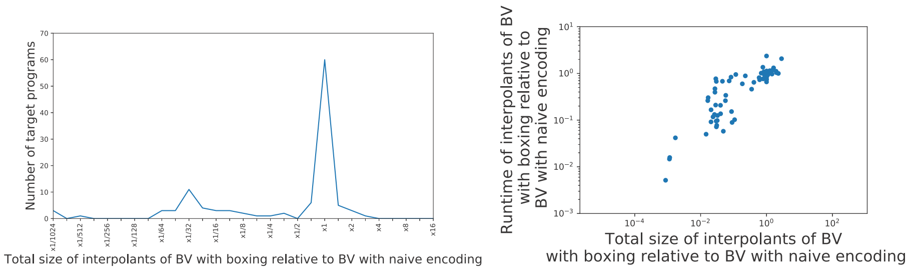


Fig. 6. Size of interpolants in boxing versus naive and its impact on performance

5 Related work

The problem of reasoning about machine arithmetic and wrapping arises not only in model checking, but abstract interpretation too, where solvers are augmented with support for relaxing abstractions by join rather than interpolation.

Despite the long-standing work [3, 7, 27] in deciding BV theories, there has been scant work on BV interpolation. Although not focussing on BV interpolation, an early work on deriving work-level interpolants [23] uses bit-vectors to interpolate equality logic. This logic supports equations of the form $x = y$ and $x = c$ where x and y are variables and c is drawn from a finite set of symbols C. Bit-vectors with width $\lceil \log_2(|C|) \rceil$ are used to bit-blast equations [29] so that formulae are encoded entirely propositionally. Then a propositional resolution proof of the inconsistence of two formulae is lifted to the work-level.

Seminal work by Griggio [16] advocated encoding BV formulae in theories of increasing complexity. The pair of BV formulae are encoded in a theory whose interpolation engine is used to find an interpolant in that theory. The interpolant is then reinterpreted as a BV formula and tested to see if it is still an interpolant the pair of BV formulae. The approach resorts to bit-blasting if no simpler theory can find an interpolant, at the cost of losing world-level information. By way of contrast, Backeman et al. [2] propose a calculus over a core language, which supports interpolation and is rich enough to describe BV formulae, even making use of Groebner bases to express polynomial equality relationships. Since interpolation is performed within their core language, they do not aim to derive a BV interpolant, and therefore their work is orthogonal to ours. Yet if Backeman's procedure returns an interpolant in their core language and it could be interpreted as an LIA formula, which would seem likely for many cases, then our work could convert the LIA formula back to BV.

Further afield, polynomial algorithms for interpolation have developed for systems of linear congruence equations [19, section 4], conjunctions of linear Diophantine equations and disequations [19, section 6], and systems of mixed integer linear equations [19, section 7]. This comprehensive study stops short of using LIA to interpolate BV formula, mentioning the problem as future work.

Abstract domains have been proposed for tracking linear modulo relationships where the module is a power of 2 [13, 22, 28]. These domains, which are essentially specialist solvers, express more than linear equalities [21], while enabling the domain operations to be realised using machine arithmetic. Surprisingly, systems of linear inequalities can be reinterpreted to model machine arithmetic by just changing the concretisation function [32] and the handling of guards [32].

6 Concluding Discussion

To repurpose efficient LIA interpolation engines to BV, we have shown how to systematically construct a BV formula so its solutions are exactly those of an LIA interpolant. Since an LIA interpolant summarises the reason for a conflict between two LIA formula, we seek to retain its compact structure by introducing no more than simple boxes around the LIA solutions which block extraneous BV solutions. When this encoding tactic, called boxing, is not applicable, gapping is used to decompose an LIA inequality into two or more inequalities which are amenable to boxing. We show how the size of the resulting BV interpolants are smaller than BV interpolants constructed by merely partitioning the LIA solutions into columns, and demonstrate how boxing and gapping improves the runtime of an interpolation-based model-checker. We instantiate a model-checker with LIA and BV to compare their performance, and conclude that with this encoding BV interpolation is feasible. Because of wrap-around, BV is substantially more complicated than LIA for interpolation, yet BV is no more than twice as slow as LIA for over half the benchmarks. Furthermore, the resulting BV interpolants can be validated, independent of LIA, just using a BV solver.

Acknowledgments We thank anonymous reviewers for their comments which helped us improve the paper. We thank Alberto Griggio for his help with MathSAT5 and looking into the intellectual property restrictions on sharing his extension to Kratos [8], a model checker based on lazy predicate abstraction. We also thank Chris Coppins with his help with IMPACT and Peter Backeman for tirelessly answering our e-mails. This work was funded, in part, by EPSRC EP/N020243/1 and by JST ERATO HASUO Metamathematics for Systems Design Project JPMJER1603.

References

1. Albarghouthi, A., McMillan, K.L.: Beautiful Interpolants. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 8044, pp. 313–329. Springer (2013)
2. Backeman, P., Rümmer, P., Zeljic, A.: Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction. In: Formal Methods in Computer Aided Design. pp. 1–10. IEEE (2018)
3. Barrett, C., Dill, D., Levitt, J.: A Decision Procedure for Bit-Vector Arithmetic. In: Design and Automation Conference. pp. 522–527 (1998)
4. Besson, F., Cornilleau, P.E., Jensen, T.: Result Certification of Static Program Analysers with Automated Theorem Provers. In: Verified Software: Theories, Tools, Experiments. Lecture Notes in Computer Science, vol. 8164, pp. 304–325. Springer (2014)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. International Journal on Software Tools for Technology Transfer **9**(5-6), 505–525 (2007)
6. Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. Impact. In: Formal Methods in Computer-Aided Design. pp. 106–113. IEEE (2012)
7. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding Bit-Vector Arithmetic with Abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 4424, pp. 358–372. Springer (2007)
8. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos – a Software Model Checker for SystemC. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 6808, pp. 123–136. Springer (2011)
9. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013)
10. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Theories. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 4963, pp. 397–412. Springer (2008)
11. Cimatti, A., Griggio, A., Sebastiani, R.: Interpolant Generation for UTVPI. In: CADE. pp. 167–182 (2009)
12. Demyanova, Y., Rümmer, P., Zuleger, F.: Systematic Predicate Abstraction Using Variable Roles. In: NASA Formal Methods. Lecture Notes in Computer Science, vol. 10227, pp. 265–281. Springer (2017)
13. Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T.: Abstract Domains of Affine Relations. ACM Transactions on Programming Languages and Systems **36** (2014)
14. Fouilhé, A., Monniaux, D., Périn, M.: Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 7935, pp. 345–365. Springer (2013)
15. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 1254, pp. 72–83. Springer (1997)
16. Griggio, A.: Effective Word-Level Interpolation for Software Verification. In: Formal Methods in Computer-Aided Design. pp. 28–36. IEEE (2011)

17. Griggio, A., Le, T.T.H., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic. Logical Methods in Computer Science **8**(3) (2010)
18. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Principles of Programming Languages. pp. 58–70 (2002)
19. Jain, H., Clarke, E.M., Grumberg, O.: Efficient Craig interpolation for linear Diophantine (dis)equations and linear modular equations. Formal Methods in System Design **35**(1), 6–39 (2009)
20. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for Data Structures. In: Foundations of Software Engineering. pp. 105–116 (2006)
21. Karr, M.: Affine Relationships among Variables of a Program. Acta Informatica **6**, 133–151 (1976)
22. King, A., Søndergaard, H.: Automatic Abstraction for Congruences. In: Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 9583, pp. 197–213. Springer (2010)
23. Kroening, D., Weissenbacher, G.: Lifting Propositional Interpolants to the Word-Level. In: Formal Methods in Computer-Aided Design. pp. 85–89. IEEE (2007)
24. McMillan, K.: An Interpolating Theorem Prover. Theoretical Computer Science **345**(1), 101–121 (2005)
25. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 4144, pp. 123–136. Springer (2006)
26. McMillan, K.L.: Interpolation and Model Checking. In: Handbook of Model Checking. pp. 421–446. Springer (2018)
27. Möller, M., Rue, H.: Solving Bit-Vector Equations. In: Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 1522, pp. 36–48 (1998)
28. Müller-Olm, M., Seidl, H.: Analysis of Modular Arithmetic. ACM Transactions on Programming Languages and Systems **29**(5), 29 (2007)
29. Pnueli, A., Rodeh, Y., Strichman, O., Siegel, M.: The Small Model Property: How small can it be? Information and Computation **178**(1), 279–293 (2002)
30. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. Journal of Symbolic Computation **45**(11), 1212–1233 (2010)
31. Services, A.W.: Amazon EC2 FAQs (2019), https://aws.amazon.com/ec2/faqs/
32. Simon, A., King, A.: Taming the Wrapping of Integer Arithmetic. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 4634, pp. 121–136. Springer (2007)

Automated and Sound Synthesis of Lyapunov Functions with SMT Solvers

Daniele Ahmed[1,2], Andrea Peruffo[1], and Alessandro Abate[1]

[1] Department of Computer Science, University of Oxford, OX1 3QD Oxford, UK
name.surname@cs.ox.ac.uk
[2] Amazon Inc, London, UK

Abstract. In this paper we employ SMT solvers to soundly synthesise Lyapunov functions that assert the stability of a given dynamical model. The search for a Lyapunov function is framed as the satisfiability of a second-order logical formula, asking whether there exists a function satisfying a desired specification (stability) for all possible initial conditions of the model. We synthesise Lyapunov functions for linear, non-linear (polynomial), and for parametric models. For non-linear models, the algorithm also determines a region of validity for the Lyapunov function. We exploit an inductive framework to synthesise Lyapunov functions, starting from parametric templates. The inductive framework comprises two elements: a *learner* proposes a Lyapunov function, and a *verifier* checks its validity - its lack is expressed via a counterexample (a point over the state space), for further use by the learner. Whilst the verifier uses the SMT solver Z3, thus ensuring the overall soundness of the procedure, we examine two alternatives for the learner: a numerical approach based on the optimisation tool Gurobi, and a sound approach based again on Z3. The overall technique is evaluated over a broad set of benchmarks, which shows that this methodology not only scales to 10-dimensional models within reasonable computational time, but also offers a novel soundness proof for the generated Lyapunov functions and their domains of validity.

Keywords: Lyapunov functions, automated synthesis, inductive synthesis, counter-example guided synthesis

1 Introduction

Dynamical systems represent a major modelling framework in both theoretical and applied sciences: they describe how objects move by means of the laws governing their dynamics in time. Often they encompass a system of ordinary differential equations (ODE) with nontrivial solutions.

This work aims at studying the stability property of general ODEs, without knowledge of their analytical solution. Stability analysis via Lyapunov functions is a known approach to assert such property. As such, the problem of constructing relevant Lyapunov functions for stability analysis has drawn much attention in the literature [1,2]. A brief introduction to the concepts of Lyapunov stability is presented in Section 3. By and large, existing approaches leverage Linear Algebra or Convex Optimisation solutions, and are not fully automated nor numerically sound.

Contributions We apply an inductive synthesis framework, known as Counter-Example Guided Inductive Synthesis (CEGIS) [3,4] and recently employed in a number of control applications [5,6,7,8], to construct Lyapunov functions for linear, polynomial and parametric ODEs, and (for non-linear ODEs) to constructively characterise their domain of validity. CEGIS, originally developed for program synthesis based on the satisfiability of second-order logical formulae, is employed in this work with template Lyapunov functions and in conjunction with a Satisfiability Modulo Theory (SMT) solver [9]. Our results offer a formal guarantee of correctness in combination with a simple algorithmic implementation.

The synthesis of a Lyapunov function V can be written as a second-order logic formula $F := \exists V \; \forall x : \psi$, where x represents the state variables and ψ represents requirements that V needs to satisfy in order to be a Lyapunov function.

The CEGIS architecture is structured as a loop between two components, a "learner" and a "verifier". The learner provides a candidate function V and the verifier checks the validity of ψ over the set of x; if the function is not valid, the verifier provides a counterexample, namely a point $\bar{x}$ in the state space where the candidate function does not satisfy ψ. The learner incorporates the generated counterexample $\bar{x}$, subsequently computes a new candidate function, and passes it back to the verifier.

We exploit SMT solvers to (repeatedly) assert the validity of ψ, given V, over a domain in the space of x. Satisfiability Modulo Theory (SMT) is a powerful tool to assert the existence of such a function. An SMT problem is a decision problem – a problem that can be formulated as a yes/no question – for logical formulae within one or more theories, e.g. the theory of arithmetics over real numbers. The generation of simple counterexamples $\bar{x}$ is a key new feature of our technique.

Furthermore, in this work we provide two alternative CEGIS implementations: 1) a numerical learner and an SMT-based verifier, and 2) an SMT-based learner and verifier. The numerical generation of Lyapunov functions is based on the optimisation tool Gurobi [10], whereas the SMT-based one leverages Z3 [11].

Related Work The construction of Lyapunov functions is recognisably an important yet hard problem, particularly for non-linear ODE models, and it has been the objective of classical studies [12,13,14]. A know constructive result has been introduced in [15], which additionally provides an estimate of the domain of attraction. It has led to further work based on recursive procedures. Broadly, these approaches are numerical and based on the solution of optimisation problems. For instance, linear programming is exploited in [16] to iteratively search for stable matrices inside a predefined convex set, resulting in an approximate Lyapunov function for the given model. Alternative approximate methods include [1] ε-bounded numerical methods, techniques leveraging series expansion of a function, the construction of functions from trajectory samples, and the framework of linear matrix inequalities. The approach in [17] uses sum-of-squares (SOS) polynomials to synthesise Lyapunov functions, however its scalability remains an issue. The work in [18] uses SOS decomposition to synthesise Lyapunov functions for (non-polynomial) non-linear systems: the algorithmic implementation is know as SOSTOOLS [19,20]. [21] focuses on an analytical result involving a summation over finite time interval, under a stability assumption. Recent developments are in [22] and subsequent work, whereas surveys on this topic are in [1,2].

In conclusion, existing constructive approaches either rely on complex candidate functions (whether rational or polynomial), on semi-analytical results, or alternatively they involve state-space partitions (for which scalability with the state-space dimension is problematic) accompanied by correspondingly complex or large optimisation problems. These approximate methods evidently lack either numerical robustness, being bound by machine precision, or algorithmic soundness: they cannot provide formal certificates of reliability which, in safety-critical applications, can be an evident limit.

In [23] Lyapunov functions are soundly found within a parametric framework, by constructing a system of linear inequality constraints over unknown coefficients. A twofold linear programming relaxation is made: it includes interval evaluation of the polynomial form and "Handelman representations" for positive polynomials. Simulations are used in [24] to generate constraints for a template Lyapunov function, which are then resolved via LP, resulting in candidate solutions. Whilst the authors refer to traces as counterexamples, they do not employ the CEGIS framework, as in this work. When no counterexamples are found, [24] further uses dReal [25] and Mathematica [26] to verify the obtained candidate Lyapunov functions. The sound technique, which is not complete, is tested on low-dimensional models with non-linear dynamics.

The cognate work in [7,8,27] is the first to employ a CEGIS-based approach to synthesise Lyapunov functions. [7,8] focuses on such synthesis for switching control models - a more general setup that ours. [7] employs an SMT solver for the learner, and towards scalability solves an optimisation problem over LMI constraints for the verifier over a given domain (unlike our approach). As such, counterexamples are matrices, not points over the state space, and furthermore the use of LMI solvers does not in principle lead to sound outcomes. Along the above line, [8] expands this approach towards robust synthesis; [27] instead employs MPC (Model Predictive Control) techniques within the learner to suggest template functions, which are later verified via semi-definite programming relaxations (again, possibly generating counterexamples by solving optimisation problems over a given domain). Whilst inspired by this line of work, our contribution provides a simple (with interpretable counterexamples that are points over the state space) yet effective (scalable to at least 10-dimensional models)

SAT-based CEGIS implementation, which automates the construction of Lyapunov functions and associated validity domains, which is is sound, and also applicable to parameterised models.

The remainder of the paper is organised as follows. In Section 2 we present the SMT Z3 solver and the inductive synthesis (IS) framework. The implementation of CEGIS, for both linear and non-linear models, is explained in Section 3. Experiments and case studies are in Section 4. Finally, conclusions are drawn in Section 5.

2 Formal Verification – Concepts and Techniques

In this work we use Z3, an SMT solver, and the CEGIS architecture, to build and to verify Lyapunov functions.

2.1 Satisfiability Modulo Theory

A Satisfiability Modulo Theory problem is a decision problem formulated within a theory, e.g. first-order logic with equality [28]. The aim is to check whether a first-order logical formula within such theory, referred to as an SMT instance, is satisfied. For example, a formula can be the inequality $3x_0 + x_1 > 0$ evaluated within the theory of linear inequalities. An SMT solver is a software that checks the satisfiability of an SMT instance, i.e. whether there exists an instantiation of the formula that evaluates to `True`. SMT solvers can be useful for function synthesis, namely to mechanically construct a function, given requirements on its output.

2.2 The Z3 SMT Solver

Z3 [11,29] is a powerful SMT solver that integrates SAT solvers, theory solvers for equalities and interpreted functions, satellite solvers for arithmetic, real, array, and other theories, and an abstract machine to handle quantifiers. Receiving an input formula, Z3 represents it as an abstract syntax tree and processes it with its SAT solver core, until it returns `SAT` if the formula is satisfiable, `UNSAT` otherwise.

Example 1 (Operation of Z3). Consider the formula $a = b \wedge f(a) = f(b)$ in the theory of equality. To verify its satisfiability, Z3 constructs a syntax tree, with nodes for each variable (a, b) and formulae ($a = b$, $f(a)$, $f(b)$, $f(a) = f(b)$). Once the tree is built, Z3 merges a with b and $f(a)$ with $f(b)$ to represent the equality operation and, in order to verify the correctness of the assertion, applies the congruence rule $\bigwedge_{i=0}^{n-1} x_i = y_i \Rightarrow f(x_0, \ldots x_{n-1}) = f(y_0, \ldots y_{n-1})$ to conclude that $a = b \Rightarrow f(a) = f(b)$. Finally, nodes $a = b$ and $f(a) = f(b)$ are merged and Z3 returns `SAT`. □

Of particular interest for the synthesis of Lyapunov functions, is the ability of Z3 to solve polynomial constraints. Z3 stores and exactly manipulates algebraic real numbers that are roots of rational univariate polynomials: this is done for an algebraic real α, by storing a polynomial $p(x)$ for which $p(\alpha) = 0$ and two rationals l, u such that $p(x) = 0$ for $x \in (l, u)$ if and only if $x = \alpha$. In this work, Z3 has been used through its Python APIs, named Z3Py. An example of a simple assertion verification follows.

Example 2 (Assertion in Z3). Consider the (valid) formula $x \geq 0 \Rightarrow 3x + 1 > 0$. The code using Z3Py results in:

```
x = Real('x')
s = Solver()
s.add(Implies(x >= 0, 3 * x + 1 > 0))
print(s.check())
```

which evaluates (as expected) to `SAT`. □

2.3 Inductive Synthesis - CEGIS

An approach to solve second-order logic problems, such as those characterising the synthesis of Lyapunov functions, is *inductive synthesis* (IS). IS infers general rules (or functions) from specific examples (observations), entailing the process of generalisation. Within the IS procedure, a synthesiser attempts the construction from a (usually small) subset of the original specifications. It then generalises to the complete specification by identifying patterns in the input data.

An exemplar of IS is the CEGIS framework. Fig. 1 depicts the relation between its two main components. It sets off with a given specification ψ over a set $\mathcal{I}$ for the synthesis. The synthesis engine (a component that will be also denoted as *learner*) provides a candidate solution for ι, a subset of $\mathcal{I}$, the space of possible inputs. This candidate solution is passed to a second component, called *verifier*, that acts as an oracle: either it approves the solution over the entire $\mathcal{I}$, so that the process terminates, or it finds an instance $\bar{x}$ (a counterexample in $\mathcal{I}$) where the candidate solution does not comply with the specifications. The learner takes $\bar{x}$ and adds it to ι, computing a new (more general) candidate solution for the problem. This cycle is repeated. Note that this algorithm might not terminate, depending on the structure of $\mathcal{I}$, or might take many cycles to find a proper solution: in those instances, tailored candidate solutions and insightful counterexamples are necessary. In this work, the IS is implemented using SMT-solvers. The verifier finds counterexamples $\bar{x}$ by seeking a witness of the negated formula $\neg\psi$, namely trying to prove that a violation of the formula exists. The learner might employ SMT solvers to solve the system of constraints generated by the counterexamples, i.e. to find a valid instance of such constraints, however in general it does not need to be sound, as it is the verifier that guarantees the soundness of the proposed solution. Section 3.1 illustrates the two CEGIS components, the learner L and the verifier Z in relation to Lyapunov function synthesis.

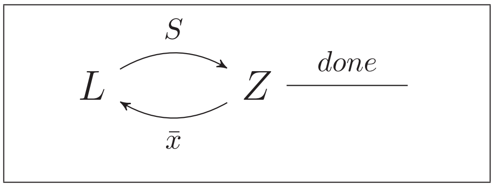
Example 3 (CEGIS Operation). Assume the task is the synthesis of a function $g(x)$ that satisfies the following formula $F(g(x))$:

$$\exists\ g(x)\ \forall x \in \mathbb{R} : \psi, \text{ where } \psi(g(x)) = g(x) + 1 > 0.$$

The learner L offers an initial (often naïve, random or default) candidate, e.g. $g(x) = x$, and passes it to the verifier Z. The verifier checks the validity of $\psi(x) = x + 1 > 0$, $\forall x \in \mathbb{R}$, by searching an instance $\bar{x}$ that might invalidate the formula. Z finds that $\bar{x} = -1$ invalidates the formula, thus sends $\bar{x}$ to L, which incorporates this counterexample to synthesise a new $g(x)$. The learner now adds a constraint on the next candidate, as

$$C := g(-1) + 1 > 0, \quad \forall x \in \mathbb{R},$$

such that the new candidate solution satisfies the formula at $\bar{x} = -1$. The learner now proposes $g(x) = x^2$, which satisfies C, and passes it to Z. The verifier searches for a counterexample to $\psi(x^2)$, but cannot find any. Thus, it exits the loop with an UNSAT answer, which proves that the synthesised function $g(x) = x^2$ is valid $\forall x \in \mathbb{R}$. □



$$\begin{tikzcd} L \arrow[r, bend left, "S"] & Z \arrow[l, bend left, "\bar{x}"] \arrow[r, dash, "done"] & {} \end{tikzcd}$$

Fig. 1. CEGIS-based inductive synthesis. The iterative procedure loops between a learner L and a verifier Z. L provides a candidate solution S to the verifier Z, which asserts its validity or outputs a counterexample $\bar{x}$. The learner provides a new solution encompassing also $\bar{x}$. The procedure stops once no counterexamples are found.

3 Automated and Sound Synthesis of Lyapunov Functions via CEGIS and SMT

Consider a dynamical system $\dot{x} = f(x)$, where $f : \mathbb{R}^n \to \mathbb{R}^n$, and assume that the point $x_e \in \mathbb{R}^n$ is an equilibrium, namely such that $f(x_e) = 0$ – without loss of generality, we assume that $x_e = 0$ (the origin). The goal is assessing the stability of such equilibrium point via the synthesis of a Lyapunov function $V(x) : \mathbb{R}^n \to \mathbb{R}$. The stability of an equilibrium guarantees that trajectories starting by the equilibrium remain close to it at all times (how close can often be quantified, as done later in this work). If $V(x)$ fulfils the following two conditions, $\forall x \in \mathcal{D}$,

$$V(x) > 0, \quad \dot{V}(x) = \nabla V(x) \cdot f(x) \leq 0, \tag{1}$$

where $\mathcal{D}$ is a domain of interest containing x_e then the Lyapunov function ensures boundedness of the trajectories. In other words, for every initial point in a neighbourhood of x_e, the trajectories of the model do not escape from $\mathcal{D}$ (with reference to notations introduced above, the condition in (1) represents the requirement ψ, and $\mathcal{D}$ denotes the set of inputs $\mathcal{I}$). We use the following polynomial expression for the Lyapunov function

$$V(x) = \sum_{l=1}^{c} (x^l)^T \; P_l \; x^l, \tag{2}$$

where x^l represents the element-wise exponentiation of vector x, i.e. element $x(j)$ to the power l, $\forall j = 1, \ldots, n$; $P_l \in \mathbb{R}^{n \times n}$ is a weighting matrix associated with x^l, and $2c$ is the order of the polynomial function. In order to obtain a proper Lyapunov function $V(x)$, the synthesiser is asked to verify the specification expressed by the formula

$$F(V(x)) : \forall x \in \mathcal{D}, V(x) > 0 \wedge \dot{V}(x) \leq 0. \tag{3}$$

This specification requires the Lyapunov function to be positive definite, and not to increase along the trajectories of the model. For linear systems, unless otherwise stated, we consider $\mathcal{D} = \mathbb{R}^n \setminus \{0\}$ and $c = 1$, as it is known that quadratic functions are sufficient to prove the stability of linear models over the whole state space. Formula (3) keeps the elements of P uninterpreted, and thus they are parameters to be found. Notice that the second-order formula

$$\exists P \in \mathbb{R}^{n \times n} : \forall x \in \mathcal{D}, V(x) > 0 \wedge \dot{V}(x) \leq 0,$$

would return a boolean value, i.e. `True` or `False`: to obtain the synthesised $V(x)$ function, we remove the existential quantifier.

3.1 The CEGIS Architecture for Lyapunov Function Synthesis

We introduce the CEGIS architecture to find Lyapunov functions. To better illustrate the methodology, we start by considering linear models (the non-linear case is further discussed in Section 3.2). As mentioned earlier, two components characterise the CEGIS approach: a learner and a verifier. The CEGIS architecture takes the system matrix A and outputs a matrix P as the key component of the function $V(x)$, verifying the conditions in Eq. (1). We denote by $\bar{P}_i$, $i = 0, 1, 2, \ldots$ the *candidate* matrices yet to be verified, i.e. the outputs of the learner. As anticipated earlier, referring to Eq. (2), we set $c = 1$ and $\mathcal{D} = \mathbb{R}^n \setminus \{0\}$.

Verifier The scope of a verifier is twofold: generate a counterexample to the validity of the candidate Lyapunov function, or certify its validity over a domain of interest. We implement the verifier in Z3.

The methodology to assert the correctness of a Lyapunov function is as follows. Assume the learner computes a candidate Lyapunov function $V(x)$ and passes it to the verifier (in case of a linear function, the learner offers a matrix $\bar{P}_i$). The goal of the verifier is to assert the validity of formula F from (3) according to the specification ψ in (1). The check is performed by negating F: if there exists a vector $\bar{x}$ that satisfies $\neg F$, it is a counterexample for F; if it does not exist, formula F is valid and the candidate Lyapunov function is an actual Lyapunov function. The domain $\mathcal{D}$ is encoded as an additional formula. Assume, as an example, the domain is an hyper-sphere of radius one: $\mathcal{D}$ can be written formally as d: $||x||^2 \leq 1$. The final formula thus results in $\neg F \wedge \mathsf{d}$.

A counterexample $\bar{x}$ must satisfy the formula $V(\bar{x}) \leq 0 \vee \dot{V}(\bar{x}) > 0$. Reasoning on either condition, it is easy to show that if there exists a counterexample $\bar{x}$ invalidating a matrix $\bar{P}$, then there exists an infinite number of counterexamples for this $\bar{P}$. Thus, particularly for high-dimensional models the generation of meaningful counterexamples is crucial to find a Lyapunov function quickly.

Let us denote $\bar{x}_i$, $i = 1, \ldots,$ the series of counterexamples provided by the verifier and $\bar{P}_i$ the series of candidate Lyapunov function matrices provided by the learner. In this setting, the learner proposes the first default candidate matrix $\bar{P}_0$; the verifier will (possibly) provide a counterexample $\bar{x}_0$; the learner includes $\bar{x}_0$ in the set of constraints (cf. Section 3.1) and offers a new candidate $\bar{P}_1$.

In this work, we let Z3 generate counterexamples without any further goals. However, counterexamples can be generated adding constraints, e.g. linear independence or orthogonality. Intuitively, more constraints might generate "better" candidates by the learner, albeit at an increase in computational cost.

As intuition suggests, if we were to work with models having a diagonal matrix A, then the synthesis of diagonal candidates $\bar{P}_i$ and of a diagonal solution P would reduce the number of variables needed, thus speeding up the computation. As such, if A is not diagonal but diagonalisable, the algorithm pre-computes the system diagonalisation and feeds it to the CEGIS architecture returning a matrix P for the diagonal system, which is then converted to a solution for the original model.

Learner A learner is the CEGIS component designated to suggest a candidate solution for the problem under consideration. Within our framework, a learner solves linear inequalities derived from $F(V(\bar{x}))$ as per Eq. (3), while memorising the set of counterexamples $\{\bar{x}_i \mid \neg F(\bar{x}_i)\}$ generated by the verifier. Whilst the verifier works over continuous domains, note that the learner only considers a *finite* number of points to synthesise the candidate Lyapunov function. At each iteration i, the learner is tasked to solve $2i$ linear inequalities: i inequalities for $V \geq 0$ and i for $\dot{V} \leq 0$ – this is two inequalities per counterexample, so a set of useful counterexamples is vital to achieve efficiency.

We implement two learners, for comparison: 1) a numerical and 2) a Z3-based learner. However, our CEGIS architecture can in principle accommodate any learner. The first learner uses Gurobi [10], a fast, commercial optimisation solver for, among others, linear and quadratic programming problems, supporting continuous variables. Notice that the synthesis is a linear program: variables $p_{i,j}$, the entries of matrix P, appear linearly within the inequalities in $F(V(\bar{x}_i))$.

Gurobi is thus expected to outperform an SMT solver in this specific task. However these variables do not represent real numbers, but floating point numbers that are approximated at machine precision. The second learner instead employs Z3, which is numerically sound and not affected by machine precision. Z3 solves an SMT instance to synthesise $V(x)$: it asserts the satisfiability of Eq. (3) $F(V(\bar{x}_i))$ for all collected counterexamples $\bar{x}_i$.

As mentioned earlier, the number of inequalities to be solved depends on the number of counterexamples, which can grow to be quite large. Whilst the verifier ought to generate useful counterexamples, the learner is optimised to output a matrix $\bar{P}_i$ that is easy to handle. The comparison between a numerical learner (running on Gurobi) and a sound one (based on Z3) shows that the compromise between speed and soundness results is evident (cf. Section 4). Z3 is sound, yet slower when compared to the numerical learner.

Z3 offers an incremental feature to the learner. During each CEGIS loop, on the verification side the memory is cleared from the previous constraints as the verifier re-initialises the verification problem with a new candidate $V(x)$. On the other hand, the learner keeps the previous synthesis instance adding a new constraint related to the latest counterexample. This incremental approach reduces the computational effort, as the learner does not initialise a new problem for every CEGIS loop.

3.2 Lyapunov Function Synthesis for Non-linear Models

The problem of synthesizing Lyapunov functions and their region of validity for a general non-linear system $\dot{x} = f(x(t))$ is approached via linearisation or via direct computation.

The linearisation approach consists of three steps for the learner: we first linearise the $f(x(t))$, obtaining

$$\dot{\tilde{x}}(t) = A_L \tilde{x}(t),$$

where A_L is the Jacobian of $f(x(t))$ evaluated at x_e; we then compute matrix P – and quadratic Lyapunov function $V(x) = x^T P x$ – on the linearised system; finally, we find $\mathcal{R}$, defined as the set in which the linear Lyapunov function is valid. Next, we detail the synthesis of region $\mathcal{R}$. Consider, without loss of generality, an autonomous non-linear system with (at least one) equilibrium point $x_e = 0$. Assume the CEGIS procedure is successful, i.e. it finds a Lyapunov function $V_L(x) = x^T P x$ that guarantees the asymptotic stability of system $\dot{\tilde{x}} = A_L \tilde{x}$ around x_e. We now compute the region where $V_L(x)$ guarantees stability with the original system, i.e. $\dot{x} = f(x)$. In view of the existence of $V_L(x)$ and by definition of linearisation, there exists a neighbourhood of the origin $\mathcal{B}_0$ in which the derivative of the Lyapunov function $\dot{V}(x)$ is non-positive; formally such set is defined as

$$\mathcal{B}_0 = \{x \in \mathbb{R}^n \backslash \{0\} \mid \dot{V}(x) \leq 0\},$$

where $\dot{V}(x)$ is computed on the original system, namely

$$\dot{V}(x) = \nabla V_L(x) \cdot f(x).$$

Let us define the boundary of $\mathcal{B}_0$ as $\partial\mathcal{B}_0 = \{x \in \mathbb{R}^n\backslash\{0\} \mid \dot{V}(x) = 0\}$. This set may be composed by single points or regions of the state space: in this case, we find r, the closest point to the equilibrium that belongs to $\partial\mathcal{B}_0$, as

$$r = \min_{x \in \partial\mathcal{B}_0} \sum_l x(l)^2.$$

We finally compute region $\mathcal{R}$ as a hyper-sphere of radius r,

$$\mathcal{R} = \{x \in \mathbb{R}^n\backslash\{0\} \mid \|x\|^2 < r^2\}, \tag{4}$$

defining the region where the Lyapunov function is valid. Finally, region $\mathcal{R}$ is tested with the verifier: formula $F(V(x))$ from Eq. (3) is passed to Z3 with $\mathcal{D} = \mathcal{R}$. Our implementation uses a numerical optimisation technique to compute a value for r that is passed to Z3, as Z3 does not natively handle non-linear optimisation problems. With this selection, the region $\mathcal{R}$ represents a sound under-approximation of the maximal stability region. The linearisation method is used in view of its rapid and effective synthesis capability. However, it produces a Lyapunov function that does not ensures global stability when one of the eigenvalues of A_L is equal to zero. This is a well-known limitation of the linearisation, which suggests a more formal approach, called *direct computation method*.

The direct computation method, as the name suggests, analytically computes $V(x)$ and $\dot{V}(x)$ from a template $V(x)$ as in Eq. (2). The learner is tasked with resolving conditions ψ obtained by a light relaxation of the two inequalities in (1), namely

$$V(x) \geq 0, \quad \dot{V}(x) = \nabla V(x) \cdot f(x) \leq 0.$$

Note that the first inequality is not strict: this relaxation allows for a faster computation of a candidate. The verifier, on the other hand, produces counterexamples for $V(x) > 0$, thus retaining soundness of the overall procedure. The CEGIS framework allows the separation between synthesis and verification. So whilst the learner might propose candidates being completely independent from domain $\mathcal{D}$, the verifier is responsible to assert or to find the domain of validity $\mathcal{D}$. Our implementation establishes that at first the verifier checks the validity of $V(x)$ on the whole state space $\mathcal{D} = \mathbb{R}^n$; if the computation is not successful – namely, the computational time is greater than a predefined timeout – the verifier checks its validity over a smaller region, e.g. $\mathcal{D} = [-1, 1]^n$, and so on. If also this program fails, the algorithm returns an empty $V(x)$. Recall that our algorithm is in general not complete - indeed, consider the trivial problem of the synthesis of a Lyapunov function for an unstable system, which is not possible: in this case, the CEGIS procedure will surely return an empty $V(x)$.

3.3 Lyapunov Function Synthesis for Parametric Models

Parametric models represent a challenge for both sound and numerical solvers. Let us remark that both Gurobi and Z3 cannot synthesise functions in the presence of uncertainty, whereas Z3 can provide counterexamples using one or more variables as fixed parameters, using the quantifier `ForAll`.

Let us consider variable x, a parameter μ and a formula $\psi(x, \mu)$: Z3 can find a counterexample for all values of μ by validating `ForAll(`μ`, `ψ`)`. If μ belongs to a range $[l, u]$, Z3 can find a counterexample by checking $\psi \wedge \mu \geq l \wedge \mu \leq u$. This provides a counterexample $(\bar{x}, \bar{\mu})$ for x and μ, respectively.

The synthesis procedure is split into two steps, in view of the inability of Z3 and Gurobi to propose parametric solutions. The first step synthesises a candidate Lyapunov function solely using the constraint $V(x) > 0$, in which no parameter appears. The second step evaluates the constraint $\dot{V} \leq 0$ to propose a parametric Lyapunov function exploiting the results from the first step. The following example details the procedure.

Example 4. Consider a two-dimensional linear parametric system [23] and a candidate Lyapunov function

$$\begin{cases} \dot{x} = y \\ \dot{y} = -(2+\mu)x - y \end{cases}, \quad V(x,y) = p_1 x^2 + p_2 y^2.$$

Assume the first guess of the learner is invalid, i.e. the verifier finds a counterexample for the validity of $V(x, y)$. The counterexample $(\bar{x}, \bar{y})$ is then sent to the learner. The synthesis procedure is split into two steps: the first step entails the synthesis solely accounting for $V(\bar{x}, \bar{y}) > 0$. The learner is tasked to solve

$$V(\bar{x}, \bar{y}) = p_1 \bar{x}^2 + p_2 \bar{y}^2 > 0,$$

where p_1, p_2 are the variables of the inequality. The learner will propose values $\bar{p}_1$ and $\bar{p}_2$ satisfying the inequality. The second step removes one of the synthesised $\bar{p}_i$, e.g. $\bar{p}_1$, in order to re-synthesise it including the parameters found in $\dot{V}$. In practical terms, the expression of $\dot{V}$ is evaluated at $\bar{x}$, $\bar{y}$ and $\bar{p}_2$, as

$$\dot{V} = 2p_1 \bar{x}\bar{y} - 2\bar{p}_2 \bar{y}^2 - 2(\mu + 2)\bar{x}\bar{y} \leq 0 \Longrightarrow p_1 \leq \bar{p}_2 \left(\frac{\bar{y}}{\bar{x}} + 2 + \mu \right).$$

We choose the value p_1 that satisfies the equality. The candidate Lyapunov function thus results in $V(x, y) = \bar{p}_2 \left(\frac{\bar{y}}{\bar{x}} + 2 + \mu \right) \cdot x^2 + \bar{p}_2 \cdot y^2$. This procedure holds as long as $\bar{x} \neq 0$: if this is not the case, we can either choose to synthesise a new value for p_2 or simply maintain the numerical values obtained after the first step. In the latter case, once the candidate Lyapunov function is passed to the verifier, a new counterexample will be generated and the procedure can be repeated until a parametric Lyapunov function is found and verified. Another possible approach is based on the mixed-terms removal: p_1 is synthesised so that the terms carrying $\bar{x}\bar{y}$ cancel out. Further, the choice of p_1 satisfying the equality is arbitrary: we can add a negative constant to its value to solve the strict inequality instead. Finally, more than one parameter $\bar{p}_i$ can be removed in the second step: this can spread the parametric coefficients among more than one p_i. However, this is likely to increase the computational cost in view of the inequality being a function of more than one variable. □

4 Case Studies and Experiments

In this Section we outline a few experiments to challenge the validity of our approach. Our technique is coded in Python 2.7 [30], using external libraries as the numerical solver Gurobi and the SMT solver Z3 (cf. Section 2). Specifically, we compare two CEGIS architectures:

1. Gurobi learner and Z3 verifier,
2. Z3 learner and Z3 verifier,

later denoted as *Gurobi-CEGIS* and *Z3-CEGIS*, respectively, against the optimisation toolbox SOSTOOLS. Whilst Z3 is an efficient verifier, it carries the weight of exact representations. We therefore compare its use within the learner to that of a numerical solver such as Gurobi - recall that the learner does not need to be sound. A relevant feature of the synthesis procedure is its *linearity* in the entries of matrix P: we expect an efficient LP solver to outperform an SMT solver. As such, we study the expected tradeoff between speed and precision. As specified earlier, the initial candidate for the learner $\bar{P}_0$ is arbitrary: we challenge the procedure by setting $\bar{P}_0 = -I$, which does not satisfy the first positivity condition for Lyapunov functions, thus showing that even with an ill-suited initial guess the procedure can rapidly synthesise a valid Lyapunov function. SOSTOOLS is a sum-of-squares optimisation toolbox available for MATLAB, equipped with the solver SeDuMi [31]. It can be used to solve a wide range of problems, from mixed continuous-discrete optimisations to finding Lyapunov functions for polynomial dynamical systems.

We consider linear, non-linear and parametric ODEs with the origin as (one of) the equilibrium(a), and aim to obtain a Lyapunov function guaranteeing the stability of such equilibrium point. The procedure entails the following steps:

a) a function $f(x)$, $x \in \mathbb{R}^n$, is fed as the input;
b) a Lyapunov function $V(x)$, as in Eq. (2), is computed;
c) in the linearisation case, the stability region $\mathcal{R}$ in Eq. (4) for $V(x)$ is found.

Let us emphasise that Z3 is unable to fully handle non-polynomial terms, which represents the only limitation of our approach. Unlike most of the literature, counterexamples are not limited to a finite set but searched over the whole $\mathbb{R}^n$.

Linear models are certainly an easier task than polynomial systems. The study with linear models focuses mainly on the scalability of the method, encompassed by the average and maximum/minimum computational time, and the number of iterations performed. We generate $N = 100$ random linear models of dimension $n \in [3, 10]$. For each linear system, the entries of matrix A range within $[-1000, 1000] \in \mathbb{R}$. For each test we set $c = 1$ (cf. Eq. (2)), namely we impose a quadratic structure to the Lyapunov function, and collect the number of iterations of the procedure, i.e. the number of counterexamples needed to compute a valid Lyapunov function, and the total elapsed time. Recall that the initial synthesiser's candidate is $\bar{P}_0 = -I$, which challenges the reliability of our method with a bad initial condition. A 180 seconds timeout is set for every run.

Results comparing the numerical learner using Gurobi and the sound learner using Z3 are reported in Table 1. The average values, as well as the minimum and maximum value among the N random systems, are computed on the synthesis tests that have not timed out. The number of timed out procedures are also listed in the Table.

With regards to non-linear and parametric models, we assess our approach over a suite of examples taken from related work on Lyapunov function synthesis [18], [19], [20], [23], which are reported in the following. The value c from Eq. (2) is set heuristically as $\texttt{ceil}(d/2)$, where d is the order of the system (this choice follows the common interpretation of Lyapunov maps as storage functions). Due to ease of implementation, only Z3-CEGIS performs the synthesis with $c > 1$ and in the case of parametric models. Results in terms of computational time and iterations are reported in Table 2. Experiments are run on a 4-core Dell laptop with Fedora 30 and 8GB RAM.

Example 5. Consider the model [18]

$$\begin{aligned} \dot{x}_1 &= -x_1^2 - 4x_2^3 - 6x_3x_4, & \dot{x}_4 &= x_1x_3 + x_3x_6 - x_4^3,\\ \dot{x}_2 &= -x_1 - x_2 + x_5^3, & \dot{x}_5 &= -2x_2^3 - x_5 + x_6,\\ \dot{x}_3 &= x_1x_4 - x_3 + x_4x_6, & \dot{x}_6 &= -3x_3x_4 - x_5^3 - x_6. \end{aligned}$$

Z3-CEGIS finds the Lyapunov function $V(x) = 2x_1^2 + 4x_2^4 + x_3^2 + 11x_4^2 + 2x_5^4 + 4x_6^2$, ensuring stability over the whole state space. SOSTOOLS fails to find a $2^{nd}-$ or $4^{th}-$order Lyapunov function for this model. □

Example 6. Consider the model [23]

$$\begin{cases} \dot{x} = -x^3 + y \\ \dot{y} = -x - y. \end{cases}$$

Gurobi-CEGIS finds the Lyapunov function $V(x) = 5 \cdot 10^{-5}x^2 + 5 \cdot 10^{-5}y^2$, whereas Z3-CEGIS finds $V(x) = 0.5x^2 + 0.5y^2$, both ensuring global stability. The linearised Gurobi-CEGIS finds $V(x) = 3.2 \cdot 10^{-3}x^2 + 3.2 \cdot 10^{-3}y^2$, whereas SOSTOOLS finds $V(x) = 0.7844(x^2 + y^2)$, also ensuring stability over the whole state space. □

Example 7. Consider the system [20]

$$\begin{cases} \dot{x}_1 = -x_1^3 - x_1x_3^2, \\ \dot{x}_2 = -x_2 - x_1^2x_2, \\ \dot{x}_3 = -x_3 - \dfrac{3x_3}{x_3^2+1} + 3x_1^2x_3. \end{cases}$$

Note that the term x_3^2+1 is always non-negative, therefore we can consider $\dot{V}(x) \cdot (x_3^2+1) \leq 0$. Gurobi-CEGIS finds the Lyapunov function $V(x) = 32 \cdot 10^{-4}x_1^2 + 32 \cdot 10^{-4}x_2^2 + 8 \cdot 10^{-4}x_3^2$, whereas Z3-CEGIS finds $V(x) = 3x_1^2 + x_2^2 + x_3^2$, and finally SOSTOOLS finds the function $V(x) = 6.659x1^2 + 4.628x2^2 + 2.073x3^2$, all ensuring global stability. □

Example 8. Consider the system [23]

$$\begin{cases} \dot{x} = -x - 1.5x^2y^3, \\ \dot{y} = -y^3 + 0.5x^3y^2. \end{cases}$$

Z3-CEGIS finds $V(x) = 1/3x^2 + y^2$, valid on the whole $\mathbb{R}^2$, whereas SOSTOOLS finds $V(x) = 0.4707x^2 + 1.412y^2$, with a stability region of radius $r = 68$. Gurobi-CEGIS returns an error, as it finds $V(x) = 1.00066454641347x^2 + 2.99933545358653y^2$ that is *not* a valid Lyapunov function. The correct solution, $V(x) = x^2 + 3y^2$, can not be attained in view of lack of convergence of the optimisation algorithm. On the other hand, the linearised Gurobi-CEGIS delivers $V(x) = 32 \cdot 10^{-4}x^2 + 2 \cdot 10^{-4}y^2$ with a radius $r = 1.25$. □

Example 9. Consider the system [23]:

$$\begin{aligned} \dot{x}_1 &= -x_1 + x_2^3 - 3x_3x_4, & \dot{x}_3 &= x_1x_4 - x_3, \\ \dot{x}_2 &= -x_1 - x_2^3, & \dot{x}_4 &= x_1x_3 - x_4^3. \end{aligned}$$

Z3-CEGIS finds the Lyapunov function $V(x) = 2x_1^2 + x_2^4 + 3201/1024x_3^2 + 2943/1024x_4^2$, ensuring global stability. SOSTOOLS, on the other hand, finds a complex 4^{th} order polynomial, omitted here for brevity, with a stability region that is hard to characterise analytically. □

Example 10. Consider the parametric linear system [23]

$$\begin{cases} \dot{x} = y, \\ \dot{y} = -(2 + \mu)x - y, \end{cases}$$

where $\mu \in (-2, 5]$. Z3-CEGIS discovers the Lyapunov function $V(x) = (\mu + 2)x^2 + y^2$, ensuring stability on the whole state space. On the other hand, SOSTOOLS fails to find a solution when setting $V(x, \mu)$ to be independent from, linear in, or quadratic in μ. □

Example 11. Consider the parametric system [23]

$$\begin{cases} \dot{x} = -(1 + \mu_1)x + (4 + \mu_2)y, \\ \dot{y} = -(1 + \mu_3)x - \mu_4y^3, \end{cases}$$

where $\mu_i \in [0, 100]$ for $i = 1, \dots 4$. Z3-CEGIS discovers the Lyapunov function $V(x) = \dfrac{\mu_3 + 1}{\mu_2 + 4}x^2 + y^2$ that asserts stability on the whole state space, whereas SOSTOOLS can not find a solution considering $V(x)$ independent from, linear in, or quadratic in μ_i, where $i = 1, \dots, 4$. □

As expected, Gurobi is faster than Z3 in terms of iterations and computational time. The gap becomes larger with a high-dimensional system, as the SMT learner does not implement any optimisation techniques. The Z3-CEGIS synthesis is performed via an SMT call, which grows in complexity as the number of constraints – related to the number of counterexamples – increases. Gurobi, on the other hand, using optimisation techniques converges faster to a candidate solution that is closer to the actual solution. Our approach outperforms SOSTOOLS in terms of computational time, and it is able to handle parametric and complex models.

Notice that the coefficients of the Lyapunov function synthesised by Gurobi are small in magnitude, as the linear programming problem can encompass the minimisation of coefficients in its setup. On the other hand those obtained from Z3 (rational fractions) are arguably more interpretable. A very interesting result comes from Example 8. Gurobi-CEGIS converges towards the correct Lyapunov function, yet it can not reach the exact numerical values in view of the algorithmic precision. Gurobi numerical guidelines [10] suggest that, as a rule of thumb, the ratio of the largest to the smallest coefficient of the LP problem should be less than 10^9. In our setting, the coefficients are the counterexamples found by Z3, which might require higher precision. In this case, the issue is (probably) caused by a counterexample $\bar{x} \simeq [-755145, 1/8]$, where the first element is actually represented as a (very long) ratio between two integers. The ratio between the two $\bar{x}$ coefficient is in the order of 10^7. Roughly speaking, the counterexamples generated by Z3 depend on the complexity of the tested model: a high-order system might generate numerically ill-conditioned counterexamples, as this example shows. It is also significant how the numerical algorithm tries to converge to a correct solution. The first candidate Lyapunov function results in $V(x) = 1.07079661938449x^2 + 2.92920338061551y^2$ and it takes 99 counterexamples to reach the final value (cf. Example 8), until the procedure stops, resulting in an infeasible problem. Even enveloping the numerical values with the Python Sympy objects `Rational`, `Decimal`, `Fraction`, or the function `simplify` do not help in this context, the limitation being Gurobi's numerical precision.

n	Gurobi-CEGIS			Z3-CEGIS		
	Iterations	Time [sec]	Oot	Iterations	Time [sec]	Oot
3	3 [3, 3]	0.48 [0.33, 0.77]	–	3.03 [3, 4]	0.49 [0.4, 0.70]	–
4	3.10 [3, 4]	0.53 [0.36, 1.20]	–	5.93 [4, 7]	0.68 [0.54,1.07]	–
5	4.15 [4, 5]	1.33 [1.08, 1.97]	–	7.38 [5, 12]	1.67 [1.10, 3.03]	–
6	6.99 [4, 10]	3.88 [2.41, 4.97]	–	9.10 [6, 10]	7.48 [2.40, 54.44]	–
7	8.56 [4, 12]	12.64 [2.9, 62.3]	–	12.88 [5, 17]	17.63 [5.41, 20.3]	1
8	9.14 [3, 13]	21.50 [3.9, 114.16]	1	16.2 [3, 25]	23.91 [4.05, 35.08]	1
9	15.72 [3, 32]	29.98 [3.87, 78.5]	2	22.47 [4, 35]	34.41 [5.67, 48.96]	5
10	18.45 [3,41]	40.63 [6.17, 46.65]	5	27.25 [5, 47]	44.63 [6.32, 101.2]	7

Table 1. Comparison between Gurobi-CEGIS and Z3-CEGIS over n-dimensional linear models. The first values are the average performance on the $N = 100$ randomly generated models, and within brackets the minimum and maximum values. Oot is the number of runs (out of N) not finishing after 180 [sec].

Example #	Gurobi-CEGIS		Z3-CEGIS		SOSTOOLS
	Time [sec]	Iterations	Time [sec]	Iterations	Time [sec]
5	–	–	18.38	4	–
6	0.32	2	1.27	5	3.66
7	0.37	4	0.60	3	4.38
8	0.16	2	0.27	2	3.83
9	–	–	9.26	3	21.31
10	–	–	0.14	3	–
11	–	–	0.23	3	–

Table 2. Comparison between Gurobi-CEGIS, Z3-CEGIS and SOSTOOLS for non-linear models (see Examples description in main text). The result for Gurobi-CEGIS in Example 8 is obtained via linearisation.

5 Conclusions and Future Work

In this work, we have studied the problem of automated and sound synthesis of Lyapunov functions. We have exploited a CEGIS framework, equipped with a sound verifier (the Z3 SMT solver) and with either a numerical LP solver (Gurobi) or a sound (Z3) learner.

We have provided a simple – yet effective – methodology to synthesise Lyapunov functions for linear, polynomial and parametric systems and shown evidence of scalability and reliability of our method using benchmarks from the literature. We have in particular synthesised quadratic Lyapunov functions for linear models and verified their validity on the whole state space. We have tackled non-linear models following two approaches: either 1) the computation of Lyapunov functions over the linearised system and the synthesis of its validity region; or 2) the direct computation of a higher-order Lyapunov function.

Future work includes the implementation of synthesis techniques for Gurobi-CEGIS for high-order and parametric models, together with the study of optimisation techniques for the synthesis in Z3-CEGIS: the tuning of the SMT solvers leaves much room, for example in order to provide insightful counterexamples or to additionally optimise an objective function. Further, we aim at embedding CEGIS with neural networks (as function approximators) to replace the learner, whilst maintaining the verification in the hands of an SMT solver - this approach has been recently pursued also in [32].

References

1. P. Giesl and S. Hafstein, "Review on Computational Methods for Lyapunov Functions," *Discrete and Continuous Dynamical Systems-Series B*, vol. 20, no. 8, pp. 2291–2331, 2015.
2. C. M. Kellett, "Classical Converse Theorems in Lyapunov's Second Method," *Discrete Continuous Dyn. Syst. Series B*, vol. 20, no. 8, pp. 2333–2360, 2015.

3. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial Sketching for Finite Programs," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 404–415, 2006.
4. C. David and D. Kroening, "Program Synthesis: Challenges and Opportunities," *Phil. Trans. R. Soc. A*, vol. 375, no. 2104, p. 20150403, 2017.
5. A. Abate, I. Bessa, D. Cattaruzza, L. Cordeiro, C. David, P. Kesseli, E. Polgreen, and D. Kroening, "Automated formal synthesis of digital controllers for state-space physical plants," in *Proceedings of CAV, LNCS 10426*, 2017, pp. 462–482.
6. A. Abate, I. Bessa, D. Cattaruzza, L. Cordeiro, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Automated formal synthesis of provably safe digital controllers for continuous plants," *Acta Informatica*, 2020.
7. H. Ravanbakhsh and S. Sankaranarayanan, "Counter-example guided synthesis of control lyapunov functions for switched systems," in *IEEE Control and Decision Conference (CDC)*, 2015, pp. 4232–4239.
8. ——, "Robust Controller Synthesis of Switched Systems Using Counterexample Guided Framework," in *ACM/IEEE Conference on Embedded Software (EMSOFT)*, 2016, pp. 8:1–8:10.
9. D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2016.
10. Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2018. [Online]. Available: http://www.gurobi.com
11. L. De Moura and N. Bjørner, "Z3: An Efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2008, pp. 337–340.
12. R. Kalman and J. Bertram, "Control System Analysis and Design via the "Second Method" of Lyapunov: Part I Continuous-time Systems," *Trans. AMSE Series D J. Basic Eng.*, vol. 82, no. 2, pp. 371–393, 1960.
13. N. N. Krasovskii, *Stability of Motion: Applications of Lyapunov's Second Method to Differential Systems and Equations With Delay.* Stanford Univ. Press, 1963.
14. J. LaSalle and S. Lefschetz, *Stability by Liapunov's Direct Method With Applications.* Academic Press, 1961.
15. V. I. Zubov, *Methods of A. M. Lyapunov and Their Application.* Noordhoff, 1964.
16. R. Brayton and C. Tong, "Stability of Dynamical Systems: A Constructive Approach," *IEEE Transactions on Circuits and Systems*, vol. 26, no. 4, pp. 224–234, 1979.
17. P. A. Parrilo, "Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization," Ph.D. dissertation, California Institute of Technology, 2000.
18. A. Papachristodoulou and S. Prajna, "On the Construction of Lyapunov Functions using the Sum of Squares Decomposition," in *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, vol. 3. IEEE, 2002, pp. 3482–3487.
19. S. Prajna, A. Papachristodoulou, and P. A. Parrilo, "SOSTOOLS: Sum of squares Optimization Toolbox for MATLAB–User's Guide," *Control and Dynamical Systems, California Institute of Technology, Pasadena, CA*, vol. 91125, 2004.
20. A. Papachristodoulou, J. Anderson, G. Valmorbida, S. Prajna, P. Seiler, and P. Parrilo, "SOSTOOLS Version 3.03. Sum of Squares Optimization Toolbox for MATLAB," 2018.
21. R. Geiselhart, R. H. Gielen, M. Lazar, and F. R. Wirth, "An Alternative Converse Lyapunov Theorem for Discrete-time Systems," *Syst. Control Lett.*, vol. 70, pp. 49–59, 2014.

22. S. F. Hafstein, “An Algorithm for Constructing Lyapunov Functions,” *Electron. J. Differ. Equ. Monograph*, vol. 8, 207.
23. S. Sankaranarayanan, X. Chen, and E. Abraham, “Lyapunov Function Synthesis using Handelman Representations,” *IFAC Proceedings Volumes*, vol. 46, no. 23, pp. 576–581, 2013.
24. J. Kapinski, J. V. Deshmukh, S. Sankaranarayanan, and N. Arechiga, “Simulation-guided Lyapunov Analysis for Hybrid Dynamical Systems,” in *Proceedings of the 17th international conference on Hybrid systems: computation and control*. ACM, 2014, pp. 133–142.
25. S. Gao, S. Kong, and E. M. Clarke, “dReal: An SMT Solver for Nonlinear Theories over the Reals,” in *International Conference on Automated Deduction*. Springer, 2013, pp. 208–214.
26. Wolfram Research, Inc., “Mathematica, Version 12.0,” 2019.
27. H. Ravanbakhsh and S. Sankaranarayanan, “Learning Control Lyapunov Functions from Counterexamples and Demonstrations,” *Autonomous Robots*, pp. 1–33, 2018.
28. E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018, vol. 10.
29. Microsoft Research, “The Z3 Theorem Prover,” https://github.com/Z3Prover/z3, accessed: 2018-07-25.
30. Python Software Foundation, “Python Language Reference, version 2.7,” http://www.python.org.
31. J. F. Sturm, “Using SeDuMi 1.02, a MATLAB Toolbox for Optimization over Symmetric Cones,” *Optimization methods and software*, vol. 11, no. 1-4, pp. 625–653, 1999.
32. Y. Chang, N. Roohi, and S. Gao, “Neural Lyapunov Control,” in *NeurIPS*, 2019, pp. 3240–3249.

A Study of Symmetry Breaking Predicates and Model Counting

Wenxi Wang[1], Muhammad Usman[1], Alyas Almaawi[1], Kaiyuan Wang[2], Kuldeep S. Meel[3], and Sarfraz Khurshid[1]

[1] University of Texas at Austin, Austin, TX, USA
[2] Google Inc., Sunnyvale, CA, USA
[3] National University of Singapore, Singapore

Abstract. Propositional model counting is a classic problem that has recently witnessed many technical advances and novel applications. While the basic model counting problem requires computing the number of all solutions to the given formula, in some important application scenarios, the desired count is not of *all* solutions, but instead, of all *unique solutions up to isomorphism*. In such a scenario, the user herself must try to either use the full count that the model counter returns to compute the count up to isomorphism, or ensure that the input formula to the model counter adequately captures the *symmetry breaking predicates* so it can directly report the count she desires.

We study the use of *CNF-level* and *domain-level* symmetry breaking predicates in the context of the state-of-the-art in model counting, specifically the leading approximate model counter ApproxMC and the recently introduced exact model counter ProjMC. As benchmarks, we use a range of problems, including structurally complex specifications of software systems and constraint satisfaction problems. The results show that while it is sometimes feasible to compute the model counts up to isomorphism using the full counts that are computed by the model counters, doing so suffers from poor scalability. The addition of symmetry breaking predicates substantially assists model counters. Domain-specific predicates are particularly useful, and in many cases can provide *full* symmetry breaking to enable highly efficient model counting up to isomorphism. We hope our study motivates new research on designing model counters that directly account for symmetries to facilitate further applications of model counting.

1 Introduction

Propositional model counting is the classic problem of counting the number of all solutions for the given formula in propositional logic. While the core problem is an integral part of complexity theory literature, advances in propositional satisfiability (SAT) solvers and other decision procedures in the last decade have led to much progress in tackling this problem in innovative ways [7, 9, 10, 15, 17, 31, 39, 40, 47, 49, 50, 56, 64]. These advances have fueled the application of model counters in various software verification and reliability domains, e.g., to perform

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 115–134, 2020.
https://doi.org/10.1007/978-3-030-45190-5_7

probabilistic analyses [13, 26, 28], check and repair string manipulation code [9, 41], and estimate information leakage using quantified information flow [19, 44].

While the basic model counting problem requires computing the number of *all* solutions, in some important application scenarios, the desired count is not of all solutions, but instead, of all *unique solutions up to isomorphism*, i.e., *non-isomorphic* (also called *non-symmetric*) solutions. For example, consider the context of software *reliability analysis* [26] where a goal is to find the number of inputs that can lead to an assertion violation, or *bounded exhaustive testing* [14, 42, 62, 68] where the goal is to estimate the total number of inputs that exist for a certain bound on the input size to decide what bound to use to stay within the testing budget. The desired counts in these cases are of non-isomorphic inputs, which are *non-equivalent* with respect to behaviors that a program can have because two inputs that are equivalent (and possibly not identical) produce the same output [66]. As another example, consider computing the number of solutions to a *constraint satisfaction problem* (CSP) [45], e.g., the number of unique ways 8 queens can be arranged on a fixed chess board such that no queen is under attack [6]. Once again, one is typically interested in the number of non-symmetric solutions because the indistinguishability of queens implies that a user does not consider two solutions obtained by swapping positions of queens to be unique.

In such scenarios, the user has two basic options. One option is to compute the full count using the model counter, and then use mathematical reasoning about symmetries to project the full count to the desired count. Doing so is straightforward in some cases, e.g., if each solution consists of n indistinguishable objects of the same type and the composition of each solution implies that each permutation of those n objects leads to a distinct (albeit isomorphic) solution, dividing the full count by $n!$ gives the count for non-isomorphic solutions; doing so is however, not always easy, for example when different solutions have different number of objects that can be permuted to form non-identical solutions. The other option is to ensure the formula that is input to the model counter includes *symmetry breaking predicates* [20, 21], i.e., additional constraints that only allow canonical solutions from each isomorphism class, so the model counter can report the desired count.

Symmetry breaking predicates can be added using three basic approaches [29]. Perhaps the most common approach is to add them at the CNF-level by using an off-the-shelf tool [8, 23], which takes as input a CNF formula and creates symmetry breaking predicates for it. Another common approach is to create them at the problem domain level using a domain-specific tool [58], and then translate the formula and predicates together to CNF. A third approach is to add them *manually* at the problem domain level [38, 59], and then translate to CNF.

A goal of our work is to study what is the best way to add symmetry breaking predicates (if at all) to obtain *precise* counts of non-isomorphic solutions. We conduct the study in the context of the state-of-the-art in model counting, specifically the leading approximate model counter *ApproxMC* [16, 17, 52] and the recently introduced exact model counter *ProjMC* [40]. ApproxMC and

ProjMC embody very different algorithms for model counting and provide us a diverse set of tools for the study. ApproxMC employs novel approximation methods to efficiently predict highly accurate model counts with formal guarantees, and is now in its third generation (called ApproxMC3 [52]). ProjMC uses a recursive algorithm and employs a disjunctive decomposition method together with a search for disjoint components, and just had its first public release.

As benchmark formulas, we use a range of problems, including structurally complex specifications of software systems [34] and constraint satisfaction problems [45]. To create the benchmark formulas, we employ the Alloy toolset [34] and its Kodkod backend [58]. Alloy allows writing formulas in relational first order logic with transitive closure, and has been used in academia and industry for design and specification of systems [11, 18, 35, 37, 65, 67, 70] as well as for various forms of analyses of code [27, 32, 36, 42, 48, 69]. The Alloy analyzer translates Alloy formulas with respect to a *scope*, i.e., bound on the universe of discourse, into propositional logic to create CNF problems that are solved using off-the-shelf SAT solvers [25]. Alloy supports fully automatic (partial) symmetry breaking at the level of Alloy specifications [51, 57] by adapting Crawford's symmetry breaking predicates [20], which are *statically* added to the formula *before* the solvers solve it. Alloy provides an ideal vehicle for evaluating the different approaches to symmetry breaking that are our focus in this study.

Similar to other techniques that use CNF-based backends, the Alloy analyzer translates problems from a higher-level (Alloy) to a lower-level (CNF). This translation often introduces new boolean variables in the resulting formula, which are not essential for creating the CNF formula but are required for a compact (feasible) encoding in CNF [60]. As a result, the translated formula is *equisatisfiable* to the original formula but may not be *equivalent* to it, and hence it may be the case that the model count for the CNF formula is very different from the original formula. Several modern model counters [16, 40, 50] readily handle this case by providing support for *projected model counting* [10], i.e., computing the model count with respect to a *subset* of all the variables. For Alloy, the subset is the *primary variables*, i.e., all boolean variables that directly correspond to the variables in the Alloy specification.

For each benchmark formula f, we create three model counting problems using automatic tools: 1) f with no symmetry breaking, which we create by setting Alloy's default symmetry breaking to *off*; 2) f with symmetry breaking predicates added at the problem domain level, which we create by having Alloy's default symmetry breaking turned *on*; and 3) f with symmetry breaking predicates added at the CNF level, which we create by first using Alloy to create a CNF formula with no domain-level symmetry breaking, and then using the BreakID [23] tool to add CNF-level symmetry breaking predicates using its default settings. In addition, for select benchmarks we create formulas with manually added domain-specific symmetry breaking predicates, which we write in Alloy following previous work [38].

The results show that while it is sometimes feasible to compute the model counts up to isomorphism using the full counts that are computed by the model counters, doing so suffers from poor scalability. The addition of symmetry breaking predicates substantially assists model counters, although it is a well-known feature in *SAT solving* supported by theory finding [46, 61]. Domain specific predicates are particularly effective, and in many cases, can provide *full* symmetry breaking to enable highly efficient model counting up to isomorphism. We were surprised by the extent of the impact. Since the addition of symmetry breaking predicates introduces new dependencies among the variables, we expected these dependencies to make the formula more complex and perhaps less amenable to efficient model counting. However, the sheer reduction in the number of solutions caused by symmetry breaking more than compensates for the additional logical complexity of the formula. In cases where it was possible to create *full* symmetry breaking predicates, the model count for the formula *with* the predicates was computed up to a few orders of magnitude faster than the formula with *no* symmetry breaking predicates.

A key lesson of our study (in the context of the model counting problems considered) is: *if non-isomorphic solution counts are desired, use full symmetry breaking predicates at the domain-level whenever feasible – even if it is straightforward to compute the number of non-isomorphic solutions from the number of all solutions, or even if the symmetry breaking constraints have to be written manually.* This paper makes the following contributions:

– **Study.** To the best of our knowledge, we present the first study of symmetry breaking in the context of model counting. As pointed out earlier, there is a tradeoff between the reduction of solution space and the likely increase in complexity due to added symmetry breaking predicates. In prior work, the benefit of symmetry breaking in SAT solving were typically observed largely for *unsatisfiable* problems [43], our study shows the importance of symmetry breaking and its deep relation to problem formulation in the context of *satisfiable* problems, albeit for model counting.
– **Dataset.** All CNF files we used for the experiments are being made publicly available: `https://github.com/wenxiwang/TACAS2020`. We expect the dataset to be useful for future work on evaluating the performance of different model counters, and of the different strategies they employ, as well as for evaluating model enumeration tools.

We believe there is an important *bi-directional* relation between symmetry breaking and model counting whereas: 1) in one direction the model counters directly support computing the counts for non-isomorphic solutions to facilitate applications that so require; and 2) in the other direction symmetry breaking helps model counters become more efficient. We hope our study motivates future work that further investigates this relation.

2 Examples

This section provides two illustrative examples that require computing the number of unique solutions up to isomorphism. We specify the examples in the Alloy

```
module nqueens -- name of the specification

sig Queen {} -- set of queen atoms

one sig Board { state: Queen -> Int -> Int } -- one board

fact StateOkay {
  all q: Queen | one q.(Board.state) -- each queen occupies exactly one cell
  all x: Queen.(Board.state).Int | ValidIndex[x] -- all x-coordinates are valid
  all y: Int.(Queen.(Board.state)) | ValidIndex[y] -- all y-coordinates are valid
  all disj q, r: Queen | q.(Board.state) != r.(Board.state) } -- queens do not share cells

pred ValidIndex[x: Int] { x.gte[0] and x.lte[(#Queen).minus[1]] } -- x >= 0 && x <= |Queen|-1

fun X[q: Queen]: Int { (q.(Board.state)).Int } -- x-coordinate of q

fun Y[q: Queen]: Int { Int.(q.(Board.state)) } -- y-coordinate of q

fun Abs[x: Int]: Int { x.lt[0] implies negate[x] else x } -- absolute value of x

pred SameRow[q, r: Queen] { X[q] = X[r] } -- q and r are in the same row

pred SameColumn[q, r: Queen] { Y[q] = Y[r] } -- q and r are in the same column

pred SameDiagonal[q, r: Queen] { -- q and r share a diagonal
  Abs[X[q].minus[X[r]]] = Abs[Y[q].minus[Y[r]]] }

pred NQueensProblem { -- no queen attacks another queen
   all disj q, r: Queen | !SameRow[q, r] and !SameColumn[q, r] and !SameDiagonal[q, r] }
```

Fig. 1: Alloy specification of n-Queens.

language, which allows us to explore different approaches for applying symmetry breaking. We provide intuitive descriptions of Alloy constructs as we introduce them; further details can be found elsewhere [34].

The first example illustrates a CSP problem [45] where Alloy's default symmetry breaking provides *full* symmetry breaking; we use ApproxMC to solve this problem (Section 2.1). The second example illustrates a software testing problem [42] where manually written symmetry breaking predicates provide full symmetry breaking; we use ProjMC to solve this problem (Section 2.2). Section 5 presents a detailed experimental evaluation where we use the two tools against many additional benchmarks.

2.1 *n*-Queens

Consider specifying the well-known *n-Queens* problem of placing n interchangeable queens[4] on a fixed $n \times n$ chess-board, and computing the number of solutions to the problem using a modern propositional model counter [16, 40, 50].

Figure 1 shows a fragment of an Alloy specification of the n-Queens problem, which has been studied before using Alloy [2, 4, 55]. The keyword `sig` introduces a set of (interchangeable) atoms. The keyword `one` makes the set a singleton. The field `state` introduces a quaternary relation of type "`Board x Queen x Int x Int`" where `Int` is a built-in type that represents integers. The *fact* `StateOkay` describes the basic constraints for the state of the board to be valid; the fact contains

[4] Here, we only consider symmetries based on permuting the queens (and not other forms, e.g., rotations of the board.)

```
Queen={Queen$0, Queen$1, Queen$2, Queen$3,
       Queen$4, Queen$5, Queen$6, Queen$7}

Board={Board$0}

Board<:state={Board$0->Queen$0->7->5, Board$0->Queen$1->6->0,
              Board$0->Queen$2->5->4, Board$0->Queen$3->4->1,
              Board$0->Queen$4->3->7, Board$0->Queen$5->2->2,
              Board$0->Queen$6->1->6, Board$0->Queen$7->0->3}
```

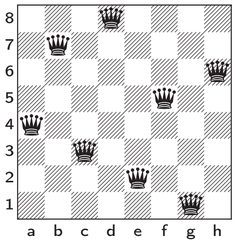


Fig. 2: A solution to 8-queens created by the Alloy analyzer illustrated.

4 sub-formulas that are implicitly conjoined; each of them uses universal quantification (`all`); the keyword `disj` constrains the quantified variables to represent distinct values. The dot operator (‘`.`’) is *relational join* [34]. A *predicate* (`pred`) is a parameterized formula that can be invoked elsewhere; likewise, a `fun` is a parameterized expression. The predicate `NQueensProblem` represents the overall specification of the n-Queens constraints. Any model of the Alloy specification must satisfy the constraints in all the facts and any predicates that are invoked (directly or transitively).

The Alloy user writes a *command* and executes it to solve desired constraints. For example, “`run NQueensProblem for 5 int, exactly 8 Queen`” asks the analyzer to find a solution to the 8-Queens problem. This command creates a constraint solving problem such that the integer bit-width is 5, and there are exactly 8 queens. Figure 2 shows a *valuation* for each set and relation created by the Alloy analyzer to solve this problem, and graphically illustrates the solution.

Next, we illustrate the use of the approximate model counter ApproxMC [16]. For the `nqueens` specification, for each $7 \leq n \leq 12$, we create three constraint solving problems: 1) no symmetry breaking (*no-sb*); 2) BreakID’s default CNF-level symmetry breaking [23] (*cnf-sb*); and 3) Alloy’s default domain-level symmetry breaking [58] (*dom-sb*). Table 1 shows the number of solutions found and time taken in each case. The model count with no symmetry breaking is the highest and takes the longest to compute; this approach times out for 8×8 and larger boards. BreakID’s default CNF-level symmetry breaking significantly reduces

Table 1: ApproxMC results for n-Queens for $7 \leq n \leq 12$. Model count (“#”) and time taken in seconds (“$t[s]$”) for different problem sizes are shown. Time-out (t.o.) is 5000 sec.

		7×7		8×8		9×9		10×10		11×11		12×12	
		#	$t[s]$	#	$t[s]$	#	$t[s]$	#	$t[s]$	#	$t[s]$	#	$t[s]$
approx	*no-sb*	208896	3727.1	-	t.o.	-	t.o.	-	t.o.	-	t.o.	-	t.o
	cnf-sb	67584	1446.4	-	t.o.	-	t.o.	-	t.o.	-	t.o.	-	t.o
	dom-sb	40	1.14	92	13.67	304	16.27	784	44.97	2752	199.77	15360	822.14
OEIS		40		92		352		724		2680		14200	
error		0		0		0.158		-0.077		-0.026		0.076	

the counts and the time. Alloy's default domain-level symmetry breaking is the most effective, and for this problem, removes all symmetries. Some of the approximate model counts reported by ApproxMC are coincidentally the *exact* counts. We validated the counts using the On-line Encyclopedia of Integer Sequences (OEIS) [6]: the sequence $\#A000170$ represents the number of solutions for the n-Queen problem. The counts computed using Alloy's default symmetry breaking with ApproxMC up to board size 8×8 form a subsequence of $A000170$. For the other board sizes, the table lists the error, which is $max(\frac{approx}{exact}, \frac{exact}{approx}) - 1$, based on multiplicative guarantees.

Note that the non-isomorphic solution count can easily be estimated from the full count for this problem. For example, for the 7×7 board we can estimate it as $\frac{208896}{7!} = 41.44$, which is quite close to the actual count of 40. While the calculation is simple, the time to compute the full count is much higher (3727.1 seconds instead of 1.14 seconds). Moreover, for larger board sizes, computing the full count times out, so using it for those sizes may be simply infeasible. This example illustrates a case where symmetry breaking predicates reduce both the model count and the time to compute it by relatively large factors.

3-queens. Table 2 shows the results for a variation of the n-queens problem where the number of queens is fixed to 3, and the board size varies. To specify this variation, we replace the expression "`(#Queen).minus[1]`" in predicate *ValidIndex* with the value of "$k - 1$" for the board size $k \times k$, and set the scope for Queen to "`exactly 3`" in the run command. We validate the ApproxMC counts using the OEIS sequence $\#A047659$ [6]. Once again, BreakID's CNF-level predicates significantly reduce the model count and time to compute it, and Alloy's domain-level predicates reduce them further. Since the number of queens is fixed to 3, the ratio of total number of solutions (*no-sb*) to number of non-isomorphic solution is $3! = 6$. For example, for 11×11 board, the ratio for ApproxMC counts is exactly 6; however, the time to compute the full count is, as before, much higher (1307.04 seconds instead of 45.1 seconds). This example shows a case where symmetry breaking predicates reduce the model count by a relatively small factor but the time to compute the counts by a much larger factor.

2.2 Data structure invariants

Next, consider the context of bounded exhaustive testing where the program under test is run against every non-equivalent input within a bound on the

Table 2: ApproxMC results for 3-Queens where 3 queens are placed on $n \times n$ board for $8 \leq n \leq 12$.

		8×8		9×9		10×10		11×11		12×12	
		#	$t[s]$	#	$t[s]$	#	$t[s]$	#	$t[s]$	#	$t[s]$
approx	*no-sb*	64512	107.56	176128	368.65	335872	695.55	688128	1307.04	1081344	4811.86
	cnf-sb	18944	30.26	51200	67.43	122880	153.16	241664	280.15	417792	567.48
	dom-sb	9728	7.94	25088	12.78	57344	26.14	114688	45.1	200704	111.76
OEIS		10320		25096		54400		107880		199400	
error		0.061		0.000		-0.051		-0.059		-0.006	

```
one sig BT {
  root: lone Node }

sig Node {
  left, right: lone Node }

pred Acyclic(t: BT) {
  all n: t.root.*(left + right) {
    n !in n.^(left + right) -- no directed cycle
    lone n.~(left + right) -- at most one parent
    no n.left & n.right }} -- children are different

pred RepOk(t: BT) { Acyclic[t] }
...
```

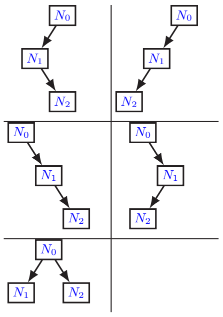


Fig. 3: (a) Alloy specification of binary trees. (b) Five non-isomorphic binary trees with 3 nodes. N_0 is the root.

input size, and the inputs are characterized by a logical formula [42]. Assume the goal is to identify a bound that will lead to a feasible number of inputs that can be executed within the testing budget. We use model counting to estimate the number of solutions for different bounds.

Assume the inputs to the program under test are *binary trees*. Figure 3a shows a partial Alloy specification for binary trees. The singleton sig `BT` represents the tree, which has a root node and an integer size; the keyword `lone` defines a partial function, so, e.g., the tree root is either exactly one node or none. Each node has an integer key and a left and a right child. The predicate `RepOk` specifies the constraints for a valid binary tree, which must be acyclic. The predicate `Acyclic` specifies acyclicity; the operator "`^`" is transitive closure, "`*`" is reflexive transitive closure, "`+`" is set union, "`&`" is set intersection, and "`~`" is transpose.

Consider the constraint solving problem for size k so that the binary tree has exactly k nodes and the keys are $1, \ldots, k$. Figure 3b illustrates the 5 non-isomorphic trees for size 3.

To show that the impact of symmetry is not limited to only approximate counting, we perform this case study with the exact model counter ProjMC [40]. Table 3 shows the model counts for different sizes. As before, CNF-level symmetry breaking reduces the model count, which is further reduced by Alloy's

Table 3: ProjMC results for binary tree constraints for trees with 6, 7, 8, 9, and 10 nodes. Time-out (t.o.) is 5000 sec.

		6		7		8		9		10	
		#	$t[s]$	#	$t[s]$	#	$t[s]$	#	$t[s]$	#	$t[s]$
exact	*no-sb*	95040	5.57	2162160	129.25	57657600	3673.89	-	t.o.	-	t.o.
	cnf-sb	61538	7.39	1538628	184.97	25955296	3466.19	-	t.o.	-	t.o.
	dom-sb	357	0.10	1866	0.70	10286	4.94	60616	40.21	373001	610.35
	man-sb	132	0.03	429	0.09	1430	0.34	4862	1.48	16796	10.53
OEIS		132		429		1430		4862		16796	

```
fact SymmetryBreaking { // pre-order
  BT.root in first[]
  all n: BT.root.*(left + right) {
    some n.left implies n.left in next[n]
    no n.left implies n.right in next[n]
    some n.right and some n.left implies
      n.right in next[max[n.left.*(left + right)]] }}
```

Fig. 4: *Full* symmetry breaking predicates in Alloy [38].

default symmetry breaking. However, unlike before, CNF-level symmetry breaking sometimes makes the model counter, which is ProjMC in this case, slower. Moreover, Alloy's default symmetry breaking does not break all symmetries. For this example, they can be broken using *manually written* predicates. Binary trees belongs to a restricted class of data structures for which *full* symmetry breaking can be achieved by writing predicates in Alloy so that only the *canonical* solution from each isomorphism class is allowed [38]. Figure 4 shows a fact that embodies this approach. Intuitively, the fact requires that a *pre-order* traversal starting at the root visits the nodes in the same order as a *pre-defined* linear ordering of the nodes; the `ordering` *module* in Alloy allows defining a linear order. The manually written predicates provide the most efficient counting. In this example the count up to isomorphism can, once again, be computed from the full count but at a much higher computational cost. For example, for 8 nodes, the full count is 57657600, which divided by 8! is 1430, i.e., the count up to isomorphism, but ProjMC takes 3673 seconds to compute the full count whereas once the manual symmetry breaking predicates are added it takes 0.34 seconds. The number of binary trees with n nodes is the OEIS sequence #A000108, which allows us to validate that the manually written predicates are indeed breaking all symmetries.

3 Background: Model counting

This section gives the relevant background on model counting, with a focus on projected and approximate model counting.

Let φ be a Boolean formula in conjunctive normal form (CNF) over the variable set X. An assignment σ of truth values to the variables in φ is called solution of φ if it makes φ evaluate to true. We denote the set of all witnesses of F by R_F. Given a set of variables $S \subseteq X$ and an assignment σ, we use $\sigma \downarrow S$ to denote the projection of σ on S. Similarly, $R_{\varphi \downarrow S}$ denotes projection of R_φ on S.

The *projected model counting problem* is to compute $|R_{\varphi \downarrow S}|$ for a given CNF formula F and sampling set $S \subseteq X$. When $S = X$, the problem is referred to as model counting. A *probably approximately correct* (or PAC) counter is a probabilistic algorithm $\mathsf{ApproxCount}(\cdot, \cdot, \cdot, \cdot)$ that takes as inputs a formula F, a sampling set S, a tolerance $\varepsilon > 0$, and a confidence $1 - \delta \in (0, 1]$, and returns a count c such that $Pr\Big[|R_{\varphi \downarrow S}|/(1+\varepsilon) \leq c \leq (1+\varepsilon)|R_{\varphi \downarrow S}|\Big] \geq 1 - \delta$. For clarity, we omit mention of S unless needed for a given context.

Projected Model counting is a fundamental problem in computer science with applications ranging from reliability of networks to information leakage. Valiant initiated complexity theoretic studies of model counting and showed that model counting is #P-hard [63]. The earliest practical approaches to model counting such as Relsat [12], were based on extending DPLL approaches. The advent of CDCL solvers led to the paradigm of combining conflict driven search with component caching leading to the development of solvers such as Cachet [49] and sharpSAT [56]. Furthermore, Darwiche and Marquis [22] pioneered a knowledge-compilation-based approach, relying on the static partitioning of the solution space, which led to development of c2d. The recent years have witnessed combination of CDCL and static approaches with solvers such as D4 and DSharp. Recently, Lagniez and Marquis proposed a recursive algorithm, called ProjMC [40], that exploits the disjunctive decomposition technique pioneered in earlier works to perform projected model counting. Concurrently, another approach, called Ganak [50], for projected model counting has been developed that provides *probabilistic exact* bounds via usage of universal hash functions. In this work, we focus on ProjMC due to its ability to provide exact counts and demonstrated scalability in comparison to other approaches.

The theoretical studies of approximation led to the introduction of PAC style, also referred to as (ε, δ), guarantees wherein the underlying algorithm returns an estimate within $(1+\varepsilon)$ factor of the exact count with confidence at least $1-\delta$. Stockmeyer [54] demonstrated that PAC guarantees can be achieved by a probabilistic polynomial Turing machine with access to NP oracle. The practical exploration of Stockmeyer's approach was pursued with Gomes et al with the development of MBound [31] and SampleCount [30]. Chakraborty, Meel, and Vardi proposed a scalable approximate counter, called ApproxMC, with formal (ε, δ) guarantees which seeks to combine the advances in SAT solving with design of efficient universal hash functions.

ApproxMC is now in its third generation, called ApproxMC3. The central idea behind ApproxMC is to employ universal hash functions, represented by randomly chosen XOR constraints, to partition the solution space into roughly equal small cells where every cell can be defined by the original constraints augmented with randomly chosen XOR constraints. ApproxMC invokes CryptoMinisat [53], a solver designed specifically for combination of CNF and XOR constraints, to enumerate solutions in a randomly chosen *small* cell. ApproxMC2 achieves a significant reduction in the number of SAT calls from linear in $|S|$ to $\log(|S|)$ by exploiting dependence among different SAT calls. Soos and Meel proposed ApproxMC3 by augmenting ApproxMC2 with a new architecture to handle CNF+XOR formulas [52].

4 Study methodology

This section describes the overall design of our study, including the model counting tools, the generation of constraint solving problems, and the measurements for evaluation.

4.1 Tools

For approximate model counting, we use ApproxMCv3 (`https://github.com/meelgroup/ApproxMC`), which is the latest public release of ApproxMC [52]. For each model counting problem, we list the primary variables in the input CNF file as a comment as required by ApproxMC. For exact model counting, we use the latest public release of ProjMC [40] (`http://www.cril.univ-artois.fr/kc/projmc.html`). For each model counting problem, we list the primary variables in a separate file as required by ProjMC.

4.2 Benchmarks

Base formulas. We use four sources of base formulas.

(1) *Alloy specs.* We consider all Alloy specifications in the standard distribution [1]; each command in an Alloy spec defines a constraint solving problem and provides a scope; we use the given scope. We remove unsatisfiable problems since their model count is 0 (regardless of symmetry breaking), and our focus in this study is on satisfiable problems. We also remove all "easy" cases that complete within 1 second for both tools and all symmetry settings. This creates a set of 47 base problems derived from Alloy specifications.

(2) *Kodkod problems.* We consider all Kodkod programs in the standard distribution [5]. Once again, we remove the unsatisfiable problems and "easy" cases. In addition, we remove problems that do not admit symmetry breaking, i.e., where Kodkod does not add any symmetry breaking by default (e.g., when there is a given partial solution, which prevents Kodkod's *greedy base partitioning* [57] from having an effect). Some of the Kodkod programs are parameterized over integer bounds and input files. We manually create those inputs in the appropriate format. This gives us a total of 13 base problems derived from Kodkod programs.

(3) n-*Queens.* We use 2 common variations of the n-Queens problem: 1) k queens are placed on a $k \times k$ board ($1 \leq k \leq 12$); 2) 3 queens are placed on a $k \times k$ board ($1 \leq k \leq 12$). This gives us a total of 24 base problems derived from the n-Queens problem[5].

(4) *Complex data structures.* We use 6 complex data structures: (1) singly-linked lists; (2) sorted lists; (3) doubly-linked lists; (4) binary trees; (5) binary search trees; and (6) red-black trees. For each structure, we bound the number of nodes to be between 6 and 9 (inclusive). This gives us a total of 24 base problems based on structural invariants.

Model counting benchmarks. For each base formula f, we create 3 model counting problems using automatic tools: 1) f with no symmetry breaking, which we create by setting Alloy's default symmetry breaking to *off*; 2) f with symmetry breaking predicates added at the CNF level, which we create by first using Alloy to create a CNF formula with no domain-level symmetry breaking, and then using the BreakID [23] tool to add CNF-level symmetry breaking predicates using the same arguments as in the SATRACE'15 competition [3]; and 3) f with symmetry breaking predicates added at the problem domain level, which we create by having Alloy's default symmetry breaking turned *on*. Moreover, for data

[5] Unfortunately, we were not able to get the results for majority of the n-Queens benchmarks with ProjMC due to an unknown issue with the tool, so we do not use the n-Queens benchmarks for experiments with ProjMC; we have requested the ProjMC team to look into the issue.

structures, we create formulas with manually added domain-specific symmetry breaking predicates, which we write in Alloy following previous work [38]. This gives us a total of 348 model counting problems.

Table 4 shows some characteristics of the benchmarks, specifically the minimum and maximum numbers of primary variables, and all variables and clauses under the different symmetry breaking settings.

4.3 Metrics

We use two key metrics – the model counts and the time to compute them – and measure them under different symmetry breaking settings. For model counts, we report the tool output and the ratio of the count under one setting to the count under another setting. For time, we report the actual wall-clock times, and the ratio of time taken under one setting to the time taken under another setting. In line with prior work [17], we report the error rate of the approximate model counting which is $max(\frac{approx}{exact}, \frac{exact}{approx}) - 1$, based on multiplicative guarantees.

5 Experimental evaluation

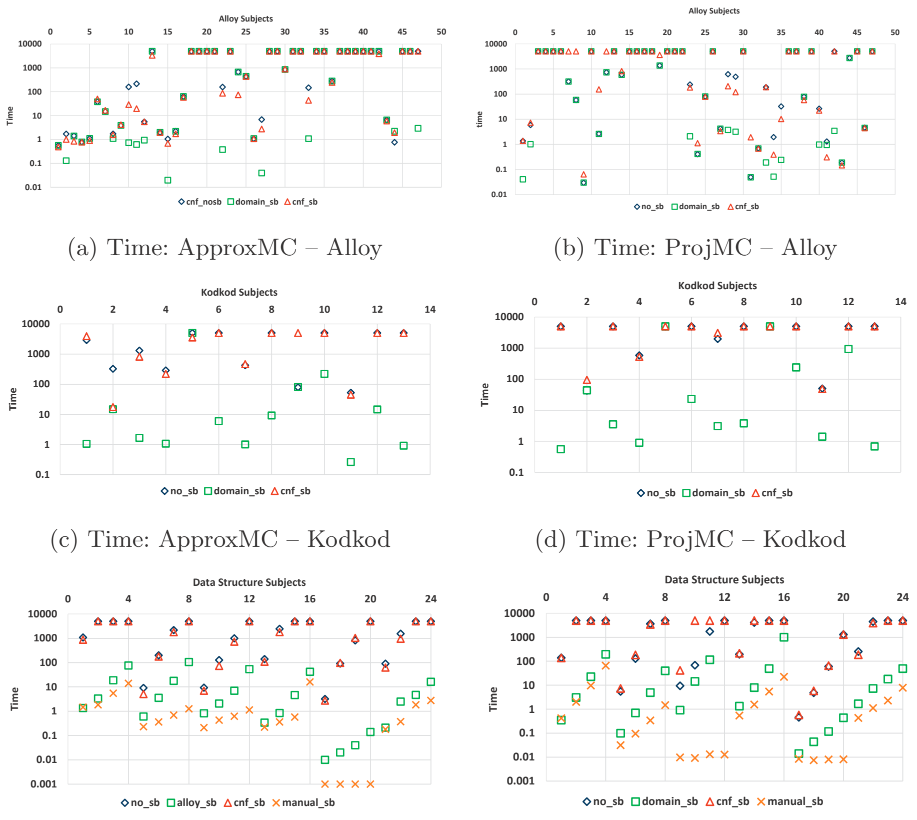
The section reports the results of the experimental evaluation. Section 5.1 describes the results for ApproxMC. Section 5.2 describes the results for ProjMC.

5.1 Symmetry breaking and approximate model counting

Time. Figures 5a, 5c, and 5e illustrate the time performance of ApproxMC on the benchmarks based on Alloy, Kodkod, and data structure invariants respectively. With no symmetry breaking, ApproxMC times out on 21 (of 47) Alloy benchmarks, 6 (of 13) Kodkod benchmarks, and 10 (of 24) data structure benchmarks. In all but 16 cases, formulas with Alloy's default symmetry breaking take less time than with CNF-level symmetry breaking. In all but 10 cases, formulas with CNF-level symmetry breaking take less time than with no symmetry breaking. Moreover, for data structure benchmarks, in all but 1 cases, formulas with manual symmetry breaking take less time than Alloy's default symmetry breaking. Among all the problems that time out with no symmetry breaking, the smallest time taken by the corresponding problem with Alloy's default symmetry breaking was 0.14 seconds, and the smallest time taken by

Table 4: Benchmark characteristics.

source	*#prim.*	*no-sb*		*cnf-sb*		*dom-sb*		*man-sb*	
		#var.	*#clause*	*#var.*	*#clause*	*#var.*	*#clause*	*#var.*	*#clause*
Alloy: min	46	384	620	522	1037	384	620	-	-
Alloy: max	2048	93764	291349	93764	289725	93764	291349	-	-
Kodkod: min	48	631	188	932	628	990	188	-	-
Kodkod: max	8188	388755	764957	397566	834629	453358	877429	-	-
n-Queens: min	1024	3762	7163	3762	7163	3762	7163	-	-
n-Queens: max	12288	200074	532527	201064	523947	269141	704396	-	-
Data Str.: min	43	992	3039	1091	3337	1209	3401	1006	3155
Data Str.: max	510	18694	48290	19045	45562	19808	50212	18993	50696



(a) Time: ApproxMC – Alloy

(b) Time: ProjMC – Alloy

(c) Time: ApproxMC – Kodkod

(d) Time: ProjMC – Kodkod

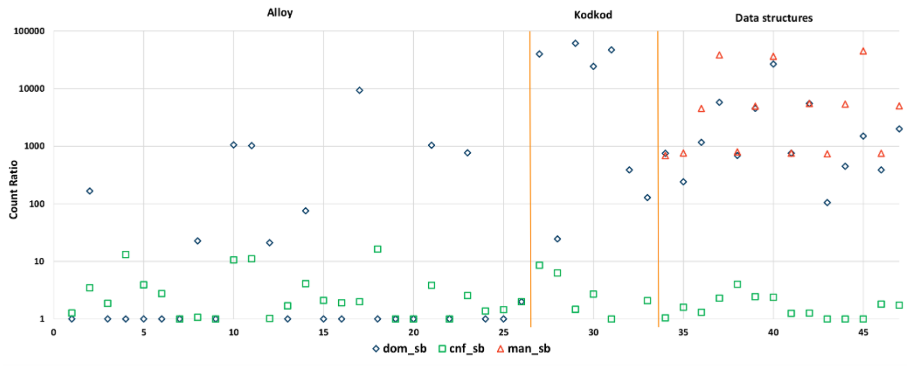
(e) Time: ApproxMC – Data structures

(f) Time: ProjMC – Data structures

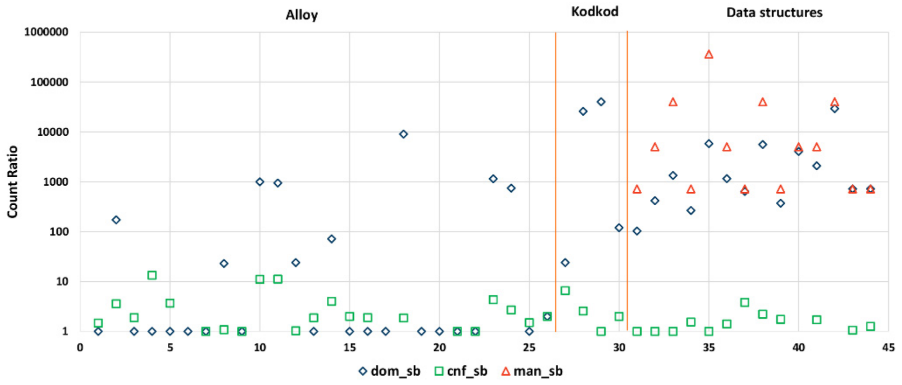
Fig. 5: Time results. x-axis has benchmark model counting problems. y-axis has time in seconds (log-scale). Benchmarks on x-axis are sorted in ascending order based on the number of primary variables; moreover, the data structure benchmarks are grouped by the type of the structure. Blue diamond is no symmetry breaking (*no-sb*) ; red triangle is CNF-level symmetry breaking (*cnf-sb*); green square is Alloy's default symmetry breaking (*dom-sb*); and orange cross is manual symmetry breaking (*man-sb*).

the corresponding problem with manual symmetry breaking was 0.008 seconds. For the Alloy benchmarks, ApproxMC does not time-out under any symmetry breaking setting for benchmarks that have up to 90 primary variables. The time results for the n-Queens benchmarks were presented in Section 2.1.

Model counts. Figure 6a graphically illustrates how the model counts vary under different symmetry breaking settings. For the Alloy and Kodkod benchmarks, in all but 10 cases the model count for the formula with Alloy's default symmetry breaking is less than the corresponding count with CNF-level sym-



(a) Model count: ApproxMC



(b) Model count: ProjMC

Fig. 6: Model count results. x-axis has benchmark model counting problems. y-axis (log-scale) has count ratio n/c where n is the model count for the formula with no symmetry breaking and c is the corresponding count with CNF-level symmetry breaking (green-square), Alloy's default symmetry breaking (blue-diamond), and manual symmetry breaking (red-triangle – only for data structures). Only cases where the calculation of n did not time out are shown.

metry breaking. For the data structures, the model count for the formula with Alloy's symmetry breaking is less than the corresponding count with CNF-level symmetry breaking in all cases; moreover, in all but 5 cases, manual symmetry breaking gives the lowest count (the 5 exceptions are due to approximation in computing the model counts). Among all problems where ApproxMC reports a count with no symmetry breaking, the largest ratio of count with no symmetry breaking to count with Alloy's default symmetry breaking was 61167, and the largest ratio of count with no symmetry breaking to count with manual symmetry breaking was 45056. The model count results for the n-Queens benchmarks were presented in Section 2.1.

Error. For the Alloy, Kodkod, and data structure benchmarks, we compute the error in ApproxMC with respect to the counts reported by ProjMC for the cases where ProjMC reported a count. The error ranges were: [0, 0.168] for the Alloy benchmarks, [0, 0.168] for the Kodkod benchmarks, and [0, 0.165] for the data structure benchmarks. Section 2.1 presented the error results for the n-Queens benchmarks with respect to the number in OEIS [6].

5.2 Symmetry breaking and exact model counting

Time. Figures 5b, 5d, and 5f illustrate the time performance of ProjMC on the benchmarks based on Alloy, Kodkod, and data structure invariants respectively. With no symmetry breaking, ProjMC times out on 21 (of 47) Alloy benchmarks (which is the same number as ApproxMC although the two sets of benchmarks are not the same), 9 (of 13) Kodkod benchmarks (which is more that the number for ApproxMC), and 9 (of 24) data structure benchmarks (which is more than ApproxMC). In all but 8 cases, formulas with Alloy's default symmetry breaking take less time than with CNF-level symmetry breaking. In all but 24 cases, formulas with CNF-level symmetry breaking take less time than with no symmetry breaking. Moreover, for data structure benchmarks, in all but 2 cases, formulas with manual symmetry breaking take less time than Alloy's default symmetry breaking. Among all the problems that time out with no symmetry breaking, the smallest time taken by the corresponding problem with Alloy's default symmetry breaking was 3.12 seconds, and the smallest time taken by the corresponding problem with manual symmetry breaking was 0.01 seconds.
Model counts. Figure 6b graphically illustrates how the model counts vary under different symmetry breaking settings. For the Alloy and Kodkod benchmarks, in all but 9 cases the model count for the formula with Alloy's default symmetry breaking is less than the corresponding count with CNF-level symmetry breaking. For the data structures, the model count for the formula with Alloy's symmetry breaking is less than the corresponding count with CNF-level symmetry breaking in all cases; moreover, in all cases, manual symmetry breaking gives the lowest count. Among all problems where ApproxMC reports a count with no symmetry breaking, the largest ratio of count with no symmetry breaking to count with Alloy's default symmetry breaking was 40320, and the largest ratio of count with no symmetry breaking to count with manual symmetry breaking was 362880.

Overall, the impact of symmetry breaking is significant for both ApproxMC and ProjMC. In majority of the cases, Alloy's default symmetry breaking is more effective than CNF-level symmetry breaking using BreakID. For data structure benchmarks, manual symmetry breaking is the most effective, and reports exactly the counts of the non-isomorphic solutions as desired; moreover, in cases where Alloy's default symmetry breaking provides *full* symmetry breaking, manual symmetry breaking provides much faster solving.

5.3 Discussion

The empirical evaluation in the preceding subsections clearly demonstrates the significant impact of symmetry breaking on ApproxMC and ProjMC. While a detailed study to explain the observed behavior is beyond the scope of this work, we offer some explanations. As pointed out by Soos and Meel [52], over 99% of the runtime of ApproxMC is consumed by the underlying SAT solver handling CNF-XOR formulas. The usage of symmetry breaking predicates for satisfiable instances typically leads to smaller overheads in runtime in the context of satisfiability queries. As discussed above, the use of symmetry breaking predicates significantly reduces the number of solutions and thereby leads to the significant reduction in the number of XORs to be added by ApproxMC. Note that the number of XORs to be added is logarithmically proportional to the number of solutions of a formula. The performance of SAT solvers has been observed to be sensitive to the number of XORs [24] and therefore, we believe that reduction in the required number of XORs is the primary reason behind the performance improvements in the context of ApproxMC.

The performance improvement of ProjMC is, however, more surprising since it is not necessarily the case that reduction in the number of solutions would lead to reduction in the size of the corresponding d-DNNF (decision-Deterministic Decomposable Negation Normal Form), which represents the trace of the execution of ProjMC [33]. Furthermore, given the lack of noticable difference in runtime performance improvement via off-the-shelf symmetry breaking tools, it would be an interesting direction of future work to understand the difference in the traces between the formulas generated via Alloy's default symmetry breaking and CNF-level symmetry breaking.

6 Conclusions

This paper presented, to the best of our knowledge, the first study of symmetry breaking and model counting. A goal of the study was to determine what is the best way to add symmetry breaking predicates (if at all) to obtain precise counts of non-isomorphic solutions. We studied two model counters from two different classes and four scenarios of applying symmetry breaking. A key lesson of our study is that domain-specific symmetry breaking predicates are most effective at enabling precise computation of model counts up to isomorphism. We believe the results of our study can provide insights into more effective use of cutting edge model counters in important domains where the number of unique solutions up to isomorphism is desired, and also enable developing novel model counting methods that exploit symmetries.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation Grant CCF-1718903, and the National Research Foundation Singapore under its AI Singapore Programme [AISG-RP-2018-005].

References

1. Alloy GitHub repository, 2019. `https://github.com/AlloyTools/org.alloytools.alloy`.
2. Alloy models repository, 2019. `https://github.com/AlloyTools/models`.
3. BreakID BitBucket repository, 2019. `https://bitbucket.org/krr/breakid/src/master/`.
4. Kodkod examples repository, 2019. `https://github.com/emina/kodkod/tree/master/examples`.
5. Kodkod GitHub repository, 2019. `https://github.com/emina/kodkod`.
6. The on-line encyclopedia of integer sequences, 2019. `https://oeis.org/`.
7. Alyas Almaawi, Nima Dini, Cagdas Yelen, Milos Gligoric, Sasa Misailovic, and Sarfraz Khurshid. Predictive constraint solving and analysis. In *International Conference on Software Engineering, New Ideas and Emerging Results (ICSE-NIER)*, 2020. To appear.
8. Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *40th Annual Design Automation Conference*, pages 836–839, 2003.
9. Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 255–272, 2015.
10. Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter J. Stuckey. Projected model counting. *CoRR*, abs/1507.07648, 2015.
11. Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. A formal approach for detection of security flaws in the android permission system. *Formal Asp. Comput.*, 30(5):525–544, 2018.
12. Roberto J. Bayardo, Jr., and J. D. Pehoushek. Counting models using connected components. In *In AAAI*, pages 157–162, 2000.
13. Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. *SIGPLAN Not.*, 49(6):123–132, June 2014.
14. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.
15. Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate probabilistic inference via word-level counting. In *Proc. of AAAI*, 2016.
16. Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Proc. of CP*, pages 200–216, 2013.
17. Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proc. of IJCAI*, 2016.
18. Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARM, and C++. *SIGPLAN Not.*, 53(4):211–225, 2018.
19. David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *Electr. Notes Theor. Comput. Sci.*, 59(3):238–251, 2001.
20. James Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In *Workshop notes, AAAI-92 workshop on tractable reasoning*, 1992.

21. James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. *KR*, 96:148–159, 1996.
22. Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Int. Res.*, 17(1):229–264, September 2002.
23. Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *TACAS*, pages 104–122, 2016.
24. Jeffrey Dudek, Kuldeep S. Meel, and Moshe Y. Vardi. Combining the k-cnf and xor phase-transitions. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 7 2016.
25. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.
26. Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *International Conference on Software Engineering*, pages 622–631, 2013.
27. J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Taco: Efficient SAT-based bounded verification using symmetry breaking and tight bounds. *Transactions on Software Engineering*, 2013.
28. Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 166–176, 2012.
29. Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, pages 329–376. 2006.
30. Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. Short XORs for model counting: From theory to practice. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 100–106, 2007.
31. Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *21st National Conference on Artificial Intelligence - Volume 1*, pages 54–61, 2006.
32. Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188, 2011.
33. Jinbo Huang and Adnan Darwiche. Dpll with a trace: From sat to knowledge compilation. In *IJCAI*, volume 5, pages 156–162, 2005.
34. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
35. Daniel Jackson and Kevin J. Sullivan. COM revisited: Tool-assisted modelling of an architectural framework. In *SIGSOFT FSE*, pages 149–158, 2000.
36. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA*, August 2000.
37. Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *ASE*, pages 13–22, 2000.
38. Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *SAT*, pages 272–286, 2003.
39. Seonmo Kim and Stephen McCamant. Bit-vector model counting using statistical estimation. In *TACAS (1)*, pages 133–151, 2018.
40. Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. *AAAI*, 33:1536–1543, 2019.
41. Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. *SIGPLAN Not.*, 49(6):565–576, June 2014.
42. Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.

43. Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. Cdclsym: Introducing effective symmetry breaking in sat solving. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–114. Springer, 2018.
44. Quoc-Sang Phan and Pasquale Malacaria. Abstract model counting: a novel approach for quantification of information leaks. In *9th ACM Symposium on Information, Computer and Communications Security*, pages 283–292, 2014.
45. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, 2009.
46. Karem Sakallah. Symmetry and satisfiability. *Frontiers in Artificial Intelligence and Applications*, 185, 01 2009.
47. Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.
48. Hesam Samimi, Ei Darli Aung, and Todd D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.
49. Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT*, 2004.
50. Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In *IJCAI*, pages 1169–1176, 2019.
51. Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
52. Mate Soos and Kuldeep S. Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 1 2019.
53. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 244–257, 2009.
54. Larry Stockmeyer. The complexity of approximate counting. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 118–126, New York, NY, USA, 1983. ACM.
55. Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Automated test generation and mutation testing for Alloy. In *ICST*, 2017.
56. Marc Thurley. SharpSAT – Counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 424–429, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
57. Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications.* PhD thesis, Cambridge, MA, USA, 2009. AAI0821754.
58. Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.
59. Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. In *MICRO*, 2018.
60. G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. 1983.
61. Alasdair Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96-97:177 – 193, 1999.
62. Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. TestMC: A framework for testing model counters. Under submission, 2020.

63. Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8:410–421, 1979.
64. Guy Van Den Broeck. First-order model counting in a nutshell. In *Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 4086–4089, 2016.
65. Marko Vasic, David Soloveichik, and Sarfraz Khurshid. CRNs exposed: Systematic exploration of chemical reaction networks. *CoRR*, abs/1912.06197, 2019.
66. E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *TSE*, 6(3):236–246, May 1980.
67. John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 190–204, 2017.
68. Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, 2005.
69. Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Contract-based data structure repair using Alloy. In *ECOOP*, pages 577–598, 2010.
70. Pamela Zave. How to make Chord correct (using a stable base). *CoRR*, abs/1502.06461, 2015.

MUST: Minimal Unsatisfiable Subsets Enumeration Tool*

Jaroslav Bendík and Ivana Černá

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbendik,cerna}@fi.muni.cz

Abstract. In many areas of computer science, we are given an unsatisfiable set of constraints with the goal to provide an insight into the unsatisfiability. One of common approaches is to identify minimal unsatisfiable subsets (MUSes) of the constraint set. The more MUSes are identified, the better insight is obtained. However, since there can be up to exponentially many MUSes, their complete enumeration might be intractable. Therefore, we focus on algorithms that enumerate MUSes *online*, i.e. one by one, and thus can find at least some MUSes even in the intractable cases.
Since MUSes find applications in different constraint domains and new applications still arise, there have been proposed several *domain agnostic* algorithms. Such algorithms can be applied in any constraint domain and thus theoretically serve as ready-to-use solutions for all the emerging applications. However, there are almost no domain agnostic tools, i.e. tools that both implement domain agnostic algorithms and can be easily extended to support any constraint domain. In this work, we close this gap by introducing a domain agnostic tool called MUST. Our tool outperforms other existing domain agnostic tools and moreover, it is even competitive to fully domain specific solutions.

Keywords: Minimal unsatisfiable subsets · Unsatisfiability analysis · Infeasibility analysis · MUS · Diagnosis.

1 Introduction

In various areas of computer science, we are given a set C of constraints with the goal to determine whether the set is satisfiable, i.e. whether all the constraints can hold simultaneously. In the case where the set is shown to be unsatisfiable, we are often interested in analysing the unsatisfiability. Identification of minimal unsatisfiable subsets (MUSes) of C is a kind of such analysis. A set $M \subseteq C$ is a MUS of C iff M is unsatisfiable and all proper subsets of M are satisfiable. The more MUSes are identified, the better insight into the unsatisfiability of C is obtained. However, the complete MUS enumeration is often intractable since

* This research was supported by ERDF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No. CZ.02.1.01/0.0/0.0/16_019/0000822).

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 135–152, 2020.
https://doi.org/10.1007/978-3-030-45190-5_8

there can be up to exponentially many MUSes w.r.t. the number of constraints in C. Therefore, several *online* MUS enumeration algorithms (e.g. [3,29,22,1,25,10]) were proposed, i.e. algorithms that identify MUSes gradually, one by one, and thus identify at least some MUSes even in the intractable cases.

Various applications of MUSes arise for example in requirements analysis [4,6], during formal equivalence checking [15], proof based abstraction refinement [23], Boolean function bi-decomposition [12], circuit error diagnosis [21], type debugging in Haskell [30], or proof explanation in symbolic model checking [20]. The domain of the constraint sets ranges from Boolean formulas [23,14], over temporal logic formulas [4,6], to transition state predicates constraining transition systems [20]. Since the list of constraint domains where MUSes find an application is quite long and new applications still arise, there have been proposed several *domain agnostic* MUS enumeration algorithms (e.g. [3,22,9,7,10]). Such algorithms can be used in an arbitrary constraint domain, and thus theoretically serve as ready-to-use solutions for any constraint domain where MUSes might eventually find an application.

Unfortunately, there is no available *domain agnostic tool implementation* of the algorithms that would actually serve as a ready-to-use solution for an arbitrary constraint domain. Although the papers that present existing domain agnostic algorithms provide results of an experimental evaluation, it is often the case that the implementation is either not publicly available [4,3], or there is a hard-coded support for a particular constraint domain [10,20]. The closest to a domain agnostic tool is a tool by Liffiton et al. [22] where the authors implement their domain agnostic MUS enumeration algorithm MARCO. Their tool currently supports the SAT and the SMT domains and can be relatively easily extended to support also another constraint domains. However, our recent evaluation [8] of contemporary domain agnostic algorithms in various constraint domains has shown that the efficiency of the algorithms (including MARCO) varies a lot in different constraint domains. There is no silver bullet algorithm that would be efficient in all the domains. Thus, to deal with a particular constraint domain, one has to wisely choose from individual algorithms.

In this work, we present the first stable release of our domain agnostic tool, called `MUST`, for MUS enumeration. The tool implements several domain agnostic MUS enumeration algorithms and currently provides support for 3 constraint domains: SAT, SMT, and LTL. Moreover, due to a modular architecture of the tool, the tool can be easily extended to support another constraint domain: it requires only to implement an API for communication with a satisfiability solver for the constraint domain. We also provide a guidance on which algorithms are suitable for which kinds of input constraint systems.

To demonstrate the efficiency of our tool, we experimentally compare it to the tool by Liffiton et al. [22] in the SAT and SMT domains, and we show that our tool clearly wins in both the domains. Moreover, we also provide a comparison with two contemporary tools that are tailored to the SAT domain: `MCSMUS` [1] and `FLINT` [25]. The results show that `MUST` is competitive to the two domain specific solutions. Moreover in case of many benchmarks, `MUST` actually significantly dominates the other tools.

Finally, to advocate the practical applicability of our tool in industrial settings, we provide a use case from the area of requirements analysis. In particular, we have employed our tool in the European Unions Horizon 2020 project called AMASS. The project focused on development and verification of cyber-physical systems in the largest industrial markets including automotive, railway, aerospace, space, and energy. One of the verification tasks is to verify that requirements on the system are consistent, i.e., to ensure that there can be even built a system that satisfies the requirements. If the requirements are found to be inconsistent, an identification of minimal inconsistent (unsatisifable) subsets of the requirements helps to fix the conflicts among the requirements. Our tool has proved to be very efficient in dealing with this task.

2 Preliminaries

2.1 Basic Definitions

We are given a set $C = \{c_1, c_2, \dots, c_n\}$ of constraints such that each subset of C is either *satisfiable* or *unsatisfiable*. The notion of satisfiability varies in particular constraint domains. We only assume that if a set N, $N \subseteq C$, is satisfiable then all subsets of N are also satisfiable. Dually, if a set K, $K \subseteq C$, is unsatisfiable then all supersets of K are also unsatisfiable. We will use C to denote the input set of constraints throughout the whole paper.

Definition 1 (MUS). *A subset N of C is a minimal unsatisfiable subset (MUS) of C if and only if N is unsatisfiable and for all $c \in N$ the set $N \setminus \{c\}$ is satisfiable.*

Note that the minimality concept used here is set minimality, not minimum cardinality. Therefore, there can be MUSes with different cardinalities. Also, there can be up to exponentially many MUSes w.r.t. the number of constraints in C (see the Sperner's theorem [28]).

Definition 2 (critical constraint). *Let U be an unsatisfiable subset of C and $c \in U$. The constraint c is* critical *for U if and only if $U \setminus \{c\}$ is satisfiable.*

Note that U is a MUS of C if and only if all constraints in U are critical for U. Furthermore, if c is critical for U then c has to be contained in every MUS of U.

Example 1. We illustrate the concepts on a small example. Assume that we are given a set C of four Boolean satisfiability constraints: $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$. Clearly, the whole set is unsatisfiable as the first two constraints are negations of each other. There are two MUSes: $\{c_1, c_2\}$, $\{c_1, c_3, c_4\}$. As for the critical constraints, we can for example see that c_1 is the only critical constraint for C, and that c_1, c_2 are critical for $\{c_1, c_2, c_3\}$.

Algorithm 1: Domain Agnostic Shrinking

```
input : an unsatisfiable set $S$ of constraints
input : a set $crits$ of constraints that are critical for $S$
output: A MUS of $S$
1 for $c \in S \setminus crits$ do
2     if not CheckSat($S \setminus \{c\}$) then
3         $S \leftarrow S \setminus \{c\}$
4 return $S$
```

2.2 Shrink

Let us now define an operation, called `Shrink`, that is used in our tool to identify individual MUSes.

– `Shrink(`S`, `$crits$`)` takes an unsatisfiable subset S of C together with a set *crits* of constraints that are critical for S and returns a MUS S_{mus} of S.

We say that S is *shrunk* into a MUS S_{mus}. The shrinking is maintained in in our algorithms as a black-box subroutine and thus can be implemented using any available single MUS extraction algorithm. Especially, we can implement the operation using a domain specific solution and thus indirectly exploit domain specific properties of particular constraint domains.

To shed more light on how a shrinking can be done, we describe in Algorithm 1 a domain agnostic single MUS extraction approach that forms the basis of many contemporary domain specific solutions. To find a MUS of S, the algorithm iteratively attempts to remove individual constraints in $S \setminus crits$ from S, checking each new set for satisfiability, and keeping only the changes that preserve S to be unsatisfiable. The most expensive part of the shrinking are the satisfiability checks. In total, the algorithm performs $|S| - |crits|$ satisfiability checks. Domain specific algorithms (e.g. [5,24,1,19]) that are based on Algorithm 1 are often able to further reduce the number of performed satisfiability checks by exploiting domain specific properties of particular constraint domains.

2.3 Unexplored Subsets

Our algorithms for MUS enumeration during their computation gradually *explore* satisfiability of individual subsets of C. The *explored* subsets are those, whose satisfiability is already known by the algorithm whereas *unexplored* subsets are those whose satisfiability is not determined yet. We use *Unexplored* to denote the set of all unexplored subsets of C. Recall that all subsets of a satisfiable set are also satisfiable. Thus, if a set S is determined to be satisfiable, then not just S but also all of its subsets become explored. Dually, if a set U is determined to be unsatisfiable, then all supersets of U become explored. We further classify unexplored subsets as follows:

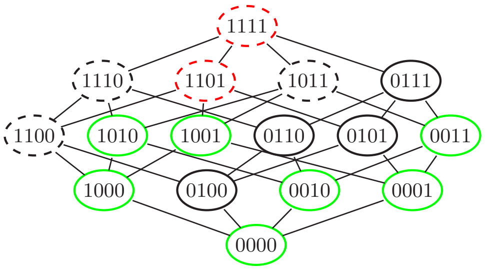


Fig. 1: Illustration of Example 2. We encode individual subsets of C as bit-vectors; for example, the subset $\{c_1, c_3, c_4\}$ is written as 1011.

Definition 3 (Minimal Unexplored Subset). *A set S is a minimal unexplored subset, if S is unexplored and for all $c \in S$ is $S \setminus \{c\}$ explored.*

Definition 4 (Maximal Unexplored Subset). *A set S is a maximal unexplored subset, if S is unexplored and for all $c \in C \setminus S$ is $S \cup \{c\}$ explored.*

Details on how we actually store, maintain, and use unexplored subsets are described later in Section 4.2. Here, we conclude by defining the concept of *minable critical* constraints:

Definition 5 (minable critical). *Let N be an unsatisfiable subset of C such that $N \in Unexplored$, and let $c \in N$. The constraint c is a* minable *critical constraint for N if $N \setminus \{c\} \notin Unexplored$.*

Example 2. Let us illustrate the concepts on an example. Assume that we are given the same set of four constraints as in Example 1: $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$. Fig. 1 shows a possible state of exploration of the power-set. Satisfiable subsets are drawn with a solid border and unsatisfiable ones with a dashed border. There are 2 explored unsatisfiable subsets (red color), 7 explored satisfiable subsets (green color), and 7 unexplored subsets (black color). There are two minimal unexplored subsets: $\{c_2\}$ and $\{c_1, c_3, c_4\}$, and three maximal unexplored subsets: $\{c_1, c_2, c_3\}$, $\{c_1, c_3, c_4\}$ and $\{c_2, c_3, c_4\}$. As for the minable critical constraints, we can for example see that c_2 is minable critical for the set $\{c_1, c_2, c_3\}$, and that all constraints are minable critical for the set $\{c_1, c_3, c_4\}$.

3 Implemented Algorithms

Our tool currently implements three domain agnostic algorithms for online MUS enumeration: MARCO [22], TOME [7], and ReMUS [9]. MARCO was originally developed by Liffiton et al. [22]; the other two algorithms are originally ours. All the three algorithms are based on a common scheme that we call *seed-shrink scheme*. In this section, we first describe the base scheme and then briefly comment also on the individual algorithms.

Algorithm 2: Seed-Shrink Scheme

```
input  : an unsatisfiable set $C$ of constraints
output: All MUSes of $C$
1 $Unexplored \leftarrow \mathcal{P}(C)$
2 while there is a seed do
3     $S \leftarrow$ find a seed
4     $crits \leftarrow$ collect minable critical constraints for $S$
5     $S_{mus} \leftarrow \texttt{Shrink}(S, crits)$
6     $Unexplored \leftarrow Unexplored \setminus \{T \mid T \subset S_{mus} \text{ or } S_{mus} \subseteq T \subseteq C\}$
7     output $S_{mus}$
```

3.1 Seed-Shrink Scheme

The *seed-shrink scheme* is shown in Algorithm 2. The computation starts by initializing the set *Unexplored* to $\mathcal{P}(C)$, i.e. all subsets of C are initially unexplored. Subsequently, the scheme iteratively identifies all MUSes of C. Each iteration starts by finding a so called *seed*, i.e. an unexplored subset that is unsatisfiable. Subsequently, the set *crits* of all constraints that are minable critical for the seed are collected and the shrinking procedure is used to find a MUS of the seed. The iteration is concluded by marking all subsets and supersets of the MUS as explored (the subsets are necessarily satisfiable, and the supersets are unsatisfiable). The computation terminates once there is no more seed.

The scheme does not specify how to find a seed; this part differs for individual algorithms implementing the scheme. In general, to find a seed, the algorithms check several unexplored subsets for satisfiability and reduce the set *Unexplored*. The difference between the algorithms is in *which* and *how many* subsets they check, and *how large* is the resultant seed. In general, the smaller the seed is, the easier is to shrink it. On the other hand, unsatisfiable subsets are naturally more concentrated among the larger subsets, thus looking for a seed among small unexplored subsets might come with the price of checking many unexplored subsets for satisfiability. Individual seed-shrink algorithms make a different trade-off between the size of identified seeds and the number of satisfiability checks that are performed to identify the seeds. In some constraint domains, it is worth to find a small seed even if it requires performing many satisfiability checks, and in other constraint domains the situation is exactly the opposite. The optimal choice of a seed-shrink algorithm thus differs for individual constraint domains.

MARCO [22] searches for a seed S among the maximal unexplored subsets and often performs only few satisfiability checks to identify a seed. Since maximal unexplored subsets are usually very large, the seeds identified by MARCO are generally hard to be shrunk. Yet, in some constraint domains, such as SAT and SMT, the size of the seed has just a negligible effect on the complexity of the shrinking. In particular, in the SAT and SMT domains, contemporary satisfiability solvers can extract an *unsat core* of the seed S, i.e. unsatisfiable, yet not necessarily minimal, subset of S. The extraction comes with almost no overhead compared to an ordinary check for satisfiability, and the unsat core is usually very close, in terms of cardinality, to a MUS of S. Thus, instead of shrinking the whole S, the unsat core is passed to the shrinking procedure.

TOME [7] identifies seeds iteratively as follows. Each iteration of the algorithm starts by picking a minimal unexplored subset N_1 and a maximal unexplored subset N_p such that $N_1 \subseteq N_p$. Subsequently, TOME builds a chain $N_1 \subset N_2 \subset \cdots \subset N_p$ of unexplored subsets. Such a chain necessarily either contains only unsatisfiable subsets, only satisfiable subsets, or it contains an element N_i such that $\forall j$, $1 \leq j < i$, is N_j satisfiable and $\forall k$, $i \leq k \leq p$, is N_k unsatisfiable. In the first case, it is guaranteed that N_1 is a MUS. In the second case, the chain does not give us any seed. Finally, in the third case, TOME finds N_i using binary search (which takes only $\mathcal{O}(\log_2 p)$ satisfiability checks). Subsequently N_i is used as a seed for the shrinking procedure and shrunk into a MUS.

There are no guarantees on distribution of satisfiable and unsatisfiable subsets on the chain, since the subsets are unexplored. In the best case, where N_1 is unsatisfiable, TOME identifies a MUS using just a single satisfiability check. In the worst case, the whole chain is satisfiable and TOME has to build another chain. Based on our experience, TOME on average performs more satisfiability checks to find a seed than MARCO does, but the seeds are much smaller than in the case of MARCO. Thus, TOME is efficient especially in constraint domains where the size of the seed highly affects the complexity of the shrinking.

ReMUS [9] is based on the following observation: if C, C^k, and M are unsatisfiable sets such that $C^k \subseteq C$ and M is a MUS of C^k, then M is necessarily also a MUS of C. Note that the smaller C^k is the smaller seeds are in C^k. ReMUS tends to identify C^k that is very small, yet contains many MUSes, and searches for seeds in C^k. In particular, the very first seed S is found among the maximal unexplored subsets of $C^0 = C$ and then shrunk to a MUS S_{mus}. To find a next seed, ReMUS chooses C^1 such that $S_{mus} \subseteq C^1 \subseteq S$, and searches for a seed S^1 among maximal unexplored subsets of C^1. If a seed S^1 is identified, then it is again shrunk to a MUS S^1_{mus} and again used to reduce the search space, i.e. the a next seed S^2 is searched for in a set C^2 such that $S^1_{mus} \subseteq C^2 \subseteq S^1$. The search space reduction is recursively repeated as long as possible. Once the current search space is completely explored, ReMUS backtracks from the recursion and searches for a seed on the previous recursion level. Moreover, ReMUS employs several heuristics to pre-emptively backtrack from a search space that contains a lot of unexplored subsets but only few MUSes.

The larger the input set C of constraints is, the more extensive recursive reduction is possible, and thus the smaller seeds can be found. We recommend to use ReMUS, rather than MARCO or TOME, if the input constraint set contains at least hundreds of constraints and hundreds of MUSes, no matter what the constraint domain is.

For a more elaborated description of the three algorithms, please refer to the original papers [22,7,9] or to our recent work [8] where we have experimentally compared the algorithms in various constraint domains.

4 Architecture of the Tool

Our tool is implemented in C++ and is available under the MIT license at:

https://github.com/jar-ben/mustool

The tool consists of six logical components: *SatSolver*, *Explorer*, *Master*, *Algorithms*, *Heuristics*, and *Initializer*. In the following section 4.1 we provide a brief description of the individual components. Subsequently, in Sections 4.2 and 4.3 we provide a more detailed description of *Explorer* and *SatSolver*. Finally, in Section 4.4, we give instructions on how to install and use our tool.

4.1 Logical Components

SatSolver *SatSolver* (declared in *SatSolver.h*) is the only domain specific part of our tool. It provides the functionality for checking sets of constraints for satisfiability, and implements the shrinking procedure. Also, *SatSolver* copes with parsing the input set of constraints (provided by the user) and exporting the identified MUSes in particular domain specific formats. A more detailed description of *SatSolver* is provided in Section 4.3.

Explorer *Explorer* (declared in *Explorer.h*) maintains the set *Unexplored* of all unexplored subsets and handles related operations including: marking sets as explored, obtaining unexplored subsets, and mining critical constraints based on the set *Unexplored*. For more information, see Section 4.2.

Master *Master* (declared in Master.h) is the coordinator of the whole computation. In particular, it holds an instance of Explorer and an instance of SatSolver and provides wrappers for calling their methods. Moreover, it runs a MUS enumeration algorithm that is specified by the user via a command line argument (see below).

Algorithms The algorithms MARCO [22], TOME [7], and ReMUS [9] are declared in Master (Master.h) and implemented in marco.cpp, tome.cpp, and remus.cpp, respectively. All calls to SatSolver and Explorer are made via the wrappers defined in Master. This means that any improvement to Explorer and especially to SatSolver (i.e. a more efficient shrinking procedure or satisfiability solver) is immediately reflected by all the algorithms.

Heuristics There are several heuristics that are bound to the wrappers defined in Master, and thus can be exploited by all the three algorithms. For example, in the wrapper for invoking the shrinking procedure, we provide two heuristics for computing critical constraints for the set that is being shrunk. One of the two heuristics uses Explorer to compute critical constraints based on the set *Unexplored*. The other heuristic uses SatSolver to obtain additional critical constraints that cannot be mined from *Unexplored*.

Initializer *Initializer* (implemented in main.cpp) parses the command line arguments provided by the user, and creates, sets-up, and runs the Master.

4.2 Explorer

Since there can be up to exponentially many unexplored subsets w.r.t. the number of constraints in C, it is intractable to represent them explicitly. Instead, we adopt a symbolic representation that was first proposed by Liffiton et al. [22] and subsequently used in many other works (e.g. [1,20,10]).

Given a set $C = \{c_1, c_2, \dots, c_n\}$ of constraints, we introduce a set $X = \{x_1, x_2, \dots, x_n\}$ of Boolean variables, and maintain two Boolean formulas, map^+ and map^-, over X such that each model of $map^+ \wedge map^-$ corresponds to an unexplored subset and vice versa. The formulas are maintained as follows:

- Initially $map^+ = map^- = True$ since all of $\mathcal{P}(C)$ are unexplored.
- To mark a satisfiable set $N \subseteq C$ and all its subsets as explored we add to map^+ the clause $\bigvee_{i:c_i \notin N} x_i$.
- Symmetrically, to mark an unsatisfiable set $N \subseteq C$ and all its supersets as explored we add to map^- the clause $\bigvee_{i:c_i \in N} \neg x_i$.

We use the SAT solver miniSAT [18] to hold and query the formulas map^+ and map^-. To get an arbitrary element of *Unexplored*, we can ask miniSAT for a model of $map^+ \wedge map^-$. However, in our algorithms, we need to be able to obtain two specific kinds of unexplored subsets.

First, given a set N, $N \subseteq C$, we need to be able to find a maximal unexplored subset of N. We exploit that miniSAT allows the user to fix values of some variables and also to set the default *polarity* of variables, i.e. the default value assignment to variables in decision points during the solving. To get a maximal unexplored subset of N, we fix the values of the variables $\{x_i | c_i \notin N\}$ to *False*, set the default polarity to *True*, and ask miniSAT for a model of $map^+ \wedge map^-$.

Second, given an *unexplored* N, $N \subseteq C$, we need to find a minimal unexplored subset B of N (this is used by TOME while constructing a chain of unexplored subsets). To do this, we fix the values of the variables $\{x_i | c_i \notin N\}$ to *False*, set the default polarity to *False*, and ask miniSAT for a model of map^+. Note that we do not include map^- in the query. Intuitively, map^- requires an absence of constraints and since N satisfies map^-, every subset of N also satisfies map^-.

As for the implementation, we integrate miniSAT via it's C API and we maintain two instances of the solver. One instance holds the formula $map^+ \wedge map^-$ whereas the other instance holds just map^+. Both the instances are used incrementally, i.e. the formulas are incrementally build during the whole MUS enumeration and simplified (internally by miniSAT) when possible. Let us note that Liffiton et al. also incrementally use miniSAT in their tool[1]. However, they maintain just the whole conjunction $map^+ \wedge map^-$ since a separate maintenance of map^- or map^+ would not bring any speed-up in case of their MUS enumeration algorithm.

[1] https://sun.iwu.edu/%7eliffito/marco/

Finally, Explorer provides one more functionality. Given an unexplored subset N, Explorer can collect minable critical constraints of N. Recall that a constraint $c \in N$ is minable critical for N iff $N \setminus \{c\}$ is explored. All the minable critical constraints can be determined based on the formula $map^+ \wedge map^-$. In particular, if we simplify the formula by fixing the variables $\{x_i | c_i \notin N\}$ to *False*, then values of some variables from $\{x_i | c_i \in N\}$ will be *implied* to be *True*. These implicants correspond to the minable critical constraints. This observation has been already exploited by Liffiton et al. [22] and they use miniSAT to obtain the implicants in their tool. However, the miniSAT's procedure for computing the implicants is not dedicated solely to this purpose; it is optimized w.r.t. the overall satisfiability solving process. Therefore, a use of miniSAT for this task brings an unnecessary overhead. In our tool, we directly compute the implicants from the formula map^+ instead of using a SAT solver to do it.

4.3 SatSolver

SatSolver (declared in *SatSolver.h*) is an abstract class stating all the domain specific functionality that needs to be implemented (in a derived class) to support a particular constraint domain in our tool. There are three methods that have to be implemented by every derived class:

- `toString(`N`)` takes as an input a set N, $N \subseteq C$, and returns a textual representation of the constraints contained in N (e.g. in the SMT-LIB 2 format if N is a set of SMT constraints). We use this method to output the identified MUSes.
- `solve(`N`, ` $core = False$`, ` $extension = False$`)` takes as an input a subset N of C and returns *True* iff N is satisfiable and *False* otherwise. Moreover, `solve` takes two optional Boolean parameters, *core* and *extension*, with default values set to *False*. If *core* is set to *True* and N is unsatisfiable, `solve` also finds an *unsat core* of N, i.e. an unsatisfiable M such that $M \subseteq N$. Similarly, if *extension* is set to *True* and N is satisfiable, `solve` finds an *extension* of N, i.e. a satisfiable set M such that $N \subseteq M \subseteq C$. We use the unsat cores in our tool to reduce seeds before shrinking. The extensions are used to further prune the set *Unexplored* when an unexplored subset is found to be satisfiable.
- `constructor(`*filepath*`)`. Every derived class of SatSolver has to implement its constructor. The constructor accepts a path *filepath* to a file that specifies the input set C of constraints in some domain specific format (e.g. SMT-LIB 2 for SMT formulae). We invoke the constructor during the initialisation phase of our tool and its goal is to parse the input set of constraints and internally store the constraints for future manipulations. SatSolver is the only one of the six logical components of our tool that directly works with particular constraints of C. All the other components work just with a bitvector representation of subsets of C. For example, if $C = \{c_1, c_2, c_3, c_4\}$ is a set of four constraints and $K = \{c_1, c_2\}$, the bitvector representation of K is 1100. Therefore, whenever another component communicates with SatSolver,

e.g. invokes the procedure `solve(`N`)`, it passes the bit-vector representation of N to SatSolver and SatSolver converts it to particular constraints.

Besides the above three methods that have to be implemented by every derived class, SatSolver defines and implements a method that can be overridden by a derived class:

– `shrink(`N`,` *crits*`)` performs the shrinking, i.e. it takes an unsatisfiable set N together with a set *crits* of constraints that are critical for N and returns a MUS of N. The default domain agnostic implementation of this method is carried out by Algorithm 1 (Section 2.2).

Currently, our tool supports 3 constraint domains via the following 4 derived classes of SatSolver:

– **MSHandle** (implemented in *MSHandle.cpp*) provides a functionality for the Boolean CNF domain, i.e. the set of constraints is a set of Boolean clauses. The input and output format is the DIMACS CNF format. For shrinking, we integrate two single MUS extraction tools: muser2 [5] by Belov and Silva, and a tool [1] by Bacchus and Katsirelos. Finally, we use miniSAT [18] to implement the method `solve`. Besides checking N for satisfiability, we also use miniSAT to obtain an unsat core or an extension of N. In particular, an unsat core is directly provided by miniSAT. To get an extension of N, we obtain a model π of N from miniSAT and collect the set $\{c | c \in C \wedge \pi \models c\}$ of all constraints in C that are satisfied by π.
– **Z3Handle** (implemented in *Z3Handle.cpp*) processes SMT constraints that are represented in the SMT-LIB2 format. We use z3 [16] to parse the input and to implement `solve`. Moreover, in the same way as in the case of MSHandle, we obtain unsat cores from z3 and we also obtain models of satisfiables formulas to compute their extensions. The shrinking is implemented using our custom procedure.
– **SpotHandle** (implemented in *SpotHandle.cpp*) supports the LTL domain. We use SPOT [17] to implement `solve` and the default domain agnostic implementation of `shrink`. In this case, we do not provide support for computing *non-trivial* unsat cores and *non-trivial* extension. Therefore, if an extension or unsat core is required while calling `solve(`N`)`, we simply use N itself (N is a trivial unsat core/extension of N).
– **NuxmvHandle** (implemented in *NuxmvHandle.cpp*) is another alternative for the LTL domain. Instead of SPOT, it uses nuXmv [11] as a satisfiability solver, which is, based on our experience, much more efficient than SPOT. However, nuXmv's license[2] is more restrictive than the SPOT's license and thus not every user of our tool might use it. In this case, we also do not support an extraction of non-trivial unsat cores and extensions.

If anyone wants to add support for another constraint domain to our tool, it is enough to implement a derived class of SatSolver. For example, the implementation of SpotHandle takes only 45 lines of code, including several empty lines

[2] https://es-static.fbk.eu/tools/nuxmv/index.php?n=Main.License

caused by formatting and lines containing only closing brackets ("}"). Therefore, we claim our tool to be indeed domain agnostic and ready-to-use solution for any constraint domain.

4.4 Installation and Execution of the Tool

For detailed installation and usage instructions, please follow the README.md file at: https://github.com/jar-ben/mustool.

Briefly, our tool can be built either in lightweight settings with support only for SAT domain, or with support also for the SMT and/or LTL domains. Whereas in the SAT domain, we use miniSAT that can be built very quickly, the z3 and SPOT solvers that we use in the SMT and LTL domains can take several hours to install. Once you have installed all the solvers you want to use, our tool can be simply built with an invocation of the command "make".

To run our tool in its default settings, execute:

```
./must input_file,
```

where `input_file` specifies the input file of constraints, and it has to have either .cnf, smt2, or .ltl extension. Based on the extension, Master selects and uses an appropriate derived class of SatSolver. To specify a MUS enumeration algorithm to be used, invoke the tool by:

```
./must -a alg input_file,
```

where `alg` can be either `marco`, `tome`, or `remus` (the default one). To see all the available settings, run

```
./must -h.
```

5 Experimental Evaluation

5.1 Evaluated Tools

The only other existing MUS enumeration tool that can be seen as domain agnostic is the implementation[3] of the domain agnostic algorithm MARCO (invented by Liffiton et al. [22] and implemented by Liffiton and Zhao). In the following, we refer to the tool as `MARCO`. Currently, `MARCO` supports the SAT and SMT domains and can also relatively easily be extended to support another constraint domains. Here, we provide results of an experimental comparison of our tool `MUST` with `MARCO` in both the SAT and SMT domains. Moreover, to demonstrate that our domain agnostic tool can be competitive even to fully domain specific solutions, we include a comparison with two state-of-the-art MUS enumeration tools from the SAT domain: `MCSMUS`[4] [2] and `FLINT`[5] [25].

Due to the space limitation, we show here only results achieved by the best (default) configurations of our tool. In particular, in both domains, we use the

[3] https://sun.iwu.edu/%7emliffito/marco/
[4] https://bitbucket.org/gkatsi/mcsmus/src
[5] The tool was kindly provided to us by its author, Nina Narodytska.

algorithm ReMUS. As for the shrinking, in SMT domain, we use our custom shrinking solution, and in the SAT domain we employ a single MUS extraction algorithm by Bacchus and Katsirelos [1]. Complete results of the evaluation are available at: https://www.fi.muni.cz/%7exbendik/research/must.

All experiments were run using a time limit of 3600 seconds and computed on an Intel(R) Core(TM) i5-4690 CPU, 3.50GHz, 16 GB memory machine running Arch Linux 4.19.69-1-lts. The comparison criterion used in our evaluation is the number of identified MUSes within the given time limit.

5.2 Benchmarks

In the SAT domain, we used a collection of 291 Boolean CNF benchmarks that were taken from the MUS track of the SAT 2011 Competition[6]. This collection has been used in many recent MUS related papers (e.g. [22,7,9,25,2]), including the ones that present `MARCO`, `FLINT`, and `MCSMUS`. The benchmarks range in their size from 70 to 16 million constraints and use from 26 to 4.4 million variables. In case of 28 benchmarks, all the evaluated algorithms identified all the MUSes within the given time limit. Since the comparison criterion of our evaluation is the number of identified MUSes, the 28 benchmarks are irrelevant for the evaluation (all three tools found the same number of MUSes for these benchmarks). Therefore, only the remaining 263 benchmarks are the subject of our evaluation.

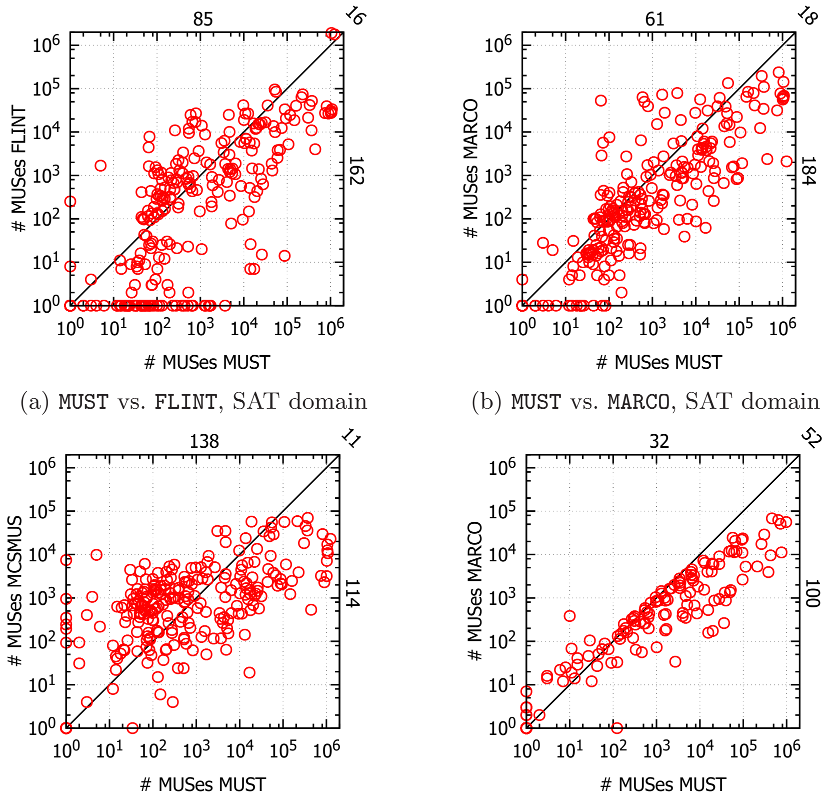
In the SMT domain, we used a collection of 433 benchmarks that were taken from the QF_UF, QF_IDL, QF_RDL, QF_LIA and QF_LRA divisions of the library SMT-LIB[7]. Also this collection has been already used in several works, e.g. in the work by Cimatti et al. [13] or in our recent papers [9,8]. The benchmarks range in their size from 70 to 16 million constraints and use from 26 to 4.4 million variables. In case of 249 benchmarks, both the evaluated algorithms identified all the MUSes. Therefore, we focus here on the remaining 184 benchmarks.

5.3 Results

In Figs. 2a, 2b, and 2c, we provide scatter plots that compare pair-wise `MUST` with the other tools in the SAT domain, and in Fig. 2d a scatter plot comparing `MUST` with `MARCO` in the SMT domain. Each point in a scatter plot corresponds to a single benchmark and shows the number of MUSes identified by the two algorithms. The x-coordinate of a point is given by the algorithm that labels the x-axis and the y-coordinate is given by the algorithm that labels the y-axis. Moreover, note that each scatter plot contains three additional numbers that are above/on right/in the right corner of the plot. These numbers show the number of points that are above/below/on the diagonal, respectively.

In the SMT domain, `MUST` conclusively dominates `MARCO`: it found more, less, and the same number of MUSes as `MARCO` in case of 100, 32, and 52 benchmarks, respectively. In the SAT domain, `MUST` outperforms on majority of benchmarks

[6] http://www.cril.univ-artois.fr/SAT11/
[7] http://www.smt-lib.org/



(a) `MUST` vs. `FLINT`, SAT domain

(b) `MUST` vs. `MARCO`, SAT domain

(c) `MUST` vs. `MCSMUS`, SAT domain

(d) `MUST` vs. `MARCO`, SMT domain

Fig. 2: Scatter plots comparing the number of produced MUSes.

both `MARCO` and `FLINT`. Finally, `MCSMUS` outperforms `MUST` in case of 52 percent of benchmarks and is worse than `MUST` in case of 43 percent of benchmarks. Still, this is a very good result since `MUST` is a domain agnostic tool whereas `MCSMUS` is tailored to the SAT domain.

Besides the pair-wise comparison of the algorithms, we also provide an overall ranking of the algorithms on individual benchmarks in the SAT domain. In particular, assume that for a benchmark B both `MUST` and `MCSMUS` found 100 MUSes, `FLINT` found 80 MUSes, and `MARCO` found 50 MUSes. In such a case, `MUST` and `MCSMUS` share the 1st (best) rank for B , `FLINT` is 3rd, and `MARCO` is on the 4th position. In Fig. 3 we show the average ranking (from all benchmarks) of all algorithms for each subsequent 60 seconds of the computation. We can see that `MARCO` ranked the worse during the whole computation. `FLINT` ranked quite well during the first 600 seconds, but then its performance degraded. Finally, `MUST` and `MCSMUS` maintained the best and the second best ranking, respectively. This might be quite surprising since `MCSMUS` is slightly better than `MUST` in Fig. 2c.

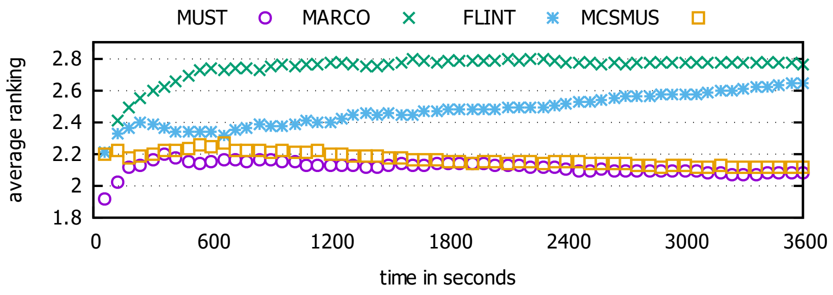


Fig. 3: Average ranking in time.

The thing is that MUST mostly ranks either as 1st or 2nd on a benchmark and rarely ranks as 4th, whereas MCSMUS more often ranks as 3rd or 4th.

Finally, let us recall that our tool contains also implementation of the algorithm MARCO and thus one might be interesting in comparing the performance of MARCO in our tool and MARCO in the tool MARCO. In the SAT domain, we found our implementation to be more efficient, equal, and less efficient than MARCO in case of 68, 6, and 26 percent of benchmarks, respectively. In the SMT domain, our implementation is better, equal, and worse in 37, 29, 34 percent of benchmarks, respectively[8]. Therefore, shall anyone want to use the algorithm MARCO, we recommend to use our implementation.

6 Case Study

During the last 4 years, we participated on the European Union's Horizon 2020 project called AMASS [26]. The project brought together researchers from academia and engineers from large industrial companies such as Honeywell, Alstom, or Infineon. The project focused on improving the process of development and certification of Cyber-Physical Systems in markets such as automotive, railway, aerospace, space, and energy. Among others, this included the development of techniques for assessing quality of system specification/requirements and this is where our tool found an application.

Establishing the requirements is an important stage in all development. In general, the requirements can be expressed either informally, e.g. using a natural language, or formally by employing a kind of mathematical logic such as the Linear Temporal Logic (LTL). The formalization removes ambiguity and allows to employ various model-based techniques, such as model checking. Moreover, we get the opportunity to verify the requirements earlier, even before any system model is built. In particular, we can verify that the requirements are consistent (satisfiable), i.e. that there can be even built a system that satisfies all the requirements. If the requirements are inconsistent, they need to be refined.

[8] See the appendix https://www.fi.muni.cz/%7exbendik/research/must

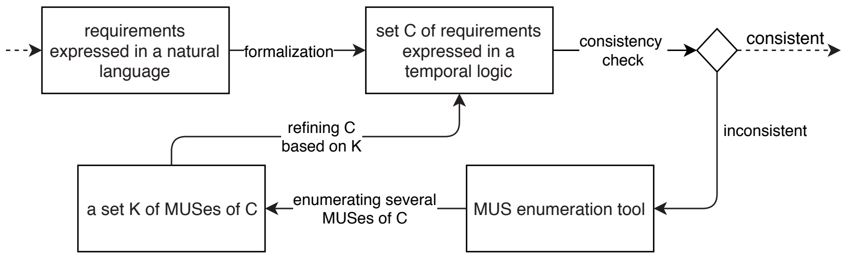

requirements expressed in a natural language
formalization
set C of requirements expressed in a temporal logic
consistency check
consistent
inconsistent
refining C based on K
a set K of MUSes of C
enumerating several MUSes of C
MUS enumeration tool

Fig. 4: Application of MUS enumeration in requirements analysis.

Within the AMASS project, we proposed a scheme [6] that exploits MUSes to help the user to establish a consistent set of requirements. A basic workflow of the scheme is depicted in Fig. 4. The process starts by introducing a set of requirements in some natural-language like format, yet using a restricted grammar that avoids ambiguities. In the next step, the requirements are formalized using LTL and gathered in a set C. Subsequently, C is checked for consistency. If C is consistent, then the software development process can continue with a next stage. Otherwise, a MUS enumeration tool is used to identify a set K of MUSes of C, and the user uses K to refine C. The MUS identification and refinement steps are repeated until the set of requirements becomes consistent.

We implemented the scheme in AMASS as a part of a so-called V&V manager [27]: a tool for validation and verification of the system model and system requirements. Our industrial partners employed the scheme on a set of industrial benchmarks, and evaluated two contemporary MUS enumeration tools from the LTL domain: our `MUST`, and `Looney` by Bauch et al. [4]. They found `MUST` to be faster by several orders of magnitude. Unfortunately, the industrial benchmarks are confidential and cannot be published in this paper. Yet, authors of `Looney` indeed acknowledge in their paper that `Looney` can handle only small input constraint sets containing just low tens of constraints. On the other hand, `MUST` was shown [8] to be able to efficiently work with hundreds of constraints.

7 Conclusion

We presented a tool, called MUST, for online enumeration of Minimal Unsatisfiable Subsets (MUSes). MUST implements three contemporary *domain agnostic* MUS enumeration algorithms, i.e. algorithms that can be applied in any constraint domain. Currently, the tool supports enumeration in the SAT, SMT and LTL domains, and can be easily extended to support another domains. Therefore, we classify the tool itself as *domain agnostic*; it serves as (an almost) ready-to-use solution for any domain where MUSes already find or eventually will find an application. We experimentally compared MUST to a domain agnostic tool by Liffiton et al. [22] in the SAT and SMT domains, and we showed that MUST conclusively dominates in both domains. Moreover, we showed that MUST is even competitive to contemporary tools that are tailored for the SAT domain.

References

1. Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV (2)*, volume 9207 of *LNCS*, pages 70–86. Springer, 2015.
2. Fahiem Bacchus and George Katsirelos. Finding a collection of MUSes incrementally. In *CPAIOR*, volume 9676 of *LNCS*, pages 35–44. Springer, 2016.
3. James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186. Springer, 2005.
4. Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. Analysing sanity of requirements for avionics systems. *FAoC*, 2016.
5. Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *JSAT*, 8:123–128, 2012.
6. Jaroslav Bendík. Consistency checking in requirements analysis. In *ISSTA*, pages 408–411. ACM, 2017.
7. Jaroslav Bendík, Nikola Beneš, Ivana Černá, and Jiří Barnat. Tunable online MUS/MSS enumeration. In *FSTTCS*, volume 65 of *LIPIcs*, pages 50:1–50:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
8. Jaroslav Bendík and Ivana Černá. Evaluation of domain agnostic approaches for enumeration of minimal unsatisfiable subsets. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 131–142. EasyChair, 2018.
9. Jaroslav Bendík, Ivana Černá, and Nikola Beneš. Recursive online enumeration of all minimal unsatisfiable subsets. In *ATVA*, volume 11138 of *LNCS*, pages 143–159. Springer, 2018.
10. Jaroslav Bendík, Elaheh Ghassabani, Michael W. Whalen, and Ivana Černá. Online enumeration of all minimal inductive validity cores. In *SEFM*, volume 10886 of *LNCS*, pages 189–204. Springer, 2018.
11. Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *CAV*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.
12. Huan Chen and João Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In *VLSI-SoC*, pages 142–147. IEEE, 2011.
13. Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *JAIR*, 40:701–728, 2011.
14. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
15. Orly Cohen, Moran Gordon, Michael Lifshits, Alexander Nadel, and Vadim Ryvchin. Designers work less with quality formal equivalence checking. In *Design and Verification Conference (DVCon)*. Citeseer, 2010.
16. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
17. Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 - A framework for LTL and ω-automata manipulation. In *ATVA*, volume 9938 of *LNCS*, pages 122–129, 2016.
18. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

19. Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient generation of inductive validity cores for safety properties. In *SIGSOFT FSE*, pages 314–325. ACM, 2016.
20. Elaheh Ghassabani, Michael W. Whalen, and Andrew Gacek. Efficient generation of all minimal inductive validity cores. In *FMCAD*, pages 31–38. IEEE, 2017.
21. Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by cs-trees with mark set in diagnosis from first principles. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999.
22. Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, pages 1–28, 2015.
23. Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, volume 2619 of *LNCS*, pages 2–17. Springer, 2003.
24. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *JSAT*, 9:27–51, 2014.
25. Nina Narodytska, Nikolaj Bjørner, Maria-Cristina Marinescu, and Mooly Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361. ijcai.org, 2018.
26. AMASS project partners. Project AMASS (Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems). https://amass-ecsel.eu/. [Online; Accessed: 2019-22-10].
27. AMASS project partners. Project AMASS, deliverable D3.6: Prototype for Architecture-Driven Assurance (c). https://amass-ecsel.eu/content/deliverables. [Online; Accessed: 2019-22-10].
28. Emanuel Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.
29. Roni Tzvi Stern, Meir Kalech, Alexander Feldman, and Gregory M. Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI*. AAAI Press, 2012.
30. Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in haskell. In *Haskell*, pages 72–83. ACM, 2003.

Timed and Dynamical Systems

Safe Decomposition of Startup Requirements: Verification and Synthesis

Alessandro Cimatti[1], Luca Geatti[1,2], Alberto Griggio[1], Greg Kimberly[3], and Stefano Tonetta[1]

[1] Fondazione Bruno Kessler, Trento, Italy
cimatti@fbk.eu, lgeatti@fbk.eu, griggio@fbk.eu, tonettas@fbk.eu
[2] University of Udine, Udine, Italy
luca.geatti@uniud.it
[3] The Boeing Company, Seattle, USA
greg.kimberly@boeing.com

Abstract. The initialization of complex cyber-physical systems often requires the interaction of various components that must start up with strict timing requirements on the provision of signals (power, refrigeration, light, etc.). In order to safely allow an independent development of components, it is necessary to ensure a safe decomposition, i.e. the specification of local timing requirements that prevent later integration errors due to the dependencies.

We propose a high-level formalism to model local timing requirements and dependencies. We consider the problem of checking the consistency (existence of an execution satisfying the requirements) and compatibility (absence of an execution that reaches an integration error) of the local requirements, and the problem of synthesizing a region of timing constraints that represents all possible correct refinements of the original specification. We show how the problems can be naturally translated into a model checking and synthesis problem for timed automata with shared variables. Exploiting the linear structure of the requirements, we propose an encoding of the problem into SMT. We evaluate the SMT-based approach using MathSAT and show how it scales better than the automata-based approach using Uppaal and nuXmv.

1 Introduction

Complex industrial cyber-physical systems often have an initialization procedure that requires to reach a startup mode within a specified design target time interval. In order for the system as a whole to complete the startup within the required interval, each subcomponent of the system may have to go through a number of intermediate phases, within their own target intervals, each of which may itself be dependent upon other subcomponents reaching startup or intermediate phases. E.g. for a power generation system to startup at full power, it may need to transition first through a low power output phase and a number of subsidiary systems (perhaps cooling or fuel supply) may first have to undergo their own phase transitions. In turn, these subsidiary systems may require transitions to occur in systems subsidiary to them and so on.

Traditionally, the integration of these distributed transition targets are validated via simulation and testing, which while sufficient to reach a desired design performance are labor and time intensive. Having a more efficient process for arriving at and validating a set of design targets that satisfy the overall system requirements is clearly beneficial in these contexts. Firstly, we would like to verify that these requirements prevent failed transitions in which the time performance of the subsidiary systems lead to outcomes where our main system (e.g., the power generation system) cannot perform a transition within its time window. For example, suppose the power system has a time window within which it must transition from low-power mode to high-power mode; in order for it to achieve this transition, however, it requires that two subsidiary systems, a cooling system and a fuel supply system, must themselves transition from a low-output mode to a high-output mode, each within their own target transition time windows. If these time windows are not compatible, the power generator may fail to provide the high power in time. Secondly, if our starting set of requirements is inadequate to provide this guarantee, we would like to be able to synthesize a set of requirements that is adequate to this task.

In this paper, we formalize the problem starting from a simple industrially relevant setting, where the components have a linear sequence of phases, must progress to the next phase within a certain interval of time, and must respect some dependencies upon the phases of other components. Dependencies are expressed as Boolean combinations of variables representing the component phases and are divided into two types: (i) *signal dependencies*, where the entering of a component into a phase is conditioned by the presence of other components in some specific phases; (ii) *state dependencies*, where a component can stay in a phase only if, during all its stay, other components are in some specific phases. We are interested in the following problems: 1) checking if the requirements are compatible, *i.e.*, if all reachable states can be extended with an execution satisfying the requirements; thus, if the components satisfy the local requirements, they cannot lead the system to an illegal state (where a component does not receive the input in time); 2) checking if the requirements are consistent, *i.e.*, there exists an execution of the components satisfying all requirements (inconsistency is actually a pathological case of incompatibility); 3) synthesizing the set of refinements (same requirements with stricter intervals) that are consistent and compatible. We show how the first two verification problems can be naturally translated into a model checking problem for timed automata with shared variables. Exploiting the linear structure of the requirements, we propose an encoding of the problem into SMT. If all intervals are bounded, the encoding is quantifier-free. Finally, both approaches have been extended to solve also the synthesis problem, using synthesis for parametrized model checking of TAs and quantifier elimination in SMT, respectively.

We implemented the SMT-based approach in a tool called TRICker and carried out experimental evaluation, comparing it with other tools for the verification of timed automata. We used Uppaal [6] and nuXmv [7] to model check TAs and MathSAT [12] to solve the SMT problems. We performed an experimental evaluation based on a test-set of randomly generated local requirements. When comparing the SMT-based approach with the automata-based one, the results highlight a better performance of the former technique on all three problems.

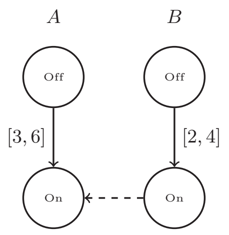
Related Work The problem of the integration and compatibility of input/output timed automata has been extensively studied in the literature. Typically, works in the literature focus on deadlock checking (see, e.g., [4,5]). The work of [2] also addresses the parameter synthesis to avoid deadlocks in timed automata. In order to check for livelocks, liveness properties can be addressed with approaches proposed in [10,7]. A general definition of illegal states for timed interface automata is given in [13]. As shown in the extended version of the paper the compatibility problem addressed in this paper can be seen as a subcase of the homonym problem for input/output timed interface automata. As we are considering a closed system, the problem reduces to the existence of a deadlock or livelock in a phase of some component (depending if the related time interval is bounded or not). Moreover, compared to the above model checking approaches we are considering a specific fragment of timed automata with a linear structure that can be exploited for specialized solutions.

Related problems have been addressed in the context of task scheduling. In the formalism introduced in [16,17], called DRT (short for *digraph real-time task* model), in which tasks and deadlines are expressed as directed graphs, the problem of determining whether a schedule exists (*feasibility problem*) bears some similarities with the consistency checking problem we study here. The DRT model allows the use of very general graph topologies, with multiple outgoing branches and loop-backs, but it does not consider dependencies across different tasks. The main difference with our work is that the problem is addressed from a *global* point of view (*i.e.*, the existence of a global scheduler that can coordinate the execution of the tasks), whereas we are interested in local solutions, in which each requirement can be considered in isolation. Another difference is the approach used to tackle the problem: while in [16] dynamic programming is used to deal with the possible explosion of the search space, we use SMT [14] as the main framework for all the three above-mentioned problems.

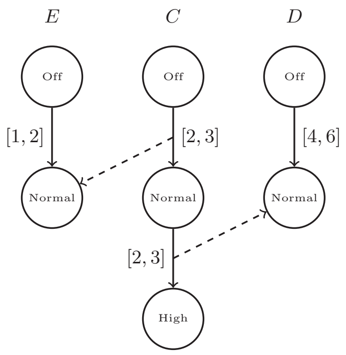
Outline. In Sec. 2, we introduce a suitable formalism to model local requirements and we formalize the three problems. In Sec. 3, we propose the reductions of *compatibility checking* and *consistency checking* into TAs and SMT. The corresponding solutions for the *synthesis* problem are then described in Sec. 4. The experimental results are described in Sec. 5. In Sec. 6, we draw some conclusions and highlight possible future directions of this work.

2 Problem Statement

Domain formalization We propose a high level formalism to model the local requirements.



(a) Example of system with two local requirements and one state dependency.



(b) Example of system with two local requirements and two signal dependencies.

Definition 1 (Local Requirements) *A specification S is given by a set of local (or component) requirements, where each local requirement $C \in S$ is given by an (ordered) sequence $\langle P_1^C, \ldots, P_n^C \rangle$ of phases. In turn, each phase P_i of C is associated when $i > 1$ with a closed real interval β_{P_i} with non-negative lower limit l_{P_i} and (finite or infinite) upper limit u_{P_i}, with a formula ϕ_{P_i} (called* signal dependency*) and, when $i > 0$ with a formula ψ_{P_i} (called* state dependency*). Both ϕ_{P_i} and ψ_{P_i} are Boolean formulae over the atoms in $\{\langle D, Q \rangle\}_{D \in S \setminus \{C\}, Q \in D}$ (i.e., the phases of other components).*

If a dependency ψ_P is just a conjunction of atoms, then we say that ψ_P is *convex*. With the notation $|C|$, we will refer to the number of phases of C.

Figs. 1a and 1b show two examples of sets of local requirements. In Fig. 1a, we have two local requirements A and B (i.e., $S = \{A, B\}$); each local requirement has two phases *Off* and *On* (i.e., $P_1^A = \mathit{Off}$ and $P_2^A = \mathit{On}$ and similarly for B); the bounds are depicted in square brackets (thus, for example $\beta_{On}^A = [3, 6]$); all dependencies are trivially true apart from the state dependency $\psi_{On}^B = \langle A, \mathit{On} \rangle$ of the local requirement B, which is plotted as an arrow from the phase *On* of B to phase *On* of A. In Fig. 1b, we have another example with three components and some signal dependencies; for example, signal dependency $\phi_{Normal}^C = \langle E, \mathit{Normal} \rangle$ is plotted as an arrow from the *transition* to phase *Normal* of C to phase *Normal* of E.

Definition 2 (Stronger local requirements) *We say that a local requirement $C' = \langle P_1^{C'}, \ldots, P_n^{C'} \rangle$ is* stronger *than $C = \langle P_1^C, \ldots, P_n^C \rangle$ (written $C' \preceq C$), iff phase $P_i^{C'}$ is identical to P_i^C except that $l_{P_i^C} \leq l_{P_i^{C'}}$ and $u_{P_i^{C'}} \leq u_{P_i^C}$, for all $1 \leq i \leq n$. Given two specifications $S = \{C_1, \ldots, C_n\}$ and $S' = \{C_1', \ldots, C_n'\}$, we say that S' is stronger than S (written $S' \preceq S$) iff for all i, $1 \leq i \leq n$, $|C_i| = |C_i'|$ and $C_i' \preceq C_i$.*

In defining the semantics of a composition of local requirements $C_1 \ldots C_n$, every local requirement C_i is associated with a local clock, which is reset each time it enters a new phase. Given a local requirements specification $\{C_1, \ldots, C_n\}$, we define its semantics formally by defining the predicate $\mathit{Reach}((C_1, j_1, t_1), \ldots, (C_n, j_n, t_n))$, which is true iff the phases $P^{C_1}_{j_1} \ldots P^{C_n}_{j_n}$ are reachable at local times $t_1 \ldots t_n$.

Definition 3 (Reachability for local requirements) *Given the specification $\{C_1 \ldots C_n\}$ and the time points $t_1 \in \mathbb{R} \ldots t_n \in \mathbb{R}$, we inductively define the predicate $\mathit{Reach}((C_1, j_1, t_1), \ldots, (C_n, j_n, t_n))$ as follows:*

– (base case) *$\mathit{Reach}((C_1, 1, 0), \ldots, (C_n, 1, 0))$ holds and for all $i \in \{1 \ldots n\}$ it holds that (state dependencies):* $((C_1, 1), \ldots, (C_n, 1)) \models \psi^{C_i}_1$
– (timed transition) *if $\mathit{Reach}((C_1, j_1, t_1), \ldots, (C_n, j_n, t_n))$ and there exists a $\delta \in \mathbb{R}$ such that $t_i + \delta \leq u^{C_i}_{j_i+1}$ for all $i \in \{1 \ldots n\}$, then $\mathit{Reach}((C_1, j_1, t_1 + \delta), \ldots, (C_n, j_n, t_n + \delta))$.*
– (discrete transition) *if $\mathit{Reach}((C_1, j_1, t_1), \ldots, (C_n, j_n, t_n))$ and there exists a $\delta \in \mathbb{R}$ and a $M \subseteq \{1, \ldots, n\}$ such that:*
  1. *for all $i \in \{1 \ldots n\}$ such that $j_i < |C_i|$, $t_i + \delta \in [l^{C_i}_{j_i+1}, u^{C_i}_{j_i+1}]$ if $i \in M$, and $t_i + \delta \leq u^{C_i}_{j_i+1}$ otherwise;*
  2. *for all $i \in M$, it holds that (signal dependencies): $((C_1, j_1), \ldots, (C_n, j_n)) \models \phi^{C_i}_{j_i+1}$*
  3. *for all $i \in M$, it holds that (state dependencies - entry): $((C_1, j_1), \ldots, (C_n, j_n)) \models \psi^{C_i}_{j_i+1}$*
  4. *for all $i \in \{1 \ldots n\}$, it holds that (state dependencies - invariant): $((C_1, j'_1), \ldots, (C_n, j'_n)) \models \psi^{C_i}_{j'_i}$*

  then it holds that $\mathit{Reach}((C_1, j'_1, t'_1), \ldots, (C_n, j'_n, t'_n))$, where $j'_i = j_i + 1$ and $t'_i = 0$ if $i \in M$ and $j_i < |C_i|$, and $j'_i = j_i$ and $t'_i = t_i + \delta$ otherwise.

We define the predicate Comp_S to be true iff there are no reachable states in S such that no component can proceed to its next phase.

Definition 4 (Compatibility for local requirements) *Given the set of local requirements $S = \{C_1 \ldots C_n\}$, the predicate Comp_S is true iff:*

$$\begin{aligned}&\forall j_1 \in \{1 \ldots |C_1| - 1\} \ldots \forall j_n \in \{1 \ldots |C_n| - 1\}\ \forall t_1 \ldots t_n \in \mathbb{R}\Big(\\ &\mathit{Reach}((C_1, j_1, t_1), \ldots, (C_n, j_n, t_n)) \Rightarrow\\ &\quad \exists M \subseteq \{1 \ldots n\}\big(M \neq \emptyset \wedge \mathit{Reach}((C_1, j'_1, t'_1), \ldots, (C_n, j'_n, t'_n))\big)\Big)\end{aligned}$$

where $j'_i = j_i + 1$ and $t'_i = 0$ for all $i \in M$, or $j'_i = j_i$ and $t'_i = t_i$ otherwise. If Comp_S holds, we say that $C_1 \ldots C_n$ are compatible, or equivalently that S is compatible.

For example, in Fig. 1a, predicate $\mathit{Reach}((A, 1, 4), (B, 1, 4))$ holds, but predicate $\mathit{Reach}((A, 1, 4), (B, 2, 0))$ does not, because for all $\delta \in \mathbb{R}$ and for all $S \subseteq \{1 \ldots n\}$, predicate $\mathit{Reach}((A, 1, 4), (B, 2, 0))$ is false.

Strict Semantics The above definition adopts a weakly-monotonic model of time, where discrete transitions are instantaneous and, therefore, the system may be in two different states at the same instant. The definition and the reductions to model checking and SMT can be easily adapted to have a strict semantics.

Verification and Synthesis Problems The core problem we address is to check if a given specification $S = \{C_1, \ldots, C_n\}$ is *compatible*, *i.e.*, if $Comp_S$ holds. The *consistency checking* problem amounts to checking if there *exists* a time point in which the final phase of all the local requirements is reached, that is it amounts to checking if the following formula holds:

$$\exists t_1 \ldots \exists t_n \; Reach((C_1, |C_1|, t_1), \ldots, (C_n, |C_n|, t_n))$$

If this is the case, then we say that S is *consistent*. Finally, we can formalize the *synthesis problem* as the problem of computing (a symbolic representation of) the set: $\{S' \mid Comp_{S'} \wedge S' \preceq S\}$

2.1 NP-hardness

In this section, we show that the simplest of the problems defined above is already NP-hard. In fact, we show a reduction from SAT to the *consistency checking* problem.

Let $\varphi(\bar{x})$ be a Boolean formula over the variables $\bar{x} = \langle x_1 \ldots x_n \rangle$; without loss of generality, we assume $\varphi(\bar{x})$ to be in negated normal form, *i.e.*, with all the negations only in front of literals. For all $1 \leq i \leq n$, we define the local requirement corresponding to variable x_i as $C_i = \langle P_1^i, P_2^i \rangle$, such that $B_{P_2^i} = [0, +\infty)$ and $\phi_{P_1^i} = \psi_{P_1^i} = \phi_{P_2^i} = \psi_{P_2^i} = \top$; the idea is to encode the values $\bot$ and $\top$ of each x_i with the two phases P_1^i and P_2^i, respectively. Moreover, we define the local requirement G, which will be useful as a gadget for the reduction, as follows: $G = \langle P_1^G, P_2^G \rangle$, where $P_2^G = \langle [0, +\infty), \varphi[x_i \mapsto \langle C_i, P_2^i \rangle, \neg x_i \mapsto \langle C_i, P_1^i \rangle], \top \rangle$. The specification S^φ corresponding to the Boolean formula $\varphi(\bar{x})$ is defined as $S^\varphi = \{G, C_1, \ldots, C_n\}$. It holds that $\varphi(\bar{x})$ is satisfiable if and only if S^φ is consistent. In fact, if S^φ is consistent, then there exists a time point in which the signal dependency of the second phase of G has been satisfied, and thus $\varphi(\bar{x})$ is satisfiable. Viceversa, let's suppose that $\varphi(\bar{x})$ is satisfiable and let M be an arbitrary model of it, expressed as the set of true atoms, in which we also substitute every x_i in it with the pair $\langle C_i, P_2^i \rangle$. Since the local requirements $C_1 \ldots C_n$ have no dependencies and, together with G, have only infinite bounds, there exists a time t such that predicate $Reach((G, P_1^G, t), (C_1, P_{b_1}^G, t_1), \ldots, (C_n, P_{b_n}^n, t_n))$ is true, where for all $1 \leq i \leq n$, $b_i = 2$ and $t_i = 0$ iff $x_i \in M$ and $t_i = t$ otherwise. By definition of *Reach* (see Definition 3), this implies that $Reach((G, P_2^G, t), (C_1, P_2^1, t), \ldots, (C_n, P_2^n, t))$ holds, *i.e.*, S is consistent.

In Sec. 3.2, we will give an encoding of the consistency checking problem based on SMT(DL) (*i.e.*, *Satisfiability Modulo Theory of Difference Logic*). In particular, we will show that the problem can be reduced to the satisfiability of a formula in SMT(DL). Since the latter belongs to NP [15], the consistency checking problem belongs to NP as well, having that *consistency checking* is NP-complete.

3 Verification

3.1 Reduction to Model Checking

In order to formalize the two verification problems into ones of model checking networks of timed automata, we use timed automata with shared variables. To this end, besides the clock constraints $\Xi(C)$, we define $L = \{l_{\mathcal{A}}, l_{\mathcal{B}}, \dots\}$ as a set of *location variables* (one for each automaton $\mathcal{A}$ in the network), and $\Theta(L)$ as the set of all Boolean combinations of atoms of type $l_{\mathcal{A}} = v_{\mathcal{A}}$, where $\mathcal{A}$ is a timed automata, $l_{\mathcal{A}} \in L$ and $v_{\mathcal{A}}$ is a state of $\mathcal{A}$.

Definition 5 (Timed Automata with Shared Variables) *A* timed automaton with shared variables *(TASV, for short)* $\mathcal{A} = \langle V_{\mathcal{A}}, v^0_{\mathcal{A}}, l_{\mathcal{A}}, C_{\mathcal{A}}, inv^{cl}_{\mathcal{A}}, inv^{loc}_{\mathcal{A}}, T_{\mathcal{A}}\rangle$ *consists of:*

- *a finite set of locations* $V_{\mathcal{A}}$*;*
- *an initial location* $v^0_{\mathcal{A}} \in V_{\mathcal{A}}$*;*
- *a location variable* $l_{\mathcal{A}}$ *with range* $V_{\mathcal{A}}$*;*
- *a finite set of clocks* $C_{\mathcal{A}}$*, where a clock is a real-valued variable;*
- *a* clock invariant $inv^{cl}_{\mathcal{A}} : V_{\mathcal{A}} \to \Xi(C_{\mathcal{A}})$ *for each location;*
- *a* location invariant $inv^{loc}_{\mathcal{A}} : V_{\mathcal{A}} \to \Theta(C_{\mathcal{A}})$ *for each location;*
- *a transition relation* $T_{\mathcal{A}} \subseteq V_{\mathcal{A}} \times 2^{C_{\mathcal{A}}} \times \Xi(C_{\mathcal{A}}) \times \Theta(L) \times V_{\mathcal{A}}$.

Given a set of clocks C, we denote with $\nu : C \to \mathbb{R}$ a *clock valuation*, that is a function assigning a rational value to each clock; with $\mathcal{V}_C$, we denote the set of all possible clock valuations over C. For $t \in \mathbb{R}$, $\nu + t$ is the clock valuation which maps every clock $c \in C$ to the value $\nu(c) + t$. For $R \subseteq C$, we define $\nu[R \mapsto 0]$ to be the valuation that maps x to 0 if $x \in R$, and to $\nu(x)$ otherwise. When defining the product of two TASVs, we will deal with tuples $(l_{\mathcal{A}_1}, \dots, l_{\mathcal{A}_n})$ of location variables; in this context, we usually denote with λ any function from the set of n-tuples of location variables to the set $V_{\mathcal{A}_1} \times \dots \times V_{\mathcal{A}_n}$. Moreover, we write that $\lambda \models \Phi$ (where $\Phi \in \Theta(L)$) iff $\Phi[l_{\mathcal{A}_i} \mapsto v_{\mathcal{A}_i}$, for all $1 \leq i \leq n]$ is true and $\lambda((\dots, l_{\mathcal{A}_i}, \dots)) = (\dots, v_{\mathcal{A}_i}, \dots)$. We give the semantics of a TASV in terms of traces and we define their product as described below.

Definition 6 (Trace of a TASV) *A* trace τ *of a TASV* $\mathcal{A} = \langle V_{\mathcal{A}}, v^0_{\mathcal{A}}, l_{\mathcal{A}}, C_{\mathcal{A}}, inv^{cl}_{\mathcal{A}}, inv^{loc}_{\mathcal{A}}, T_{\mathcal{A}}\rangle$ *is a (either finite or infinite) sequence of* states *of the form:*

$$\langle v_0, \nu_0, \lambda_0\rangle \xrightarrow{\alpha_1} \langle v_1, \nu_1, \lambda_1\rangle \xrightarrow{\alpha_2} \langle v_2, \nu_2, \lambda_2\rangle \xrightarrow{\alpha_3} \dots$$

such that $v_i \in V_{\mathcal{A}}$*,* $\alpha_i \in \mathbb{R} \cup \{\tau\}$*,* $\nu_i \in \mathcal{V}_{C_{\mathcal{A}}}$ *and* $\lambda_i \in \mathcal{V}_L$ *for all* $i \geq 0$*, and:*

- (initiation) $v_0 = v^0_{\mathcal{A}}$, $\nu_0(x) = 0$ *for all* $x \in C_{\mathcal{A}}$, $\nu_0 \models inv^{cl}_{\mathcal{A}}(v^0_{\mathcal{A}})$, $\lambda_0(l_{\mathcal{A}}) = v_0$ *and* $\lambda_0 \models inv^{loc}_{\mathcal{A}}(v^0_{\mathcal{A}})$*;*
- (consecution)*: for all* $i \geq 0$
  - *(timed transition) if* $\alpha \in \mathbb{R}$*, then* $v_{i+1} = v_i$ *and* $\nu_{i+1} = \nu_i + \alpha$*,* $\nu_i + \delta \models inv^{cl}_{\mathcal{A}}(v_i)$*, for all* $0 \leq \delta \leq \alpha$*, and* $\lambda_{i+1}(l_{\mathcal{A}}) = v_i$*;*

- *(discrete transition) if $\alpha = \tau$ then there is a tuple $(v_i, R_i, \Xi_i, \Phi_i, v_{i+1}) \in T_{\mathcal{A}}$ such that: $\nu_i \models inv^{cl}_{\mathcal{A}}(v_i) \wedge \Xi_i$; $\lambda_i \models \Phi_i$; $\nu_{i+1} = \nu_i[R_i \mapsto 0]$; $\nu_{i+1} \models inv^{cl}_{\mathcal{A}}(v_{i+1})$; $\lambda_{i+1}(l_{\mathcal{A}}) = v_{i+1}$, and $\lambda_{i+1} \models inv^{loc}_{\mathcal{A}}(v_{i+1})$.*

Definition 7 (Product of TASVs) *Given two TASVs $\mathcal{A}$ and $\mathcal{B}$, their product is the TASV $\mathcal{A} \otimes \mathcal{B}$ defined as follows:*

- *$V_{\mathcal{A}\otimes\mathcal{B}} = V_{\mathcal{A}} \times V_{\mathcal{B}}$ and $v^0_{\mathcal{A}\otimes\mathcal{B}} = (v^0_{\mathcal{A}}, v^0_{\mathcal{B}})$;*
- *$l_{\mathcal{A}\otimes\mathcal{B}} = (l_{\mathcal{A}}, l_{\mathcal{B}})$;*
- *$C_{\mathcal{A}\otimes\mathcal{B}} = C_{\mathcal{A}} \cup C_{\mathcal{B}}$;*
- *$inv^{cl}_{\mathcal{A}\otimes\mathcal{B}}(v,u) = inv^{cl}_{\mathcal{A}}(v) \wedge inv^{cl}_{\mathcal{B}}(u)$, for all $(v,u) \in V_{\mathcal{A}\otimes\mathcal{B}}$;*
- *$inv^{loc}_{\mathcal{A}\otimes\mathcal{B}}(v,u) = inv^{loc}_{\mathcal{A}}(v) \wedge inv^{loc}_{\mathcal{B}}(u)$, for all $(v,u) \in V_{\mathcal{A}\otimes\mathcal{B}}$;*
- *the transition relation is defined as follows:*

$$\begin{aligned} T_{\mathcal{A}\otimes\mathcal{B}} = & \{((v,u), R, \Xi, \Phi, (v',u)) \mid (v, R, \Xi, \Phi, v') \in T_{\mathcal{A}}\} \cup \\ & \{((v,u), R, \Xi, \Phi, (v,u')) \mid (u, R, \Xi, \Phi, u') \in T_{\mathcal{B}}\} \end{aligned}$$

It is worth noting that each TASV corresponds to a timed automaton defined in the standard way [1], and viceversa. We define now the TASV corresponding to a local requirement.

Definition 8 (TASV for a Local Requirement) *Let $C = \langle P^C_1, \ldots, P^C_n \rangle$ be a local requirement. We define the corresponding TASV $\mathcal{A} = \{V_{\mathcal{A}}, v^0_{\mathcal{A}}, l_{\mathcal{A}}, C_{\mathcal{A}}, inv^{cl}_{\mathcal{A}}, inv^{loc}_{\mathcal{A}}, T_{\mathcal{A}}\}$ as follows:*

- *for each phase P^C_i of local requirement C, $v^i_{\mathcal{A}}$ is the corresponding location in $V_{\mathcal{A}}$; P^C_0 corresponds to $v^0_{\mathcal{A}}$ and $C_{\mathcal{A}} = \{c_{\mathcal{A}}\}$;*
- *for each phase P^C_i (but the last) of C, $inv^{cl}_{\mathcal{A}}(v^i_{\mathcal{A}}) := c_{\mathcal{A}} \leq u_{P^C_{i+1}}$;*
- (discrete transition) *for each phase P^C_i (but the last) of C, it holds that $(v^i_{\mathcal{A}}, \{c_{\mathcal{A}}\}, \Xi^C_i, \Phi_{P^C_{i+1}} \wedge \Psi_{P^C_{i+1}}, v^{i+1}_{\mathcal{A}}) \in T_{\mathcal{A}}$, where $\Xi^C_i := l_{P^C_{i+1}} \leq c_{\mathcal{A}} \leq u_{P^C_{i+1}}$.*
- (state deps) *for each phase P^C_i of C, it holds that $inv^{loc}_{\mathcal{A}}(v^i_{\mathcal{A}}) := \Psi_{P^C_i}$;*

where $\Phi_P := \phi_P[(d,j) \mapsto (l_d = v_j)]$, for each phase P (the same holds for Ψ);

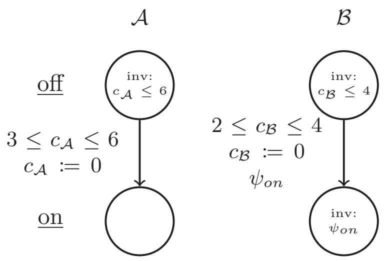

$\mathcal{A}$
$\mathcal{B}$
off
inv: $c_{\mathcal{A}} \leq 6$
inv: $c_{\mathcal{B}} \leq 4$
$3 \leq c_{\mathcal{A}} \leq 6$
$c_{\mathcal{A}} := 0$
$2 \leq c_{\mathcal{B}} \leq 4$
$c_{\mathcal{B}} := 0$
ψ_{on}
on
inv: ψ_{on}

Fig. 2: Example of TASV corresponding to a local requirement.

Example. Consider Fig. 1a: the corresponding TASV is depicted in Fig. 2. Each phase of each local requirement corresponds to a location of the corresponding TASV; in the example, phase *off* is mapped into location off. The first locations of automata $\mathcal{A}$ and $\mathcal{B}$ have attached the invariants $c_{\mathcal{A}} \leq 6$ and $c_{\mathcal{B}} \leq 4$, respectively. Automaton $\mathcal{A}$ proceeds to location on (corresponding to phase $A.on$) by a transition labelled with clock constraint $3 \leq c_{\mathcal{A}} \leq 6$ and clock reset $c_{\mathcal{A}} := 0$. Since the second phase of local requirement A has no dependencies, the transition to

$\underline{on}$ has no constraints on the location variables. The situation is different for automaton $\mathcal{B}$, for which the transition to $\underline{on}$ is labelled with $2 \leq c_{\mathcal{B}} \leq 4$ and $c_{\mathcal{B}} := 0$, and also with $\psi_{on} := (l_{\mathcal{A}} = on)$, that is the state dependency of phase $B.on$; moreover, ψ_{on} is also an invariant for the second location of automaton $\mathcal{B}$, since it is a state dependency.

Given a network $\mathcal{S} := \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ of TASVs, the problem of *consistency checking* can be expressed as the reachability of location $(\mathcal{A}_1.last, \ldots, \mathcal{A}_n.last) \in V_{\mathcal{S}}$. A *deadlock* of a TASV $\mathcal{A}$ is defined as a state $(v, t) \in V_{\mathcal{A}} \times \mathbb{R}$ such that $\mathcal{A}$ can take neither a timed nor a discrete transition from (v, t). We call *livelock* a state (v, t) such that $\mathcal{A}$ can take only timed transitions. The *compatibility checking* problem can be expressed as the problem of checking if there exists a trace of $\mathcal{S}$ such that (i) either the trace is finite and its final state is a deadlock of $\mathcal{S}$; we can check this property by adding a *sink* location to the TASV $\mathcal{S}$ to which all locations can transition to and by checking the reachability of it; (ii) or the trace is infinite and there exists a location $v \in V_{\mathcal{S}}$ and a point $k \geq 0$ such that $l_{\mathcal{S}} = v \neq (\mathcal{A}_1.last, \ldots, \mathcal{A}_n.last)$, for all the states after k in the trace, where the i^{th} component of v together with the time of the current state is a *livelock* for automata $\mathcal{A}_i$, for some $1 \leq i \leq n$. The second point is fundamental for local requirements featuring *infinite bounds*: in these automata, it is not sufficient to check for deadlocks, since a timed transition could be always enabled; instead, an illegal state can be described by a trace of the system that reaches a *livelock* whose location has no invariants attached and then stays constantly in this location. Having reached a livelock, the automaton can proceed only with timed moves: in particular, it can't proceed to the next location because its dependencies are violated. We can check the second point in this way: we first add a sink location $sink_v^{\mathcal{A}_i}$ for each location $v \in \mathcal{A}_i$ (and of course a transition from the latter to the former), for each $1 \leq i \leq n$, and we attach to it the invariant $\neg inv_{\mathcal{A}_i}^{loc}(v)$. Now, in the product S of these modified automata, we look for a trace such that, from a certain time point onwards, it stays constantly in a location $(l_1, \ldots, l_n)$ such that at least one l_i is a sink state. This property can be formalized in *Linear Temporal Logic* as $FG(\bigvee_{1 \leq i \leq n, v \in \mathcal{A}_i} sink_v^{\mathcal{A}_i})$.

3.2 Encoding into SMT(DL)

We describe the encoding into SMT(DL) (Satisfiability Modulo Theory of Difference Logic) for the problems of *consistency checking* and *compatibility checking*. For all $1 \leq c \leq n$ and $1 \leq i \leq |c|$, we introduce the following variables: (i) $r_i^c \in \mathbb{B}$ represents the fact that phase i of local requirement c is *reachable*; (ii) $s_i^c = (t_i^c, p_i^c)$ represents the superdense time instant in which local requirement c enters phase i, where $t_i^c \in \mathbb{R}$ and $p_i^c \in \mathbb{N}$. We can compare two superdense-valued variables (t, p) and (t', p') with the lexicographical order, which we define as follows: $(t, p) \preceq (t', p')$ iff $t \leq t' \wedge (t = t' \rightarrow p \leq p')$. We now give the set of (conjunctively related) constraints which form our SMT(DL) encoding.

Initialization. Each local requirement starts in its first phase at the same time, *i.e.*, the real time point 0. Hence, for all $1 \leq c \leq n$, we add the constraint $t_0^c = 0$.

Reachability. For all local requirements c and all phases i, it holds that if $i-1$ is not reachable then so is phase i, *i.e.*, $\neg r^c_{i-1} \rightarrow \neg r^c_i$. Moreover, we require the monotonicity over time, *i.e.*, $r^c_i \rightarrow (s^c_{i-1} \prec s^c_i)$.

Bounds. For all local requirements c and all phases i, c can move to i only if it respects the bounds $[l^c_i, u^c_i]$ of phase i, namely $r^c_i \rightarrow (l^c_i \leq t^c_i - t^c_{i-1} \leq u^c_i)$. If $u^c_i = \infty$, then we add only the left-most inequality.

Signal and State dependencies. Consider a local requirement c and one of its phases i. Since we have only a *finite* number of phases, we can preprocess both signal and state dependencies to remove from them all negations, as explained in the extended version of the paper [4]; this means that every atom in ϕ^c_i and ψ^c_i occurs positive.

We want c to reach i only if all its signal and state dependencies are satisfied. For signal dependencies, we require the time point in which c enters i to be strictly greater[5] than the time point of the entry of the target phase and smaller than or equal to the time point of the exit of the target phase.

$$r^c_i \rightarrow \phi^c_i[(d,j) \mapsto (r^d_j \wedge s^d_j \prec s^c_i \preceq s^d_{j+1})]$$

Moreover, we have to guarantee that the state dependencies hold as well. In particular, if phase i is reachable, then surely the time point in which c enters i has to be strictly greater than the time point in which the other local requirement reaches the target phase.

$$r^c_i \rightarrow \psi^c_i[(d,j) \mapsto (r^d_j \wedge s^d_j \prec s^c_i)]$$

Since state dependencies are invariant properties, *i.e.*, they have to hold for each time instant a local requirement is in a particular phase, if one state dependency is violated at some time point of phase $i-1$, then phase i is not reachable. The contrapositive means that if phase i is reachable, then the state dependencies of phase $i-1$ have to be invariant for phase $i-1$, namely:

$$r^c_i \rightarrow \forall \widetilde{s}(s^c_{i-1} \preceq \widetilde{s} \preceq s^c_i \rightarrow \psi^c_{i-1}[(d,j) \mapsto (r^d_j \wedge s^d_j \prec \widetilde{s} \preceq s^d_{j+1})]) \qquad (1)$$

Illegal States. If phase i of local requirement c is not reachable, *i.e.*, i is an *illegal state*, then there exists a time point s^c_{ill} such that, for all the next (remaining) time points $\bar{s}$ between s^c_{ill} and the upperbound of the transition, at least one dependency is not satisfied.

$$(r^c_{i-1} \wedge \neg r^c_i) \rightarrow \exists s^c_{ill} \forall \bar{s}(s^c_{ill} \preceq \bar{s} \preceq s^c_{i-1} + u^c_{i-1} \rightarrow \text{VIOLATION}(\bar{s})) \qquad (2)$$

[4] http://users.dimi.uniud.it/~luca.geatti/tricker.html

[5] This allows us to model the observability of the events: c first observes d entering its phase j and *then* moves.

where

$$\begin{aligned}\text{VIOLATION}(\bar{s}) \coloneqq\ & \neg\phi_i^c[(d,j) \mapsto (r_j^d \wedge s_j^d \prec \bar{s} \preceq s_{j+1}^d)] \vee & (3)\\ & \neg\psi_i^c[(d,j) \mapsto (r_j^d \wedge s_j^d \prec \bar{s})] \vee & (4)\\ & \exists \widetilde{s}(s_{i-1}^c \preceq \widetilde{s} \preceq \bar{s} \wedge \neg\psi_{i-1}^c[(d,j) \mapsto (r_j^d \wedge s_j^d \prec \widetilde{s} \preceq s_{j+1}^d)]) & (5)\end{aligned}$$

We interpret $\bar{s} \preceq s_{i-1}^c + u_i^c$ as $\forall \bar{p}(\bar{s} \preceq s_{i-1}^c + (u_i^c, \bar{p}))$ and the $+$ symbol as the pairwise sum. In the case the upperbound of the transition is infinite, we simply do not add the $\bar{s} \preceq s_{i-1}^c + u_i^c$ inequality. We refer to the conjunction of all these constraints as W.

For *consistency checking*, we define END $\coloneqq \bigwedge_{1\le c\le n} r_{|c|}$ and we call W^{cons} the conjunction of W with END. We check consistency by checking the satisfiability of W^{cons}.

For *compatibility checking*, we define ILL $\coloneqq \bigvee_{\substack{1\le c\le n\\ 1\le i\le |c|}} \neg r_i^c$ and we call W^{ill} the conjunction of W with ILL. We check the existence of an illegal state in the system by checking the satisfiability of W^{ill}, *i.e.*, W^{ill} is satisfiable iff the local requirements are *not* compatible.

Strict Semantics In the strict semantics setting, we forbid two events to occur at the same real-time point. For strict semantics, the encoding is equal to W except that we interpret $\prec$ and $\preceq$ as $<$ and $\le$, respectively, and all the s_i^c variables as single real-valued variables $t_i^c \in \mathbb{R}$. We call S this encoding and we define S^{cons} and S^{ill} as above.

Finite bounds and convex dependencies. Despite being very close to the problem formalization, the W encoding features a high number of quantifications, also in alternation; therefore, in the general case, it is very burdensome for an SMT solver to first perform quantifier elimination on W and then to solve the resulting formula. Nevertheless, if we make some restrictions on the type of local requirements we consider, we are able to remove *upfront* all the quantifiers from W, without the need to use quantifier elimination techniques. In fact, suppose we consider only local requirements with *finite bounds* and *convex* state dependencies (see Sec. 2). We call $\widehat{\text{W}_{\text{fin}}^{\text{ill}}}$ the encoding equal to W except that Eq. (1) is replaced by:

$$r_i^c \to \psi_{i-1}^c[(d,j) \mapsto (r_j^d \wedge s_i^c \preceq s_{j+1}^d)] \tag{6}$$

and we add the following constraint:

$$(r_{i-1}^c \wedge \neg r_i^c) \to (t_i^c = t_{i-1}^c + u_{i-1}^c) \tag{7}$$

and we replace Eq. (2) with:

$$(r_{i-1}^c \wedge \neg r_i^c) \to \text{WEAKVIOL}(t_i^c) \tag{8}$$

where:

$$\begin{aligned}\text{WEAKVIOL}(t_i^c) \;\coloneqq\; & \neg\phi_i^c[(d,j) \mapsto (r_j^d \wedge t_j^d \leq t_i^c < t_{j+1}^d)] \vee \\ & \neg\psi_i^c[(d,j) \mapsto (r_j^d \wedge t_j^d \leq t_i^c)] \vee \\ & \neg\psi_{i-1}^c[(d,j) \mapsto (r_j^d \wedge t_i^c \leq t_{j+1}^d)])\end{aligned} \tag{9}$$

We can prove that W^{ill} and $\widehat{\text{W}^{\text{ill}}_{\text{fin}}}$ are equisatisfiable for every set of local requirements with only finite bounds and convex dependencies. Notably, there are no quantifiers in $\text{W}^{\text{ill}}_{\text{fin}}$: as said before, this makes the encoding dramatically more efficient with respect to W: in Sec. 5, we will consider only local requirements of this type. The details of the proofs are reported in the extended version of the paper in which, given that the proofs are a bit involved, we proceed incrementally, showing first how we can remove upfront the quantifiers in case of finite bounds with strict semantics, then in the case with weak semantics and finally in case of convex dependencies.

4 Synthesis

In this section, we tackle the *synthesis* problem, *i.e.*, computing the set of *all stronger* local requirements (as defined in Def. 2) of the initial local requirements such that their composition is *compatible*. We solve this problem by reducing it to a *parameter synthesis* problem (see [9] for a more detailed description); given a local requirement C, its corresponding *parametric local requirement* $\langle C, \pi \rangle$ is defined as C (see Sec. 2), except that the bounds l_P and u_P of each phase P are now the parameters $\bar{l}_P$ and $\bar{u}_P$, respectively, and $\pi \coloneqq \{\bar{l}_P \mid P \text{ is a phase of } C\} \cup \{\bar{u}_P \mid P \text{ is a phase of } C\}$. Given a set of local requirements $S = \{C_1, \ldots, C_n\}$, we write $\langle S, \Pi \rangle$ for its parametric version $\{\langle C_1, \pi_1 \rangle, \ldots, \langle C_n, \pi_n \rangle\}$, where the set of parameters is defined as $\Pi \coloneqq \bigcup_{i=1}^n \pi_i$. A parameter valuation $\gamma : \Pi \to \mathbb{Q}$ assigns a rational value to each parameter; moreover, for each $1 \leq i \leq n$, it also induces a (concrete) local requirement $\langle C_i, \gamma(\pi_i) \rangle$, obtained from $\langle C_i, \pi_i \rangle$ by replacing every parameter $p \in \pi_i$ with the concrete value $\gamma(p)$. In the same way, we can define the *concrete* version $\langle S, \gamma(\pi) \rangle$ of $\langle S, \pi \rangle$. γ is said to be *feasible for S* if $\langle C_i, \gamma(\pi_i) \rangle$ is a *stronger* local requirement of C_i, for all $1 \leq i \leq n$, and $\langle S, \gamma(\pi) \rangle$ is *compatible*. A *feasible region* is a set $\mathcal{R} \coloneqq \{\gamma \mid \gamma \text{ is feasible for } S\}$. Also in this case, we can either use parameter synthesis algorithms over timed automata [3] or reduce the problem to SMT(LRA); we focus on the latter and in particular, we will synthesize a symbolic representation of the region $\mathcal{R}$, namely an SMT formula $\varphi_{\mathcal{R}}$ with the following property: $\gamma \models \varphi_{\mathcal{R}}$ iff $\gamma \in \mathcal{R}$, for each valuation γ.

Let $\overline{\text{W}^{\text{ill}}}$ be the encoding equal to W^{ill} except that each number l_i^c (resp. u_i^c) is replaced with the variable $\bar{l}_i^c$ (resp. $\bar{u}_i^c$) and each phase is required to have finite bounds. We define the sets of variables $\mathbb{R} \coloneqq \{r_i^c \mid c \in S,\ i \text{ is a phase of } c\}$ and $\mathbb{S} \coloneqq \{s_i^c \mid c \in S,\ i \text{ is a phase of } c\}$: these are the variables we are going to remove by means of quantifier elimination. Finally, we define:

$$\text{DOMAIN} \coloneqq \bigwedge_{\substack{1\le c\le n\\1\le i\le |c|}} (\bar{l}_i^c \geq 0 \land \bar{l}_i^c \leq \bar{u}_i^c)$$

$$\text{REFINE} \coloneqq \bigwedge_{\substack{1\le c\le n\\1\le i\le |c|\\u_i^c\neq\infty}} (a_i^c \leq \bar{l}_i^c \land \bar{u}_i^c \leq b_i^c)$$

The symbolic representation of the *feasible region* $\mathcal{R}$ is given by:

$$\text{SYNTH} \coloneqq \text{DOMAIN} \land \text{REFINE} \land \neg\exists\mathbb{S},\mathbb{R}\left(\overline{\text{W}^{\text{ill}}}\right) \tag{10}$$

By removing the existential quantification on $\mathbb{S}$ and $\mathbb{R}$ (this can be done by means of quantifier elimination techniques), we obtain a quantifier-free formula over the variables in Π. By construction, we have that each model γ of SYNTH is a feasible valuation, and viceversa. Therefore SYNTH is the symbolic representation of the feasible region $\mathcal{R}$.

5 Experimental Evaluation

We implemented the encoding described in Sec. 3.2 in a tool called TRICker (Timing Requirements Integration Checker) [6], which uses MathSAT [12] as the backend SMT engine. We compared TRICker with Uppaal [6] and Timed-nuXmv [8], both using the automata-based encoding described in Sec. 3.1.

The test set is partitioned into three categories: (i) `bounded convex` contains only systems with finite bounds and convex state dependencies; (ii) `bounded` contains systems with only finite bounds, but with arbitrary dependencies (not necessarily convex); (iii) `general` contains systems with infinite bounds and arbitrary dependencies (this is the most general fragment). Each category in turn consists of ca. 500 randomly-generated systems, divided in 10 sub-categories, namely 2c3p, 2c15p, 5c3p, 5c20p, 10c4p, 10c30p, 50c5p, 50c30p, 100c3p and 100c10p, where NcMp is the category containing only systems with N components and (approximately) M phases for each component. Inside each sub-category, each benchmark is randomly generated, meaning that the exact number of phases for each component and the density of its signal and state dependencies was chosen uniformly at random. For each benchmark, we compare the time spent by the three tools on the *consistency checking* and *compatibility checking* problems. We ran the experiments on a cluster of Linux machines with a 2.27GHz Xeon CPU, with a timeout of 360 seconds for each instance.

We consider first the `bounded convex` category. Fig. 3 shows the comparison of TRICker with Timed-nuXmv and Uppaal on the two verification problems. In both cases, Timed-nuXmv runs the infinite-state variant of IC3 described in [11] after discretizing the timed automata. As for Uppaal, we verify a property in the form $EF\varphi$, where φ is a Boolean formula. For both problems, the SMT-based approach implemented in TRICker outperforms the model checkers. While there are a number of instances for which the model checkers perform better

[6] http://users.dimi.uniud.it/~luca.geatti/tricker.html

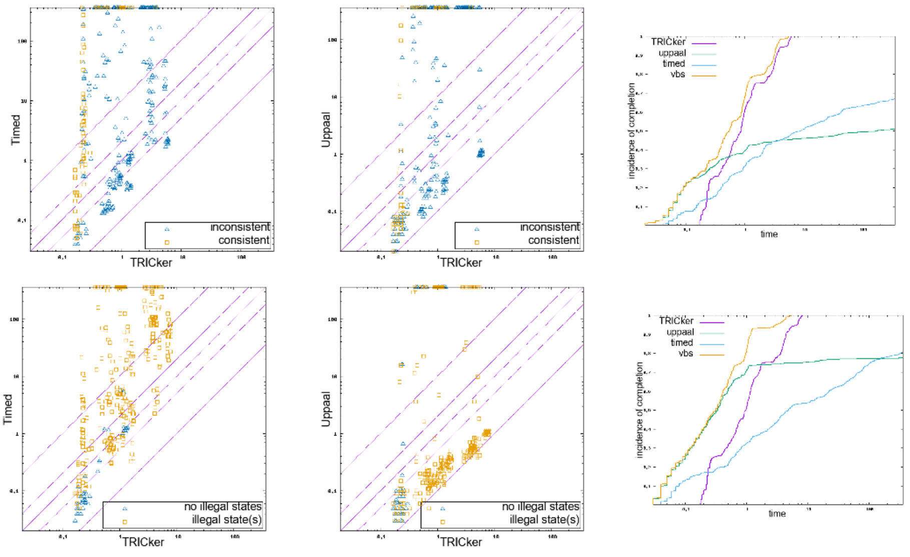


Fig. 3: Comparison on the `bounded convex` category (*consistency checking* on the first row and *compatibility checking* on the second).

than TRICker (especially for Uppaal), the latter overall solves a significantly larger amount of problems within the timeout, showing a clear improvement in scalability. This can be seen also in the survival plots comparing the three tools with the Virtual Best Solver (*vbs* for short). We can make similar considerations for the `bounded` and `general` categories, shown respectively in Fig. 4 and Fig. 5. (Note that for the `general` case, we could not evaluate Uppaal as it does not support the verification of fairness properties.) We remark that we did not note any kind of correlation between the number of signal or state dependencies in the benchmarks and the time spent by the solver. Finally, Fig. 6 shows the correlation between the memory (measured in MB) and the time (in seconds) spent by TRICker on consistency and compatibility checking, respectively.

We also evaluated the parameter synthesis algorithm described in Sec. 4. Since Uppaal currently does not support parameter synthesis for timed automata, we could not include it in the comparison. We therefore compared TRICker with Timed-nuXmv, for which we used the ParamIC3 parameter synthesis algorithm described in [9]. The algorithm is based on the inverse method, *i.e.*, it finds a bad configuration for the parameters and it tries to generalize it, maximizing the set of bad parameters removed from the current approximation of the region. We took all the consistent benchmarks of the previous test sets, which amounts to approximately 100 instances (note that for each instance of the class NcMp, the number of parameters is $\approx 2 \cdot N \cdot M$[7]). The results of the comparison are shown in Fig. 7; as in the previous cases, TRICker shows better performance and

[7] recall that both the lower and the upper bounds are parameters.

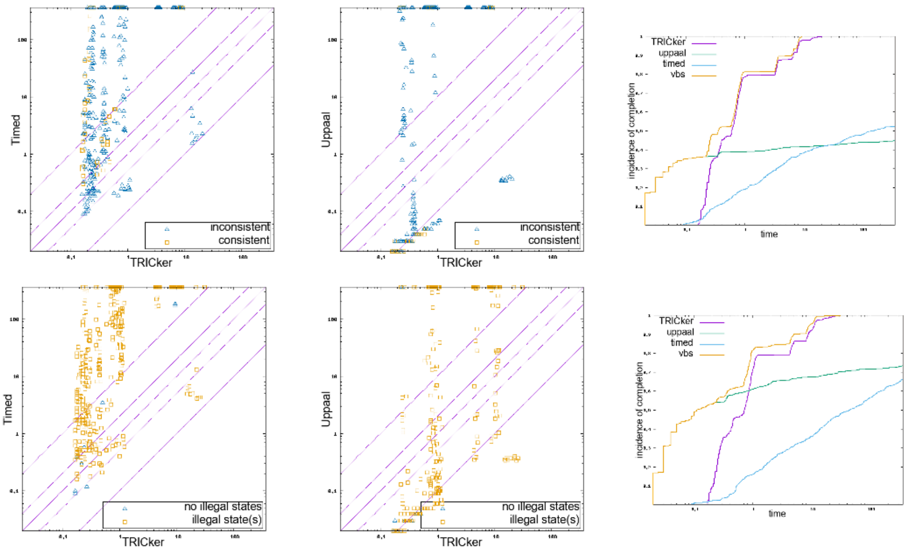


Fig. 4: Comparison on the `bounded` category (*consistency checking* on the first row and *compatibility checking* on the second).

scalability than ParamIC3, though there are several instances for which synthesis via quantifier elimination is still very expensive.

6 Conclusions

In this paper, we defined verification and synthesis problems of industrial relevance focused on the decomposition of startup requirements into local timing constraints and dependencies on components. Namely, we addressed the problem of checking if the local requirements are free of integration errors (i.e., consistent and compatible), and the problem of synthesizing the region of refinements of the original specification that are error free. The problem can be naturally translated into model checking and synthesis problems for timed automata with shared variables. Exploiting the structure of the requirements, we provide an encoding into SMT where consistency and incompatibility correspond to satisfiability queries, while synthesis is resolved by means of quantifier elimination.

In the future, we will consider various directions, such as extending the applicability of the approach to more general structures with loops, enriching the synthesis problem with cost functions to repair the specification driven by specific industrial goals, and considering more complex representations of signals exchanged between components.

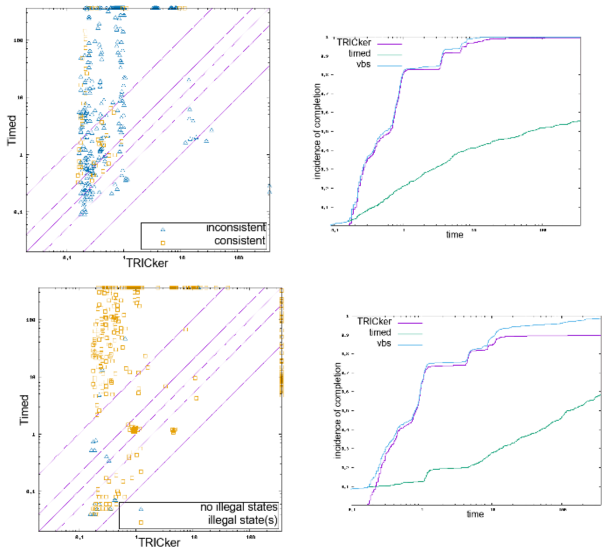


Fig. 5: Comparison on the `general` category (*consistency checking* on the first row and *compatibility checking* on the second).

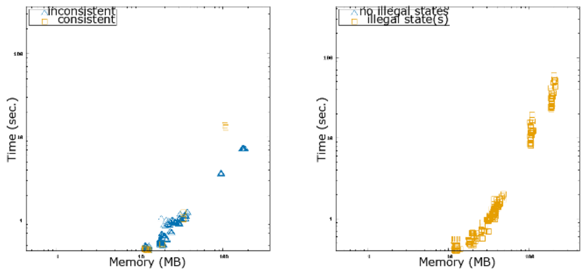


Fig. 6: Comparison between time and memory consuption of TRICker (*consistency checking* on the left and *compatibility checking* on the right).

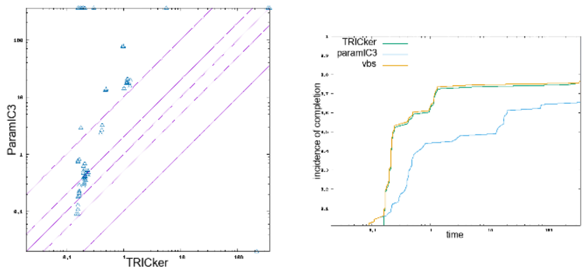


Fig. 7: Comparison on parameter synthesis.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science **126**(2), 183–235 (1994)
2. André, É.: Parametric Deadlock-Freeness Checking Timed Automata. In: Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings. pp. 469–478 (2016). https://doi.org/10.1007/978-3-319-46750-4_27
3. André, É., Chatain, T., Fribourg, L., Encrenaz, E.: An inverse method for parametric timed automata. International Journal of Foundations of Computer Science **20**(05), 819–836 (2009)
4. Astefanoaei, L., Rayana, S.B., Bensalem, S., Bozga, M., Combaz, J.: Compositional Invariant Generation for Timed Systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. pp. 263–278 (2014). https://doi.org/10.1007/978-3-642-54862-8_18
5. Astefanoaei, L., Rayana, S.B., Bensalem, S., Bozga, M., Combaz, J.: Compositional Verification of Parameterised Timed Systems. In: NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings. pp. 66–81 (2015). https://doi.org/10.1007/978-3-319-17524-9_6
6. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0 (2006)
7. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Extending nuXmv with Timed Transition Systems and Timed Temporal Properties. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. pp. 376–386 (2019). https://doi.org/10.1007/978-3-030-25540-4_21
8. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Smt-based satisfiability of first-order ltl with event freezing functions and metric operators (2019)
9. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with ic3. In: 2013 Formal Methods in Computer-Aided Design. pp. 165–168. IEEE (2013)
10. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Verifying LTL Properties of Hybrid Systems with K-Liveness. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. pp. 424–440 (2014). https://doi.org/10.1007/978-3-319-08867-9_28
11. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. Formal Methods in System Design **49**(3), 190–218 (2016). https://doi.org/10.1007/s10703-016-0257-4
12. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 93–107. Springer (2013)
13. De Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed interfaces. In: International Workshop on Embedded Software. pp. 108–122. Springer (2002)
14. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)
15. Niemelä, I.: Stable models and difference logic. Annals of Mathematics and Artificial Intelligence **53**(1-4), 313–329 (2008)

16. Stigge, M., Ekberg, P., Guan, N., Yi, W.: The digraph real-time task model. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 71–80. IEEE (2011)
17. Stigge, M., Yi, W.: Combinatorial abstraction refinement for feasibility analysis of static priorities. Real-Time Systems **51**(6), 639–674 (2015). https://doi.org/10.1007/s11241-015-9220-5

Multi-Agent Safety Verification using Symmetry Transformations[⋆]

Hussein Sibai, Navid Mokhlesi, Chuchu Fan, and Sayan Mitra
{sibai2,navidm2,cfan10,mitras}@illinois.edu

University of Illinois, Urbana IL 61801, USA

Abstract. We show that symmetry transformations and caching can enable scalable, and possibly unbounded, verification of multi-agent systems. Symmetry transformations map any solution of the system to another solution. We show that this property can be used to transform cached reachsets to compute new reachsets, for hybrid and multi-agent models. We develop a notion of a *virtual system* which defines symmetry transformations for a broad class of agent models that visit waypoint sequences. Using this notion of a virtual system, we present a prototype tool CacheReach that builds a cache of reachsets, in a way that is agnostic of the representation of the reachsets and the reachability analysis method used. Our experimental evaluation of CacheReach shows up to 64% savings in safety verification computation time on multi-agent systems with 3-dimensional linear and 4-dimensional nonlinear fixed-wing aircraft models following sequences of waypoints. These savings and our theoretical results illustrate the potential benefits of using symmetry-based caching in the safety verification of multi-agent systems.

1 Introduction

As the cornerstone for safety verification of dynamical and hybrid systems, reachability analysis has attracted attention and has delivered automatic analysis of automotive, aerospace, and medical applications [2,24,17,11]. Notable advances from the last few years include the development of the generalized star data-structure [14] and the HyLaa tool [3] which can analyze massive linear models [4]; Taylor model based reachability analysis algorithms for nonlinear systems and their implementations in Flow* [7]; and a simulation-based algorithm that guarantees locally optimal precision [15].

Exact symbolic reachability analysis of nonlinear models is generally hard. One prominent approach is based on generalizing individual behaviors or simulations to cover a whole set of behaviors. The idea was pioneered in [10] and implemented in Breach [9] with sound generalization guarantees for linear models based on *sensitivity analysis*. Subsequently, the idea has been significantly extended to cover nonlinear, hybrid, and black-box models and it has been implemented in tools like C2E2 and DryVR [12,19,17,16].

In all of the above, a single behavior ξ of the system from an initial state, is generalized to a *compact set* of *neighboring* behaviors that contains all the behaviors starting

[⋆] The authors are supported by a research grant from The Boeing Company and a research grant from NSF (CPS 1739966). We would like to thank John L. Olson and Arthur S. Younger from The Boeing Company for valuable technical discussions.

from a small neighborhood around the initial state of ξ. Thus, the computed neighboring set of behaviors always contains ξ and its size is determined by the algorithms for sensitivity analysis. In contrast, the type of generalization we pursue here uses *symmetry transforms* on the state space. Given a group Γ of operators on the state space, and a single behavior ξ, we can generalize ξ to $\gamma(\xi)$, for each $\gamma \in \Gamma$. Symmetry transformations can be applied to sets of behaviors symbolically. Not only can this type of generalization work in conjunction with sensitivity analysis, it captures structural properties of the system that make behaviors similar in a way that is not covered by sensitivity analysis.

In our recent work [29], we showed how symmetry transforms can be used to produce new reachsets from other previously computed reachsets for *non-parameterized* dynamical systems. In this paper, we introduce the use of symmetry transforms of *parameterized* dynamical systems for safety verification. We present an algorithm `symComputeReachtube` (Algorithm 1) which caches and reuses reachsets, avoiding repeating expensive computations. We show how an infinite number of reachsets can be obtained by transforming a single one using symmetry transforms (Corollary 2). Building on it, we provide unbounded time safety guarantees using finite cached safety checking results (Theorem 6).

The key contributions of this paper are as follows.

First, we show how symmetry transformations for parameterized dynamical systems can be used to compute reachable states (Theorem 2). Going well beyond the previous theory [29], this enables *cached reachtubes* to be reused for verification across different modes and across multiple agents.

We develop a notion of *virtual system* (Section 4) which automatically defines symmetry transformations for a broad swathe of hybrid and dynamical systems modeling agents visiting a sequence of waypoints (see Theorem 3 and Examples 3 and 4). That is, reachability analysis of a multi-agent system, with possibly different dynamics and different parameters, can be performed in a common transformed coordinate system, and thus, increases the possibility of reuse. We show how this principle can make it possible to verify systems over unbounded time and with infinite number of agents (Theorem 6), provided that no new unproven scenarios appear for the virtual system.

We present a prototype implementation of a tool that uses `symComputeReachtube`. We name it CacheReach. It builds a cache of reachtubes for the virtual system, from different sets of initial states. In performing reachability analysis of a multi-agent hybrid or dynamical system, for each agent and each mode, the algorithm proceeds as follows: (1) transform the initial set X to an initial set of the virtual system to get $\gamma(X)$. (2) If the transformed set $\gamma(X)$ has already been stored in the cache, then extract it and apply γ^{-1} to get the actual reachset. (3) Otherwise, compute the reachset from $\gamma(X)$ and cache it. Our algorithm `symComputeReachtube` and its implementation in CacheReach are agnostic of the representation of the reachsets and the reachability analysis subroutine, and therefore, any of the ever-improving libraries can be plugged-in for step 3.

Our experimental evaluation of CacheReach shows safety verification computation time savings of up to 64% on scenarios with multiple agents with 3-dimensional linear and 4-dimensional nonlinear fixed-wing aircraft model following sequences of waypoints. These savings illustrate the potential benefits of using symmetry transformations and caching in the safety verification of multi-agent systems.

2 Model and problem statement

Notations. We denote by $\mathbb{N}$, $\mathbb{R}$, and $\mathbb{R}^{\geq 0}$ the sets of natural numbers, real numbers and non-negative reals. Given a finite set S, its cardinality is denoted by $|S|$. Given $N \in \mathbb{N}$, we denote by $[N]$ the set $\{1, \ldots, N\}$. Given a vector $v \in \mathbb{R}^n$ and a set $L \subseteq [n]$, we denote the projection of v to the indices in L by $v[L]$. We define an n-dimensional hyper-rectangle by a 2d-array specifying its bottom-left and upper-right corners. We denote the projection of a hyper-rectangle H on the set of dimensions L by $H[L]$. Given a function $\gamma : \mathbb{R}^k \to \mathbb{R}^k$ and a set $S \subseteq \mathbb{R}^k$, we abuse notation and define $\gamma(S) = \{\gamma(x) \mid x \in S\}$. Moreover, given $S \in 2^{\mathbb{R}^k} \times \mathbb{R}^{\geq 0}$, we define $\gamma(S) = \{(\gamma(X), t) \mid (X, t) \in S\}$.

2.1 Agent mode dynamics

In this section, we define the syntax and semantics of the model that determines the dynamics of an agent. We present the syntax first.

Definition 1 (syntax). *The agent dynamics are defined by a tuple $A = \langle S, P, f \rangle$, where $S \subseteq \mathbb{R}^n$ is its state space, $P \subseteq \mathbb{R}^m$ is its parameter or mode space, and the dynamic function $f : S \times P \to S$ that is Lipschitz in the first argument.*

The semantics of an agent dynamics is defined by trajectories, which describe the evolution of states over time.

Definition 2 (semantics). *For a given agent $A = \langle S, P, f \rangle$, we call a function $\xi : S \times P \times \mathbb{R}^{\geq 0} \to S$ a* trajectory *if ξ is differentiable in its third argument, and given an initial state $\mathbf{x}_0 \in S$ and a mode $p \in P$, $\xi(\mathbf{x}_0, p, 0) = \mathbf{x}_0$ and for all $t > 0$,*

$$\frac{d\xi}{dt}(\mathbf{x}_0, p, t) = f(\xi(\mathbf{x}_0, p, t), p). \tag{1}$$

We say that $\xi(\mathbf{x}_0, p, t)$ is the state of A at time t when it starts from $\mathbf{x}_0$ in mode p.

Given an initial state $\mathbf{x}_0 \in S$ and mode $p \in P$, the trajectory $\xi(\mathbf{x}_0, p, \cdot)$ is the unique solution of the ordinary differential equation (ODE) (1) since f is Lipschitz continuous.

Given a compact initial set $K \subseteq S$, a parameter $p \in P$, the set of *reachable states* of A over a time interval $[\mathit{ftime}, \mathit{etime}]$ is defined as

$$\texttt{Reach}(K, p, [\mathit{ftime}, \mathit{etime}]) = \{\mathbf{x} \in S \mid \exists \mathbf{x}_0 \in K, t \in [\mathit{ftime}, \mathit{etime}], \mathbf{x} = \xi(\mathbf{x}_0, p, t)\}. \tag{2}$$

We let $\texttt{Reach}(K, p, t)$ denote the set of reachable states at time t. Unbounded reachset from K and p is $\texttt{Reach}(K, p, [\mathit{ftime}, \infty))$.

The *bounded time safety verification* problem requires one to check if any state reachable by A for a given initial set K and mode p is unsafe within a given time bound. That is, given a time bound $T > 0$, $p \in P$, and an unsafe set $U \subseteq S$, we want to check whether $\texttt{Reach}(K, p, [0, T]) \cap U = \emptyset$.

2.2 Reachtubes

Computing reachsets exactly is theoretically hard [22]. There are many reachability analysis tools [8,1,3] that can compute bounded-time over-approximations of the reachsets. Generally, given an initial set K for a set of ODEs, these tools can return a sequence of sets that contain the exact reachset over small time intervals. Motived by this, we define reachtubes as sequences of time-annotated over-approximations of exact reachsets:

Definition 3. *For a given agent $A = \langle S, P, f \rangle$, an initial set $K \subseteq T$, a mode $p \in P$, and a time interval $[ftime, etime]$, a $(K, p, [ftime, etime])$-reachtube* $\texttt{ReachTb}(K, p, [0, T])$ *is a sequence $\{(X_i, [\tau_{i-1}, \tau_i])\}_{i=1}^{j}$ such that* $\texttt{Reach}(K, p, [\tau_{i-1}, \tau_i]) \subseteq X_i$*, and $\tau_0 = ftime < \tau_1 < \cdots < \tau_j = etime$. Without loss of generality, we assume equal separation between the time points, i.e. $\exists\ \tau_s > 0, \forall i \in [j], \tau_i - \tau_{i-1} = \tau_s$.*

For a given $(K, p, [ftime, etime])$-reachtube *rtube*, we denote its parameters by *rtube.K*, *rtube.p*, *rtube.ftime*, and *rtube.etime*, respectively, and its cardinality by *rtube.len*.

We define union, truncation, concatenate, and time-shift operators on reachtubes. Fix $rtube_1 = \{(X_{i,1}, [\tau_{i-1,1}, \tau_{i,1}])\}_{i=1}^{j_1}$ and $rtube_2 = \{(X_{i,2}, [\tau_{i-1,2}, \tau_{i,2}])\}_{i=1}^{j_2}$ to be two reachtubes, where $j_1 = rtube_1.len$ and $j_2 = rtube_2.len$. If $\tau_{i,1} = \tau_{i,2}$ for all $i \in [\min(j_1, j_2)]$, we say they are *time-aligned*. Without loss of generality, assume that $j_1 \leq j_2$. The operators are defined as follows:

- *timeShift*$(rtube_1, t_s) = \{(X_{i,1}, [t_s + \tau_{i-1,1}, t_s + \tau_{i,1}])\}_{i=1}^{j_1}$,
- *union*: $rtube_1 \cup rtube_2 = \{(X_{i,1} \cup X_{i,2}, [\tau_{i-1,1}, \tau_{i,1}])\}_{i=1}^{j_1} \cup \{(X_{i,2}, [\tau_{i-1,2}, \tau_{i,2}])\}_{i=j_1+1}^{j_2}$,
- *concatenation*: $rtube_1 \frown rtube_2 = rtube_1 \cup \{(X_{i,2}, [\tau_{j_1,1} + \tau_{i-1,2}, \tau_{j_1,1} + \tau_{i,2}))\}_{i=1}^{j_2}$,
- *truncate*$(rtube_1, t_c) = \{(X_{i,1}, [\tau_{i-1,1}, \tau_{i,1}])\}_{i=1}^{k}$, where $\tau_{k,1} \geq t_c$ and $\tau_{k-1,1} < t_c$.

A *simulation* of system (1) is a reachtube with X_0 being a singleton state $x_0 \in K$. That is, a simulation is a representation of $\xi(x_0, p, \cdot)$. Several numerical solvers can compute such simulations as VNODE-LP[1] and CAPD Dyn-Sys library [2].

Example 1 (Fixed-wing aircraft following a single waypoint). Consider an agent with state space $S = \mathbb{R}^4$, parameter space $P = \mathbb{R}^4$, and $f : S \times P \to S$ defined as follows: for any $\mathbf{x} \in S$ and $p \in P$,

$$f(\mathbf{x}, p) = [\frac{T_c - c_{d1}\mathbf{x}[0]^2}{m}, \frac{g}{\mathbf{x}[0]} \sin\phi, \mathbf{x}[0] \cos \mathbf{x}[1], \mathbf{x}[0] \sin \mathbf{x}[1]],$$

where $T_c = k_1 m(v_c - \mathbf{x}[0])$, $\phi = k_2 \frac{v_c}{g}(\psi_c - \mathbf{x}[1])$, $\psi_c = \arctan_2(\frac{\mathbf{x}[2]-p[2]}{\mathbf{x}[3]-p[3]})$, and k_1, k_2, m, g, c_{d1}, and v_c are positive constants. The agent models a fixed-wing aircraft starting from a waypoint and following another in the 2D plane: $\mathbf{x}[0]$ is its speed, $\mathbf{x}[1]$ is its heading angle, $(\mathbf{x}[2], \mathbf{x}[3])$ is its position in the plane, $[p[0], p[1]]$ is the position of the source waypoint, and $(p[2], p[3])$ is the position of the destination one. Note that the source waypoint does not affect the dynamics, but will be useful later in the paper.

[1] http://www.cas.mcmaster.ca/~nedialk/vnodelp/
[2] http://capd.sourceforge.net/capdDynSys/docs/html/odes_rigorous.html

3 Symmetry and Equivariant Dynamical Systems

Symmetry plays a fundamental role in the analysis of dynamical systems. It has been used for studying stability of feedback systems [25], designing observers [5] and controllers [30], and analyzing neural networks [20]. In this section, we present definitions of symmetries and their implications on systems that posses them.

3.1 Symmetry of systems with inputs

In the following, symmetry transformations are defined by the ability of computing new solutions of (1) using already computed ones. First, let Γ be a group of smooth maps acting on S.

Definition 4 (Definition 2 in [27]). *We say that $\gamma \in \Gamma$ is a symmetry of (1) if for any solution $\xi(\mathbf{x}_0, p, \cdot)$, $\gamma(\xi(\mathbf{x}_0, p, \cdot))$ is also a solution.*

Using γ-symmetry, we can get a new trajectory without simulating the system but instead by just transforming the entire old trajectory using γ.

In the following definition we characterize the conditions under which a transformation is a symmetry of (1).

Definition 5. *The dynamic function $f : S \times P \to S$ is said to be Γ-equivariant if for any $\gamma \in \Gamma$, there exists $\rho : P \to P$ such that for all $\mathbf{x} \in S$, $\frac{\partial \gamma}{\partial \mathbf{x}} f(\mathbf{x}, p) = f(\gamma(\mathbf{x}), \rho(p))$.*

The following theorem shows that it is enough to check the condition in Definition 5 to prove that a transformation is a symmetry of (1).

Theorem 1 (part of Theorem 10 in [27]). *If f is Γ-equivariant, then all maps in Γ are symmetries of (1). Moreover, for any solution $\xi(\mathbf{x}_0, p, \cdot)$ and $\gamma \in \Gamma$, $\gamma(\xi(\mathbf{x}_0, p, \cdot)) = \xi(\gamma(\mathbf{x}_0), \rho(p), \cdot)$, where ρ is the transformation associated with γ in Definition 5.*

Proof. Let $\mathbf{y} = \gamma(\mathbf{x})$, then $\dot{\mathbf{y}} = \frac{\partial \gamma}{\partial \mathbf{x}}(\dot{\mathbf{x}}) = \frac{\partial \gamma}{\partial \mathbf{x}}(f(\mathbf{x}, p)) = f(\gamma(\mathbf{x}), \rho(p)) = f(\mathbf{y}, \rho(p))$. The second equality is a result of the derivative chain rule. The 3^{rd} equality uses Definition 5.

Remark 1. If γ in Theorem 1 is linear, the condition in Definition 5 for a map γ to be a symmetry becomes $\gamma(f(\mathbf{x}, p)) = f(\gamma(\mathbf{x}), \rho(p))$.

Example 2 (Fixed-wing aircraft coordinate transformation symmetry). Consider the fixed-wing aircraft model of Example 1. Fix $goal \in \mathbb{R}^2$ and $\theta \in \mathbb{R}$. Let $\gamma : \mathbb{R}^4 \to \mathbb{R}^4$ and $\rho : \mathbb{R}^4 \to \mathbb{R}^4$ be defined as:

$$\begin{aligned}\gamma(\mathbf{x}) = [&\mathbf{x}[0], \mathbf{x}[1] + \theta, (\mathbf{x}[2] - goal[0])\cos(\theta) + (\mathbf{x}[3] - goal[1])\sin(\theta),\\ &-(\mathbf{x}[2] - goal[0])\sin(\theta) + (\mathbf{x}[3] - goal[1])\cos(\theta)] \text{ and} \qquad (3)\\ \rho(p) = [&0, 0, (p[2] - goal[0])\cos(\theta) + (p[3] - goal[1])\sin(\theta),\\ &-(p[2] - goal[0])\sin(\theta) + (p[3] - goal[1])\cos(\theta)]. \qquad (4)\end{aligned}$$

Then, for all $\mathbf{x} \in S$ and $p \in P$, $\gamma(f(\mathbf{x}, p)) = f(\gamma(\mathbf{x}), \rho(p))$, where f is as in Section 2.1. The transformation γ would change the origin of S from $[0, 0, 0, 0]$ to $[0, 0, goal[0], goal[1]]$.

Then, it would rotate the third and four axes counter-clockwise by θ. Moreover, ρ would set the first two coordinates of the parameters to zero as they do not affect the dynamics, translate the origin of the parameter space P to $[0, 0, goal[0], goal[1]]$, and rotate the third and fourth axes counter-clockwise by θ. For the aircraft, this means translating and rotating the plane where the aircraft and the waypoint positions reside.

3.2 Symmetry and reachtubes

Computing reachtubes is computationally expensive as it requires non-trivial optimization problems and integrating non-linear functions [13,15,16,8,6]. Compared with that, transforming reachtubes is much cheaper, especially if the transformation is linear.

In our previous work [29], we showed how to get reachtubes of autonomous systems from previously computed ones using symmetry transformations. In this paper, we show how to do that for systems with parameters. This allows different modes of a hybrid system and different agents with similar dynamics to share reachtube computations. That was not possible when the theory was limited to non-parameterized systems.

Theorem 2. *Let (1) be Γ-equivariant. Then for any $\gamma \in \Gamma$ and its corresponding ρ, any $K, p, [ftime, etime]$ and $\{(X_i, [\tau_{i-1}, \tau_i])\}_{i=1}^{j}$ as a $(K, p, [ftime, etime])$-reachtube,*

$$\forall i \in [j], \texttt{Reach}(\gamma(K), \rho(p), [\tau_{i-1}, \tau_i]) = \gamma(\texttt{Reach}(K, p, [\tau_{i-1}, \tau_i])) \subseteq \gamma(X_i).$$

Proof. (Sketch) The first part $\texttt{Reach}(\gamma(K), \rho(p), [\tau_{i-1}, \tau_i]) = \gamma(\texttt{Reach}(K, p, [\tau_{i-1}, \tau_i]))$ follows directly from Theorem 1. The second part $\gamma(\texttt{Reach}(K, p, [\tau_{i-1}, \tau_i])) \subseteq \gamma(X_i)$ follows from the reachtube $\texttt{ReachTb}(K, p, [t_b, t_e])$ being an over-approximation of the exact reachset during the small time intervals $[\tau_{i-1}, \tau_i]$.

Theorem 2 says that we can transform a computed reachtube $\texttt{ReachTb}(K, p, [t_1, t_2]) = \{(X_i, [\tau_{i-1}, \tau_i])\}_{i=1}^{j}$ to get another reachtube $\{(\gamma(X_i), [\tau_{i-1}, \tau_i])\}_{i=1}^{j}$, which is an over-approximation of the reachsets starting from $\gamma(K)$.

The results of this section subsume the results about transforming reachtubes of autonomous systems-dynamical systems without parameters as presented in [29].

4 Virtual system

The challenge in safety verification of multi-agent systems is that the dimensionality of the problem grows rapidly with the number of agents. However, often agents share the same dynamics. For instance, several fixed-wing aircrafts of the type described in Example 1 share the same dynamics but may have different initial conditions and follow different waypoints. This commonality has been exploited in developing specialized proof techniques [23]. For reachability analysis, using symmetry transforms of the previous section, reachtubes of one agent in one mode can be used to get the reachtubes of other modes and even other agents.

Fix a particular value $p_v \in P$ and call it the *virtual* parameter. Assume that for all $p \in P$, there exists a pair of transformations (γ_p, ρ_p) such that $\rho_p(p) = p_v$, γ_p is invertible, and $\gamma_p(f(\mathbf{x},p)) = f(\gamma_p(\mathbf{x}),\rho_p(p_v)) = f(\gamma_p(\mathbf{x}),p_v)$. Consider the resulting ODE:

$$\frac{d\xi}{dt}(\mathbf{y},p_v,t) = f(\xi(\mathbf{y},p_v,t),p_v). \tag{5}$$

Following [27], we call (5) a *virtual system*. Correspondingly, we call (1), the *real system* for the rest of the paper. The virtual system unifies the behavior of all modes of the real system in one representative mode, the virtual one p_v.

Example 3 (Fixed-wing aircraft virtual system). Consider the fixed-wing aircraft agent described in Example 1 and the corresponding transformations described in Example 2. Fix $p \in P$, we set *goal* in the transformation of Example 2 to $[p[2],p[3]]$ and θ to $\arctan_2(p[0]-p[2],p[3]-p[1])$ and let γ_p and ρ_p be the resulting transformations. Then, for all $p \in P$, $\rho_p(p) = [0,0,0,0]$. Hence, $p_v = [0,0,0,0]$ and the virtual system is that of Example 1 with the parameter $p = p_v$. For the aircraft, γ_p would translate the origin of the plane to the destination waypoint and rotate its axes so that the y-axis is aligned with the segment between the source and destination waypoints. Hence, in the constructed virtual system, the destination waypoint is the origin of the plane. The source waypoint is the origin as well as it does not affect the dynamics.

The solutions of the virtual system can be transformed to get solutions of all other modes in P using $\{\gamma_p^{-1}\}_{p\in P}$. This is shown in the following theorem.

Theorem 3. *Given any initial state $\mathbf{y}_0 \in S$, and any mode $p \in P$, $\gamma_p^{-1}(\xi(\mathbf{y}_0,p_v,\cdot))$ is a solution of the real system (1) with mode p starting from $\gamma_p^{-1}(\mathbf{y}_0)$. Similarly, given any $\mathbf{x}_0 \in S$, $\gamma_p(\xi(\mathbf{x}_0,p,\cdot))$ is the solution of the virtual system (5) starting from $\gamma_p(x_0)$.*

Proof. Lets start with the first part of the theorem. Fix $p \in P$ and let $\mathbf{x}_0 = \gamma_p^{-1}(\mathbf{y}_0)$. Using Theorem 1, $\gamma_p(\xi(\mathbf{x}_0,p,\cdot)) = \xi(\gamma_p(\mathbf{x}_0),\rho_p(p),\cdot))$ and is the solution of the real system (1). Furthermore, $\rho_p(p) = p_v$, by definition, and $\gamma_p(\mathbf{x}_0) = \gamma_p(\gamma_p^{-1}(\mathbf{y}_0)) = \mathbf{y}_0$. Hence, $\gamma_p(\xi(\mathbf{x}_0,p,\cdot)) = \xi(\mathbf{y}_0,p_v,\cdot)$. Applying γ_p^{-1} on both sides implies the first part of the theorem. The second part is a direct application of Theorem 1.

The following corollary extends the result of Theorem 3 to reachtubes. It follows from Theorem 2.

Corollary 1. *Given a $K_v \subseteq S$ and a mode $p \in P$, $\gamma_p^{-1}(\texttt{ReachTb}(K_v,p_v,[t_b,t_e]))$ is a reachtube of the real system (1) with mode p starting from $\gamma_p^{-1}(K_v)$. Similarly, given any initial set $K \subset S$, $\gamma_p(\texttt{ReachTb}(K,p,[t_b,t_e]))$ is a reachtube of the virtual system (5) starting from $\gamma_p(K)$.*

Consequently, we get a solution or a reachtube for each mode $p \in P$ of the real system by simply transforming a single solution or a single reachtube of the virtual system using the transformations $\{\gamma_p\}_{p\in P}$ and their inverses. This will be the essential idea behind the savings in computation time of the new symmetry-based reachtube computation algorithm and symmetry-based safety verification algorithms presented next. It will be also the essential idea behind proving safety in the case of unbounded time and infinite number of modes.

Example 4 (Fixed-wing aircraft infinite number of reachtubes resulting from transforming a single one). Consider the real system in Example 1 and the virtual one in Example 3. Fix the initial set, which is represented as a hyper-rectangle, $K_r = [[1, \frac{\pi}{4}, 3, 1], [2, \frac{\pi}{3}, 4, 2]]$, the real mode $p_r = [2.5, 0.5, 13.3, 5]$, and the time bound 20 seconds. Then, similar to Example 3, we fix $\theta = \arctan_2(2.5 - 13.3, 5 - 0.5) = -1.176$ rad and $goal = [13.3, 5]$. We call the resulting transformations from Example 3, γ_{p_r} and ρ_{p_r}. Let $K_v = \gamma_{p_r}(K_r)$ and $p_v = \rho_{p_r}(p_r) = [0, 0, 0, 0]$. Assume that we have the reachtube $rtube_r = \texttt{ReachTb}(K_r, p_r, T)$. Then, using Corollary 1, we can get $rtube_v = \texttt{ReachTb}(K_v, p_v, T)$ by transforming $rtube_r$ using γ_{p_r}. The benefit of the corollary appears in the following: for any $p \in P = \mathbb{R}^4$, we can get the corresponding reachtube $\texttt{ReachTb}(\gamma_p^{-1}(K_v), p, T)$ by transforming $rtube_v$ using γ_p^{-1}.

The projection of K_v on its last two coordinates $K_v[2:3]$ represents the possible initial position of the aircraft in the plane relative to the destination waypoint. It would be a rotated square with angle θ. The distance from $K_v[2:3]$ center to the origin would be equal to the distance from $K[2:3]$ center to the destination waypoint. Moreover, the angle between the y-axis and the line connecting the origin with the center of $K_v[2:3]$ would be equal to the angle from the segment connecting the source and destination waypoints to the line connecting the destination waypoint with the center of $K[2:3]$. On the other hand, $K_v[0] = K[0]$ and $K_v[1] = K[1] + \theta$.

In summary, the absolute positions of the aircraft and waypoints do not matter. What matters is their relative positions. The virtual system stores what matters and whenever a reachtube is needed for a new absolute position, we can transform it from the virtual one.

5 Symmetry-based verification algorithm

In this section, we introduce a novel safety verification algorithm, `symSafetyVerif`, which uses existing reachability subroutines, but exploits symmetry, unlike existing algorithms. In our earlier work [29], we introduced reachtube transformations using symmetry for single mode dynamical systems. Here, we extend the method across modes, introduce the virtual system, and develop the corresponding verification algorithm.

In Section 5.1, we define *tubecache*—a data-structure for storing reachtubes; in 5.2, we present the symmetry-based reachtube computation algorithm `symComputeReachtube` that reuses reachtubes stored in *tubecache*; finally, in 5.3, we define the *safetycache* data-structure which stores previously computed safety verification results. These results would be used by the `symSafetyVerif` algorithm.

5.1 *tubecache*: shared memory for reachtubes

We show how we use the virtual system (5) to create a shared memory for the different modes of the real system (1) to reuse each others' computed reachtubes. We call this shared memory *tubecache*.

Definition 6. *A tubecache is a data structure that stores a set of reachtubes of the virtual system (5). It has two methods: getTube, for retrieving stored tubes and storeTube, for storing a newly computed one.*

The function getTube returns a set of reachtubes $\{\texttt{ReachTb}(K_i, p_v, [0, T_i])\}_{i \in [h]}$, for some $h \in \mathbb{N}$, that are already stored in *tubecache*. Moreover, the union of K_is is the largest subset of K that can be covered by the initial sets of the reachtubes in *tubecache*. Formally,

$$\textit{tubecache}.\texttt{getTube}(K) = \underset{\{\texttt{ReachTb}(K_i, p_v, [0, T_i]) \in \textit{tubecache}\}_i}{\operatorname{argmax}} \operatorname{Vol}(K \cap \cup_i K_i), \tag{6}$$

where $\operatorname{Vol}(\cdot)$ is the Lebesgue measure of the set. Note that for any $K \subset \mathbb{R}^n$, a maximizer of (6) would be the set of all reachtubes in *tubecache*. However, this is very inefficient and it would be too conservative to be useful for checking safety. Therefore, getTube should return the minimum number of reachtubes that maximize (6). Note that the reachtubes in *tubecache* may have different time bounds. We will truncate or extend them when used.

5.2 symComputeReachtube: symmetry-based reachtube computation

Given an initial set $K \subset S$, a mode $p \in P$, and time bound T, there are dozens of tools that can return a $\texttt{ReachTb}(K, p, [0, T])$. See [13,8,9] for examples of such tools and [26] for a comprehensive survey. We denote this procedure by $\texttt{computeReachtube}(K, p, [0, T])$.

Whenever a reachtube is needed, instead of calling computeReachtube, we will use symmetry to retrieve corresponding reachtubes that are already stored in *tubecache* and only compute what is not stored. We introduce Algorithm 1 which implements this idea and name it symComputeReachtube.

It takes as input the initial set of the virtual system K_v, the time bound T, and *tubecache*. It returns a reachtube of the virtual system starting from K_v and running for T time units. Hence, to get a reachtube of the real system starting from an initial set K and having a mode p and time bound T, we transform K using γ_p to get K_v, call symComputeReachtube, and transform the result using γ_p^{-1}.

First, it initializes $\textit{restube}_v$ as an empty tube of the virtual system (5) to store the result in line 2. It then gets the reachtubes from *tubecache* that corresponds to K_v using the getTube method in line 3. Now that it has the relevant tubes in *storedtubes*, it adjusts their lengths based on the time bound T. For a retrieved tube with a time bound less than T in line 5, symComputeReachtube extends the tube for the remaining time using computeReachtube in lines 6-7, store the resulting tube in *tubecache* instead of the shorter one in line 8. If the retrieved tube is longer than T (line 9), it trims it in line 10. However, we keep the long one in the *tubecache* to not lose a computation we already did. Then, the tube with the adjusted length is added to the result tube $\textit{restube}_v$ in line 11.

The union of the initial sets of the tubes retrieved *storedtubes* may not contain all of the initial set K_v. That uncovered part is called K_v' in line 12. The reachtube starting from K_v' would be computed from scratch using computeReachtube in line 13, stored in *tubecache* in line 14, and added to $\textit{restube}_v$ in line 15. The resulting tube of the virtual system (5) is returned in line 16. This tube would be transformed by the calling algorithm using γ_p^{-1} to get the corresponding tube of the real system (5).

Theorem 4. *The output of Algorithm 1 is an over-approximation of the reachtube* $\texttt{ReachTb}(K_v, p_v, [0, T])$.

Algorithm 1 `symComputeReachtube`

```
1: **input:** $K_v, T, tubecache$
2: $restube_v \leftarrow \emptyset$
3: $storedtubes \leftarrow tubecache.\texttt{getTube}(K_v)$
4: **for** $i \in [|storedtubes|]$ **do**
5:     **if** $storedtubes[i].T < T$ **then**
6:         $(K_i, [\tau_i, T_i]) \leftarrow storedtubes[i].end$
7:         $tube_i \leftarrow storedtubes[i] \frown \texttt{computeReachtube}(K_i, p_v, [0, T - \tau_i])$
8:         $tubecache.\texttt{storeTube}(tube_i)$
9:     **else if** $storedtubes[i].T > T$ **then**
10:        $tube_i \leftarrow storedtubes[i].truncate(T)$
11:    $restube_v \leftarrow restube_v \cup tube_i$
12: $K'_v \leftarrow K_v \setminus \cup_i storedtubes[i].K$
13: $tube' = \texttt{computeReachtube}(K'_v, p_v, [0, T])$
14: $tubecache.\texttt{storeTube}(tube')$
15: $restube_v \leftarrow restube_v \cup tube'$
16: **return:** $restube_v$
```

Proof. The function `computeReachtube` always returns over-approximations of the reachset from a given initial set and for a given time bound. The set *restube* contains reachtubes that were computed by `computeReachtube` at some point. There are three types of reachtubes in *restube*:

1. When the time bound T_i of the stored reachtube $storedtubes[i]$ is less than T, we need to extend $storedtubes[i]$ until time T by concatenating the original tube with `computeReachtube`$(K_i, p_v, [0, T - \tau_i])$, where $(K_i, [\tau_i, T_i])$ is the last pair in $storedtubes[i]$. The result is a valid $(storedtubes[i].K, p_v, [0, T])$-reachtube.
2. When the time bound T_i of the stored reachtube $storedtubes[i]$ is more than T, the truncated reachtube is also a valid $(storedtubes[i].K, p_v, [0, T])$-reachtube.
3. For K'_v that is not contained in the union of the initial sets in *storedtubes*, the function `computeReachtube` will return a valid $(K'_v, p_v, [0, T])$-reachtube.

The union of the initial sets of the tubes in *storedtubes* and K'_v contains K_v, so the union of the reachtubes the algorithm returns a $(K_v, p_v, [0, T])$-reachtube.

The importance of `symComputeReachtube` lies in that if a mode p required a computation of a reachtube and the result is saved in *tubecache*, another mode with a similar scenario with respect to the virtual system would reuse that tube instead of computing one from scratch. Moreover, reachtubes of the same mode might be reused as well if the scenario was repeated again.

5.3 Bounded time safety

In this section, we show how to use *tubecache* and `symComputeReachtube` of the previous section for bounded and unbounded time safety verification of the real system (1). We consider a scenario where the safety verification of multiple modes of the real system (1) starting from different initial sets and for different time horizons is needed. We will use the virtual system (5) and the transformations $\{\gamma_p\}_{p \in P}$ to share safety computations across modes, initial sets, time horizons, and unsafe sets.

We first introduce *safetycache*, a shared memory to store the results of intersecting reachtubes of the virtual system (5) with different unsafe sets. It will prevent repeating safety checking computations of different modes under similar scenarios and can be used in finding unbounded time safety properties of the real system (1).

Definition 7. *A safetycache is a data structure that stores the results of intersecting reachtubes of the virtual system (5) with unsafe sets. It has two functions:* `getIntersect`*, for retrieving stored results and* `storeIntersect`*, for storing a newly computed one.*

Given an initial set K_v, a time bound T, and an unsafe set U_v, the reachtube $rtube = \texttt{ReachTb}(K_v, p_v, [0,T])$ is unsafe if there is another one $rtube' = \texttt{ReachTb}(K_v', p_v, [0,T'])$, is unsafe, and is an under-approximation of *rtube*. Similarly, if *rtube*$'$ is an over-approximation of *rtube* and is safe, then *rtube* is safe. Formally, the `getIntersect` function of *safetycache* returns the truth value of the predicate $\texttt{ReachTb}(K_v, p_v, [0,T]) \cap U_v = \emptyset$ if a subsuming computation is stored, and returns $\perp$, otherwise.

Formally, *safetycache*.$\texttt{getIntersect}(K_v, T, U_v) =$

$$\begin{cases} 0, \text{ if } \exists\, K_v', T', U_v' \mid K_v \supseteq K_v', T \geq T', U_v \supseteq U_v', \mathit{safetycache}(K_v', T', U_v') = 0, \\ 1, \text{ if } \exists\, K_v', T', U_v' \mid K_v \subseteq K_v', T \leq T', U_v \subseteq U_v', \mathit{safetycache}(K_v', T', U_v') = 1, \text{ and} \\ \perp, \text{ otherwise,} \end{cases}$$

where 0 means *unsafe* and 1 means *safe*.

It is equivalent to check the intersection of a reachtube of the real system (1) with an unsafe set U and to check the intersection of the corresponding reachtube and unsafe set of the virtual one. This is formalized in the following lemma.

Lemma 1. *Consider an unsafe set $U \subseteq \mathbb{R}^n \times \mathbb{R}^+$ and $rtube = \texttt{ReachTb}(K, p, [t_1, t_2])$. Then, for any invertible $\gamma : \mathbb{R}^n \to \mathbb{R}^n$, $rtube \cap U \neq \emptyset$ if and only if $\gamma(rtube) \cap \gamma(U) \neq \emptyset$.*

Now that we have established the equivalence of safety checking between the real and virtual systems, we present Algorithm 2 denoted by `symSafetyVerif`. It uses *safetycache*, *tubecache*, and `symComputeReachtube` in order to share safety verification computations across modes. The method `symSafetyVerif` would be called several times to check safety of different scenarios and *safetycache* and *tubecache* would be maintained across calls.

The function `symSafetyVerif` takes as input an initial set K, a mode p, a time bound T, an unsafe set U, the transformation γ_p, and *safetycache* and *tubecache* that resulted from previous runs of the algorithm.

It starts by transforming the initial and unsafe sets K and U to a virtual system initial and unsafe sets K_v and U_v using γ_p in line 2. It then checks if a subsuming result of the safety check for the tuple (K_v, T, U_v) exists in *safetycache* using its method `getIntersect` in line 3. If it does exist, it returns it directly in line 8. Otherwise, the approximate reachtube is computed using `symComputeReachtube` in line 5. The returned tube is intersected with U_v in line 6 and the result of the intersection is stored in *safetycache* in line 7 and returned in line 8.

Algorithm 2 `symSafetyVerif`

```
1: input: $K, p, T, U, \gamma_p, safetycache, tubecache$
2: $K_v \leftarrow \gamma_p(K), U_v \leftarrow \gamma_p(U)$
3: $result \leftarrow safetycache.$getIntersect$(K_v, T, U_v)$
4: if $result = \perp$ then
5:     $rtube \leftarrow$ symComputeReachtube $(K_v, T, tubecache)$
6:     $result \leftarrow (tube \cap U_v = \emptyset)$
7:     $safetycache.$storeIntersect$(K_v, T, U_v, result)$
8: return: $result$
```

Theorem 5. *If* `symSafetyVerif` *returns safe, then* $\texttt{ReachTb}(K, p, [0,T]) \cap U = \emptyset$.

Proof. From Theorem 4, if the result is not stored in *safetycache*, we know that *rtube* in line 5 is an over-approximation of $\texttt{ReachTb}(K_v, p_v, [0,T])$. Moreover, we know from Corollary 1 that $\texttt{ReachTb}(K, p, [0,T]) \subseteq \gamma_p^{-1}(rtube)$. But, from Lemma 1, we know that the truth value of the predicate $(rtube \cap U_v = \emptyset)$ is equal to that of $(\gamma_p^{-1}(rtube) \cap U = \emptyset)$ and hence *result* is *safe* if $\gamma_p^{-1}(rtube) \cap U = \emptyset$ and thus it is *safe* if $\texttt{ReachTb}(K, p, T) \cap U = \emptyset$. Finally, the stored values in *safetycache* are results from previous runs, and hence have the same property.

However, if `symSafetyVerif` returns *unsafe*, it might be that *rtube* in line 5 intersected the unsafe set because of an over-approximation error. There are two sources of such errors: first, the method `computeReachtube` used by `symComputeReachtube` can itself result in over-approximation errors. Actually, it will, most of the time [13,8]. But it may be exact too [3]. Second, the *tubecache*.`getTube` method which would return a list of tubes with the union of their initial sets strictly over-approximating the needed initial set. The first problem can be solved by asking the method `computeReachtube` to compute tighter reachtubes. Existing methods provide this option at the expense of worse computational complexity [13,8]. However, we can use symmetry in these tightening computations as well, as we did in [29]. We can also replace saved tubes in *tubecache* with newly computed tighter ones. The second problem can be solved by asking *tubecache*.`getTube` to return only the tubes with initial sets that are fully contained in the asked initial set. This would decrease the savings from transforming cached results, but it would reduce the false-positive error, saying *unsafe* while it is *safe*.

5.4 Unbounded time safety

In this section, we show how infinite number of results of safety checks, i.e. results of intersections of reachtubes with unsafe sets, can be deduced from finite ones. The following corollary applies Lemma 1 to the transformations $\{\gamma_p\}_{p \in P}$ that map the different modes of the real system (1) to the unique virtual one (5).

Corollary 2 (Infinite safety verification results from a single one). *Fix* $U \subseteq \mathbb{R}^n$ *and* $rtube = \texttt{ReachTb}(K_v, p_v, [0,T])$. *If* $rtube \cap U = \emptyset$, *then* $\forall p \in P$, $\gamma_p^{-1}(rtube) \cap \gamma_p^{-1}(U) = \emptyset$.

The corollary means that from a single scenario safety check, i.e. an intersection operation between a reachtube $\texttt{ReachTb}(K, p_v, [0,T])$ and unsafe set U, we can deduce the safety of any mode $p \in P$ starting from $\gamma_p^{-1}(K)$ and running for T time units with respect to the corresponding unsafe set $\gamma_p^{-1}(U)$. This would, for example, imply unbounded time safety of a hybrid automaton under the assumption that the unsafe sets of the modes are at the same relative position with respect to the reachtube. But, *safetycache* stores a number of results of such operations. We can infer from each one of them the safety of infinite scenarios. This is formalized in the following theorem which follows directly from Corollary 2.

Theorem 6 (Infinite safety verification results from finite ones). *For any mode $p \in P$, initial set $K \subseteq S$, time bound $T \geq 0$, and unsafe set $U \subset S \times \mathbb{R}^{\geq 0}$, such that $K \subseteq \gamma_p^{-1}(K')$, $U \subseteq \gamma_p^{-1}(U')$, and safetycache$(K', T, U') = 1$, system (1) is safe.*

As more results are added to *safetycache*, then we can deduce the safety of more scenarios in all modes. If at a given point of time, we are sure that no new scenarios would appear, we can deduce the safety for unbounded time and unbounded number of agents with the same dynamics having scenarios already covered.

Example 5 (Fixed-wing aircraft infinite number of safety verification results from computing a single one). Consider the initial set K, mode p, time bound T, their corresponding virtual ones K_v and p_v, and the symmetry transformation γ_{p_r} considered in Example 4. Let the unsafe set be $U = [[0, -\infty, 11.9, 5.1], [\infty, \infty, 12.9, 6.1]] \times \mathbb{R}^{\geq 0}$ and $U_v = \gamma_{p_r}(U)$. Assume that $rtube_v \cap U_v = \emptyset$ and the result is stored in *safetycache*. Then, for all $p \in P$, $\gamma_p^{-1}(rtube_v) \cap \gamma_p^{-1}(U_v) = \emptyset$.

For the aircraft, U could represent a mountain. Crashing with the mountain at any speed, heading angle, and time is unsafe. U_v represents the relative position of the mountain with respect to the segment of waypoints. Theorem 6 says that for any initial set of states K of the aircraft and time bound T, if the relative positions of the aircraft, unsafe set, and the segment of waypoints are the same or subsumed by those of K_v, U_v, and the origin, we can infer safety irrespective of their absolute positions.

6 Experimental evaluation

We implemented a software safety verification tool for multi-agent hybrid systems based on `symComputeReachtube` using Python 3. We named it CacheReach. By hybrid, we mean systems that transition between different modes under different conditions. We tested it on a linear dynamical system and the aircraft model of Example 1, following sequences of waypoints, using DryVR [18] and Flow* [8] as reachability subroutines. Our code is available in a figshare repository [28] and has been tested on an Ubuntu virtual machine available in another figshare repository [21].

6.1 CacheReach: multi-agent safety verification tool

Our tool CacheReach takes as input a JSON file specifying a list of N agents of dimension n. It also specifies the python file that contains the dynamics function f of

Definition 1 and two symmetry-related functions: *symGamma* and *symGammaInv*. Given a $p \in P$ and a polytope[3] *poly* of dimension n representing a set of states of the agent, *symGamma* returns $\gamma_p(poly)$, where γ_p is the symmetry map to the virtual system. Similarly, *symGammaInv* would return $\gamma_p^{-1}(poly)$. The list of modes that the i^{th} agent transition between sequentially and their corresponding transitions conditions, denoted by guards, are specified as well and denoted by H_i. The guard of the j^{th} mode of the i^{th} agent $H_i[j].guard$ is a hyper-rectangle in the state space which when the agent reaches, it transitions to the $(j+1)^{st}$ mode. The guard $H_i[j]$ has time bound $H_i[j].T$ on how long the agent can stay in the mode. Moreover, it specifies the initial set of states for each agent as a hyper-rectangle. Finally, it specifies the static unsafe set U and the subset of dimensions $O \subseteq [n]$ that is relevant for dynamic safety checking between agents. If the reachtubes of two agents projected on O intersect each other, it would model a collision between the agents. For example, O would be $\{2,3\}$ for the aircraft model in Example 1 as $(\mathbf{x}[2], \mathbf{x}[3])$ represents its position.

CacheReach would return *unsafe* if the reachtubes of the agents starting from their initial sets of states and following the sequence of modes intersect a static unsafe set, or when projected to O, intersect each other. It would return *safe*, otherwise. Currently, CacheReach assumes that all agents share the same dynamics but do not interact. Hence, it has a single *tubecache* that is shared by all.

CacheReach computes the reachtubes of individual agents iteratively. It would compute the reachtube $mtube_i$ of the j^{th} mode of the i^{th} agent using `symComputeReachtube`. Then, it intersects it with the guard using the function *guardIntersect* to get the initial set $initset_i$ for the next mode. In addition to $initset_i$, *guardIntersect* computes the minimum and maximum times: $mintime_i$ and $maxtime_i$, respectively, at which $mtube_i$ intersects the guard. The value $mintime_i$ is the time at which a trajectory of the next mode may start at and $maxtime_i$ is the maximum such time. These values are used to check safety against time-annotated unsafe sets such as collision between agents.

The computed tube $mtube_i$ gets appended to $atube_i$ storing the full reachtube of the i^{th} agent. The benefit of this method is that now all modes of all agents can be mapped to a single virtual system. They can resuse each others reachtubes using *tubecache* that is getting updated at every call to `symComputeReachtube`. Moreover, the static safety is done in the usual way.

The collision between agents is done by the function checkDynamicSafety. It takes two full reachtubes of two agents $atube_1$ and $atube_2$ along with two arrays $lookback_1$ and $lookback_2$. For agent i, the array $lookback_i$ consists of pairs of integers $(ind_j, timerange_j)$ specifying the index identifying the beginning of the j^{th} mode tube in $atube_i$ and the uncertainty in the starting time of the trajectories from its initial set. checkDynamicSafety would use this information to time-align parts of $atube_1$ and $atube_2$ so that the intersection check happens only between two sets that may have been reached at the same time by the two agents.

[3] https://github.com/tulip-control/polytope

6.2 Experimental results

We ran experiments using our tool CacheReach on two models: a 3-dimensional linear dynamical system example and the nonlinear aircraft model described in Example 1. The linear model is of the form $\dot{\mathbf{x}} = A(\mathbf{x} - p[3:5])$, where $A = [[-3,1,0],[0,-2,1],[0,0,-1]]$, $\mathbf{x} \in \mathbb{R}^3$, and $p \in \mathbb{R}^6$. We considered scenarios with single, two, and three agents for each model following different sequences of waypoints. The sequences of waypoints for the linear model are translations and rotations of a digital-*S* shaped path. For the aircraft model, the paths are random crossing paths going north-east. In every scenario, all the agents have the same model. In the aircraft scenarios, the agent would switch to the next waypoint once its x, y position is within 0.5 units from the current waypoint in each dimension. The initial set of the aircraft was of size 1 in the position components, 0.1 in the speed, and 0.01 in the heading angle. We used Flow* [8] and DryVR [18] to compute reachtubes from scratch for the linear example. We only used DryVR for the aircraft model since our C++ Flow* wrapper does not handle a model having $\arctan_2$ in the dynamics. We ran all scenarios in CacheReach with and without using *tubecache*. The symmetry used for the aircraft was the one we showed in Example 3. For the linear model, the symmetry transformation γ_p that was used to map the state to the virtual system was a coordinate transformation where the new origin is at the next waypoint $p[3:5]$ and rotating the *xy*-plane by the angle between the previous and the next waypoints $p[0:2]$ and $p[3:5]$ projected to the plane. We compared the computation time with and without symmetry and show the results in Table 1. The reachtubes for three nonlinear and three linear agents are shown in Figure 1. The different colors represent reachtubes of different agents, the black points represent the waypoints, the black segments connect consecutive waypoints, and the red rectangles represent the unsafe sets. The figures on the top represent the real reachtubes while those on the bottom represent the ones corresponding to the virtual system saved in *tubecache*.

Table 1: Results.

tool \ agent model		Linear(1,2,and 3 agents)			aircraft(1,2,and 3 agents)		
Sym-DryVR	computed	57	90	90	635.23	1181.38	1550.62
	transformed	42	165	264	20.76	286.62	501.38
	time (min)	0.093	0.163	0.187	3.42	8.2	10.59
Sym-Flow*	computed	39.8	61.14	66.15			
	transformed	19.2	84.85	143.85	NA	NA	NA
	time (min)	0.387	0.62	0.684			
NoSym-DryVR	computed	99	255	354	656	1468	2052
	time (min)	0.062	0.355	0.52	3.71	10.78	15.47
NoSym-Flow*	computed	59	151	210			
	time (min)	0.53	1.328	1.5	NA	NA	NA

In Table 1, we call CacheReach, when ran with DryVR while using *tubecache*, Sym-DryVR, for symmetric DryVR. We call it Sym-Flow* if we are using Flow* instead. If we are not using *tubecache*, we call them NoSym-DryVR and NoSym-Flow*, respectively. Remember in `symComputeReachtube`, some tubes may be cached but they have shorter time horizons than the needed tube. So, we compute the rest from scratch. Here, we report the fractions of tubes computed from scratch and tubes that were transformed from cached ones. Moreover, we report the execution time till the tubes are

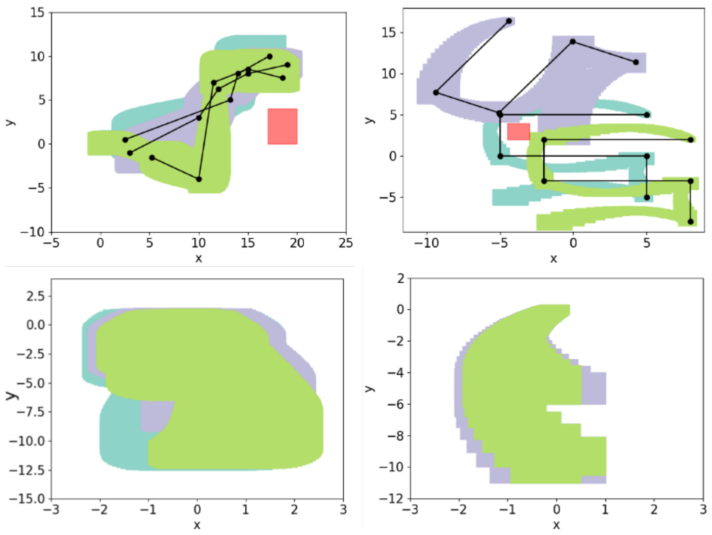


Fig. 1: Reachtubes for three fixed-wing aircrafts (left) and three linear models (right). Real reachtubes (top) vs. the virtual ones saved in *tubecache* (bottom).

computed. In the experiments, we always compute the full tubes even if it was detected to be unsafe earlier to have a fair comparison of running times. Moreover, the execution time does not include dynamic safety checking as the four versions of the experiments are doing the same computations for that purpose. We are using CacheReach in all scenarios with other reachability computation tools to decrease the degrees of freedom and show the benefits of transforming reachtubes over computing them. The Sym versions result in decrease of running time up-to 64% in the linear case with three agents. The ratio of transformed vs. computed tubes increases as the number of agents increase. This means that different agents are sharing reachtubes with each other in the virtual system. The total number of reachtubes is the same, whether *tubecache* is used or not. This means that the quality of the tubes, i.e. how tight they are, is the same whether we are transforming from *tubecache* or computing from scratch since the initial sets of modes are computed from intersections of reachtubes with guards. The fatter the reachtube is, the larger the initial set gets and the larger the number of reachtubes need to be computed.

7 Discussion and conclusions

In this paper, we investigated how symmetry transformations and caching can help achieve scalable, and possibly unbounded, verification of multi-agent systems. We developed a notion of *virtual system* which define symmetry transformations for a broad class of hybrid and dynamical agent models visiting waypoint sequences. Using virtual system, we present a prototype tool called CacheReach that builds a cache of reachtubes for the transformed virtual system, in a way that is agnostic of the representation of the reachsets and the reachability analysis subroutine used. Our experimental evaluation show significant improvement in computation time on simple examples and increased savings as number of agents increase.

References

1. Althoff, M.: An introduction to cora 2015. In: Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
2. Althoff, M., Dolan, J.M.: Online verification of automated road vehicles using reachability analysis. IEEE Trans. Robotics **30**(4), 903–918 (2014). https://doi.org/10.1109/TRO.2014.2312453
3. 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)
4. Bak, S., Tran, H., 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, Montreal, QC, Canada, April 16-18, 2019. pp. 23–32 (2019). https://doi.org/10.1145/3302504.3311792
5. Bonnabel, S., Martin, P., Rouchon, P.: Symmetry-preserving observers. IEEE Transactions on Automatic Control **53**(11), 2514–2526 (2008)
6. Chen, X.: Reachability analysis of non-linear hybrid systems using taylor models (2015)
7. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 8044, pp. 258–263. Springer Berlin Heidelberg (2013)
8. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 8044, pp. 258–263. Springer Berlin Heidelberg (2013)
9. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Computer Aided Verification (CAV 2010), Lecture Notes in Computer Science, vol. 6174, pp. 167–170. Springer (2010)
10. Donzé, A., Maler, O.: Systematic simulation using sensitivity analysis. In: Hybrid Systems: Computation and Control, pp. 174–189. Springer (2007)
11. Duggirala, P.S., Fan, C., Mitra, S., Viswanathan, M.: Meeting a powertrain verification challenge. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 536–543 (2015). https://doi.org/10.1007/978-3-319-21690-4_37
12. Duggirala, P.S., Mitra, S., Viswanathan, M.: Verification of annotated models from executions. In: EMSOFT (2013)
13. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2e2: A verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 68–82. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
14. Duggirala, P.S., Viswanathan, M.: Parsimonious, simulation based verification of linear systems. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. pp. 477–494 (2016). https://doi.org/10.1007/978-3-319-41528-4_26
15. Fan, C., Kapinski, J., Jin, X., Mitra, S.: Locally optimal reach set over-approximation for nonlinear systems. In: EMSOFT. pp. 6:1–6:10. ACM (2016)
16. Fan, C., Mitra, S.: Bounded verification with on-the-fly discrepancy computation. In: ATVA. Lecture Notes in Computer Science, vol. 9364, pp. 446–463. Springer (2015)
17. Fan, C., Qi, B., Mitra, S.: Data-driven formal reasoning and their applications in safety analysis of vehicle autonomy features. IEEE Design & Test **35**(3), 31–38 (2018). https://doi.org/10.1109/MDAT.2018.2799804
18. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: Data-driven verification and compositional reasoning for automotive systems. In: Computer Aided Verification. pp. 441–461. Springer International Publishing (2017)

19. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. pp. 531–538 (2016). https://doi.org/10.1007/978-3-319-41528-4_29
20. G'erard, L., Slotine, J.J.E.: Neuronal networks and controlled symmetries, a generic framework (2006)
21. Hartmanns, A., Seidl, M.: tacas20ae.ova. figshare (2019). https://doi.org/10.6084/m9.figshare.9699839.v2
22. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences **57**(1), 94 – 124 (1998), http://www.sciencedirect.com/science/article/pii/S0022000098915811
23. Johnson, T., Mitra, S.: A small model theorem for rectangular hybrid automata networks (2012)
24. Kushner, T., Bequette, B.W., Cameron, F., Forlenza, G.P., Maahs, D.M., Sankaranarayanan, S.: Models, devices, properties, and verification of artificial pancreas systems. In: Automated Reasoning for Systems Biology and Medicine, pp. 93–131 (2019). https://doi.org/10.1007/978-3-030-17297-8_4
25. Mehta, P., Hagen, G., Banaszuk, A.: Symmetry and symmetry-breaking for a wave equation with feedback. SIAM J. Applied Dynamical Systems **6**, 549–575 (01 2007). https://doi.org/10.1137/060666044
26. Mitra, S.: Verifying Cyberphysical Systems: A path to safe autonomy. To be published by MIT Press, Cambridge, MA, USA (2020), https://sayanmitracode.github.io/cpsbooksite/
27. Russo, G., Slotine, J.J.E.: Symmetries, stability, and control in nonlinear systems and networks. Physical Review E **84**(4), 041929 (2011)
28. Sibai, H., Mokhlesi, N., Fan, C., Mitra, S.: Cachereach: multi-agent safety verification using symmetry transformations software tool (2020). https://doi.org/10.6084/m9.figshare.11874375
29. Sibai, H., Mokhlesi, N., Mitra, S.: Using symmetry transformations in equivariant dynamical systems for their safety verification. In: Automated Technology for Verification and Analysis. pp. 1–17 (2019)
30. Spong, M.W., Bullo, F.: Controlled symmetries and passive walking. IEEE Transactions on Automatic Control **50**(7), 1025–1031 (July 2005). https://doi.org/10.1109/TAC.2005.851449

Relational Differential Dynamic Logic*

Juraj Kolčák[1], Jérémy Dubut[2,3], Ichiro Hasuo[2,4], Shin-ya Katsumata[2], David Sprunger[2], and Akihisa Yamada[2]

[1] LSV, CNRS & ENS Paris-Saclay, Université Paris-Saclay, Cachan, France
`kolcak@lsv.fr`
[2] National Institute of Informatics, Tokyo, Japan
`{dubut,hasuo,s-katsumata,sprunger,akihisayamada}@nii.ac.jp`
[3] Japanese-French Laboratory for Informatics, CNRS IRL 3527, Tokyo, Japan
[4] The Graduate University for Advanced Studies (SOKENDAI), Tokyo, Japan

Abstract. In the field of quality assurance of hybrid systems, Platzer's *differential dynamic logic* (dL) is widely recognized as a deductive verification method with solid mathematical foundations and sophisticated tool support. Motivated by case studies provided by our industry partner, we study a *relational extension* of dL, aiming to formally prove statements such as "an earlier engagement of the emergency brake yields a smaller collision speed." A main technical challenge is to combine two dynamics, so that the powerful inference rules of dL (such as the differential invariant rules) can be applied to such relational reasoning, yet in such a way that we relate two different time points. Our contributions are a semantical theory of *time stretching*, and the resulting *synchronization* rule that expresses time stretching by the syntactic operation of Lie derivative. We implemented this rule as an extension of KeYmaera X, by which we successfully verified relational properties of a few models taken from the automotive domain.

Keywords: hybrid system · cyber-physical system · formal verification · theorem proving · dynamic logic.

1 Introduction

Hybrid Systems *Cyber-physical systems* (CPSs) have been studied as a subject in their own right for over a decade, but the rise of *automated driving* in the last few years has created a panoply of challenges in the quality assurance of these systems. In the foreseeable future, millions of cars will be driving on streets

* Thanks are due to Stefan Mitsch, André Platzer, and Yong Kiam Tan for useful tips on the KeYmaera X source code; and to Kenji Kamijo, Yoshiyuki Shinya, and Takamasa Suetomi from Mazda Motor Corporation for helpful discussions. The authors are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. I.H. is supported by Grant-in-Aid No. 15KT0012, JSPS. J.D. is supported by Grant-in-aid No. 19K20215, JSPS. The work was done during J.K.'s internship at the National Institute of Informatics, Tokyo, Japan.

with unprecedented degrees of automation; ensuring the safety and reliability of these automated driving systems is a pressing social and economic challenge.

The *hybridity* of cyber-physical systems, the combination of continuous physical dynamics and discrete digital control, poses unique scientific challenges. To address these challenges, two communities have naturally joined forces: *control theory* whose traditional application domain is continuous dynamics and *formal methods* that have mainly focused on the analysis of software systems. This has been a fruitful cross-pollination: techniques from formal methods such as bisimilarity [9] and temporal logic specification [8] have been imported to control theory, and conversely, control theory notions such as Lyapunov functions have been used in formal methods [26].

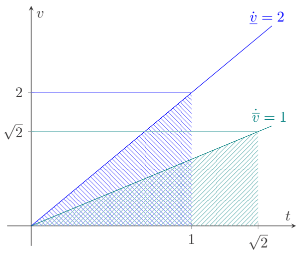
Deductive Verification of Hybrid Systems In the formal methods community, two major classes of techniques are *model checking* (usually automata-based and automatic) and *deductive verification* (based on logic and can be automated or interactive). Model checking techniques rely on exhaustive search in state spaces and therefore cannot be applied *per se* to hybrid systems with infinite state spaces. This has led to the active study of *discrete abstraction* of hybrid dynamics, see e.g. [9]; or of *bounded model checking*, see [5].

In contrast, nothing immediately rules out the use of the deductive approach for hybrid systems. Finitely many variables in logical formulas can represent infinitely many states, and proofs in suitably designed logics are valid even when the semantic domain is uncountable. That said, designing such a logic, proving the soundness of its rules, and showing that logics is actually *useful* in hybrid system verification is a difficult task.

Platzer's *differential dynamic logic* dL [21] is a remarkable success in this direction. Its syntax is systematic and intuitive, extending the classic formalism of *dynamic logic* [10] with differential equations as programs. Its proof rules encapsulate several essential proof principles about differential equations, including a *differential invariant* (DI) rule for universal properties and *side deduction* for existential properties. The logic dL has served as a general platform that accommodates a variety of techniques, including those which come from real algebraic geometry [22]. Furthermore, dL comes with sophisticated tool support: the latest tool KeYmaera X [15] comes with graphical interface for interactive proving and a number of automation heuristics.

Relational Reasoning on Hybrid Systems In this work, we introduce proof-based techniques for *relational reasoning* to the deductive verification of hybrid systems. Here, by relational reasoning we mean analyzing how changes in the system will affect the overall system behavior. One of the applications of such reasoning in our mind is to deduce the safety of a system by checking the most aggressive settings. To make such reduction sound, we need to verify that less aggressive versions result in less dangerous outcomes than the aggressive ones. As a simple example, consider the following case distilled from our collaboration with an industrial partner.

Example 1 (leading example: collision speed). Consider two cars $\overline{\mathrm{C}}$ and $\underline{\mathrm{C}}$, whose positions and velocities are real numbers denoted by $\overline{x}, \underline{x}$ and $\overline{v}, \underline{v}$,



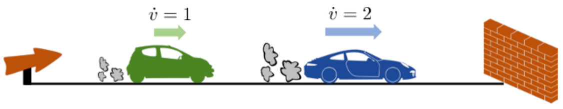
The two hatched areas designate the traveled distances ($\overline{x} = \underline{x} = 1$). We can compute the collision speeds ($\overline{v} = \sqrt{2}$ and $\underline{v} = 2$) via the closed-form solutions of the differential equations (1), concluding $\overline{v} \leq \underline{v}$ when $\overline{x} = \underline{x} = 1$.

Fig. 1. An ad-hoc proof for Example 1

respectively. Their dynamics are governed by the following differential equations:

$$\dot{\overline{x}} = \overline{v}, \quad \dot{\overline{v}} = 1; \qquad \dot{\underline{x}} = \underline{v}, \quad \dot{\underline{v}} = 2. \tag{1}$$

Both cars start at the same position at rest ($\overline{x} = \underline{x} = 0 \wedge \overline{v} = \underline{v} = 0$), and both drive towards a wall at position 1. We consider this question: *which car is traveling faster when it hits the wall?*



The second car, $\underline{\mathrm{C}}$, has strictly greater acceleration all the time, so we can imagine that $\underline{\mathrm{C}}$ hits the wall harder. This hypothesis turns out to be correct, but we are more interested in how this claim could be proven.

A simple proof would be to solve the differential equation exactly and notice $\underline{\mathrm{C}}$ has greater velocity at the end of its run. However, it is known that closed-form solutions are scarce for ODEs—we want a proof method that is more general.

Another possible argument is based on the relationship between the accelerations. Since the second car's acceleration is greater at every point in time, we might be tempted to conclude that the second car's velocity must always be greater than the first car's, based on the monotonicity of integration: $\overline{a}(t) \leq \underline{a}(t) \Rightarrow \overline{v}(t) = \int_0^T \overline{a}(t)\,\mathrm{d}t \leq \int_0^T \underline{a}(t)\,\mathrm{d}t = \underline{v}(t)$. However, this reasoning has a flaw. $\underline{\mathrm{C}}$ reaches the wall at an earlier point in time than $\overline{\mathrm{C}}$, and therefore $\overline{\mathrm{C}}$ has more time to accelerate. In the end, we have to compare $\int_0^{\overline{T}} \overline{a}(t)\,\mathrm{d}t$ and $\int_0^{\underline{T}} \underline{a}(t)\,\mathrm{d}t$ where $\overline{a}(t) \leq \underline{a}(t)$ for all $t \in [0, \underline{T}]$ but $\overline{T} > \underline{T}$, as depicted in figure 1.

Our solution, roughly stated, is to compare the two cars at the same points *in space* by reparametrizing time for one of the two cars. This parametrization is specially chosen to ensure the two cars pass through the same points in space at the same points in time.

Our current work is about a logical infrastructure needed to support this kind of relational reasoning comparing two different dynamics, based on dL. Our semantical theory, as well as the resulting syntactic extension of dL by what we call the synchronization rule, generalizes the kind of reasoning in Example 1 using the notion of *time stretching*.

Technical Contributions We make the following technical contributions.

1. **Formulation of relational reasoning in dL**. We find that relational properties are expressible in dL, using disjoint variables in a sequential composition. This representation, however, does not allow the use of the rich logical infrastructure of dL (such as the (DI) rule).
2. **Time stretching, semantically and syntactically**. To alleviate this difficulty, we first develop the theory of *time stretching*, so that we can compare two dynamics at different timepoints (cf. Example 1). Accommodating this semantical notion in dL and KeYmaera X is not possible *per se*. We introduce an indirect syntactic alternative, which turns out to be better suited in fact to many case studies (where we compare the two dynamics at the same "position," much like in Example 1). The resulting *synchronization* rule in dL has a clean presentation (Theorem 24), owing to the syntactic Lie derivative operator in dL.
3. **Implementation and case studies**. We implemented the new synchronization rule as an extension of KeYmaera X. We used it successfully for establishing nontrivial relational properties in case studies taken from the automotive domain.

Relational Reasoning in Practice We contend relational reasoning has practical significance based on our collaboration with an industry partner. Relational properties, especially with an aspect of *monotonicity*, abound in real-world examples. In particular, we have often encountered situations where we have a parametrized model $M(p)$ and need to show a property of the form:

$$p_1 < p_2 \text{ implies } M(p_2) \text{ is less safe than } M(p_1). \tag{2}$$

These properties occur especially in the context of *product lines*, where the same model can come in many slight variants. Example 1 is such a situation.

Relational statements (such as monotonicity) are easy to state and interpret. Intuitions about the *direction of* the change in a behavior of a system resulting from the change of a parameter are more often valid than intuitions about the *amount of* such a change. These kinds of simple statements are often used by engineers to establish the basic credibility of a model. Qualitative, relational properties also tend to be easier to prove than exact, quantitative properties.

Finally, monotonicity can serve as a powerful technique in *test-case reduction*. If a safety property is too complex to be deductively verified, one usually turns to testing. It is often still possible to establish a simple monotonicity property of the form (2). This can powerfully boost testing efforts: one can focus exclusively on establishing safety for the extreme case $M(p_{\max})$.

Related Work Since this work is about its relational extension, the works we mentioned on dL are naturally relevant. We discuss other related works here.

Simulink (Mathworks, Inc.) is an industry standard in modeling hybrid systems, but unfortunately Simulink models do not come with rigorously defined semantics. Therefore, while integration with Simulink is highly desirable any quality assurance methods for hybrid systems, formal verification methods require some work to set up the semantics for Simulink models. The recent work [12] tackles this problem, identifying a fragment of Simulink, and devising a translator from Simulink models to dL programs. Their translation is ingenious, and their tool is capable of proving rather complicated properties when used in combination with KeYmaera X [15].

Relational extensions of the *Floyd–Hoare logic*—which can be thought of as a discrete-time version of dL—have been energetically pursued especially in the context of *differential privacy* [4,2,3].

In deductive verification of hybrid systems, an approach alternative to dL uses *nonstandard analysis* [23] and regards continuous dynamics as if they were discrete due to the existence of infinitesimal elements [24,25]. The logic used in that framework is exactly the same as the classic Floyd–Hoare logic, and the soundness of the logic in the hybrid setting is shown by a model-theoretic result called the *transfer principle*. Its tool support has been pursued as well [11].

This is not the first time that relational reasoning—in a general sense—has been pursued in dL. Specifically, Loos and Platzer introduce the *refinement* primitive $\beta \leq \alpha$, which asserts a refinement relation between two hybrid dynamics, meaning the set of successor states of β is included in that of α [14]. This kind of relation is inspired by the software engineering paradigm of incremental modeling (supported by languages and tools such as Event-B [1,6]); the result is a rigorous deductive framework for refining an abstract model (with more nondeterminism) into a more concrete one (with less nondeterminism). In contrast, we compare one concrete model (not necessarily with nondeterminism) with another. Thus, our notion of relational reasoning builds more on relational extensions of the Floyd–Hoare logic [4,2,3] than on Event-B. Combining these two orthogonal kinds of relational extensions of dL is important future work.

Organization In Section 2, we recall some basics of differential dynamic logic dL: its syntax, semantics and some proof rules. Our main goal, relational reasoning, is formulated in Section 3, where we identify difficulties in doing so in the original dL. In Section 4 we introduce the semantical notion of time stretching, and turn its theory into the new synchronization rule. After introducing our implementation in Section 5, we describe our three case studies in Section 6.

The appendix containing omitted proofs and details, the source code and the artifact are found at http://group-mmm.org/rddl_tacas_2020/.

2 Preliminaries: Syntax and Semantics of the Logic dL

We recall some of the basics of *differential dynamic logic* (dL). The interested reader is referred to [19,20] for full details.

Definition 2 (language). We fix a set $\mathcal{V}$ of *variables*, denoted by $x, y, \ldots$. The set of *terms* is defined by the following grammar:

$$e, f, g, \ldots \ ::= \ x \mid n \mid -e \mid e + f \mid e \cdot f \mid e/f$$

where $x \in \mathcal{V}$ and $n \in \mathbb{N}$. First-order *formulas* are defined by

$$P, Q, \ldots \ ::= \ e \leq f \mid \neg P \mid P \wedge Q \mid \forall x.\, P$$

A *state* is a function mapping each variable to a real number, $\omega : \mathcal{V} \to \mathbb{R}$. We denote the set of all states by $\mathbb{R}^{\mathcal{V}}$. Given a state, each term has a valuation in the reals, and each formula has a valuation in Booleans defined by the usual induction. We denote these by $[\![e]\!]_\omega \in \mathbb{R}$ and $[\![P]\!]_\omega \in \{\text{TRUE}, \text{FALSE}\}$, respectively. The *models* of a first-order formula P are the states satisfying P, $[\![P]\!] := \{\omega \in \mathbb{R}^{\mathcal{V}} \mid [\![P]\!]_\omega = \text{TRUE}\}$.

We use classical shorthands, including $e = f \ := \ e \leq f \ \wedge \ f \leq e$, $P \vee Q \ := \ \neg(\neg P \wedge \neg Q)$, $\exists x.\, P \ := \ \neg(\forall x.\, \neg P)$, and $\top := 0 \leq 0$. We denote a vector $(e_1, \ldots, e_n)$ of terms (or variables) by $\mathbf{e}$ when the length n is irrelevant or clear from the context.

We now introduce the syntax of hybrid programs.

Definition 3 (hybrid programs). The set $\mathcal{HP}(\mathcal{V})$ of *hybrid programs* over variables $\mathcal{V}$ is given by the following grammar:

$$\alpha_1, \alpha_2, \ldots \ ::= \ ?P \ \mid \ x := e \mid \dot{x}_1 = e_1, \ldots, \dot{x}_n = e_n \,\&\, Q \ \mid \ \alpha_1; \alpha_2 \ \mid \ \alpha_1 \cup \alpha_2 \mid \alpha_1^*$$

We may also abbreviate $\dot{x}_1 = e_1, \ldots, \dot{x}_n = e_n$ by $\dot{\mathbf{x}} = \mathbf{e}$. Hybrid programs of the form $\dot{\mathbf{x}} = \mathbf{e} \,\&\, Q$ are especially important in this work. We call such a program *differential dynamics*, where $\dot{\mathbf{x}} = \mathbf{e}$ is its *differential equation* and the first-order formula Q is its *evolution domain constraint*. The intuitive meaning of such a program is that the values of the variables $\mathbf{x}$ evolve continuously in time according to $\dot{\mathbf{x}} = \mathbf{e}$, as long as Q is satisfied at the current value of $\mathbf{x}$. If we see differential dynamics as a continuous analog of loops, then Q plays the role of guard and $\dot{\mathbf{x}} = \mathbf{e}$ plays the role of body.[5] We write $\dot{\mathbf{x}} = \mathbf{e}$ instead of $\dot{\mathbf{x}} = \mathbf{e} \,\&\, \top$.

Definition 4 (solutions). A mapping $\psi : [0, T) \to \mathbb{R}^{\mathcal{V}}$ with $T \in [0, \infty]$ is called a *solution* of a differential equation $\dot{x}_1 = e_1, \ldots, \dot{x}_n = e_n$ if ψ is differentiable in $[0, T)$ and, whenever $t \in [0, T)$, $\dot{\psi}(t)(x_i) = [\![e_i]\!]_{\psi(t)}$ for $i \in \{1, \ldots, n\}$ and $\dot{\psi}(t)(y) = 0$ for any $y \in \mathcal{V} \setminus \{x_1, \ldots, x_n\}$.

According to the Picard–Lindelöf theorem [13], for each differential equation $\dot{\mathbf{x}} = \mathbf{e}$ and each state ω, there is a unique maximal solution $\psi_\omega : [0, T_\omega) \to \mathbb{R}^{\mathcal{V}}$ of the differential equation satisfying $\psi_\omega(0) = \omega$.

Definition 5 (semantics of hybrid programs). The *semantics* of a hybrid program α is a relation $-[\![\alpha]\!]\!\to \ \subseteq \mathbb{R}^{\mathcal{V}} \times \mathbb{R}^{\mathcal{V}}$ on states, defined by:

[5] This analogy is not perfect: a typical while loop can only exit when its guard is false, whereas a hybrid program can exit the differential dynamics while Q is satisfied.

1. $-\!\llbracket ?P \rrbracket\!\!\rightarrow = \{(\omega,\omega) \mid \omega \in \llbracket P \rrbracket\}$,
2. $-\!\llbracket x := e \rrbracket\!\!\rightarrow = \{(\omega,\omega') \mid \omega'(x) = \llbracket e \rrbracket_\omega$ and $\omega'(y) = \omega(y)$ for all $y \neq x\}$,
3. $-\!\llbracket \dot{\mathbf{x}} = \mathbf{e} \,\&\, Q \rrbracket\!\!\rightarrow = \{(\omega, \psi_\omega(t)) \mid \omega \in \mathbb{R}^{\mathcal{V}},\ t \in [0, T_\omega), \psi_\omega([0,t]) \subseteq \llbracket Q \rrbracket\}$,
4. $-\!\llbracket \alpha_1 \cup \alpha_2 \rrbracket\!\!\rightarrow = -\!\llbracket \alpha_1 \rrbracket\!\!\rightarrow \cup -\!\llbracket \alpha_2 \rrbracket\!\!\rightarrow$,
5. $-\!\llbracket \alpha_1 ; \alpha_2 \rrbracket\!\!\rightarrow = -\!\llbracket \alpha_1 \rrbracket\!\!\rightarrow ; -\!\llbracket \alpha_2 \rrbracket\!\!\rightarrow$ where ; denotes relation composition, and
6. $-\!\llbracket \alpha^* \rrbracket\!\!\rightarrow = (-\!\llbracket \alpha \rrbracket\!\!\rightarrow)^*$ where * denotes the reflexive transitive closure.

Definition 6 (dL formulas). *Modal formulas* extend first-order formulas and are defined by the following grammar:

$$\varphi, \varphi_1, \varphi_2, \ldots ::= e \le f \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \forall x.\, \varphi \mid [\alpha]\varphi.$$

As usual, we write $\langle\alpha\rangle\varphi$ to abbreviate $\neg[\alpha]\neg\varphi$. We will also call modal formulas "dL formulas" since these are the widest class of formulas in dL.

The Boolean valuation $\llbracket \varphi \rrbracket_\omega$ of a modal formula φ in a state ω is defined in the same way as for first-order formulas, with the addition of $\llbracket [\alpha]\varphi \rrbracket_\omega = \text{TRUE}$ if and only if $\llbracket \varphi \rrbracket_{\omega'} = \text{TRUE}$ for all ω' such that $\omega -\!\llbracket \alpha \rrbracket\!\!\rightarrow \omega'$.

We take the sequent-calculus style proof system for dL, following [22]. It has judgments of the form $\Gamma \vdash \varphi$, where Γ is a set of modal formulas and φ is a single modal formula. One of the most fundamental axiom is

$$[\dot{\mathbf{x}} = \mathbf{e} \,\&\, Q]\phi \Longleftrightarrow \forall t \ge 0.\, (\forall v \in [0,u].\ [x := f(v)]Q) \Rightarrow [x := f(u)]\phi \quad \text{(solve)}$$

where $f(t)$ is a term with a fresh variable t such that $\llbracket f \rrbracket$ is a solution of $\dot{\mathbf{x}} = \mathbf{e}$ and $\llbracket f(0) \rrbracket = \mathsf{id}$.

Some other rules of dL, such as the differential invariant rule (DI) that is central in many proofs, are introduced later in Definition 13.

3 Relational Differential Dynamic Logic

Intuitively, we want a way to describe two dynamics that are executed in parallel, and compare their outputs. In terms of (nondeterministic) transition systems, parallel composition is available via tensor products.

Definition 7 (tensor product). Given two transition systems (S, R) and (S', R'), their *tensor product* $(S \times S', R \otimes R')$ is defined to be the transition system whose transition relation is given by

$$R \otimes R' := \{(s, s'), (t, t') \mid (s,t) \in R, (s', t') \in R'\}.$$

No extension of the dL syntax is needed to model tensor products: disjointness of the variables of the two systems suffices. From now on we split variables into two disjoint sets: $\mathcal{V} = \overline{\mathcal{V}} \uplus \underline{\mathcal{V}}$. We denote variables in $\overline{\mathcal{V}}$ by $\overline{x}, \overline{y}, \ldots$ and those in $\underline{\mathcal{V}}$ by $\underline{x}, \underline{y}, \ldots$. Terms in $\mathcal{T}(\overline{\mathcal{V}})$, first-order formulas in $\mathcal{F}ml(\overline{\mathcal{V}})$, and programs in $\mathcal{HP}(\overline{\mathcal{V}})$ are denoted by $\overline{e}, \overline{f}, \ldots, \overline{P}, \overline{Q}, \ldots$, and $\overline{\alpha}, \overline{\beta}, \ldots$, and similarly for the corresponding constructs with $\underline{\mathcal{V}}$.

An easy proof of the following fact can be found in the appendix.

Proposition 8. $\dashv\!\llbracket\overline{\alpha}\rrbracket\!\mapsto \otimes \dashv\!\llbracket\underline{\alpha}\rrbracket\!\mapsto = \dashv\!\llbracket\overline{\alpha};\underline{\alpha}\rrbracket\!\mapsto$ □

Scenarios with two parallel differential dynamics are the main focus of this work. We formalize an assertion relating two dynamics using the following format. It is a syntactic counterpart of Proposition 8.

Definition 9 (relational differential dynamics). We call hybrid programs of the following form *relational differential dynamics (RDD)*

$$\dot{\overline{\mathbf{x}}} = \overline{\mathbf{e}} \,\&\, \overline{Q}\ ; \quad \dot{\underline{\mathbf{x}}} = \underline{\mathbf{e}} \,\&\, \underline{Q} \tag{3}$$

Now that we have ways to express separate systems evolving in parallel, we turn to the construction of proofs which reason about their relationships.

Example 10. Using RDD, the problem in Example 1 is expressed as $\Gamma_C \vdash \left[\overline{\delta_C};\underline{\delta_C}\right]\phi_C$ where $\overline{\delta_C} := \left(\dot{\overline{x}} = \overline{v}, \dot{\overline{v}} = 1\right)$, $\underline{\delta_C} := (\dot{\underline{x}} = \underline{v}, \dot{\underline{v}} = 2)$, $\Gamma_C := \{\overline{x} = \underline{x} = 0,\ \overline{v} = \underline{v} = 0\}$ is the precondition, and $\phi_C := (\overline{x} = \underline{x} = 1 \Rightarrow \overline{v} \leq \underline{v})$ is the postcondition.

Let us prove, in KeYmaera X, the RDD sequent $\Gamma_C \vdash \left[\overline{\delta_C};\underline{\delta_C}\right]\phi_C$. In KeYmaera X, the only applicable rule to this sequent turns it into $\Gamma_C \vdash \left[\overline{\delta_C}\right]\left[\underline{\delta_C}\right]\phi_C$. We then explicitly "solve" the second dynamics, yielding the following goal:

$$\Gamma_C \vdash \left[\overline{\delta_C}\right]\forall \underline{t} \geq 0.\left(\overline{x} \ = \ \underline{x} + \underline{v}\cdot\underline{t} + \underline{t}^2 = 1 \ \Rightarrow \ \overline{v} \ \leq \ \underline{v} + \underline{t}\right) \tag{4}$$

where $\underline{x}$ and $\underline{v}$ in ϕ_C are replaced by their explicit solutions with respect to the freshly introduced time variable $\underline{t}$. Again differential invariant rules do not apply to (4), so one must solve the first dynamics, too, yielding

$$\Gamma_C \vdash \forall \overline{t} \geq 0.\,\forall \underline{t} \geq 0.\left(\overline{x} + \overline{v}\cdot\overline{t} + \overline{t}^2/2 \ = \ \underline{x} + \underline{v}\cdot\underline{t} + \underline{t}^2 \ = \ 1 \ \Rightarrow \ \overline{v} + \overline{t} \ \leq \ \underline{v} + \underline{t}\right)$$

Since this goal is first order, the quantifier elimination, a central proof technique in KeYmaera X [18], proves the goal.

The above example worked out since it admits explicit solutions expressible in dL. This is not always the case as the following example demonstrates.

Example 11. We consider two objects moving through fluids subjected to different kinds of drag. One object moves through a viscous fluid and is therefore subject to linear drag; its dynamics are $\underline{\delta_F} := (\dot{\underline{x}} = \underline{v}, \dot{\underline{v}} = -\underline{v})$.

The other object moves through a less viscous fluid and is subject to turbulent drag; its dynamics are $\overline{\delta_F} := (\dot{\overline{x}} = \overline{v}, \dot{\overline{v}} = -\overline{v}^2)$. Our goal is to show that the latter has higher speed when both objects reach a certain point in space $(\overline{x} = \underline{x} = l)$.

The following functions $\underline{v}^*, \underline{x}^*, \overline{v}^*$ and $\overline{x}^*$ are solutions of the dynamics.

$$\underline{v}^*(\underline{t}) = \underline{v}_0 \cdot e^{-\underline{t}} \qquad\qquad \underline{x}^*(\underline{t}) = \underline{x}_0 + \underline{v}_0 \cdot (1 - e^{-\underline{t}})$$

$$\overline{v}^*(\overline{t}) = \frac{\overline{v}_0}{1 + \overline{v}_0 \cdot \overline{t}} \qquad\qquad \overline{x}^*(\overline{t}) = \overline{x}_0 + \log(1 + \overline{v}_0 \cdot \overline{t})$$

where $\underline{v}_0$ etc. denote the initial values. Unfortunately, we cannot express exponentiations and logarithms in KeYmaera X, and thus the "solve" rule that we used in Example 10 cannot be applied here.

One obvious solution to this would be to add support for exponentiations and logarithms in KeYmaera X, but this would break the decidability of the underlying first order logic, which is a major feature of dL [18]. In fact, the same issue occurs even in standard use cases of KeYmaera X, and motivated the introduction of proof rules which do not demand explicit solutions to differential dynamics [20,22] using the *Lie derivative*.

Definition 12 (formal Lie derivative in dL from [20,22]). The formal Lie derivative of a term f along dynamics $\delta \equiv (\dot{\mathbf{x}} = \mathbf{e} \,\&\, Q)$ of dimension n is a dL term $\mathcal{L}_\delta\, f \in \mathcal{T}(\mathcal{V})$ given by[6]

$$\mathcal{L}_\delta\, f := \tfrac{\partial}{\partial x_1} f \cdot e_1 + \cdots + \tfrac{\partial}{\partial x_n} f \cdot e_n$$

Definition 13 (proof rules from [20,22]). The following rules are sound:

$$\frac{\Gamma, Q \vdash f \sim 0 \quad \Gamma \vdash [\delta]\mathcal{L}_\delta\, f \simeq 0}{\Gamma \vdash [\delta] f \sim 0}\ \text{DI} \qquad \frac{\Gamma \vdash p \sim 0 \quad Q \vdash \mathcal{L}_\delta\, p \simeq g \cdot p}{\Gamma \vdash [\delta] p \sim 0}\ \text{Dbx}$$

where $\delta \equiv (\dot{\mathbf{x}} = \mathbf{e} \,\&\, Q)$, $(\sim, \simeq) \in \{\, (=,=), (>,\geq), (\geq,\geq) \,\}$, and g is any term without division.

The differential invariant rule (DI) is the central rule for proving safety properties [20,22]: it reduces a global property of the dynamics to local reasoning by means of Lie derivatives. The Darboux inequality rule (Dbx) is derived from real algebraic geometry; see e.g. [22].

Example 14. Consider an example differential dynamics in one variable, $\dot{x} = 2$. Suppose we want to show that $x \geq 0$ holds after following these dynamics for any amount of time, starting from $x = 1$. One way to do this is to show that (1) this predicate holds initially and (2) the time derivative of x is always nonnegative. These are precisely the two premises of the (DI) rule: to show the sequent $x = 1 \vdash [\dot{x} = 2]x \geq 0$ (DI) requires us to prove (1) $x = 1 \vdash x \geq 0$ and (2) $x = 1 \vdash [\dot{x} = 2]\mathcal{L}_{\dot{x}=2}\, x \geq 0$, where $\mathcal{L}_{\dot{x}=2}\, x = 2$. Note that we give an initial condition $x = 1$ in the precedent of this sequent.

4 Synchronizing Dynamics

The intuitive explanation of the RDD construction of Definition 9 is a "serialization" of two dynamics. This construction however does not match the (DI)

[6] It is easy to see that the *derivative* of a term $t \in \mathcal{T}(\mathcal{V})$ with respect to $x \in \mathcal{V}$ can be given as a dL term $\frac{\partial}{\partial x} e \in \mathcal{T}(\mathcal{V})$ such that $[\![\frac{\partial}{\partial x} e]\!] = \frac{\partial}{\partial x} [\![e]\!]$. The definition of $\frac{\partial}{\partial x} e$ is inductive with respect to the term e.

and (Dbx) rules, as they accept only one dynamics followed by a comparison. In order to make use of these rules in our relational reasoning, we introduce another proof method. It "synchronizes" two dynamics.

After some theoretical preparations we define the new rule and prove its soundness. We will illustrate the usefulness of this rule in Section 6, through some case studies that are inspired by our collaboration with the industry.

4.1 Time Stretching

A key theoretical tool towards the soundness of our synchronization rule is called *time stretching*. Its idea is very similar to the technique of *time-reparametrization* for ODEs [7].

Definition 15 (time stretch function). Let $T \in \mathbb{R}_{\geq 0}$. A function $K : [0,T] \to \mathbb{R}_{\geq 0}$ is a *time stretch function* if $K(0) = 0$, K is continuously differentiable and $\dot{K}(t) > 0$ for each $t \in [0,T]$.

Remark 16. The condition $\dot{K}(t) > 0$ ensures that K is strictly increasing and is a bijection from $[0,T]$ to $[0,K(T)]$. The inverse of K is $K^{-1} : [0,K(T)] \to [0,T]$, and it is straightforward to check K^{-1} is another time stretch function.

The next results tell us how to turn an ODE into another, given a time stretching function K, so that a time-stretch $\psi \circ K$ of a solution ψ of one becomes a solution of the other.

Lemma 17. *Suppose $f : \mathbb{R}^{\mathcal{V}} \to \mathbb{R}^{\mathcal{V}}$ is a vector field and $K : [0,T] \to [0,K(T)]$ is a time stretch function. If $\psi : [0,K(T)) \to \mathbb{R}^{\mathcal{V}}$ satisfies $\dot{\psi}(s) = f(\psi(s))$ for all $s \in [0,K(T))$, then the function $\rho = \psi \circ K : [0,T) \to \mathbb{R}^{\mathcal{V}}$ satisfies $\dot{\rho}(t) = \dot{K}(t) \cdot f(\rho(t))$ for all $t \in [0,T)$.*

Proof. We have $\dot{\rho}(t) = \dot{K}(t) \cdot \dot{\psi}(K(t)) = \dot{K}(t) \cdot f(\psi(K(t))) = \dot{K}(t) \cdot f(\rho(t))$, where the first equality is by the definitions and the chain rule, the second equality is by the assumption on $\dot{\psi}$, and the last equality is by the definition of ρ. □

Since the inverse of a time stretch function is another time stretch function, we obtain the following corollary of Lemma 17.

Corollary 18. *Let $K : [0,T] \to [0,K(T)]$ be a time stretch function. Let $\rho : [0,T) \to \mathbb{R}^{\mathcal{V}}$ satisfy $\dot{\rho}(t) = \dot{K}(t) \cdot f(\rho(t))$ whenever $0 \leq t < T$. Then the function $\psi : [0,K(T)) \to \mathbb{R}^{\mathcal{V}}$, defined by $\psi(s) := \rho(K^{-1}(s))$, satisfies $\dot{\psi}(s) = f(\psi(s))$ whenever $0 \leq s < K(T)$.* □

4.2 Towards a Syntactic Representation

So far our time-stretch function K has been a semantical object. Here we introduce a syntactic way of reasoning via time-stretch functions. Since a desired time-stretch function is not necessarily expressible in dL, our syntactic reasoning uses an indirect method that exploits a pair of functions called a synchronizer. We will be eventually led to a syntactic reasoning rule (Sync) (Thm. 24).

Given a term $g \in \mathcal{T}(X)$ and a mapping $\psi : [0,T) \to \mathbb{R}^X$, we define $g_\psi : [0,T) \to \mathbb{R}$ by

$$g_\psi(t) := [\![g]\!]_{\psi(t)}. \tag{5}$$

Intuitively, $g_\psi(t)$ is the value of g at time t when we follow the dynamics whose solution is ψ.

Definition 19 (synchronizers). Let $(\overline{\delta}, \underline{\delta})$ be a pair of dynamics, $(\overline{\omega}, \underline{\omega}) \in \mathbb{R}^{\overline{\mathcal{V}}} \times \mathbb{R}^{\underline{\mathcal{V}}}$ be a pair of states, and $\overline{\psi} : [0, \overline{T}) \to \mathbb{R}^{\overline{\mathcal{V}}}$ and $\underline{\psi} : [0, \underline{T}) \to \mathbb{R}^{\underline{\mathcal{V}}}$ be the unique solutions of $\overline{\delta}$ and $\underline{\delta}$ from $\overline{\omega}$ and $\underline{\omega}$, respectively. We say a pair of dL terms $(\overline{g}, \underline{g}) \in \mathcal{T}(\overline{\mathcal{V}}) \times \mathcal{T}(\underline{\mathcal{V}})$ *synchronizes* $(\overline{\delta}, \underline{\delta})$ from $(\overline{\omega}, \underline{\omega})$ if the following hold.

- $\overline{g}_{\overline{\psi}}(0) = \underline{g}_{\underline{\psi}}(0)$
- The derivatives of $\overline{g}_{\overline{\psi}}$ and $\underline{g}_{\underline{\psi}}$ are both strictly positive.

The following lemma ensures that, for any synchronizer, a corresponding time stretch function exists.

Lemma 20. *In the setting of Definition 19, let $\overline{t} \in [0, \overline{T})$ and $\underline{t} \in [0, \underline{T})$ be such that $\overline{g}_{\overline{\psi}}(\overline{t}) = \underline{g}_{\underline{\psi}}(\underline{t})$. Then the function K, defined by $K(s) := \underline{g}_{\underline{\psi}}{}^{-1}(\overline{g}_{\overline{\psi}}(s))$, is a time stretch function from $[0,\overline{t}]$ to $[0,\underline{t}]$. Moreover we have $\dot{K}(s) = \frac{\dot{\overline{g}_{\overline{\psi}}}(s)}{\dot{\underline{g}_{\underline{\psi}}}(K(s))}$.*

Proof. Since $\underline{g}_{\underline{\psi}}$ is strictly monotonic on $[0,\underline{t}]$, it has an inverse $\underline{g}_{\underline{\psi}}{}^{-1}$ defined from $\underline{g}_{\underline{\psi}}([0,\underline{t}])$ to $[0,\underline{t}]$. By assumption we have $\overline{g}_{\overline{\psi}}(0) = \underline{g}_{\underline{\psi}}(0)$, and thus $K(0) = \underline{g}_{\underline{\psi}}{}^{-1}(\overline{g}_{\overline{\psi}}(0)) = \underline{g}_{\underline{\psi}}{}^{-1}(\underline{g}_{\underline{\psi}}(0)) = 0$. Also since $\overline{g}_{\overline{\psi}}(\overline{t}) = \underline{g}_{\underline{\psi}}(\underline{t})$, we see that $\underline{g}_{\underline{\psi}}{}^{-1}$ is defined from $\overline{g}_{\overline{\psi}}([0,\overline{t}])$ to $[0,\underline{t}]$. Thus $K = \underline{g}_{\underline{\psi}}{}^{-1} \circ \overline{g}_{\overline{\psi}}$ is defined from $[0,\overline{t}]$ to $[0,\underline{t}]$.

$$\begin{aligned}
\dot{K}(s) &= \dot{\overline{g}_{\overline{\psi}}}(s) \cdot \big(\dot{\underline{g}_{\underline{\psi}}{}^{-1}}\big)(\overline{g}_{\overline{\psi}}(s)) && \text{derivative of } K = \underline{g}_{\underline{\psi}}{}^{-1} \circ \overline{g}_{\overline{\psi}} \\
&= \frac{\dot{\overline{g}_{\overline{\psi}}}(s)}{\dot{\underline{g}_{\underline{\psi}}}(\underline{g}_{\underline{\psi}}{}^{-1}(\overline{g}_{\overline{\psi}}(s)))} && \text{derivative of } \underline{g}_{\underline{\psi}}{}^{-1} \\
&= \frac{\dot{\overline{g}_{\overline{\psi}}}(s)}{\dot{\underline{g}_{\underline{\psi}}}(K(s))}
\end{aligned}$$

whose value is positive by assumptions on the derivatives of $\overline{g}_{\overline{\psi}}$ and $\underline{g}_{\underline{\psi}}$. □

We remark that time stretch functions we obtain in Lemma 20 are not necessarily expressible as a dL term, as exemplified by the following example.

Example 21. Consider two dynamics $\overline{\delta_F} := (\dot{\overline{x}} = \overline{v}, \dot{\overline{v}} = -\overline{v}^2)$ and $\underline{\delta} := (\dot{\underline{x}} = 1)$. Their solutions $\overline{\psi}, \underline{\psi} : \mathbb{R}_{\geq 0} \to \mathbb{R}^2$ from initial value $x = 0, v = 1$ are

$$\overline{\psi}(s) = \big(\log(1+s), (1+s)^{-1}\big) \qquad\qquad \underline{\psi}(s) = (s, 0)$$

Now let $\overline{g} = \overline{x}$ and $\underline{g} = \underline{x}$. Then $\overline{g}_{\overline{\psi}}(s) = \log(1+s)$, $\underline{g}_{\underline{\psi}} = \underline{g}_{\underline{\psi}}{}^{-1} = \mathsf{id}$ and thus $K(s) = \underline{g}_{\underline{\psi}}{}^{-1}(\overline{g}_{\overline{\psi}}(s)) = \log(1+s)$. This is not rational and not expressible in dL.

Using the syntactic Lie derivative (Definition 12), we state a sound inference rule that does not need K to be represented explicitly. We note that there is strong support for Lie derivatives in the tool KeYmaera X, as a key syntactic operation behind the differential invariant (DI) rule (Definition 13).

Definition 22. Let $\overline{\delta} := \left(\dot{\overline{\mathbf{x}}} = \overline{\mathbf{e}} \,\&\, \overline{Q}\right)$ and $\underline{\delta} := \left(\dot{\underline{\mathbf{x}}} = \underline{\mathbf{e}} \,\&\, \underline{Q}\right)$ be two dynamics and let $(\overline{g}, \underline{g}) \in \mathcal{T}(\overline{\mathcal{V}}) \times \mathcal{T}(\underline{\mathcal{V}})$ (which is supposed to be a synchronizer). We define the *synchronized dynamics* of $(\overline{\delta}, \underline{\delta})$ with respect to $(\overline{g}, \underline{g})$ as follows:

$$\overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta} \;:=\; \left(\dot{\overline{\mathbf{x}}} = \overline{\mathbf{e}},\; \dot{\underline{\mathbf{x}}} = \frac{\mathcal{L}_{\overline{\delta}}\, \overline{g}}{\mathcal{L}_{\underline{\delta}}\, \underline{g}} \cdot \underline{\mathbf{e}}\right) \,\&\, \left(\overline{Q} \wedge \underline{Q} \wedge \mathcal{L}_{\overline{\delta}}\, \overline{g} > 0 \wedge \mathcal{L}_{\underline{\delta}}\, \underline{g} > 0\right)$$

Lemma 23. *Let $(\overline{g}, \underline{g})$ be a synchronizer of $(\overline{\delta}, \underline{\delta})$ from $(\overline{\omega}_0, \underline{\omega}_0)$. The following are equivalent, where the semantical transition relations are from Definition 5.*

1. $(\overline{\omega}_0, \underline{\omega}_0) \dashv\!\llbracket \overline{\delta};\ \underline{\delta} \rrbracket\!\mapsto (\overline{\omega}, \underline{\omega})$ *and* $(\overline{\omega}, \underline{\omega}) \in \llbracket \overline{g} = \underline{g} \rrbracket$
2. $(\overline{\omega}_0, \underline{\omega}_0) \dashv\!\llbracket \overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta} \rrbracket\!\mapsto (\overline{\omega}, \underline{\omega})$

Proof. We first prove ($1 \Rightarrow 2$). In the proof of Lemma 20, we can observe that $\dot{\overline{g_{\psi}}}(s) = \llbracket \mathcal{L}_{\overline{\delta}}\, \overline{g} \rrbracket_{\overline{\psi}(s)}$, and analogously, $\dot{\underline{g_{\psi}}}(s) = \llbracket \mathcal{L}_{\underline{\delta}}\, \underline{g} \rrbracket_{\underline{\psi}(s)}$. Hence we obtain

$$\dot{K}(s) = \frac{\llbracket \mathcal{L}_{\overline{\delta}}\, \overline{g} \rrbracket_{\overline{\psi}(s)}}{\llbracket \mathcal{L}_{\underline{\delta}}\, \underline{g} \rrbracket_{\underline{\psi}(K(s))}} = \left\llbracket \frac{\mathcal{L}_{\overline{\delta}}\, \overline{g}}{\mathcal{L}_{\underline{\delta}}\, \underline{g}} \right\rrbracket_{\rho(s)} \tag{6}$$

where $\rho : [0, \overline{t}) \to \mathbb{R}^{\overline{\mathcal{V}} \uplus \underline{\mathcal{V}}}$ is defined by $\rho(s) := \left(\overline{\psi}(s), \underline{\psi}(K(s))\right)$.

We note that $K : [0, \overline{t}] \to [0, K(\overline{t})]$ is a time-stretch function, and that $\underline{\psi}$ is a solution of $\dot{\underline{\mathbf{x}}} = \underline{\mathbf{e}}$, that is, $\dot{\underline{\psi}}(u) = \llbracket \underline{\mathbf{e}} \rrbracket_{\underline{\psi}(u)}$ whenever $0 \leq u < \underline{t} = K(\overline{t})$. Combined with Lemma 17, we obtain

$$\left(\underline{\psi} \overset{\cdot}{\circ} K\right)(s) = \dot{K}(s) \cdot \llbracket \underline{\mathbf{e}} \rrbracket_{\underline{\psi}(K(s))} = \dot{K}(s) \cdot \llbracket \underline{\mathbf{e}} \rrbracket_{\rho(s)} \qquad \text{whenever } 0 \leq s < \overline{t}.$$

Hence, with the fact that $\overline{\psi}$ is a solution of $\dot{\overline{\mathbf{x}}} = \overline{\mathbf{e}}$, we obtain

$$\dot{\rho}(s) = \left(\dot{\overline{\psi}}(s), \left(\underline{\psi} \overset{\cdot}{\circ} K\right)(s)\right) = \left(\llbracket \overline{\mathbf{e}} \rrbracket_{\rho(s)},\ \dot{K}(s) \cdot \llbracket \underline{\mathbf{e}} \rrbracket_{\rho(s)}\right) = \left\llbracket \left(\overline{\mathbf{e}},\ \frac{\mathcal{L}_{\overline{\delta}}\, \overline{g}}{\mathcal{L}_{\underline{\delta}}\, \underline{g}} \cdot \underline{\mathbf{e}}\right) \right\rrbracket_{\rho(s)}$$

whenever $0 \leq s < \overline{t}$. Here the last equality is from (6). This concludes that ρ is a solution of the dynamics $\overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta}$. It remains to prove that for all $\tau \in [0, \overline{t}]$, $\llbracket \overline{Q} \wedge \underline{Q} \wedge \mathcal{L}_{\overline{\delta}}\, \overline{g} > 0 \wedge \mathcal{L}_{\underline{\delta}}\, \underline{g} > 0 \rrbracket_{\rho(\tau)}$ is true. This is an easy consequence of item 1, and the fact that $(\overline{g}, \underline{g})$ is a synchronizer of $(\overline{\delta}, \underline{\delta})$ from $(\overline{\omega}_0, \underline{\omega}_0)$.

For the direction ($2 \Rightarrow 1$), let $(\overline{\xi}, \underline{\xi}) : [0, T) \to \mathbb{R}^{\overline{\mathcal{V}}} \times \mathbb{R}^{\underline{\mathcal{V}}}$ be the unique solution of $\overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta}$ from $(\overline{\omega}_0, \underline{\omega}_0)$. Then there is $t \in [0, T)$ such that $(\overline{\xi}(t), \underline{\xi}(t)) = (\overline{\omega}, \underline{\omega})$. Let us prove that $(\overline{\omega}, \underline{\omega}) \in \llbracket \overline{g} = \underline{g} \rrbracket$. The function $h : s \in [0, T) \mapsto \llbracket \overline{g} \rrbracket_{\overline{\xi}(s)} - \llbracket \underline{g} \rrbracket_{\underline{\xi}(s)}$ is equal to 0 at $s = 0$ and its derivative is given by:

$$\dot{h}(s) = \llbracket \mathcal{L}_{\overline{\delta}}\, \overline{g} \rrbracket_{\overline{\xi}(s)} - \llbracket \mathcal{L}_{\underline{\delta}}\, \underline{g} \rrbracket_{\underline{\xi}(s)} \cdot \frac{\llbracket \mathcal{L}_{\overline{\delta}}\, \overline{g} \rrbracket_{\overline{\xi}(s)}}{\llbracket \mathcal{L}_{\underline{\delta}}\, \underline{g} \rrbracket_{\underline{\xi}(s)}} = 0$$

Consequently, h is the constant function equal to 0, which implies that $(\overline{\omega}, \underline{\omega}) \in [\![\overline{g} = \underline{g}]\!]$. By definition, $\overline{\xi}$ is a solution of $\overline{\delta}$, so $\overline{\omega}_0 \dashv\![\![\overline{\delta}]\!]\!\mapsto \overline{\omega}$. Furthermore, by Corollary 18, $\underline{\xi} \circ K^{-1}$ is a solution of $\underline{\delta}$. Thus $\underline{\omega_0} \dashv\![\![\underline{\delta}]\!]\!\mapsto \underline{\omega}$ and

$$(\overline{\omega}_0, \underline{\omega_0}) \dashv\![\![\overline{\delta};\ \underline{\delta}]\!]\!\mapsto (\overline{\omega}, \underline{\omega}). \qquad \square$$

The above lemma is a key observation in the current work. It allows us to turn the relational dynamics $\overline{\delta}; \underline{\delta}$—expressed as a sequential composition in dL—into a combined dynamics $\overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta}$. Moreover, we can do so in a way that the two dynamics are synchronized in a reparametrized manner, as specified by $(\overline{g}, \underline{g})$. Such combination of two dynamics is crucial in exploiting the logical infrastructure of dL and KeYmaera X—we emphasize again that the (DI) rule does not support invariant reasoning about the relationship between $\overline{\delta}$ and $\underline{\delta}$, when the relational dynamics is expressed in the original form $\overline{\delta};\ \underline{\delta}$.

The following is an incarnation of Lemma 23 as a proof rule. We assume that a postcondition is a conditional form $E \Rightarrow \varphi$; E is called an *exit condition*. By assuming that E implies $\overline{g} = \underline{g}$, we enforce the second condition $(\overline{\omega}, \underline{\omega}) \in [\![\overline{g} = \underline{g}]\!]$ in item 1 of Lemma 23. The first three premises are there to ensure that $(\overline{g}, \underline{g})$ is a synchronizer. Under these premises (the first four), the rule allows one to transform its conclusion (about $\overline{\delta}; \underline{\delta}$) into one about the combined dynamics $\overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta}$, which is amenable to application of the (DI) rule, for example.

Theorem 24 (synchronization rule). *The following inference rule is sound:*

$$\frac{\begin{array}{ll} \Gamma \vdash [\,\overline{\delta}\,]\mathcal{L}_{\overline{\delta}}\,\overline{g} > 0 & \Gamma \vdash \overline{g} = \underline{g} \\ \Gamma \vdash [\,\underline{\delta}\,]\mathcal{L}_{\underline{\delta}}\,\underline{g} > 0 & E \vdash \overline{g} = \underline{g} \quad \Gamma \vdash [\,\overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta}\,](E \Rightarrow \varphi) \end{array}}{\Gamma \vdash [\,\overline{\delta}; \underline{\delta}\,](E \Rightarrow \varphi)}\ (\mathit{Sync})$$

Recall the definition of $\overline{\delta} \otimes_{(\overline{g},\underline{g})} \underline{\delta}$ (Definition 22), where time stretching for the second dynamics $\underline{\delta}$ is expressed syntactically by Lie derivatives. We call the four premises $\Gamma \vdash \overline{g} = \underline{g}$, $E \vdash \overline{g} = \underline{g}$, $\Gamma \vdash [\,\overline{\delta}\,]\mathcal{L}_{\overline{\delta}}\,\overline{g} > 0$, and $\Gamma \vdash [\,\underline{\delta}\,]\mathcal{L}_{\underline{\delta}}\,\underline{g} > 0$ the *synchronizability conditions*. These obligations are usually easy to discharge. The last premise, which we call the *synchronized formula*, is typically the core remaining obligation.

Remark 25 (choice of $(\overline{g}, \underline{g})$). In applying the (Sync) rule, one still has to find a suitable synchronizer $(\overline{g}, \underline{g})$. This turns out to be straightforward in many examples. In all the case studies in Section 6 and in Example 1, the exit condition E is of the form $\overline{x} = \underline{x} = C$ where C is a constant. This suggests the use of $\overline{g} = \overline{x}$, $\underline{g} = \underline{x}$. Indeed, all our proofs use this choice of $(\overline{g}, \underline{g})$.

5 Implementation

KeYmaera X [17] is an interactive theorem prover based on the sequent calculus formulation of dL. It is implemented in Scala, replacing its former system KeYmaera [16]. It has a web-based GUI environment, and a support of automated theorem proving using computer algebra systems such as Mathematica [27].

For the formalization of case studies in Section 6, we extended KeYmaera X version 4.7 (available at [17]) with the (Sync) rule. This extension of KeYmaera X, together with our proofs in case studies, are currently available at http://group-mmm.org/rddl_tacas_2020/.

The KeYmaera X implementation is structured in a flexible manner, from which we benefited. To add a rule to KeYmaera X, one has to implement a Scala program that take the conclusion of the rule and generate the premises of the rule as subgoals. The fact that any Scala program is allowed here enabled us to implement complex algorithms, such as inductive translation of formulas.

In implementing the (Sync) rule, the functions in KeYmaera X called *helpers* helped us, such as in the Lie derivative computation and the functionality to simplify formulas into equivalent ones. The bulk of our effort regarded the $\otimes_{(\overline{g},\underline{g})}$ operator. There we did a bit more general than we stated in the paper: not only taking dynamics of form $\dot{\mathbf{x}} = \mathbf{e} \,\&\, Q$, we also allow sequences of dynamics possibly interleaved by guards and nondeterministic choices. This feature was utilized in the case study that will be described in Section 6.3.

6 Case Studies

We describe three case studies where we proved relational properties of hybrid dynamics. We did so formally in our extension of KeYmaera X described in Section 5. In all the examples, we apply the (Sync) rule as a main proof step, in conjunction with the existing rules in dL. Below, we describe our example systems and outline the important steps in the formal proofs.

6.1 Collision Speed with Constant Acceleration

In this section we apply the (Sync) rule to the running Example 1. For this example we consider two dynamics $\overline{\delta_C} := \left(\dot{\overline{x}} = \overline{v}, \dot{\overline{v}} = \overline{a}\right)$ and $\underline{\delta_C} := \left(\dot{\underline{x}} = \underline{v}, \dot{\underline{v}} = \underline{a}\right)$. Both dynamics represent a car with constant acceleration. Our claim is that if acceleration is larger in the first system, then the first car is necessarily faster than the second car after traveling the same distance l; formally,

$$\Gamma \vdash \left[\overline{\delta_C}; \underline{\delta_C}\right](\overline{x} = l \wedge \underline{x} = l \Rightarrow \underline{v} \leq \overline{v}) \tag{7}$$

where

$$\Gamma := \{0 = \overline{x} = \underline{x},\quad 0 < \overline{v} = \overline{v}_0,\quad \underline{v} = \underline{v}_0,\quad \overline{v}_0 \geq \underline{v}_0,\quad 0 \leq \underline{a} \leq \overline{a}\}$$

We apply the (Sync) rule, where $\overline{g} := \overline{x}$ and $\underline{g} := \underline{x}$. The first two synchronizability conditions are $\Gamma \vdash \underline{x} = \overline{x}$ and $\underline{x} = l,\ \overline{x} = l \vdash \underline{x} = \overline{x}$, which are trivial. The last two synchronizability conditions are

$$\Gamma \vdash \left[\overline{\delta_C}\right]\mathcal{L}_{\overline{\delta_C}}\,\overline{g} = \overline{v} > 0 \qquad\qquad \Gamma \vdash \left[\underline{\delta_C}\right]\mathcal{L}_{\underline{\delta_C}}\,\underline{g} = \underline{v} > 0$$

which are proven using differential invariants (DI). The synchronized formula is

$$\Gamma \vdash \left[\overline{\delta_C}, \dot{\underline{x}} = \underline{v}\cdot(\overline{v}/\underline{v}), \dot{\underline{v}} = \underline{a}\cdot(\overline{v}/\underline{v}) \,\&\, \overline{v} > 0 \wedge \underline{v} > 0\right](\overline{x} = l \wedge \underline{x} = l \Rightarrow \underline{v} \leq \overline{v})$$

One might try to show the inequality $\overline{v} - \underline{v} \geq 0$ by the differential invariant (DI) rule, but the Lie derivative of the term $\overline{v} - \underline{v}$ is $\overline{a} - \underline{a} \cdot (\overline{v}/\underline{v})$, which is not obviously nonnegative. Instead, a trickier expression $\overline{a} \cdot (\underline{v}^2 - \underline{v}_0^2) - \underline{a} \cdot (\overline{v}^2 - \overline{v}_0^2) = 0$ turns out to be an invariant. Its Lie derivative is $\overline{a} \cdot (2\underline{v}) \cdot \underline{a} \cdot (\overline{v}/\underline{v}) - \underline{a} \cdot (2\overline{v}) \cdot \overline{a}$, which is clearly 0, since we also know $\underline{v} > 0$.

We do not have an intuitive explanation for this invariant, but it was found by a template-based search, like many other invariants in dL. By positing the existence of a polynomial invariant of a certain degree, we can find conditions on the coefficients by requiring its Lie derivative and initial value are zero. Solving these conditions for a second-degree invariant on the velocities in the system yielded the invariant above.

After finding our invariant, we additionally have to show the invariant entails our desired result, $\underline{v} \leq \overline{v}$. This can be shown with a standard monotonicity property of modal logics: from $\phi \vdash \psi$ and $\Gamma \vdash [\alpha]\phi$, we can conclude $\Gamma \vdash [\alpha]\psi$, where ϕ states the expression above is an invariant and the velocities are always greater than their initial value, and ψ is our goal: $\underline{v} \leq \overline{v}$.

6.2 Collision Speed with Different Kinds of Friction

Here we continue Example 11, where we consider two dynamics $\overline{\delta_F} \equiv (\dot{\overline{x}} = \overline{v}, \dot{\overline{v}} = -\overline{v}^2)$ and $\underline{\delta_F} \equiv (\dot{\underline{x}} = \underline{v}, \dot{\underline{v}} = -\underline{v})$. Our goal is $\Gamma_F \vdash [\,\overline{\delta_F}; \underline{\delta_F}\,](\overline{x} = \underline{x} = l \Rightarrow \underline{v} \leq \overline{v})$, with $\Gamma_F := \{\overline{x} = \underline{x} = 0,\ 0 < \underline{v} \leq \overline{v} \leq 1\}$.

First, we establish the fact that the objects in this example always have positive velocity. We show this by the (Dbx) rule (Definition 13), where $\mathcal{L}_{\overline{\delta_F}}\, \overline{v} = -\overline{v}^2$ and $\mathcal{L}_{\underline{\delta_F}}\, \underline{v} = -\underline{v}$. This allows us to infer $\overline{v} > 0$ and $\underline{v} > 0$ hold at all times.

We apply the (Sync) rule along $\overline{x} = \underline{x}$, yielding the synchronized dynamics

$$\dot{\overline{x}} = \overline{v},\ \dot{\overline{v}} = -\overline{v}^2,\ \dot{\underline{x}} = \underline{v} \cdot (\overline{v}/\underline{v}),\ \dot{\underline{v}} = -\underline{v} \cdot (\overline{v}/\underline{v})\ \&\ \overline{v} > 0 \wedge \underline{v} > 0$$

Note that the new evolution domain condition $\underline{v} > 0$ allows us to rewrite $\underline{v} \cdot (\overline{v}/\underline{v})$ to $\overline{v}$. The synchronizability conditions follow immediately from the fact that $\underline{v} > 0$ and $\overline{v} > 0$. For the synchronized formula, we apply the (DI) rule, so the desired inequality $\overline{v} \geq \underline{v}$ is reduced to $\overline{v}^2 \leq \overline{v}$, that is, $\overline{v} \leq 1$. To this end, $\overline{v} > 0$ tells us that the derivative of $\overline{v}$, that is, $-\overline{v}^2$, is always negative, therefore $\overline{v} \leq 1$.

6.3 Model Refinement

In this example, we consider two abstract models of cars. The first car is able to provide a high amount of constant acceleration a at low velocities, but at a certain velocity v_{cut} the engine switches to a different mode and then provides a lesser, but still constant acceleration a_{cut}. The second car is an abstracted version of the first, which ignores this mode change and provides the same constant amount of acceleration a at all velocities. Our aim in this example is to establish a safety envelope around the first car's behavior using the more simply stated second car's dynamics. Hence we show that the second car's velocity is greater than the first's at any position $\overline{x} = \underline{x} = l$. More formally, the behavior of the first car is expressed as a hybrid program $\overline{\alpha} := (\overline{\delta_1};\ ?\overline{v} = v_{cut};\ \overline{\delta_2})$ with two modes: $\overline{\delta_1} := (\dot{\overline{x}} = \overline{v}, \dot{\overline{v}} = a \,\&\, \overline{v} \leq v_{cut})$ and $\overline{\delta_2} := (\dot{\overline{x}} = \overline{v}, \dot{\overline{v}} = a_{cut})$. The second car follows the simple dynamics $\underline{\delta} := (\dot{\underline{x}} = \underline{v}, \dot{\underline{v}} = a)$. Our goal is to prove the sequent $\Gamma \vdash \left[\overline{\alpha}; \underline{\delta}\right](\underline{x} = \overline{x} = l \Rightarrow \underline{v} \leq \overline{v})$, where the initial conditions are given by

$$\Gamma := (\overline{x} = \underline{x} = 0, 0 < \overline{v} = \underline{v} = v_0, 0 < v_{cut}, 0 < a_{cut} \leq a)$$

Technically, the (Sync) rule merges one differential dynamics with another, but the program the first car executes is a more complicated composition of dynamics and testing. However, it is possible to synchronize *piecewise*, first synchronizing $\underline{\delta}$ with $\overline{\delta_1}$ until the first car changes modes, then synchronizing $\underline{\delta}$ with $\overline{\delta_2}$ for the remainder of their runs. This slightly generalized synchronization procedure means that we can instead show

$$\Gamma \vdash \left[\overline{\delta_1} \otimes_{(\overline{x},\underline{x})} \underline{\delta}; ?\overline{v} = v_{cut}; \overline{\delta_2} \otimes_{(\overline{x},\underline{x})} \underline{\delta}\right](\underline{x} = \overline{x} = l \Rightarrow \underline{v} \leq \overline{v})$$

There are also now two sets of synchronizability conditions to satisfy, but both are again straightforward. Since $\overline{\delta_1}$ and $\underline{\delta}$ are nearly identical (except for the evolution domain constraint), their synchronization $\overline{\delta_1} \otimes_{(\overline{x},\underline{x})} \underline{\delta}$ basically identifies the two dynamics. The synchronization of $\overline{\delta_2}$ and $\underline{\delta}$ is exactly the synchronization performed above in Section 6.1, and proceeds in the same way.

7 Conclusions and Future Work

In this paper, we present a relational extension of the differential dynamic logic based on time stretching of dynamics. This reparametrization enables us to enforce that comparisons between two systems occur when certain conditions are satisfied, for example when two cars are passing through the same position. While such reparametrizations can be thought of as stretching or compressing time for one of the dynamics, we also show they can be conducted by a transformation of the dynamics themselves, based on Lie derivatives. We call this process *synchronizing* the dynamics (Definition 19), and it leads us to a new dL proof rule, the (Sync) rule (Theorem 24). We implemented the new rule in the KeYmaera X tool and use our extension to demonstrate several nontrivial relational properties of dynamical systems.

In the future, we think it would be interesting to combine our relational logic with orthogonal relational extensions of dL [14] which focus on *refinement relations* with varying levels of nondeterminism. We also hinted in our last case study that it is possible to synchronize wider classes of hybrid programs than just two differential dynamics. We also think that the level of automated proof search available in KeYmaera X may enable the automatic detection of monotonic properties in *product lines*. This may be useful in industry both to provide sanity checks on formalized models of products, as well as enabling strong guarantees to be more easily obtained for those models.

References

1. Abrial, J.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
2. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.: A relational logic for higher-order programs. PACMPL **1**(ICFP), 21:1–21:29 (2017). https://doi.org/10.1145/3110265
3. Azevedo de Amorim, A., Gaboardi, M., Hsu, J., Katsumata, S.: Probabilistic Relational Reasoning via Metrics. In: LICS 2019. pp. 1–19. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785715
4. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) POPL 2004. pp. 14–25. ACM (2004). https://doi.org/10.1145/964001.964003
5. Bryce, D., Sun, J., Bae, K., Zuliani, P., Wang, Q., Gao, S., Schmarov, F., Kong, S., Chen, W., Tavares, Z.: dReach homepage. http://dreal.github.io/dReach/
6. Butler, M.J., Abrial, J., Banach, R.: Modelling and Refining Hybrid Systems in Event-B and Rodin. In: Petre, L., Sekerinski, E. (eds.) From Action Systems to Distributed Systems: The Refinement Approach, pp. 29–42. Chapman and Hall/CRC (2016). https://doi.org/10.1201/b20053-5
7. Chicone, C.: Ordinary Differential Equations with Applications, Texts in Applied Mathematics, vol. 34. Springer-Verlag New York, 2 edn. (2006)
8. Fainekos, G.E., Pappas, G.J.: Robustness of Temporal Logic Specifications. In: Havelund, K., Núñez, M., Rosu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Revised Selected Papers. LNCS, vol. 4262, pp. 178–192. Springer (2006). https://doi.org/10.1007/11940197_12
9. Girard, A., Pappas, G.J.: Approximate Bisimulation: A Bridge Between Computer Science and Control Theory. Eur. J. Control **17**(5–6), 568–578 (2011). https://doi.org/10.3166/ejc.17.568-578
10. Harel, D., Tiuryn, J., Kozen, D.: Dynamic Logic. MIT Press, Cambridge, MA, USA (2000)
11. Hasuo, I., Suenaga, K.: Exercises in Nonstandard Static Analysis of Hybrid Systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 462–478. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_34
12. Liebrenz, T., Herber, P., Glesner, S.: Deductive Verification of Hybrid Control Systems Modeled in Simulink with KeYmaera X. In: Sun, J., Sun, M. (eds.) ICFEM 2018. LNCS, vol. 11232, pp. 89–105. Springer (2018). https://doi.org/10.1007/978-3-030-02450-5_6
13. Lindelöf, E.: Sur l'application de la méthode des approximations successives aux équations différentielles ordinaires du premier ordre. Journal de mathématiques pures et appliquées 4e série **10**, 117–128 (1894)
14. Loos, S.M., Platzer, A.: Differential Refinement Logic. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) LICS 2016. pp. 505–514. ACM (2016). https://doi.org/10.1145/2933575.2934555
15. Mitsch, S., Platzer, A.: The KeYmaera X Proof IDE – Concepts on Usability in Hybrid Systems Theorem Proving. In: Dubois, C., Masci, P., Méry, D. (eds.) F-IDE@FM 2016. EPTCS, vol. 240, pp. 67–81 (2016). https://doi.org/10.4204/EPTCS.240.5
16. Platzer, A.: KeYmaera homepage. http://symbolaris.com/info/KeYmaera.html

17. Platzer, A.: KeYmaera X homepage. http://www.ls.cs.cmu.edu/KeYmaeraX/index.html
18. Platzer, A.: Differential Dynamic Logic for Hybrid Systems. J. Autom. Reasoning **41**(2), 143–189 (2008). https://doi.org/10.1007/s10817-008-9103-8
19. Platzer, A.: The Complete Proof Theory of Hybrid Systems. In: LICS 2012. pp. 541–550. IEEE Computer Society (2012). https://doi.org/10.1109/LICS.2012.64
20. Platzer, A.: A Complete Uniform Substitution Calculus for Differential Dynamic Logic. J. Autom. Reasoning **59**(2), 219–265 (2017). https://doi.org/10.1007/s10817-016-9385-1
21. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer (2018). https://doi.org/10.1007/978-3-319-63588-0
22. Platzer, A., Tan, Y.K.: Differential Equation Axiomatization: The Impressive Power of Differential Ghosts. In: Dawar, A., Grädel, E. (eds.) LICS 2018. pp. 819–828. ACM (2018). https://doi.org/10.1145/3209108.3209147
23. Robinson, A.: Non-standard analysis. Princeton University Press (1966)
24. Suenaga, K., Hasuo, I.: Programming with Infinitesimals: A While-Language for Hybrid System Modeling. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6756, pp. 392–403. Springer (2011). https://doi.org/10.1007/978-3-642-22012-8_31
25. Suenaga, K., Sekine, H., Hasuo, I.: Hyperstream processing systems: nonstandard modeling of continuous-time signals. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013. pp. 417–430. ACM (2013). https://doi.org/10.1145/2429069.2429120
26. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and Repulsing Supermartingales for Reachability in Probabilistic Programs. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 476–493. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_28
27. Wolfram Research, Inc.: Mathematica, Version 12.0 (2019), https://www.wolfram.com/mathematica, Champaign, IL

Verifying Concurrent Systems

Assume, Guarantee or Repair*

Hadar Frenkel[1], Orna Grumberg[1], Corina Pasareanu[2] and Sarai Sheinvald[3]

[1] Department of Computer Science, The Technion, Haifa, Israel
[2] Carnegie Mellon University and NASA Ames Research Center, CA, USA
[3] Department of Software Engineering, Braude College of Engineering, Karmiel, Israel

Abstract. We present Assume-Guarantee-Repair (AGR) – a novel framework which not only verifies that a program satisfies a set of properties, but also *repairs* the program in case the verification fails. We consider *communicating programs* – these are simple C-like programs, extended with synchronous communication actions over communication channels. Our method, which consists of a learning-based approach to assume-guarantee reasoning, performs verification and repair simultaneously: in every iteration, AGR either makes another step towards proving that the (current) system satisfies the specification, or alters the system in a way that brings it closer to satisfying the specification. We manage handling infinite-state systems by using a finite abstract representation, and reduce the semantic problems in hand – satisfying complex specifications that also contain first-order constraints – to syntactic ones, namely membership and equivalence queries for regular languages. We implemented our algorithm and evaluated it on various examples. Our experiments present compact proofs of correctness and quick repairs.

1 Introduction

Verification of large-scale systems is a main challenge in the field of formal verification. Often, the verification process of such a system does not scale well. *Compositional verification* aims to verify small components of a system separately, and from the correctness of the individual components, to conclude the correctness of the entire system. This, however, is not always possible, since the correctness of a component often depends on the behavior of its environment.

The Assume-Guarantee (AG) style compositional verification [22,26] suggests a solution to this problem. The simplest AG rule checks if a system composed of components M_1 and M_2 satisfies a property P by checking that M_1 under assumption A satisfies P and that any system containing M_2 as a component satisfies A. Several frameworks have been proposed to support this style of reasoning. Finding a suitable assumption A is then a common challenge in such frameworks.

In this work, we present *Assume-Guarantee-Repair* (AGR) – a fully automated framework which applies the Assume-Guarantee rule, and while seeking a suitable assumption A, incrementally repairs the given program in case the verification fails. Our framework is inspired by [24], which presented a learning-based method to finding an assumption A, using the L^* [5] algorithm for learning regular languages.

Our AGR framework handles *communicating programs*. These are infinite-state C-like programs, extended with the ability to synchronously read and write messages over communication channels. We model such programs as finite-state automata over an *action alphabet*, which reflects the program statements. The accepting states in these automata model points of interest in the program that the specification can relate to. The automata representation is similar in nature to that of control-flow graphs. Its advantage, however, is in the ability to exploit an automata-learning algorithm such as L^*.

* This research was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel National Cyber Directorate and the Israel Science Foundation (ISF)

The composition of the two program components, M_1 and M_2, denoted $M_1||M_2$, synchronizes on read-write actions on the same channel. Between two synchronized actions, the individual actions of both systems interleave.

```
while(true)
    password:=readInput;
    while(password≤ 999)
        password:=readInput;
    password2:=encrypt(password);
```

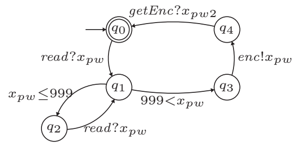


Fig. 1: Modeling a communicating program as an automaton M_2

Figure 1 presents the code of a communicating program (left) and its corresponding automaton M_2 (right). The automaton alphabet consists of constraints (e.g. $x_{pw} \leq 999$), assignment actions (e.g. $y_{pw} := 2 \cdot y_{pw}$ in M_1 of Figure 2), and communication actions (e.g. $enc!x_{pw}$ sends the value of variable x_{pw} over channel enc, and $getEnc?x_{pw2}$ reads a value to x_{pw2} on channel $getEnc$).

The specification P is modeled as an automaton that does not contain assignment actions. It may contain communication actions in order to specify behavioral requirements, as well as constraints over the variables of both system components, that express requirements on their values in various points in the runs.

Consider, for example, the program M_1 and the specification P seen in Figure 2, and the program M_2 of Figure 1. M_2 reads a password on channel $read$ to the variable x_{pw}, and once it is long enough (has at least four digits), it sends the value of x_{pw} to M_1 through channel enc. M_1 reads this value to variable y_{pw} and then applies a simple function that changes its value, and sends the changed variable back to M_2. The property P reasons about the parallel run of the two programs. The pair $(getEnc?x_{pw2}, getEnc!y_{pw})$ denotes a synchronization of M_1 and M_2 on channel $getEnc$. P makes sure that the parallel run of M_1 and M_2 always reads a value and then encrypts it – a temporal requirement. In addition, it makes sure that the value after encryption is different than the original value, and that there is no overflow – both are semantic requirements on the program variables. That is, P expresses temporal requirements that contain first order constraints. In case one of the requirements does not hold, P reaches the state r_4 which is an error state. Note that P here is not complete, for simplicity of presentation (see Definition 5 for a formal definition of a complete program).

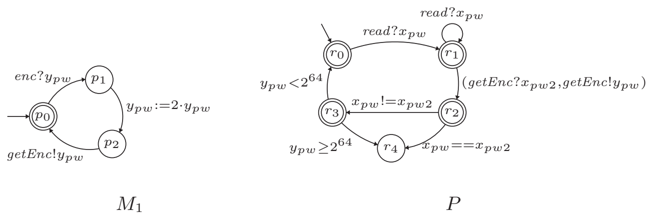


Fig. 2: The programs M_1, M_2, and the specification P

The L^* algorithm aims at learning a regular language U. Its entities consist of a *teacher* – an oracle who answers *membership queries* ("is the word w in U?") and *equivalence queries* ("is $\mathcal{A}$ an automaton whose language is U?"), and a *learner*, who iteratively constructs a finite deterministic automaton $\mathcal{A}$ for U by submitting a sequence of membership and equivalence queries to the teacher.

In using the L^* algorithm for learning an assumption A for the AG-rule, membership queries are answered according to the satisfaction of the specification P: If $M_1||t$ satisfies P, then the trace t in hand should be in A. Otherwise, t should not be in A. Once the learner constructs a stable system A, it submits an equivalence query. The teacher then checks whether A is a suitable assumption, that is, whether $M_1||A$ satisfies P, and whether the language of M_2 is contained in the language of A. According to the results, the process either continues or halts with an answer to the verification problem. The learning procedure aims at learning the weakest assumption A_w, which contains all the traces that in parallel with M_1 satisfy P . The key observation that guarantees termination in [24] is that the components in this procedure – M_1,M_2, P and A_w – are all regular.

Our setting is more complicated, since the traces in the components – both the programs and the specification – contain constraints, which are to be checked semantically and not syntactically. These constraints may cause some traces to become infeasible. For example, if a trace contains an assignment $x := 3$ followed by a constraint $x \geq 4$ (modeling an "if" statement), then this trace does not contribute any concrete runs, and therefore does not affect the system behavior. Thus, we must add feasibility checks to the process.

Constraints in the specification also pose a difficulty, as satisfiability of a specification is determined by the semantics of the constraints and not only by the language syntax, and so there is more here to check than standard language containment. Moreover, in our setting A_w above may no longer be regular, see Example 3. However, our method manages to overcome this problem.

As we have described above, not only do we construct a learning-based method for the AG-rule for communicating programs, but we also repair the programs in case the verification fails. An AG-rule can either conclude that $M_1||M_2$ satisfies P, or return a real, non-spurious counterexample of a computation t of $M_1||M_2$ that violates P. In our case, instead of returning t, we repair M_2 in a way that eliminates this counterexample. Our repair is both syntactic and semantic, where for semantic repair we use *abduction* [25] to infer a new constraint which makes the counterexample t infeasible.

Consider again M_1 and P of Figure 2 and M_2 of Figure 1. The composition $M_1||M_2$ does not satisfy P. For example, if the initial value of x_{pw} is 2^{63}, then after encryption the value of y_{pw} is 2^{64}, violating P. Our algorithm finds a bad trace during the AG stage which captures this bad behavior, and the abduction in the repair stage finds a constraint that eliminates it: $x_{pw} < 2^{63}$, and inserts this constraint to M_2.

Following this step we now have an updated M_2, and we continue with applying the AG-rule again, using information we have gathered in the previous steps. In addition to removing the error trace, we update the alphabet of M_2 with the new constraint.

Continuing our example, in a following iteration AGR will verify that the repaired M_2 together with M_1 satisfy P, and terminate.

Thus, AGR operates in a verify-repair loop, where each iteration runs a learning-based process to determine whether the (current) system satisfies P, and if not, eliminates bad behaviors from M_2 while enriching the set of constraints derived from these bad behaviors, which often leads to a quicker convergence. In case the current system does satisfy P, we return the repaired M_2 together with an assumption A that abstracts M_2 and acts as a smaller proof for the correctness of the system.

We have implemented a tool for AGR and evaluated it on examples of various sizes and of various types of errors. Our experiments show that for most examples, AGR converges and finds a repair after 2-5 iterations of verify-repair. Moreover, our tool generates assumptions that are significantly smaller then the (possibly repaired) M_2, thus constructing a compact and efficient proof of correctness.

Contributions To summarize, the main contributions of this paper are:

1. A learning-based Assume-Guarantee algorithm for infinite-state communicating programs, which manages to overcome the difficulties such programs present. In particular, our algorithm overcomes the inherent irregularity of the first-order constraints in these programs, and offers syntactic solutions to the semantic problems they impose.
2. An Assume-Guarantee-Repair algorithm, in which the Assume-Guarantee and the Repair procedures intertwine to produce a repaired program which, due to our construction, maintains many of the "good" behaviors of the original program. Moreover, in case the original program satisfies the property, our algorithm is guaranteed to terminate and return this conclusion.
3. An incremental learning algorithm that uses query results from previous iterations in learning a new language with a richer alphabet.
4. A novel use of abduction to repair communicating programs over first order constraints.
5. An implementation of our algorithm, demonstrating the effectiveness of our framework.

Related Work Assume-guarantee style compositional verification [22,26] has been extensively studied. The assumptions necessary for compositional verification were first produced manually, limiting the practicality of the method.

More recent works [9,16,14,6] proposed techniques for automatic assumption generation using learning and abstraction refinement techniques, making assume-guarantee verification more appealing. In [24,6] alphabet refinement has been suggested as an optimization, to reduce the alphabet of the generated assumptions, and consequently their sizes. This optimization can easily be incorporated into our framework as well.

Other learning-based approaches for automating assumption generation have been described in [7,17,8]. All these works address non-circular rules and are limited to finite state systems. Automatic assumption generation for circular rules is presented in [12,13], using compositional rules similar to the ones studied in [21,23].

Our approach is based on a non-circular rule but it targets complex, infinite-state concurrent systems, and addresses not only verification but also repair. The compositional framework presented in [19] addresses L^*-based compositional verification and synthesis but it only targets finite state systems.

Also related is the work in [18], which addresses automatic synthesis of circular compositional proofs based on logical abduction; however the focus of that work is sequential programs, while here we target concurrent programs. A sequential setting is also considered in [3], where abduction is used for automatically generating a program environment. Our computation of abduction is similar to that of [3]. However, we require our constraints to be over a predefined set of variables, while they look for a minimal set.

The approach presented in [27] aims to compute the *interface* of an infinite-state component. Similar to our work, the approach works with both over- and under- approximations but it only analyzes one component at a time. Furthermore, the component is restricted to be deterministic (necessary for the permissiveness check). In contrast we use both components of a system to compute the necessary assumptions, and as a result they can be much smaller than in [27]. Furthermore, we do not restrict the components to be deterministic and, more importantly, we also address the system repair in case of dissatisfaction.

2 Communicating Programs

In this section we present the notion of *communicating programs*. These are C-like programs, extended with the ability to synchronously read and write messages over communication channels. We model such programs as automata over an *action alphabet* that reflects the program statements. The alphabet includes *constraints*, which are quantifier-free first-order formulas, representing the conditions in *if* and *while* statements. It also includes *assignment statements* and $read$ and $write$ *communication actions*. The automata representation is similar in nature to that of a control-flow graph. Its advantage, however, is in the ability to exploit an automata-learning algorithm such as L^* for its verification.

We first formally define the alphabet over which communicating programs are defined. Let G be a finite set of communication channels. Let X be a finite set of variables (whose ordered vector is $\bar{x}$) and D be a (possibly infinite) data domain. For simplicity, we assume that all variables are defined over D. The elements of D are also used as constants in arithmetic expressions and constraints.

Definition 1. *An* action alphabet *is* $\alpha = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ *where:*

1. $\mathcal{G} \subseteq \{ g?x_1, g!x_1, (g?x_1, g!x_2), (g!x_1, g?x_2) \mid g \in G, x_1, x_2 \in X\}$ *is a finite set of* communication actions. $g?x$ *is a* read *action of a value to the variable* x *through channel* g*, and* $g!x$ *is a* write *action of the value of* x *on channel* g*. We use* $g * x$ *to indicate some action, either* read *or* write*, through* g*. The pairs* $(g?x_1, g!x_2)$ *and* $(g!x_1, g?x_2)$ *represent a synchronization of two programs on read-write actions over channel* g *(defined later).*
2. $\mathcal{E} \subseteq \{ x := e \mid e \in E, x \in X\}$ *is a finite set of* assignment statements*, where* E *is a set of expressions over* $X \cup D$*.*
3. $\mathcal{C}$ *is a finite set of constraints over* $X \cup D$*.*

Definition 2. *A* communicating program *(or, a program) is* $M = \langle Q, X, \alpha, \delta, q_0, F\rangle$*, where:*

1. Q *is a finite set of states and* $q_0 \in Q$ *is the initial state.*
2. X *is a finite set of variables that range over* D*.*
3. $\alpha = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ *is the action alphabet of* M*.*
4. $\delta \subseteq Q \times \alpha \times Q$ *is the transition relation, where for each* $q \in Q$*, only one of the following holds:*
   – $\alpha \in \mathcal{C}$ *for all* $(q, \alpha, q') \in \delta$
   – $\alpha \in \mathcal{G} \cup \mathcal{E}$ *for all* $(q, \alpha, q') \in \delta$
   That is, for each state it holds that either all outgoing edges are labeled with constraints, or that all outgoing edges are labeled with assignments or communication actions.
5. $F \subseteq Q$ *is the set of accepting states.*

The words that are read along a communicating program are a *symbolic representation* of the program behaviors. We refer to such a word as a *trace*. Each such trace induces *concrete runs* of the program, which are formed by concrete assignments to the program variables in a way that conforms with the actions along the word.

We now formally define these notions.

Definition 3. *A* path *in a program* M *is a finite sequence of states and actions* $p = (q_0, a_1, q_1, \ldots, a_n, q_n)$*, starting with the initial state* q_0*, such that* $\forall 0 \leq i < n$ *we have* $(q_i, a_{i+1}, q_{i+1}) \in \delta$*. The* induced trace *of* p *is the sequence* $t = (a_1, \ldots, a_n)$ *of the actions in* p*. If* q_n *is accepting, then* t *is an* accepted trace *of* M*.*

From now on we assume that every trace we discuss is induced by some path. We turn to define the concrete runs of the program.

Definition 4. *Let* $t = (a_1, \ldots, a_n)$ *be a trace and let* $(\beta_0, \ldots, \beta_n)$ *be a sequence of valuations (i.e., assignments to the program variables)*[4]*. Then a sequence* $r = (\beta_0, a_1, \beta_1, a_2, \ldots, a_n, \beta_n)$ *is a* run *of* t *if the following holds.*

[4] Such valuations are usually referred to as states. We do not use this terminology here in order not to confuse them with the states of the automaton.

1. *β_0 is an arbitrary valuation.*
2. *If $a_i = g?x$, then $\beta_i(y) = \beta_{i-1}(y)$ for every $y \neq x$. Intuitively, x is arbitrarily assigned by the read action, and the rest of the variables are unchanged.*
3. *If a_i is an assignment $x := e$, then $\beta_i(x) = e[\bar{x} \leftarrow \beta_{i-1}(\bar{x})]$ and $\beta_i(y) = \beta_{i-1}(y)$ for every $y \neq x$.*
4. *If $a_i = (g?x, g!y)$ or $a_i = (g!y, g?x)$ then $\beta_i(x) = \beta_{i-1}(y)$ and $\beta_i(z) = \beta_{i-1}(z)$ for every $z \neq x$. That is, the effect of a synchronous communication on a channel is that of an assignment.*
5. *If a_i does not involve a read or an assignment, then $\beta_i = \beta_{i-1}$.*
6. *Finally, if a_i is a constraint in $\mathcal{C}$, then $\beta_i(\bar{x}) \vDash a_i$ (and since a_i does not change the variable assignments, then $\beta_{i-1}(\bar{x}) \vDash a_i$ holds as well).*

We say that t is feasible *if there exists a run of t.*

The *symbolic language* of M, denoted $\mathcal{T}(M)$, is the set of all *accepted* traces induced by paths of M. The *concrete language* of M is the set of all runs of accepted traces in $\mathcal{T}(M)$. We will mostly be interested in feasible traces, which represent (concrete) runs of the program. Intuitively, the symbolic language of a program M corresponds to its syntactic behavior, while the concrete language corresponds to the semantics of the program.

Example 1.
- The trace $(x := 2 \cdot y,\ g?x,\ y := y + 1,\ g!y)$ is feasible, as it has a run $(x = 1, y = 3), (x = 6, y = 3), (x = 20, y = 3), (x = 20, y = 4), (x = 20, y = 4)$.
- The trace $(g?x,\ x := x^2\ , x < 0)$ is not feasible since no β can satisfy the constraint $x < 0$ if $x := x^2$ is executed beforehand.

2.1 Parallel Composition

We now describe and define the parallel run of two communicating programs, and the way in which they communicate.

Let M_1 and M_2 be two programs, where $M_i = \langle Q_i, X_i, \alpha_i, \delta_i, {q_0}^i, F_i \rangle$ for $i \in \{1, 2\}$. Let G_1, G_2 be the sets of communication channels occurring in actions of M_1, M_2, respectively. We assume $X_1 \cap X_2 = \emptyset$.

The *interface alphabet* αI of M_1 and M_2 consists of all communication actions on channels that are common to both components. That is, $\alpha I = \{\, g?x,\ g!x \mid g \in G_1 \cap G_2,\ x \in X_1 \cup X_2 \}$.

In *parallel composition*, the two components synchronize on their communication interface only when one component writes data through a channel, and the other reads it through the same channel. The two components cannot synchronize if both are trying to read or both are trying to write. We distinguish between communication of the two components with each other (on their common channels), and their communication with their environment. In the former case, the components must "wait" for each other in order to progress together. In the latter case, the communication actions of the two components interleave asynchronously.

Formally, the *parallel composition* of M_1 and M_2, denoted $M_1||M_2$, is the program $M = \langle Q, x, \alpha, \delta, q_0, F \rangle$ defined as follows.

1. $Q = (Q_1 \times Q_2) \cup (Q_1' \times Q_2')$, where Q_1' and Q_2' are new copies of Q_1 and Q_2, respectively. The initial state is $q_0 = (q_0^1, q_0^2)$.
2. $X = X_1 \cup X_2$.
3. $\alpha = \{\, (g?x_1, g!x_2), (g!x_1, g?x_2) \mid g * x_1 \in (\alpha_1 \cap \alpha I)$ and $g * x_2 \in (\alpha_2 \cap \alpha I)\} \cup ((\alpha_1 \cup \alpha_2) \setminus \alpha I)$. That is, the alphabet includes pairs of read-write communication actions on channels common to M_1 and M_2. It also includes individual actions of M_1 and M_2 – assignment actions, constraints and communication actions which are not communications on common channels.

4. δ is defined as follows.
   (a) For $(g * x_1, g * x_2) \in \alpha$:
      i. $\delta((q_1, q_2), (g * x_1, g * x_2)) = (q_1', q_2')$.
      ii. $\delta((q_1', q_2'), x_1 == x_2) = (\delta_1(q_1, g * x_1), \delta_2(q_2, g * x_2))$.
   That is, when a communication is performed synchronously in both components, the data is transformed through the channel from the writing component to the reading component. As a result, the values of x_1 and x_2 equalize. This is enforced in M by adding a transition labeled by the constraint $x_1 == x_2$ that immediately follows the synchronous communication.
   (b) For $a \in \alpha_1 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (\delta_1(q_1, a), q_2)$.
   (c) For $a \in \alpha_2 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (q_1, \delta_2(q_2, a))$.
   That is, on actions that are not in the interface alphabet, the two components interleave.
5. $F = F_1 \times F_2$

Figure 3 demonstrates the parallel composition of components M_1 and M_2 of Figures 1 and 2. The program $M = M_1||M_2$ reads a password from the environment through channel $read$. The two components synchronize on channels enc and $getEnc$.

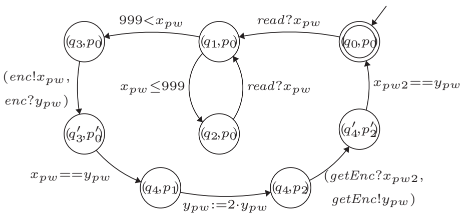


Fig. 3: Parallel composition $M = M_1||M_2$ of components M_1 and M_2 from Figures 1, 2

3 Regular Properties and Their Satisfaction

In this section we define the syntax and semantics of the properties that we consider. These are properties that can be represented as finite automata, hence the name *regular*. However, the alphabet of such automata includes communication actions and first-order constraints over program variables. Thus, such automata are suitable for specifying the desired and undesired behaviors of communicating programs over time.

In order to define our properties, we first need the notion of a *deterministic and complete* program. The definition is somewhat different from the standard definition for finite automata, since it takes the semantic meaning of constraints into account.

Intuitively, in a deterministic and complete program, every concrete run has exactly one trace that induces it.

Definition 5. *A program over alphabet α is* deterministic and complete *if for every state q and for every action $a \in \alpha$ the following hold:*

1. There is exactly one state q' such that (q, a, q') is in δ.[5]

[5] in our examples we sometimes omit the actions that lead to a rejecting sink for the sake of clarity.

2. *If (q, c_1, q') and (q, c_2, q'') are in δ for constraints $c_1, c_2 \in \mathcal{C}$ and $q' \neq q''$, then $c_1 \wedge c_2 \equiv false$.*
3. *Let C_q be the set of all constraints on transitions leaving q. Then $(\bigvee_{c \in C_q} c) \equiv true$.*

A *property* is a deterministic and complete program with no assignment actions.

A trace is accepted by a property P if it reaches a state in F, the set of accepting states of P. Otherwise, it reaches a state in $Q \setminus F$, and is rejected by P.

Next, we define the satisfaction relation $\vDash$ between a program and a property. Intuitively, a program M satisfies a property P (denoted $M \vDash P$) if all runs induced by accepted traces of M reach an accepting state in P.

A property P specifies the behavior of a program M by referring to communication actions of M and imposing constraints over the variables of M. Thus, the set of variables of P is identical to that of M. Let $\mathcal{G}$ be the set of communication actions of M. Then, αP includes a subset of $\mathcal{G}$ as well as constraints over the variables of M. The *interface* of M and P, which consists of the communication actions that occur in P, is defined as $\alpha I = \mathcal{G} \cap \alpha P$.

In order to capture the satisfaction relation between M and P, we define a *conjunctive composition* between M and P, denoted $M \times P$. In conjunctive composition, the two components synchronize on their common communication actions when both read or both write through the same communication channel. They interleave on constraints and on actions of αM that are not in αP.

Definition 6. *Let $M = \langle Q_M, X_M, \alpha M, \delta_M, q_0^M, F_M \rangle$ be a program and $P = \langle Q_P, X_P, \alpha P, \delta_P, q_0^P, F_P \rangle$ be a property, where $X_M = X_P$. The* conjunctive composition *of M and P is $M \times P = \langle Q, X, \alpha, \delta, q_0, F \rangle$, where:*

1. *$Q = Q_M \times Q_P$. The initial state is $q_0 = (q_0^M, q_0^P)$.*
2. *$X = X_M = X_P$.*
3. *$\alpha = \{\, g!x, g?x, (g?x, g!y), (g!x, g?y) \mid g * x, (g * x, g * y) \in \alpha I \} \cup ((\alpha M \cup \alpha P) \setminus \alpha I))$[6]. That is, the alphabet includes communication actions on channels common to M and P. It also includes individual actions of M and P.*
4. *δ is defined as follows.*
   (a) For $a = (g * x, g * y) \in \alpha I$, or $a = g * x \in \alpha I$: $\delta((q_1, q_2), a) = (\delta_M(q_1, a), \delta_P(q_2, a))$.
   (b) For $a \in \alpha M \setminus \alpha I$: $\delta((q_1, q_2), a) = (\delta_M(q_1, a), q_2)$.
   (c) For $a \in \alpha P \setminus \alpha I$: $\delta((q_1, q_2), a) = (q_1, \delta_P(q_2, a))$.
   That is, on actions that are not common communication actions to M and P, the two components interleave.
5. *$F = F_M \times B_P$, where $B_P = Q_P \setminus F_P$.*

Note that accepted traces in $M \times P$ are those that are accepted in M and rejected in P. Such traces are called *error traces* and their corresponding runs are called *error runs*. Intuitively, an error run is a run along M which violates the properties modeled by P. Such a run either fails to synchronize on the communication actions, or reaches a point in the computation in which its assignments, coming from M, violate some constraint described by P. These runs are manifested in the traces that are accepted in M but are composed with matching traces that are rejected in P. We can now formally define when a program satisfies a property.

Definition 7. *For a program M and a property P, we define $M \vDash P$ iff $M \times P$ contains no feasible accepted traces.*

[6] Note that communication actions of the form $(g * x, g * y)$ can only appear if M is a parallel composition of two programs.

Thus, a feasible error trace in $M \times P$ is an evidence to $M \not\models P$, since it indicates the existence of a run that violates P.

Example 2. Consider the program M of Figure 3 and the property P of Figure 2. As we discussed in Section 1, $M \not\models P$. The trace $t = \langle read?x_{pw}, 999 < x_{pw}, (enc!x_{pw}, enc?y_{pw}), x_{pw} == y_{pw}, y_{pw} := 2 \cdot y_{pw}, (getEnc?x_{pw2}, getEnc!y_{pw}), x_{pw2} == y_{pw}, x_{pw}! = x_{pw2}, y_{pw} \geq 2^{64}\rangle$ is a feasible error trace in $M \times P$ proving that an overflow is possible.

4 The Assume-Guarantee-Repair (AGR) Framework

In this section we discuss our Assume-Guarantee-Repair (AGR) framework for communicating programs. The framework consists of a learning-based Assume-Guarantee algorithm, called AG_{L^*}, and a REPAIR procedure, which are tightly joined.

Let M_1 and M_2 be two programs, and let P be a property. The classical Assume-Guarantee (AG) proof rule [26] assures that if we find an assumption A (in our case, a communicating program) such that $M_1||A \models P$ and $M_2 \models A$ both hold, then $M_1||M_2 \models P$ holds as well. For LTSs [9], the AG-rule is guaranteed to either prove correctness or return a real (non-spurious) counterexample. The work in [9] relies on the L^* algorithm [5] for learning an assumption A for the AG-rule. In particular, L^* aims at learning A_w, the weakest assumption for which $M_1||A_w \models P$ holds. A crucial point of this method is the fact that A_w is *regular* [15], and thus can be learned by L^*.

Lemma 1. *For infinite-state communicating programs, the weakest assumption A_w is not always regular.*

Example 3. Consider the programs M_1, M_2 and the property P of Figure 4. The weakest assumption with which M_1 satisfies P should contain exactly all traces (over the alphabet of M_2) that contain equally many actions of the form $x := x + 1$ and $y := y + 1$. This set of traces is not regular, and therefore cannot be learned by L^*.

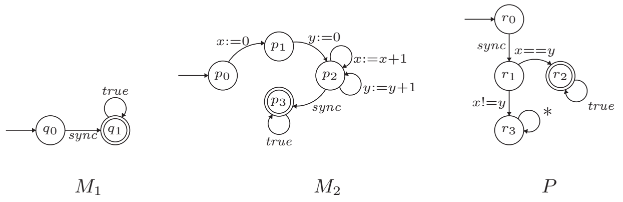


Fig. 4: A system for which the weakest assumption is not regular

To cope with this difficulty, we change the target of learning. Instead of learning the (possibly) non-regular language of A_w, we learn $\mathcal{T}(M_2)$, the set of accepted traces of M_2. This language is guaranteed to be regular, as it is represented by the automaton M_2.

Note that in case that $M_1||M_2 \models P$, repair is never needed, and M_2 is a valid assumption. In the worst case, the procedure halts once it has learned M_2. In particular, in case there are no error traces, termination of our algorithm is guaranteed. If $M_1||M_2 \not\models P$ then there does not exist a matching assumption, and attempting to learn M_2 will reveal this. Therefore, using $\mathcal{T}(M_2)$ as a learning goal matches the AG rule. The nature of

AG_{L^*} is such that the assumptions it learns before it reaches M_2 may contain the traces of M_2 and more, but still be represented by a smaller automaton. Therefore, similarly to [9], AG_{L^*} often terminates with an assumption A that is much smaller than M_2. Indeed, our tool often produces very small assumptions (see Section 5).

As mentioned before, not only that we determine whether $M_1||M_2 \models P$, but we also repair the program in case it violates the specification. When $M_1||M_2 \not\models P$, the AG_{L^*} algorithm returns an error trace t as a witness for the violation. In this case, we initiate the REPAIR procedure, which eliminates t from M_2. REPAIR applies abduction in order to learn a new constraint which, when added to t, creates an infeasible trace.[7] The new constraint enriches the alphabet in a way which may make similar traces infeasible as well. We elaborate on our use of abduction in Section 4.2. The removal of t and the addition of the new constraint result in a new goal M_2' for AG_{L^*} to learn. We now return to AG_{L^*} to search for a new assumption A' that allows to verify $M_1||M_2' \models P$.

An important feature of our AGR algorithm is its *incrementality*. When learning an assumption A' for M_2' we can use the membership queries previously asked for M_2, since the answer for them has not been changed. In the full version [1] we prove that the difference between the languages of M_2 and M_2' lies in words (traces) whose membership has not yet been queried on M_2. This allows the learning of M_2' to start from the point where the previous learning has left off, resulting in a more efficient algorithm.

As opposed to the case where $M_1||M_2 \models P$, we cannot guarantee the termination of the repair process in case $M_1||M_2 \not\models P$. This, since we are only guaranteed to remove one (bad) trace and add one (infeasible) trace in every AGR REPAIR iteration (although in practice, every iteration may remove a larger set of traces). Thus, we may never converge to a repaired system. Nevertheless, in case of property violation, our algorithm always finds an error trace, thus a progress towards a "less erroneous" program is guaranteed.

It should be noted that the AG_{L^*} part of our AGR algorithm deviates from the AG-rule of [9] in two important ways. First, since the goal of our learning is M_2 rather than A_w, our membership queries are different in type and order. Second, in order to identify real error traces and send them to REPAIR as early as possible, we add additional queries to the membership phase that reveal such traces. We then send them to REPAIR without ever passing through equivalence queries, which improves the overall efficiency. Indeed, our experiments include several cases in which all repairs were invoked from the membership phase. In these cases, AGR ran an equivalence query only when it has already successfully repaired M_2, and terminated.

4.1 The Assume-Guarantee-Repair (AGR) Algorithm

We now describe our AGR algorithm in more detail (see Algorithm 1). Figure 5 describes the flow of the algorithm. AGR comprises two main parts, namely AG_{L^*} and REPAIR.

The input to AGR are the components M_1 and M_2, and the property P. While M_1 and P stay unchanged during AGR, M_2 keeps being updated as long as the algorithm recognizes that it needs repair (we can guarantee termination in certain cases, as we discuss in Section 4.4).

The algorithm works in iterations, where in every iteration the next updated M_2^i is calculated, starting with iteration $i = 0$, where $M_2^0 = M_2$. An iteration starts with the membership phase in line 2, and ends either when AG_{L^*} successfully terminates (line 16) or when procedure REPAIR is called (lines 7 and 24). When a new system M_2^i is constructed, AG_{L^*} does not start from scratch. The information that has been used in previous iterations is still valid for M_2^i. The new iteration is given additional new trace(s) that have been added or removed from the previous M_2^i (lines 9,11,20, 27).

AG_{L^*} consists of two phases: membership, and equivalence.

The membership phase (lines 2-11) consists of a loop in which the learner constructs the next assumption A_j^i according to answers it gets from the teacher on a sequence of membership queries on various traces.

[7] There are also cases in which we do not use abduction, as discussed in Section 4.3

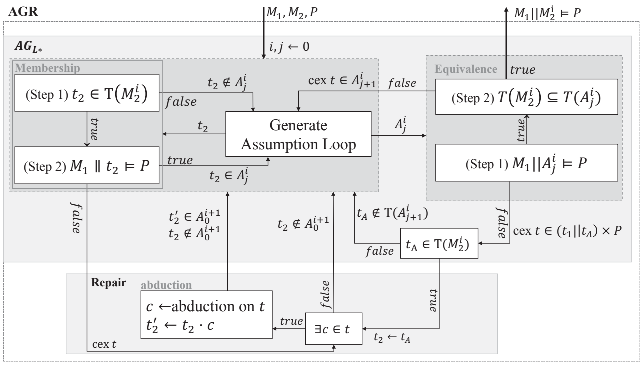


Fig. 5: The flow of AGR

These queries are answered in accordance with traces we allow in A_j^i: traces in M_2^i that in parallel with M_1 satisfy P. If a trace $t \in \mathcal{T}(M_2^i)$ in parallel with M_1 does not satisfy P, then t is a bad behavior of M_2. Therefore, if such a t is found during the membership phase, REPAIR is invoked.

Once the learner reaches a stable assumption A_j^i, it passes it to the equivalence phase (lines 12-27). A_j^i is a suitable assumption if both $M_1||A_j^i \models P$ and $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A_j^i)$ hold. In this case, AGR terminates and returns M_2^i as a successful repair of M_2. If $M_1||A_j^i \not\models P$, then a counterexample t is returned, that is composed of bad traces in M_1, A_j^i, and P. If the bad trace t_2, the restriction of t to the alphabet of A_j^i, is also in M_2^i, then t_2 is a bad behavior of M_2^i, and here too the REPAIR phase is invoked. Otherwise, AGR returns to the membership phase with t_2 as a trace that should not be in A_j^i, and continues to learn A_{j+1}^i.

As we have described, REPAIR is called when a bad trace t is found in $(M_1||M_2^i) \times P$ and should be removed. If t contains no constraints then its sequence of actions is illegal and its subtrace t_2 from M_2^i should be removed from M_2^i. In this case, REPAIR returns to AG_{L^*} with a new learning goal M_2^{i+1} such that $\mathcal{T}(M_2^{i+1}) \subseteq \mathcal{T}(M_2^i) \setminus \{t_2\}$, along with the answer "no" to the membership query on t_2. In 4.3 we discuss different methods for removing t_2 from M_2^i.

The more interesting case is when t contains constraints. In this case, we not only remove the matching t_2 from M_2^i, but we also add a new constraint c to the alphabet of M_2^{i+1}, which causes t_2 to be infeasible. This way we eliminate t_2, and may also eliminate a family of bad traces that violate the property in the same manner. We deduce c using abduction, see Section 4.2. As before, REPAIR returns to AG_{L^*} with a new goal to be learned, but now also with an extended alphabet. The membership phase is then provided with two new answers to the membership query: t_2 that should *not* be included in the new assumption, and $(t_2 \cdot c)$ that should be included.

Incremental learning One of the advantages of AGR is that it is *incremental*, in the sense that membership answers from previous iterations remain unchanged for the repaired system. Indeed, since this is the first time that AG_{L^*} queries t_2, we can return to AG_{L^*} with the answer $t_2 \notin \mathcal{T}(M_2^{i+1})$, without contradicting any previous queries. In addition, t_2' obtained by abduction is a new word (over a new alphabet), which also was not queried earlier. Therefore, we can incrementally add t_2 and t_2' as answers from the teacher, and continue to use answers from previous queries on all other traces.

Algorithm 1 AGR

```
1: function $AG_{L^*}$
2:     //Membership Queries
3:     Let $t_2 \in (\alpha M_2^i)^*$.
4:     if $t_2 \in \mathcal{T}(M_2^i)$ then
5:         if $M_1 || t_2 \nvDash P$ then
6:             Let $t \in (M_1 || t_2) \times P$ be an error trace.        ▷ $t$ is a cex proving $M_1 || M_2^i \nvDash P$
7:             REPAIR($M_2^i, t$)
8:         else                                                           ▷ $M_1 || t_2 \vDash P$
9:             Return to $AG_{L^*}$ in Line 2 with $t_2 \in \mathcal{T}(A_j^i)$.
10:    else                                                               ▷ $t_2 \notin \mathcal{T}(M_2^i)$
11:        Return to $AG_{L^*}$ in Line 2 with $t_2 \notin \mathcal{T}(A_j^i)$.
12:    //Equivalence Queries
13:    Let $A_j^i$ be the candidate assumption generated by the learner.
14:    if $M_1 || A_j^i \vDash P$ then
15:        if $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A_j^i)$ then
16:            Terminate and return $M_1 || M_2^i \vDash P$.
17:        else
18:            Let $t_2 \in \mathcal{T}(M_2^i) \setminus \mathcal{T}(A_j^i)$.
19:            Set $j := j + 1$
20:            Return to $AG_{L^*}$ in Line 2 with $t_2 \in \mathcal{T}(A_j^i)$.
21:    else                                                               ▷ $M_1 || A_j^i \nvDash P$
22:        let $t \in (M_1 || A_j^i) \times P$ be an error trace, and denote $t = (t_1 || t_A) \times t_P$.
23:        if $t_A \in \mathcal{T}(M_2^i)$ then
24:            REPAIR($M_2^i, t_A$)                                     ▷ $t_A$ is a cex proving $M_1 || M_2^i \nvDash P$
25:        else
26:            Set $j := j + 1$.
27:            Return to $AG_{L^*}$ in Line 2 with $t_A \notin \mathcal{T}(A_j^i)$.
28: function REPAIR($M_2^i, t$)
29:     Let $t_1 \in M_1, t_2 \in M_2^i, t_p \in P$ such that $t = (t_1 || t_2) \times t_p$.
30:     if $t$ does not contain constraints then
31:         Return to $AG_{L^*}$ in Line 2 with $M_2^{i+1}$ such that $\mathcal{T}(M_2^{i+1}) = \mathcal{T}(M_2^i) \setminus \{t_2\}$ and $t_2 \notin \mathcal{T}(A_0^{i+1})$.
32:     else                                                              ▷ $t$ contains constraints
33:         Use abduction to eliminate $t$.
34:         Let $c$ be the new constraint learned during abduction.
35:         Update $\alpha M_2^{i+1} = \alpha M_2^i \cup \{c\}$.
36:         Let $t_2' = t_2 \cdot c$ be the output of the abduction.
37:         Return to $AG_{L^*}$ in Line 2 with $M_2^{i+1}$ such that $\mathcal{T}(M_2^{i+1}) = (\mathcal{T}(M_2^i) \setminus \{t_2\}) \cup \{t_2'\}$,
38:         and $t_2 \notin \mathcal{T}(A_0^{i+1}), t_2' \in \mathcal{T}(A_0^{i+1})$
```

4.2 Repair by Abduction

We now describe the repair we apply to M_2^i, in case the error trace t contains constraints (see Algorithm 1, line 32). Error traces with no constraints are removed from M_2^i syntactically (line 31), while in abduction we *semantically* eliminate t by making it infeasible. The new constraints are then added to the alphabet of M_2^{i+1} in a way that may eliminate additional error traces. Note that even-though we add new alphabet letters to M_2, we do not add new *feasible traces*, since the constraints added by abduction can only restrict the behavior of M_2, making more traces infeasible. Therefore, we do not add counterexamples to M_2.

The process of inferring new constraints from known facts about the program is called *abduction* [11]. We now describe how we apply it. Given a trace t, let φ_t be the first-order formula (a conjunction of constraints), which constitutes the SSA representation of t [4]. In order to make t infeasible, we look for a formula ψ such that $\psi \wedge \varphi_t \rightarrow \mathit{false}$[8].

Note that $t \in \mathcal{T}(M_1||M_2^i) \times P$, and so it includes variables both from X_1, the set of variables of M_1, and from X_2, the set of variables of M_2^i. Since we wish to repair M_2^i, the learned ψ is over the variables in X_2 only.

The formula $\psi \wedge \varphi_t \rightarrow \mathit{false}$ is equivalent to $\psi \rightarrow (\varphi_t \rightarrow \mathit{false})$. Thus, $\psi = \forall x \in X_1(\varphi_t \rightarrow \mathit{false}) \equiv \forall x \in X_1(\neg\varphi_t)$, is such a desired constraint: ψ makes t infeasible and is defined only over X_2. We now use quantifier elimination [28] to produce a quantifier-free formula over X_2. Computing ψ is similar to the abduction suggested in [11], but the focus here is on finding a formula over X_2 rather than over any minimal set of variables. We use Z3 [10] to apply quantifier elimination and to generate the new constraint. After generating $\psi(X_2)$, we add it to the alphabet of M_2^{i+1} (line 35 of Algorithm 1). In addition, we produce a new trace $t'_2 = t_2 \cdot \psi(X_2)$. The trace t'_2 is returned as the output of the abduction.

Example 4. Recall the error trace $t = \langle \mathit{read}?x_{pw}, 999 < x_{pw}, (\mathit{enc}!x_{pw}, \mathit{enc}?y_{pw}), x_{pw} == y_{pw}, y_{pw} := 2 \cdot y_{pw}, (\mathit{getEnc}?x_{pw2}, \mathit{getEnc}!y_{pw}), x_{pw2} == y_{pw}, x_{pw}! = x_{pw2}, y_{pw} \geq 2^{64}\rangle$ of Example 2. From t we create the formula $\varphi_t = (999 < x_{pw}) \wedge (y_{pw} = x_{pw}) \wedge (y'_{pw} = 2 \cdot y_{pw}) \wedge (x_{pw2} = y'_{pw}) \wedge (x_{pw} \neq x_{pw2}) \wedge (y'_{pw} \geq 2^{64})$. We then apply quantifier elimination and simplification on the formula $\forall y_{pw} \forall y'_{pw}(\neg\varphi_t)$ and get the new constraint $x_{pw} < 2^{63}$.

Lemma 2. *Let* $t = (t_1||t_2) \times t_P$*. If* t_2 *is infeasible, then* t *is infeasible as well.*

This is due to the fact that t_P can only restrict the behaviors of t_1 and t_2, thus if t_2 is infeasible, t cannot be made feasible. See the full version of the paper [1] for a formal proof. Therefore, by making t_2 infeasible, we eliminate the error trace t.

We now want to build a repaired component M_2^{i+1} of M_2^i, which includes $t_2 \cdot \psi(X_2)$ but not t_2. To do so, we split the state q that t_2 reaches in M_2^i into two states q, q', and add a transition labeled $\psi(X_2)$ from q to q', where only q' is now accepting[9]. Thus, we eliminated a violating trace from $M_1||M_2^i$. AGR now returns to AG_{L^*} in order to learn an assumption for the repaired component M_2^{i+1}, which now includes t'_2 but not t_2.

4.3 Removal of Error Traces

Recall that the goal of REPAIR is to remove a bad trace t from M_2 once it is found by AG_{L^*}. If t contains constraints, we remove it using abduction. Otherwise, we can remove t by constructing a system whose language is $\mathcal{T}(M_2) \setminus \{t\}$. We call this the *exact* method for repair. However, removing a single trace at a time may lead to slow convergence, and to an exponential blow-up in the size of the repaired systems. Moreover, as we have discussed, in some cases there are infinitely many such error traces, in which case AGR may never terminate.

For faster convergence, we have implemented two additional heuristics, namely *approximate* and *aggressive*. These heuristics may remove more than a single trace at a time, while keeping the size of the systems small. While "good" traces may be removed as well, the correctness of the repair is maintained, since no bad traces are added. Moreover, an error trace is likely to be in an erroneous part of the system, and in these cases our heuristics manage removing a set of error traces in a single step.
We briefly survey the three methods.

[8] Usually, in abduction, we look for ψ such that $\psi \wedge \varphi_t$ is not a contradiction. In our case, however, since φ_t is a violation of the specification, we want to infer a formula that makes φ_t unsatisfiable.

[9] Note that q is an accepting state in M_2^i since $t \in \mathcal{T}(M_2^i)$.

- *Exact.* To eliminate only t from M_2, we construct a program (an automaton) A_t that accepts only t, and complement it to construct A'_t that accepts all traces except for t. Finally, we intersect A'_t with M_2.
- *Approximate.* Similarly to our repair via abduction in Section 4.2, we prevent the last transition that t takes from reaching an accepting state. Let q be the state that t reaches. We mark q as non-accepting, and add an accepting state q', to which all in-going transitions to q are diverted, except for the last transition on t. This way, some traces that lead to q are preserved by reaching q' instead, and the traces that share the last transition of t are eliminated along with t. As we have argued, these transitions may also be erroneous.
- *Aggressive.* In this simple method, we remove q, the state that t reaches, from the set of accepting states. This way we eliminate t along with all other traces that lead to q. In case that every accepting state is reached by some error trace, this repair might result in an empty language, creating a trivial repair. However, our experiments show that in most cases, this method quickly leads to a non-trivial repair.

4.4 Correctness and Termination

For this discussion, we assume a sound and complete teacher who can answer the membership and equivalence queries in AG_{L^*}, which require verifying communicating programs and properties with first-order constraints.

As we have discussed earlier, AGR is not guaranteed to terminate, and there are cases where the REPAIR stage may be called infinitely many times. However, in case that no repair is needed, or if a repaired system is obtained after finitely many calls to REPAIR, then AGR is guaranteed to terminate with a correct answer.

To see why, consider a repaired system M_2^i for which $M_1||M_2^i \models P$. Since the goal of AG_{L^*} is to syntactically learn M_2^i, which is regular, this stage will terminate at the latest when AG_{L^*} learns exactly M_2^i (it may terminate sooner if a smaller appropriate assumption is found). Notice that, in particular, if $M_1||M_2 \models P$, then AGR terminates with a correct answer in the first iteration of the verify-repair loop.

REPAIR is only invoked when a (real) error trace t is found in M_2^i, in which case a new system M_2^{i+1}, that does not include t, is produced by REPAIR. If $M_1||M_2^i \not\models P$, then an error trace is guaranteed to be found by AG_{L^*} either in the membership or equivalence phase. Therefore, also in case that M_2^i violates P, the iteration is guaranteed to terminate. To conclude, we have the following.

Theorem 1. – *An iteration i of AGR ends with an error trace t iff $M_1||M_2^i \not\models P$, where M_2^i is the repaired system at iteration i.*
– *If, after finitely many iterations, a repaired program M'_2 is such that $M_1||M'_2 \models P$, then AGR terminates with a correct answer.*

We have shown that every iteration of AGR is guaranteed to terminate with a correct answer. The detailed correctness proofs are in the full version of this paper [1].

In particular, since every iteration of AGR finds and removes an error trace t, and no new erroneous traces are introduced in the updated system, then in case that M_2 has finitely many error traces, AGR is guaranteed to terminate with a correctly repaired system.

5 Experimental Results and Conclusions

We implemented our AGR framework in Java, integrating L^* implementation from the LTSA tool [20]. We used Z3 [10] as the teacher for the satisfaction queries in AG_{L^*}, and for abduction in REPAIR.

Table 1 displays some results of running AGR on various examples, varying in their sizes, types of errors – semantic and syntactic – and their amount. Additional results are in the full version of this paper [1], and the full examples are available on [2]. The *iterations* column indicates the number of iterations of the verify-repair loop, until a repaired M_2 is achieved. Examples with no errors were verified in the first iteration, and are indicated by *verification*. We tested the three repair methods described in Section 4.3 for counterexamples without constraints, and used abduction when needed. Figure 6 presents comparisons between the three methods in terms of run-time and the size of the repair and assumptions (note that the graphs are given in logarithmic scale).

Table 1: AGR algorithm results on various examples

Example	M_1 **Size**	M_2 **Size**	P **Size**	**Time (sec.)**	A **size**	**Repair Size**	**Repair Method**	**#Iterations**
#4	64	64	3	95	7	verification		
#6	2	27	2	0.106	5	27	aggress.	2
				0.126	6	28	approx.	2
				0.132	8	81	exact	2
#7	2	81	2	0.13	6	81	aggress.	2
				0.138	7	82	approx.	2
				0.165	9	243	exact	2
#8	2	243	2	0.15	8	243	aggress.	2
				0.17	8	244	approx.	2
				0.223	10	729	exact	2
#11	5	256	6	4.88	92	verification		
#14	5	256	6	4.44	109	verification		
#15	3	16	5	0.69	12	16	aggress.	5
				0.28	13	18	approx.	3
				4.27	44	864	exact	5
#16	4	256	8	6.63	113	256	aggress.	2
				5.94	113	257	approx.	2
				12.87	155	1280	exact	2
#19	3	16	5	1.07	18	18	aggress.	3
				1.12	18	18	approx.	3
				1.26	18	18	exact	3
#22	2	4	2	0.09	1	4 (trivial)	aggress.	4
				0.21	6	8	approx.	5
				timeout			exact	timeout

Most of our examples model multi-client-server communication protocols, with varying sizes. Our tool managed repairing all these examples when needed.

As can be seen in Table 1, our tool successfully generates assumptions that are significantly smaller than the repaired and the original M_2.

For the examples that needed repair, in most cases our tool needed 2-5 iterations of verify-repair in order to successfully construct a repaired component. Interestingly, in example $\#15$ the *aggressive* method converged slower than the *approximate* method. This is due to the structure of M_2, in which different error traces lead to different states. Marking these states as non-accepting removed each trace separately. However, some of these traces have a common transition, and preventing this transition from reaching an accepting state, as done in the *approximate* method, managed removing several error traces in a single repair. This example also includes repairs by abduction (as do examples $\#16$, $\#18$ and $\#19$).

Example $\#22$ models a simple structure in which, due to a loop in M_2, the same alphabet sequence can generate infinitely many error traces. The *exact* repair method timed out, since it attempted removing one

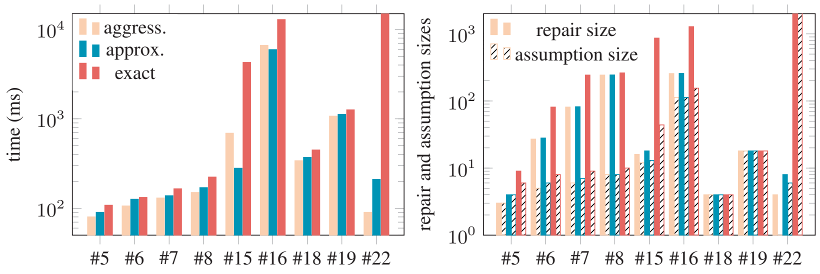


Fig. 6: Comparing repair methods: time and repair size (logarithmic scale).

error trace at a time. On the other hand, the *aggressive* method removed all accepting states, creating an empty program – a trivial (yet valid) repair. However, the *approximate* method created a valid, non-trivial repair.

Conclusion AGR offers a new take on the learning-based approach to assume-guarantee verification, and manages coping with complex properties and repairing infinite-state programs. Our experimental results show that using existing semantic tools, AGR produces very succinct proofs, and quickly and efficiently repairs flawed communicating programs.

References

1. http://hfrenkel.cswp.cs.technion.ac.il/agr-full-version/.
2. https://www.dropbox.com/sh/oi1joxvjuv5p3ag/AACOMDB6wGevkFogilQUyfXqa?dl=0.
3. A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.
4. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, 1988.
5. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
6. S. Chaki and O. Strichman. Optimized l*-based assume-guarantee reasoning. In *TACAS*, 2007.
7. Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang. Automated assume-guarantee reasoning through implicit learning. In *CAV*, 2010.
8. Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. In *TACAS*, 2009.
9. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, 2003.
10. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.
11. I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In *CAV*, 2013.
12. K. A. Elkader, O. Grumberg, C. S. Pasareanu, and S. Shoham. Automated circular assume-guarantee reasoning. In *FM*, 2015.
13. K. A. Elkader, O. Grumberg, C. S. Pasareanu, and S. Shoham. Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement. In *CAV*, 2016.
14. M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, 2007.
15. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*. IEEE Computer Society, 2002.
16. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005.

17. A. Gupta, K. L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. *Formal Methods in System Design*, 32(3):285–301, 2008.
18. B. Li, I. Dillig, T. Dillig, K. L. McMillan, and M. Sagiv. Synthesis of circular compositional program proofs via abduction. In *TACAS*, 2013.
19. S. Lin and P. Hsiung. Compositional synthesis of concurrent systems through causal model checking and learning. In *FM*, 2014.
20. J. Magee and J. Kramer. *Concurrency - state models and Java programs*. Wiley, 1999.
21. K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, 1999.
22. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
23. K. S. Namjoshi and R. J. Trefler. On the competeness of compositional reasoning. In *CAV*, 2000.
24. C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 2008.
25. C. Peirce and C. Hartshorne. *Collected Papers of Charles Sanders Peirce*. Belknap Press, 1932.
26. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, NATO ASI Series*, 1985.
27. R. Singh, D. Giannakopoulou, and C. S. Pasareanu. Learning component interfaces with may and must abstractions. In *CAV*, 2010.
28. V. Weispfenning. Quantifier elimination and decision procedures for valued fields. *Models and Sets.Lecture Notes in Mathematics (LNM)*, 1103:419—472, 1984.

Structural Invariants for the Verification of Systems with Parameterized Architectures

Marius Bozga[1], Javier Esparza[2], Radu Iosif[1], Joseph Sifakis[1] and Christoph Welzel[2]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP[⋆⋆], Verimag
[2] Technische Universität München

We consider parameterized concurrent systems consisting of a finite but unknown number of components, obtained by replicating a given set of finite state automata. Components communicate by executing atomic interactions whose participants update their states simultaneously. We introduce an interaction logic to specify both the type of interactions (e.g. rendez-vous, broadcast) and the topology of the system (e.g. pipeline, ring). The logic can be easily embedded in monadic second order logic of $\kappa \geq 1$ successors (WSκS), and is therefore decidable.

Proving safety properties of such a parameterized system, like deadlock freedom or mutual exclusion, requires to infer an inductive invariant that contains all reachable states of all system instances, and no unsafe state. We present a method to automatically synthesize inductive invariants directly from the formula describing the interactions, without costly fixed point iterations. We experimentally prove that this invariant is strong enough to verify safety properties of a large number of systems, including textbook examples (dining philosophers, synchronization schemes), classical mutual exclusion algorithms, cache-coherence protocols and self-stabilization algorithms, for an arbitrary number of components.

1 Introduction

The problem of parameterized verification asks whether a system composed of n replicated processes is safe, for all $n \geq 2$. By safety we mean that every execution of the system stays clear of a set of global error configurations, such as deadlocks or mutual exclusion violations. Even if we assume each process to be finite-state and every interaction to be a synchronization of actions without exchange of data, ranging over large or infinite domains, the problem remains challenging because we ask for a general proof of safety that works for any number of processes.

Parameterized verification is undecidable, even if processes only manipulate data from a bounded domain [6]. Various restrictions of communication and architecture[3] define decidable subproblems [18,31,27,5]. Seminal works consider *rendez-vous* communication, with participants placed in a ring [18,27] or a clique [31] of arbitrary size. Recently, MSO-definable graphs (with bounded tree- and clique-width) and point-to-point rendez-vous communication have been considered [5]. Most approaches to define decidable problems focus on manually proving a *cut-off* bound $c \geq 2$ such that

[⋆⋆] Institute of Engineering Univ. Grenoble Alpes

[3] We use the term architecture for the shape of the graph along which the interactions take place.

correctness for at most c processes implies correctness for any number of processes [18,27,26,7,34]. Other methods identify systems with well-structured transition relations [31,1,29]. An exhaustive chart of decidability results for verification of parameterized systems is drawn in [12]. When decidability is not of concern, over-approximation and semi-algorithmic techniques such as *regular model checking* [36,2], SMT-based *bounded model checking* [4,21], *abstraction* [10,14] and *automata learning* [19] can be used to deal with more general classes of systems.

The efficiency of a verification method crucially relies on its ability to synthesize an *inductive safety invariant*, i.e., an infinite set of configurations that contains the initial configurations, is closed under the transition relation, and excludes the error configurations. In general, automatically synthesizing invariants requires computationally expensive fixpoint iterations [22]. In the particular case of parameterized systems, invariants can be either *global*, relating the local states of all processes [23], or *modular*, relating the local states of a few processes whose identity is irrelevant [38,20].

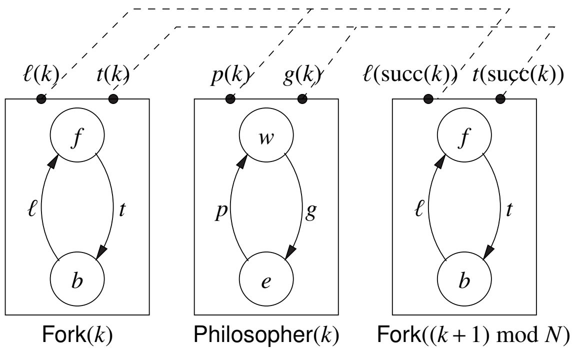
Our Contributions. The novelty of the approach described in this paper is three-fold:

1. The architecture of the system is not fixed a priori, but given as a parameter of the verification problem. In fact, we describe parameterized systems using the Behavior-Interaction-Priorities (BIP) framework [9], in which processes are instances of finite-state *component types*, whose interfaces are sets of *ports*, labeling transitions between local states, and interactions are sets of strongly synchronizing ports, described by formulae of an *interaction logic*. An interaction formula captures the architecture of the interactions (pipeline, ring, clique, tree) and the communication scheme (rendez-vous, broadcast), which are not hardcoded, but rather specified by the designer of the system.
2. We synthesize parameterized invariants directly from the interaction formula of a system, without iterating its transition relation. Such invariants depend only on the structure (and not on the operational semantics) of an infinite family of Petri Nets, one for each instance of the system, and are thus *structural* invariants. Essentially, the invariants we infer use the *traps*[4] of the system, which are sets W of local states with the property that, if a process is initially in a state from W, then always some process will be in a state from W. Following [11,17], we call them (parameterized) *trap invariants*. Computing trap invariants only requires a simple syntactic transformation of the interaction formula and the result is expressed using $\mathsf{WS}\kappa\mathsf{S}$, the weak monadic second order logic of $\kappa \geq 1$ successor functions. Thus invariant computation is very cheap, and the verification problem (proving the emptiness of the intersection between the invariant and the set of error states) is reduced to the unsatisfiability of a $\mathsf{WS}\kappa\mathsf{S}$ formula with a single quantifier alternation. In practice, this check can be carried out quite efficiently by existing tools, such as Mona [33].
3. We refine the approach by considering so called 1-*invariants*, that can also be derived cheaply from the interaction formula of the system. We show that 1-invariants in conjunction with trap invariants successfully verify additional examples.

Comparison to related work. Trap invariants have been very successfully used in the verification of non-parameterized systems [11,28,13]. The technique was lifted to parameterized systems in [17], but the work there is only applicable to clique architec-

[4] Called in this way by analogy with the notion of traps for Petri Nets [39].

tures, in which processes are indistinguishable, and the system can be described by one single Petri Net with an infinite family of initial markings. Here, for the first time, we show that the trap technique can be extended to pipelines, token rings and trees, where the system is defined by an infinite family of Petri Nets, each with a different structure. These systems cannot be analyzed using the techniques of [31,1,29], because they do not yield well-structured transition systems. Contrary to [18,27,26,7,34], our approach does not require a manual cut-off proof. Contrary to regular model checking and automata learning [2,19], it does not require any symbolic state-space exploration. Finally, our approach produces an explanation of why the property holds in terms of the trap invariant and 1-invariants used. Summarizing, our approach provides a comparatively cheap technique for parameterized verification, that succeeds in numerous cases. It is ideal as preprocessing step that can very quickly lead to success with a very clear explanation of why the property holds, and otherwise provides at least a strong invariant that can be used for further analysis.



$$\Gamma_{philo} = (g(i) \wedge t(i) \wedge t(\text{succ}(i))) \vee (p(i) \wedge \ell(i) \wedge \ell(\text{succ}(i)))$$

Fig. 1: Parameterized Dining Philosophers

Running Example. Consider the dining philosophers system in Fig. 1, consisting of $n \geq 2$ components of type Fork and Philosopher respectively, placed in a ring of size $2n$. The k-th philosopher has a left fork, of index k, and a right fork, of index $(k+1)$ mod n. Each component is an instance of a finite state automaton with states f(ree) and b(usy) for Fork, respectively w(aiting) and e(ating) for Philosopher. A fork goes from state f to b via a t(ake) transition and from f to b via a ℓ(eave) transition. A philosopher goes from w to e via a g(et) transition and from e to w via a p(ut) transition. The g action of the k-th philosopher is executed jointly with the t actions of the k-th and $[(k+1)$ mod $n]$-th forks, in other words, the philosopher takes both its left and right forks simultaneously. Similarly, the p action of the k-th philosopher is executed simultaneously with the ℓ action of the k-th and $[(k+1)$ mod $n]$-th forks, i.e. each philosopher leaves both its left and right forks at the same time. We describe these interactions by the *interaction formula*:

$$\Gamma_{philo} = \ (g(i) \wedge t(i) \wedge t(\text{succ}(i))) \ \vee \ (p(i) \wedge \ell(i) \wedge \ell(\text{succ}(i))) \qquad (1)$$

where the free variable i refers at some arbitrary component index.

Intuitively, the transitions of the system with n dining philosophers and n forks are given by the *minimal models* of the disjuncts of Γ_{philo} with universe $\{0, 1, \ldots, n-1\}$, and succ interpreted as "successor modulo n'. In particular, for each $0 \leq k \leq n-1$ the first disjunct has a minimal model that interprets the predicates g and t as the sets $\{k\}$ and $\{k, (k+1) \bmod n\}$. This model describes the interaction in which the k-th philosopher takes a g-transition (from waiting to eating), while, simultaneously, the k-th and $(k+1)$-th forks take t-transitions (from free to busy). This is graphically represented by one of the dashed lines in Fig. 1. Observe that the ring topology of the system is implicit in the modulo-n interpretation of the successor function.

Since philosophers can only grab their two forks simultaneously, the system is deadlock-free for any number $n \geq 2$ of philosophers. An automatic proof requires to compute an invariant, and prove that it has an empty intersection with the set of deadlock configurations defined by the $\mathsf{WS}\kappa\mathsf{S}$ formula

$$deadlock(X_w, X_e, X_f, X_b) = \forall i \,.\, [\neg X_w(i) \vee \neg X_f(i) \vee \neg X_f(\mathrm{succ}(i))] \wedge [\neg X_e(i) \vee \neg X_b(i) \vee \neg X_b(\mathrm{succ}(i))] \quad (2)$$

where X_w, X_e, X_f, X_b are set variables, the intended meaning of $X_w(i)$ resp. $X_e(i)$ is that the i-th philosopher is waiting, resp. eating, and the intended meaning of $X_f(i)$ resp. $X_b(i)$ is that the i-th fork is free, resp. busy. Our method automatically computes from Γ_{philo} a formula $trap\text{-}invariant_{\mathcal{S}}$ which formalizes an inductive invariant of the system. Moreover, we express the consistency requirement that every component is in one of its state at all times in a formula $marking_{\mathcal{S}}$ and derive the deadlock-freeness for any number of philosophers by the unsatisfiability of the formula

$$deadlock \wedge trap\text{-}invariant_{\mathcal{S}} \wedge marking_{\mathcal{S}}\,.$$

2 Parameterized Component-based Systems

A *component type* is a tuple $C = \langle \mathsf{P}, \mathsf{S}, s_0, \Delta \rangle$, where $\mathsf{P} = \{p, q, r, \ldots\}$ is a finite set of *ports*, S is a finite set of *states*, $s_0 \in \mathsf{S}$ is an initial state and $\Delta \subseteq \mathsf{S} \times \mathsf{P} \times \mathsf{S}$ is a set of *transitions* denoted $s \xrightarrow{p} s'$, for $s, s' \in \mathsf{S}$ and $p \in \mathsf{P}$. We assume there are no two different transitions with the same port.

A *component-based system* $\mathcal{S} = \langle C^1, \ldots, C^N, \Gamma \rangle$ consists of a fixed number $N \geq 1$ of component types $C^k = \langle \mathsf{P}^k, \mathsf{S}^k, {s_0}^k, \Delta^k \rangle$ and an *interaction formula* Γ. In the dining philosophers there are two component types, Philosopher and Fork, each with two states and two transitions, as shown in Fig. 1. We assume that $\mathsf{P}^i \cap \mathsf{P}^j = \emptyset$ and $\mathsf{S}^i \cap \mathsf{S}^j = \emptyset$, for all $1 \leq i < j \leq N$. We denote the component type of a port p or a state s by $type(p)$ and $type(s)$, respectively. For instance, in Fig. 1 we have $type(p) = type(g) = type(w) = type(e) =$ Philosopher and $type(t) = type(\ell) = type(f) = type(b) =$ Fork.

The interaction formula Γ determines the family of systems we can construct out of these components. It does so by specifying, for each possible number of replicated instances (for example, 3 philosophers and 3 forks), which are the possible interactions between them. An interaction consists of a set of transitions that are executed simultaneously. For example, in an interaction philosopher 3 executes a g(et) transition simultaneously with t(ake) transitions of the forks 2 and 3. Before formalizing this, we introduce the syntax and semantics of Interaction Logic.

Interaction Logic. For a constant $\kappa \geq 1$, fixed throughout the paper, the *Interaction Logic* $\mathsf{IL}\kappa$ is built on top of a countably infinite set Var of variables, the set $\mathsf{Pred} = \bigcup_{k=1}^{N} \mathsf{P}^k$ of monadic predicate symbols ranged over by pr (i.e. the logic has a predicate symbol for each port), the binary predicate $\leq$, and the *successor* functions $\mathrm{succ}_0, \ldots, \mathrm{succ}_{\kappa-1}$, of arity one. The formulae of $\mathsf{IL}\kappa$ are generated by the syntax

$$\begin{array}{ll} t := i \in \mathsf{Var} \mid \mathrm{succ}_0(t) \mid \ldots \mid \mathrm{succ}_{\kappa-1}(t) & \text{terms} \\ \phi := t_1 \leq t_2 \mid \mathsf{pr}(t) \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \exists i \,.\, \phi_1 & \text{formulae} \end{array}$$

Abbreviations like $t_1 = t_2$, $t_1 < t_2$, $\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2$, and $\forall i \,.\, \phi$ are defined as usual. $\mathsf{IL}\kappa$ is interpreted over finite ranked trees of arity κ, which we identify with a prefix-closed language of words, also called *nodes*, over the alphabet $\{0, \ldots, \kappa-1\}$. The root of the tree is the empty word ϵ, and the children of w are $w0, w1, \ldots, w(\kappa-1)$. Formally, an *interpretation* or *structure* is a pair $\mathcal{I} = (\mathfrak{U}, \iota)$, where the universe $\mathfrak{U}$ is a tree and ι assigns a node to each variable and a set of nodes to each predicate in Pred. The predicate $\leq$ and the functions $\mathrm{succ}_0, \ldots, \mathrm{succ}_{k-1}$ have the usual fixed interpretations: If t and t' are interpreted as w and w', then $t_1 \leq t_2$ holds iff w is a prefix of w', and $\mathrm{succ}_i(t)$ is interpreted as the node wi, if $wi \in \mathfrak{U}$, and as the root ϵ otherwise. So, loosely speaking, successor functions wrap around to the root.

When $\kappa = 1$, formulae are interpreted on languages $\{\epsilon, 0, 00, \ldots, 0^{n-1}\}$ for some number n. To simplify notation, in this case we assume that they are interpreted over the set $\{0, 1, \ldots, n-1\}$, and succ_0 is the usual successor function on numbers, modulo n.

Intuitively, a universe $\mathfrak{U}$ determines an instance of the component-based system, with one instance of each component for each $w \in \mathfrak{U}$. So, for example, for $\kappa = 1$ and $\mathfrak{U} = \{0, 1, 2, \ldots, n-1\}$ in our running example we have philosophers $0, 1, \ldots, n-1$ and forks $0, 1, \ldots, n-1$. Generally, with $\kappa = 1$ we can describe pipeline and token-ring architectures, whereas higher values describe tree-shaped architectures.

Interaction formulae. A formula of $\mathsf{IL}\kappa$ is an *interaction formula* if it is the conjunction of the following formula:

$$\forall i \forall j \,.\, \bigwedge_{\substack{p,q \in \mathsf{Pred} \\ type(p) = type(q)}} p(i) \wedge q(j) \rightarrow i \neq j \tag{3}$$

with a finite disjunction of formulae of the form:

$$\mathfrak{C}(i_1, \ldots, i_\ell) \stackrel{\mathsf{def}}{=} \varphi \;\wedge\; \bigwedge_{j=1}^{\ell} p_j(i_j) \;\wedge\; \bigwedge_{j=1}^{m} \forall k \,.\, \psi_j \rightarrow q_j(k) \tag{4}$$

where $\varphi, \psi_1, \ldots, \psi_m$ are conjunctions of atomic formulae of the form $t_1 \leq t_2$ and their negations. Intuitively, formula (3) is a generic axiom that prevents two ports of the same instance of a component type from interacting. The formulae of form (4) are called the *clauses* of the interaction formula.

Example 1. Consider a component-based system $\mathcal{S} = \langle C^1, C^2, \Gamma \rangle$, where C^1 and C^2 have ports p_1 and p_2, respectively, and Γ has one single clause

$$\mathfrak{C}(i, j, k) = \;(i < j \wedge k = \mathrm{succ}(j)) \;\wedge\; (p_1(i) \wedge p_2(j)) \;\wedge\; \forall i . i > k \rightarrow p_1(i)$$

Γ states that an interaction consists of: the i-th process of type C^1 executes transition p_1; the j-th process of type C^2 executes p_2; and, for every $i > (j+1) \bmod n$, the i-th process of type C^1 executes transition p_1 as well; all this happens simultaneously in one atomic step. ■

Loosely speaking, (4) states that in an interaction ℓ components can simultaneously engage in a multiparty rendez-vous, together with a broadcast to the ports $q_1, \ldots, q_m$ of the components whose indices satisfy the constraints $\psi_1, \ldots, \psi_m$, respectively. An example of peer-to-peer rendez-vous with no broadcast is the dining philosophers system in Fig. 1, whereas examples of broadcast are found among the benchmarks in §5. In the next section we show that, despite this generality, it is possible to construct a trap invariant for any interaction formula in a purely syntactic way.

Observe that the interaction formula does not explicitly specify that every other process remains idle. Formally, as we will see in the next section, the system has an interaction for each *minimal model* of (4), which allows us not to have to specify idleness. Given structures $\mathcal{I}_1 = (\mathfrak{U}, \iota_1)$ and $\mathcal{I}_2 = (\mathfrak{U}, \iota_2)$ sharing the same universe $\mathfrak{U}$, we say $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2$ if and only if $\iota_1(\mathsf{pr}) \subseteq \iota_2(\mathsf{pr})$ for every $\mathsf{pr} \in \mathsf{Pred}$. Given a formula ϕ, a structure $\mathcal{I}$ is a *minimal model* of ϕ if $\mathcal{I} \models \phi$ and, for all structures $\mathcal{I}'$ such that $\mathcal{I}' \sqsubseteq \mathcal{I}$ and $\mathcal{I}' \neq \mathcal{I}$, we have $\mathcal{I}' \not\models \phi$.

2.1 Execution Semantics of Component-based Systems

The semantics of a component-based system $\mathcal{S} = \langle C^1, \ldots, C^N, \Gamma \rangle$ is an infinite family of Petri Nets, one for each universe of Γ. The reachable markings and actions of the Petri Net characterize the reachable global states and transitions of the system, respectively. To fix notations, we recall several basic definitions.

Preliminaries: Petri Nets. A *Petri Net* (PN) is a tuple $\mathsf{N} = \langle S, T, E \rangle$, where S is a set of *places*, T is a set of *transitions*, $S \cap T = \emptyset$, and $E \subseteq (S \times T) \cup (T \times S)$ is a set of *arcs*. The elements of $S \cup T$ are called *nodes*. Given nodes $x, y \in S \cup T$, we write $E(x,y) \overset{\text{def}}{=} 1$ if $(x,y) \in E$ and $E(x,y) \overset{\text{def}}{=} 0$, otherwise. For a node x, let ${}^\bullet x \overset{\text{def}}{=} \{y \in S \cup T \mid E(y,x) = 1\}$, $x^\bullet \overset{\text{def}}{=} \{y \in S \cup T \mid E(x,y) = 1\}$ and lift these definitions to sets of nodes.

A *marking* of N is a function $\mathrm{m} : S \to \mathbb{N}$. A transition t is *enabled* in m if and only if $\mathrm{m}(s) > 0$ for each place $s \in {}^\bullet t$. For all markings m, m′ and transitions t, we write $\mathrm{m} \xrightarrow{t} \mathrm{m}'$ whenever t is enabled in m and $\mathrm{m}'(s) = \mathrm{m}(s) - E(s,t) + E(t,s)$, for all $s \in S$. Given two markings m and m′, a finite sequence of transitions $\sigma = t_1, \ldots, t_n$ is a *firing sequence*, written $\mathrm{m} \xrightarrow{\sigma} \mathrm{m}'$ if and only if either (i) $n = 0$ and $\mathrm{m} = \mathrm{m}'$, or (ii) $n \geq 1$ and there exist markings $\mathrm{m}_1, \ldots, \mathrm{m}_{n-1}$ such that $\mathrm{m} \xrightarrow{t_1} \mathrm{m}_1 \ldots \mathrm{m}_{n-1} \xrightarrow{t_n} \mathrm{m}'$.

A *marked Petri Net* is a pair $\mathcal{N} = (\mathsf{N}, \mathrm{m}_0)$, where m_0 is the *initial marking* of N. A marking m is *reachable* in $\mathcal{N}$ if there exists a firing sequence σ such that $\mathrm{m}_0 \xrightarrow{\sigma} \mathrm{m}$. We denote by $\mathcal{R}(\mathcal{N})$ the set of reachable markings of $\mathcal{N}$. A marked PN $\mathcal{N}$ is 1-*safe* if $\mathrm{m}(s) \leq 1$, for each $s \in S$ and $\mathrm{m} \in \mathcal{R}(\mathcal{N})$. All PNs considered in the following will be 1-safe and we shall silently blur the distinction between a marking $\mathrm{m} : S \to \{0,1\}$ and the boolean valuation $\beta_\mathrm{m} : S \to \{\bot, \top\}$ defined as $\beta_\mathrm{m}(s) = \top \iff \mathrm{m}(s) = 1$. A set of markings $\mathcal{M}$ is an *inductive invariant* of $\mathcal{N} = (\mathsf{N}, \mathrm{m}_0)$ if and only if $\mathrm{m}_0 \in \mathcal{M}$ and for each $\mathrm{m} \xrightarrow{t} \mathrm{m}'$ such that $\mathrm{m} \in \mathcal{M}$, we have $\mathrm{m}' \in \mathcal{M}$.

Petri Net Semantics of Component-Based Systems. We define the semantics of a component-based system as an infinite family of 1-safe Petri Nets. For $k = 1, \ldots, N$ let $C^k = \langle \mathsf{P}^k, \mathsf{S}^k, s_0{}^k, \Delta^k \rangle$ be a component type and, then, let $\mathcal{S} = \langle C^1, \ldots, C^N, \Gamma \rangle$ be a system. Fix a universe $\mathfrak{U}$ of Γ. We define a marked Petri Net $\mathcal{N}_\mathcal{S}^\mathfrak{U} \overset{\text{def}}{=} (\langle S, T, E \rangle, \mathrm{m}_0)$ as follows:

- $S \overset{\text{def}}{=} \left(\bigcup_{k=1}^{N} \mathsf{S}^k\right) \times \mathfrak{U}$. That is, the net has a place (s,u) for each state s of each component type, and for each node u.
- For each minimal model $\mathcal{I} = (\mathfrak{U}, \iota)$ of a clause $\mathfrak{C}$ of Γ, the set T contains a transition $\mathsf{t}_\iota \in T$, and the set E contains edges $((s,u), \mathsf{t}_\iota)$ and $(\mathsf{t}_\iota, (s',u))$ for every $s \xrightarrow{p} s' \in \left(\bigcup_{k=1}^{N} \Delta^k\right)$ such that $u \in \iota(p)$. Nothing else is in T or E. Intuitively, t_ι "synchronizes" all the transitions $s \xrightarrow{p} s'$ of the different components occurring in the interaction.
- For each $1 \leq k \leq N$, each $s \in \mathsf{S}^k$ and each $u \in \mathfrak{U}$, $\mathrm{m}_0((s,u)) = 1$ if $s = s_0{}^k$ and $\mathrm{m}_0((s,u)) = 0$, otherwise. That is, m_0 contains the places (s,u) such that s is an initial state.

It follows immediately from this definition that $\mathcal{N}_S^{\mathfrak{U}}$ is a 1-safe Petri Net. Indeed, for every $u \in \mathfrak{U}$, for every component-type C^k, and for every reachable marking m, we have $\sum_{s \in \mathsf{S}^k} m((s,u)) = 1$. This reflects that the instance of C^k at u is always in exactly one of the states of S^k; if s is that state, then (s,u) is the place carrying the token.

Example 2. Consider our running example, with $\mathfrak{U} = \{0, 1, \ldots, n-1\}$, i.e., n philosophers and n forks. Since the interaction formula (1) has no constants, its models are pairs $(\mathfrak{U}, \iota)$, where ι gives the interpretation of the free variable i and the predicates g, t, etc. The first disjunct of (1) is $[g(i) \wedge t(i) \wedge t(\mathrm{succ}(i))]$. It has a minimal model for each $k \in \mathfrak{U}$, namely the model with $\iota(i) = k$, $\iota(g) = \{k\}$ and $\iota(t) = \{k, (k+1) \bmod n\}$. In the interaction produced by this model, the k-th philosopher executes transition g(et), the forks with numbers k and $(k+1) \bmod n$ execute transition t(ake), and all other philosophers and forks remain idle. The second disjunct yields the interactions in which a philosopher puts down its forks. Fig. 2 shows the Petri Net $\mathcal{N}_S^{\mathfrak{U}}$ for universe $\mathfrak{U} = \{0,1,2\}$. For clarity,

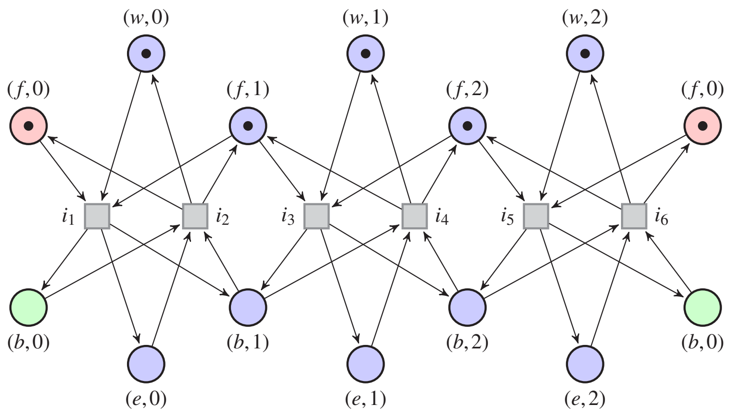


Fig. 2: Petri Net of the dining philosophers for the universe $\mathfrak{U} = \{0,1,2\}$. In reality, the two pink and green places are only one place.

the places $(f,0)$ and $(b,0)$ have been duplicated; in reality the two copies are merged. The places of each philosopher are $\{(w,i),(e,i)\}$ for $i = 0,1,2$. For example, transition

i_3 corresponds to the minimal model $\{(g,1),(t,1),(t,2)\}$, in which philosopher 1 takes forks 1 and 2.

3 Trap Invariants

Given a Petri Net $\mathsf{N} = (S,T,E)$, a set of places $W \subseteq S$ is called a *trap* if and only if $W^\bullet \subseteq {}^\bullet W$. A trap W of N is an *initially marked trap* (IMT) of the marked PN $\mathcal{N} = (\mathsf{N},\mathrm{m}_0)$ if and only if $\mathrm{m}_0(s) = \top$ for some $s \in W$.

Example 3. $\{(f,1),(b,1)\}$ and $\{(f,0),(b,1),(f,2),(e,2)\}$ are two traps of the Petri Net in Figure 2.

An IMT defines an invariant of the Petri Net, because some place in the trap will always be marked, no matter which sequence of transitions is fired. The *trap invariant* of $\mathcal{N}$ is the set of markings that mark each IMT of $\mathcal{N}$. Clearly, since marked traps remain marked, the set of reachable markings is contained in the trap invariant. Hence, to prove that a certain set of markings is unreachable, it is sufficient to prove that the set has empty intersection with the trap invariant. For self-completeness, we briefly discuss the computation of the trap invariant for a given marked Petri Net of fixed size, before explaining how this can be done for the infinite family of marked Petri Nets defining the executions of parameterized systems.

The *trap constraint* of a Petri Net $\mathsf{N} = (S,T,E)$ is the formula:

$$\Theta(\mathsf{N}) \stackrel{\text{def}}{=} \textstyle\bigwedge_{t \in T} \left(\bigvee_{x \in {}^\bullet t} x\right) \rightarrow \left(\bigvee_{y \in t^\bullet} y\right)$$

where each place $x, y \in S$ is viewed as a propositional variable. It is not hard to show[5] that any boolean valuation $\beta : S \rightarrow \{\bot,\top\}$ that satisfies the trap constraint $\Theta(\mathsf{N})$ defines a trap W_β of N in the obvious sense $W_\beta = \{s \in S \mid \beta(s) = \top\}$. Further, if $\mathrm{m}_0 : S \rightarrow \{0,1\}$ is the initial marking of a 1-safe Petri Net N and $\mu_0 \stackrel{\text{def}}{=} \bigvee_{\mathrm{m}_0(s)=1} s$ is a propositional formula, then every valuation of $\mu_0 \wedge \Theta(\mathsf{N})$ defines an IMT of $(\mathsf{N},\mathrm{m}_0)$. Usually, computing invariants requires building a sequence of underapproximants whose limit is the least fixed point of an abstraction of the transition relation of the system [22]. This is not the case of the trap invariant, that can be directly computed from the trap constraint and the initial marking [11,17].

In the rest of the section we construct a parameterized trap constraint that characterizes the traps, not of one single net, as $\Theta(\mathsf{N})$, but of the infinite family of Petri Nets obtained from a component-based system. The parameterized trap constraint is a formula of $\mathsf{WS}\kappa\mathsf{S}$. In Section 3.1 we first explain how to embed our interaction logic into $\mathsf{WS}\kappa\mathsf{S}$, and in Section 3.2 we construct the parameterized trap constraint.

3.1 From $\mathsf{IL}\kappa$ to $\mathsf{WS}\kappa\mathsf{S}$

We briefly recall the syntax and semantics of $\mathsf{WS}\kappa\mathsf{S}$, the monadic second order logic $\mathsf{WS}\kappa\mathsf{S}$ of κ successors (see e.g. [37]). Let SVar be a countably infinite set of second-

[5] See e.g. [8] for a proof.

order variables (also called set variables), denoted as $X, Y, \ldots$ in the following. The syntax of $\mathsf{WS}\kappa\mathsf{S}$ is:

$$\begin{array}{ll} t := \overline{\epsilon} \mid x \mid \mathrm{succ}_0(t) \mid \ldots \mid \mathrm{succ}_\kappa(t) & \text{terms} \\ \phi := t_1 = t_2 \mid \mathsf{pr}(t) \mid X(t) \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \exists x \,.\, \phi_1 \mid \exists X \,.\, \phi_1 & \text{formulae} \end{array}$$

So $\mathsf{WS}\kappa\mathsf{S}$ extends $\mathsf{IL}\kappa$ with the constant symbol $\overline{\epsilon}$, atoms $X(t)$ and monadic second order quantifiers $\exists X \,.\, \phi$. We can consider w.l.o.g. equality atoms $t_1 = t_2$ instead of the inequalities $t_1 \leq t_2$ in $\mathsf{IL}\kappa$, because the latter can be defined in $\mathsf{WS}\kappa\mathsf{S}$ as usual:

$$x \leq y \stackrel{\text{def}}{=} \forall X \,.\, \mathit{closed}(X) \wedge X(x) \rightarrow X(y) \qquad \mathit{closed}(X) \stackrel{\text{def}}{=} \forall x \,.\, X(x) \rightarrow \bigwedge_{i=0}^{\kappa-1} X(\mathrm{succ}_i(x))$$

Like $\mathsf{IL}\kappa$, the formulae of $\mathsf{WS}\kappa\mathsf{S}$ are interpreted on ordered trees of arity κ. The models of $\mathsf{WS}\kappa\mathsf{S}$ are structures $(\mathfrak{U}, \iota)$, where ι assigns the root of the tree to $\overline{\epsilon}$, a node $\iota(x)$ to each variable $x \in \mathsf{Var}$ and a set $\iota(X) \subseteq \mathfrak{U}$ to each set variable $X \in \mathsf{SVar}$. The satisfaction relation $(\mathfrak{U}, \iota) \models_{\mathsf{WS}\kappa\mathsf{S}} \phi$ is defined as for $\mathsf{IL}\kappa$, with one difference: in $\mathsf{IL}\kappa$, the successor of a leaf of a tree is the root of the tree, while in $\mathsf{WS}\kappa\mathsf{S}$ the successor of leaf is, by convention, the leaf itself [37, Example 2.10.3]. This is the only reason why $\mathsf{IL}\kappa$ is not just a fragment of $\mathsf{WS}\kappa\mathsf{S}$.

We define an embedding of $\mathsf{IL}\kappa$ formulae, without occurrences of predicates and set variables, into $\mathsf{WS}\kappa\mathsf{S}$. W.l.o.g. we consider $\mathsf{IL}\kappa$ formulae that have been previously flattened, i.e the successor function occurs only within atomic propositions of the form $\mathrm{succ}_i(x) = y$. This is done by replacing each atomic proposition of the form $\mathrm{succ}_{i_1}(\ldots \mathrm{succ}_{i_n}(x) \ldots) = y$ by the formula $\exists x_1 \ldots \exists x_n \,.\, x_n = \mathrm{succ}_{i_n}(x) \wedge y = \mathrm{succ}_{i_1}(x_1) \wedge \bigwedge_{j=1}^{n-1} x_j = \mathrm{succ}_{i_j}(x_{j+1})$. The translation of an $\mathsf{IL}\kappa$ formula ϕ into $\mathsf{WS}\kappa\mathsf{S}$ is the formula $\mathit{Tr}(\phi)$, defined recursively on the structure of ϕ such that Tr simply preserves first-order connectives and, secondly, yields:

$$\mathit{Tr}(\mathrm{succ}_i(x) = y) \stackrel{\text{def}}{=} (\neg \max(x) \wedge \mathrm{succ}_i(x) = y) \vee (\max(x) \wedge y = \overline{\epsilon}).$$

We show that a formula ϕ of $\mathsf{IL}\kappa$ and its $\mathsf{WS}\kappa\mathsf{S}$ counterpart $\mathit{Tr}(\phi)$ are equivalent:

Lemma 1. *Given an $\mathsf{IL}\kappa$ formula ϕ, for any structure $\mathcal{I} = (\mathfrak{U}, \iota)$, we have $\mathcal{I} \models_{\mathsf{IL}} \phi \iff \mathcal{I} \models_{\mathsf{WS}\kappa\mathsf{S}} \mathit{Tr}(\phi)$.*

3.2 Defining Parameterized Trap Invariants in $\mathsf{WS}\kappa\mathsf{S}$

Fix a component-based system $\mathcal{S} = \langle C^1, \ldots, C^N, \Gamma \rangle$ and recall that every universe $\mathfrak{U}$ induces a Petri Net $\mathsf{N}_{\mathcal{S}}^{\mathfrak{U}}$ whose set of places is $\bigcup_{k=1}^{N} \mathsf{S}^k \times \mathfrak{U}$. For every state $s \in \bigcup_{i=1}^{N} \mathsf{S}^i$, let X_s be a monadic second-order variable, and let $\overline{X}$ be the tuple of these variables in an arbitrary but fixed order. We define a formula $\mathit{trap\text{-}pred}_{\mathcal{S}}(\overline{X})$, with $\overline{X}$ as set of free variables, that characterizes the traps of the infinitely many Petri Nets $\mathsf{N}_{\mathcal{S}}^{\mathfrak{U}}$ corresponding to $\mathcal{S}$. Formally, $\mathit{trap\text{-}pred}_{\mathcal{S}}(\overline{X})$ has the following property:

For every universe $\mathfrak{U}$ and for every set $P \subseteq \bigcup_{k=1}^{N} \mathsf{S}^k \times \mathfrak{U}$ of places of $\mathsf{N}_{\mathcal{S}}^{\mathfrak{U}}$:
P is a trap of $\mathsf{N}_{\mathcal{S}}^{\mathfrak{U}}$ iff the assignment $X_q \mapsto \{u \in \mathfrak{U} \mid (q, u) \in P\}$ satisfies $\mathit{trap\text{-}pred}_{\mathcal{S}}(\overline{X})$.

Observe that every assignment to $\overline{X}$ encodes a set of places, and vice versa. So, abusing language, we can speak of the set of places $\overline{X}$.

We define auxiliary predicates that capture the intersection of the set of places $\overline{X}$ with the pre (${}^\bullet t$) and postset ($t^\bullet$) of a transition t in $\mathsf{N}_{\mathcal{S}}^{\mathfrak{U}}$. For every clause $\mathfrak{C}$ of Γ, of the form (4), we define the $\mathsf{WS\kappa S}$ formulae:

$$\textit{intersects-pre}^{\mathfrak{C}}_{\mathcal{S}}(\overline{X}, x_1, \ldots, x_\ell) = \bigvee_{j=1}^{\ell} X_{\bullet p_j}(x_j) \vee \bigvee_{j=\ell+1}^{\ell+m} \exists x_j \,.\, Tr(\psi_j) \wedge X_{\bullet p_j}(x_j) \text{ and}$$

$$\textit{intersects-post}^{\mathfrak{C}}_{\mathcal{S}}(\overline{X}, x_1, \ldots, x_\ell) = \bigvee_{j=1}^{\ell} X_{p_j \bullet}(x_j) \vee \bigvee_{j=\ell+1}^{\ell+m} \exists x_j \,.\, Tr(\psi_j) \wedge X_{p_j \bullet}(x_j).$$

Now we can define $\textit{trap-pred}_{\mathcal{S}}(\overline{X})$ as the conjunction of the following formulae, one for each clause $\mathfrak{C}$ (in the form described in (4)) of Γ

$$\begin{aligned} \forall x_1 \ldots \forall x_\ell \,.\, \Big[Tr(\varphi) \wedge \textit{intersects-pre}^{\mathfrak{C}}_{\mathcal{S}}(\overline{X}, x_1, \ldots, x_\ell) \Big] \\ \rightarrow \textit{intersects-post}^{\mathfrak{C}}_{\mathcal{S}}(\overline{X}, x_1, \ldots, x_\ell). \end{aligned} \tag{5}$$

So, intuitively, $\textit{trap-pred}_{\mathcal{S}}(\overline{X})$ states that for every transition of the Petri Net, if the set $\overline{X}$ of places intersects the preset of the transition, then it also intersects its postset. This is the condition for the set of places to be a trap. Formally, we obtain:

Lemma 2. *Given a component-based system* $\mathcal{S} = \langle C^1, \ldots, C^N, \Gamma \rangle$ *and a structure* $\mathcal{I} = (\mathfrak{U}, \iota)$*, where* ι *is an interpretation of the set variables* $\overline{X}$*, the set* $P = \{\langle s, u \rangle \in \bigcup_{k=1}^{N} \mathsf{S}^k \times \mathfrak{U} \mid u \in \iota(X_s)\}$ *is a trap of* $\mathsf{N}^{\mathfrak{U}}_{\mathcal{S}}$ *if and only if* $(\mathfrak{U}, \iota) \models_{\mathsf{WS\kappa S}} \textit{trap-pred}_{\mathcal{S}}(\overline{X})$*.*

Parameterized Trap Invariants in $\mathsf{WS\kappa S}$. Loosely speaking, the intended meaning of $\textit{trap-pred}_{\mathcal{S}}(\overline{X})$ is "the set of places $\overline{X}$ is a trap". Our goal is to construct a formula stating: "the marking m marks all initially marked traps".

Recall that the Petri Nets obtained from component-based systems are always 1-safe, and so a marking is also a set of places. Recall, however, that all reachable markings have the property that they place exactly one token in the set of places modeling the set of states of a component (loosely speaking, the set of places of the k-th philosopher is (w, k) and (e, k), and there is always one token in the one or the other). So we define a formula $\textit{marking}_{\mathcal{S}}(\overline{X})$ with intended meaning "the set of places $\overline{X}$ is a legal marking", and another one, $\textit{trap-invariant}_{\mathcal{S}}(\overline{X})$ with intended meaning "the set of places $\overline{X}$ marks every initially marked trap".

In addition to the tuple of set variables $\overline{X}$ defined above, we consider now the "copy" tuple $\overline{X'} \overset{\mathsf{def}}{=} \langle X'_s \rangle_{s \in \mathsf{S}^i, 1 \leq i \leq N}$. Intuitively, $\overline{X}$ and $\overline{X'}$ represent one set of places each. First, we define a (1-safe) marking as a set of places that marks exactly one state of each copy of each component:

$$\textit{marking}_{\mathcal{S}}(\overline{X}) = \forall x \,.\, \bigwedge_{1 \leq i \leq N} \bigvee_{s \in S^i} \left(X_s(x) \wedge \bigwedge_{s' \in S^i \setminus \{s\}} \neg X_{s'}(x) \right).$$

Second, we give a formula describing the intersection of two sets of places:

$$\textit{intersection}_{\mathcal{S}}(\overline{X}, \overline{X'}) = \exists x \,.\, \bigvee_{s \in \bigcup_{1 \leq i \leq N} \mathsf{S}^i} (X_s(x) \wedge X'_s(x)).$$

Finally, to actually capture IMTs we need to determine if a trap is initially marked. However, this can be easily described by the formula:

$$\textit{initially-marked}_{\mathcal{S}}(\overline{X}) = \exists x \,.\, \bigvee_{1 \leq i \leq N} X_{s_0{}^i}(x).$$

So we can define the *trap-invariant* by the $\mathsf{WS\kappa S}$ formula:

$$\begin{aligned} \textit{trap-invariant}_{\mathcal{S}}(\overline{X}) = \forall \overline{X'} \,.\, \Big[\textit{trap-pred}_{\mathcal{S}}(\overline{X'}) \wedge \textit{initially-marked}_{\mathcal{S}}(\overline{X'}) \Big] \\ \rightarrow \textit{intersection}_{\mathcal{S}}(\overline{X}, \overline{X'}). \end{aligned} \tag{6}$$

Relying on Lemma 2 we are assured that the set represented by $\overline{X}$ intersects all IMTs. Now, let $\varphi(\overline{X})$ be any formula that defines a set of *good* global states of the component-based systems (or, equivalently, a good set of markings of their corresponding Petri nets), with the intuition that, at any moment during execution, the current global state of the component-based system should be good. We can now state the following theorem, that captures the soundness of the verification method based on trap invariants:

Theorem 1. *Given a component-based system $\mathcal{S}$ and a* WSκS *formula $\varphi(\overline{X})$, if the formula*

$$\exists \overline{X} \; . \; marking_{\mathcal{S}}(\overline{X}) \wedge trap\text{-}invariant_{\mathcal{S}}(\overline{X}) \wedge \neg\varphi(\overline{X}) \tag{7}$$

is unsatisfiable, then for every universe $\mathfrak{U}$, the property defined by the formula $\varphi(\overline{X})$ holds in every reachable marking of $\mathcal{N}_{\mathcal{S}}^{\mathfrak{U}}$.

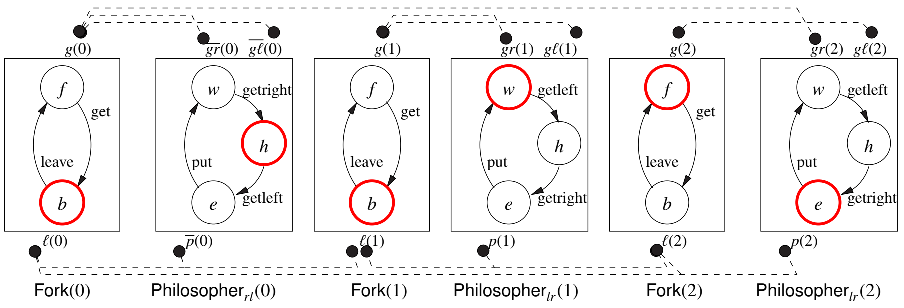
In the light of the above theorem, verifying the correctness of a component-based system with any number of active components boils down to deciding the satisfiability of a WSκS formula. The latter problem is known to be decidable, albeit with non-elementary worst-case complexity. A closer look at the verification conditions of the form (7) generated by our method suffices to see that the quantifier alternation is finite, which implies that the time needed to decide the (un)satisfiability of (7) is elementary. Moreover, our experiments show that these checks are very fast (less than 1 second on an average machine) for a non-trivial set of examples.

4 Refining Trap Invariants

Since the safety verification problem is undecidable for parameterized systems [6], the verification method based on trap invariants cannot be complete. As an example, consider the alternating dining philosophers system, of which an instance (for $n = 3$) is shown in Fig. 3. The system consists of two philosopher component types, namely Philosopher$_{rl}$, which takes its right fork before its left fork, and Philosopher$_{lr}$, taking the left fork before the right one. Each philosopher has two interaction ports for taking the forks, namely $g\ell$ (get left) and gr (get right) and one port for releasing the forks p (put). The ports of the Philosopher$_{rl}$ component type are overlined, in order to be distinguished. The Fork component type is the same as in Fig. 1. The interaction formula for this system Γ_{philo}^{alt}, shown in Fig. 3, implicitly states that only the 0-index philosopher component is of type Philosopher$_{rl}$, whereas all other philosophers are of type Philosopher$_{lr}$. Note that the interactions on ports $\overline{g\ell}$, $\overline{gr}$ and $\overline{p}$ are only allowed if $zero(x) \overset{\text{def}}{=} \forall y \; . \; x \leq y$ holds, in other words if x is interpreted as the root of the universe (in our case, 0 since $\mathfrak{U} = \{0, \ldots, n-1\}$).

It is well-known that any instance of the parameterized alternating dining philosophers system consisting of at least one Philosopher$_{rl}$ and one Philosopher$_{lr}$ is deadlock-free. However, trap invariants are not enough to prove deadlock freedom, as shown by the global state $\{\langle b,0\rangle, \langle h,0\rangle, \langle b,1\rangle, \langle w,1\rangle, \langle f,2\rangle, \langle e,2\rangle\}$, marked with thick red lines in Fig. 3. Note that no interaction is enabled in this state. Moreover, this state intersects with any trap of the marked PN that defines the executions of this particular instance, as proved below. Consequently, the trap invariant contains a deadlock configuration, and the system cannot be proved deadlock-free by this method.

Fig. 3: Alternating Dining Philosophers



$$\Gamma^{alt}_{philo} = zero(x) \wedge [(\overline{g\ell}(x) \wedge g(x)) \vee (\overline{gr}(x) \wedge g(s(x))) \vee (\overline{p}(x) \wedge \ell(x) \wedge \ell(s(x)))] \vee$$
$$\neg zero(x) \wedge [(g\ell(x) \wedge g(x)) \vee (gr(x) \wedge g(s(x))) \vee (p(x) \wedge \ell(x) \wedge \ell(s(x)))]$$

Proposition 1. *Consider an instance of the alternating dining philosophers system in Fig. 3, consisting of components* Fork(0)*,* Philosopher$_{rl}$(0)*,* Fork(1)*,* Philosopher$_{lr}$(1)*,* Fork(2) *and* Philosopher$_{lr}$(2) *placed in a ring, in this order. Then each nonempty trap of this system contains one of the places* $\langle b,0\rangle, \langle h,0\rangle, \langle b,1\rangle, \langle w,1\rangle, \langle f,2\rangle$ *or* $\langle e,2\rangle$.

However, the configuration is unreachable by a real execution of the PN, started in the initial configuration that marks $\langle f,i\rangle$ and $\langle w,i\rangle$, for all $i = 0,1,2$. An intuitive reason is that, in any reachable configuration, each fork is in state f(ree) only if none of its neighboring philosophers is in state e(ating). In order to prove deadlock freedom, one must learn this and other similar constraints. Next, we present a heuristic method for strengthening the trap invariant that infers such universal constraints.

4.1 One Invariants

As shown by the example above, trap constraints do sometimes fail to prove interesting properties. Hence, it is desirable to refine the overapproximation of viable markings to exclude more spurious counterexamples. In order to do so, we consider a special class of *linear invariants*, called 1*-invariants* in the following. Although linear invariants are not structural and rely on the set of reachable markings of a marked Petri Net, the set of 1-invariants can be sufficiently under-approximated by structural conditions.

Definition 1. *Given a marked PN* $\mathcal{N} = ((S,T,E),\mathrm{m}_0)$*, with* $S = \{s_1,\ldots,s_n\}$*, a vector* $\mathbf{a} = (a_1,\ldots,a_n) \in \{0,1\}^n$ *is a* 1-invariant *of* $\mathcal{N}$ *if and only if, for each reachable marking* $\mathrm{m} \in \mathcal{R}(\mathcal{N})$*, we have* $\sum_{i=1}^{n} a_i \cdot \mathrm{m}(s_i) = 1$.

The following lemma relates 1-invariants to some structural properties. However, there are 1-invariants not captured by these conditions. Taking the intersection of this set of 1-invariants defines a weaker invariant, which is sound for our verification purposes.

Lemma 3. *Given a marked PN* $\mathcal{N} = ((\mathsf{S},T,E),\mathrm{m}_0)$*, a set of places* $\mathfrak{F} \subseteq \mathsf{S}$ *is a* 1*-invariant if the following hold:*

1. $\sum_{s\in\mathfrak{F}} \mathrm{m}_0(s) = 1$,
2. *either* $\|\mathfrak{F} \cap {}^\bullet t\| = \|\mathfrak{F} \cap t^\bullet\| = k$ *with* $k \in \{0,1\}$ *or* $\|\mathfrak{F} \cap {}^\bullet t\| > 1$ *for every* $t \in T$.

We devote the rest of this section to describe WSκS formulae which capture the structural properties necessary to define 1-invariants as laid down by Lemma 3 (2). As demonstrated in Section 3 the pre- and postset of transitions, as well as general sets of places in a PN describing the execution semantics can be defined in WSκS. Hence, we present the definitions of the following formulae only in the full version of this article [16] and just give the intuitions here.

As before, we fix two tuples of set variables $\overline{X}$ and $\overline{X'}$, with one variable X_s for each state $s \in \bigcup_{i=1}^{N} \mathsf{S}^i$ and define the following formulae:

- $\textit{unique-init}_{\mathcal{S}}(\overline{X})$, which captures that the set of places induced by an interpretation of $\overline{X}$ uniquely intersects the set of all initial states, and
- $\textit{unique-intersection}_{\mathcal{S}}(\overline{X},\overline{X'})$, which states that the set of places induced by an interpretation of $\overline{X}$ and $\overline{X'}$ share precisely one place.

Given a transition t of the marked Petri Net $\mathcal{N}_{\mathcal{S}}^{\mathfrak{U}}$ defining the execution semantics of a component-based system $\mathcal{S}$, for a universe $\mathfrak{U}$, we consider the following formulae:

- $\textit{uniquepre}_{\mathcal{S}}^{\mathfrak{C}}(\overline{X},x_1,\ldots,x_\ell)$, which describes that the set of places encoded by the interpretation of $\overline{X}$ uniquely intersects ${}^\bullet t$, and
- $\textit{uniquepost}_{\mathcal{S}}^{\mathfrak{C}}(\overline{X},x_1,\ldots,x_\ell)$, which in the same sense captures the unique intersection with $t^\bullet$.

Now we define a predicate $\textit{1-pred}_{\mathcal{S}}$ which consists of a conjunction of $\textit{unique-init}_{\mathcal{S}}$ and the formulae:

$$\begin{aligned}\forall x_1,\ldots,\forall x_\ell\ .\ (Tr(\varphi) \rightarrow [\neg\,\textit{intersects-pre}_{\mathcal{S}}^{\mathfrak{C}} \wedge \neg\,\textit{intersects-post}_{\mathcal{S}}^{\mathfrak{C}} \\ \vee\,\textit{uniquepre}_{\mathcal{S}}^{\mathfrak{C}} \wedge \textit{uniquepost}_{\mathcal{S}}^{\mathfrak{C}} \\ \vee\,\textit{intersects-pre}_{\mathcal{S}}^{\mathfrak{C}} \wedge \neg\,\textit{uniquepre}_{\mathcal{S}}^{\mathfrak{C}}])\end{aligned} \tag{8}$$

one for each clause $\mathfrak{C}$ in Γ. We show the soundness of this definition, by the following:

Lemma 4. *Let $\mathcal{S} = \langle C^1,\ldots,C^N,\Gamma\rangle$ be a component-based system and let $\overline{X}$ be a tuple of set variables, one for each state in a component of $\mathcal{S}$. Then, for any structure $(\mathfrak{U},\iota)$ such that ι interprets the variables in $\overline{X}$, the set $P = \{\langle s,u\rangle \in \bigcup_{i=1}^{N} \mathsf{S}^i \times \mathfrak{U} \mid u \in \iota(X_s)\}$ is a 1-invariant of $\mathcal{N}_{\mathcal{S}}^{\mathfrak{U}}$ if $(\mathfrak{U},\iota) \models_{\mathsf{WS\kappa S}} \textit{1-pred}_{\mathcal{S}}(\overline{X})$.*

We may now define the 1-*invariant* analogously to the trap-invariant before:

$$\textit{1-invariant}_{\mathcal{S}}(\overline{X}) = \forall\overline{X'}\ .\ \textit{1-pred}_{\mathcal{S}}(\overline{X'}) \rightarrow \textit{unique-intersection}_{\mathcal{S}}(\overline{X},\overline{X'}). \tag{9}$$

Reasoning as before we obtain a refinement of Theorem 1 since every reachable marking has to satisfy both invariants.

Theorem 2. *Given a component-based system $\mathcal{S}$ and a WSκS formula $\varphi(\overline{X})$, if the formula:*

$$\exists\overline{X}\ .\ \textit{marking}_{\mathcal{S}}(\overline{X}) \wedge \textit{1-invariant}_{\mathcal{S}}(\overline{X}) \wedge \textit{trap-invariant}_{\mathcal{S}}(\overline{X}) \wedge \neg\varphi(\overline{X}) \tag{10}$$

is unsatisfiable, then for every universe $\mathfrak{U}$, the property defined by the formula $\varphi(\overline{X})$ holds in every reachable marking of $\mathcal{N}_{\mathcal{S}}^{\mathfrak{U}}$.

5 Experiments

We have implemented a prototype (called `ostrich` [15]) of this verification procedure to evaluate the viability of our approach. The current version of the prototype can only handle token-ring and pipeline topologies, but not trees; for these topologies the verification reduces to checking satisfiability of a formula of $\mathsf{WS1S}$. We have also considered one example with tree-topology (see below), for which the formula was constructed manually. Satisfiability of $\mathsf{WS1S}$ and $\mathsf{WS}\kappa\mathsf{S}$ formulae was checked using version 1.4/17 of Mona [33]. We consider various examples separated in categories:

Cache Coherence. Following [24] we formalized and checked the described safety properties and deadlock-freedom of the following cache coherence protocols: Illinois, Berkeley, Synapse, Firefly, MESI, MOESI, and Dragon.

Mutual Exclusion. We modelled and checked for deadlock-freedom and mutual exclusion Burns' [35], Dijkstra's and Szymanski's [3] algorithms as well as a formulation of Dijkstra's algorithm on a ring structure with token passing [30]. Furthermore, we check synchronization via a semaphore which is atomically aquired and by broadcasting to ensure everyone else is not in the critical section.

Dining Philosophers. This is the classical problem of dining philosophers which all take first the right fork and then the left fork. We consider the following "flavors" of this problem:
- there is one philosopher who takes first her left and then her right fork,
- as above but the forks remember whom took them, and
- there are two global forks everyone grabs in the same order.

Preemptive Tasks. There are tasks which can be either waiting, ready, executing or preempted. Initially one task is executing while all others are waiting. At any point a task may become ready and any ready task may preempt the currently executing task. Upon finishing the executing task re-enables one preempted task. Here, we have additionally two alternatives: Firstly, we consider the case where always the agent with highest index resumes execution. Secondly, we let the processes establish the initial condition from a position where everyone is waiting (referenced later as *uninitialized*).

Dijkstra-Scholten. This is an algorithm that is used to detect termination of distributed systems by message passing along a tree [25]. Since the prototype only supports linear topologies we can generate the necessary formula automatically only for this case.

Herman. This algorithm implements self-stabilizing token passing in rings. The formulation is modelled after [19]. This applies for all following examples. Hence, we describe the examples only in little detail.

Israeli-Jalfon. This is another self-stabilizing token passing algorithm in rings.

Lehmann-Rabin. This is a randomized solution to the dining philosophers problem.

Dining Cryptographers. A group of cryptographers want to determine if one of them paid for a meal or a stranger but do not reveal how they acted individually.

The results are shown in Table 1. The first column reports the size of the example in terms of the amount of states (#st.) and clauses (#cls.). The second column indicates which properties could (✓) and could not be verified (×) because the conjunction of trap and one-invariant was not strong enough to prove the given property. The third column reports the time (in second) it takes to prove all considered properties. These results are measured on the provided virtual machine for artifacts [32] where the host system is an average laptop. To understand the next four columns, recall that Mona constructs for a given formula $\phi(X)$ a finite automaton recognizing all the sets X for which φ holds. Since the automaton can have very large alphabets, its transition relation is encoded as a binary decision diagram (BDD). The columns report the number of states and the number of nodes of the BDD for different formulas. More precisely, the columns trap, trap-inv, flow, and flow-inv give the sizes of the automata for the formula $\mathit{trap\text{-}pred}_{\mathcal{S}} \wedge \mathit{initially\text{-}marked}_{\mathcal{S}}$, $\mathit{trap\text{-}invariant}_{\mathcal{S}}$, $\mathit{1\text{-}pred}_{\mathcal{S}}$ and $\mathit{1\text{-}invariant}_{\mathcal{S}}$ respectively. We write "n.a." (for "not available") to indicate that Mona timed out before the automaton was computed.

The first observation is that the satisfiability checks often can be done in very short time. This is surprising, because the formulas to be checked, namely (7) and (10), exhibit one quantifier alternation (recall that $\mathit{trap\text{-}invariant}_{\mathcal{S}}$ and $\mathit{1\text{-}invariant}_{\mathcal{S}}$ contain universal quantifiers). More specifically, since $\mathit{trap\text{-}invariant}_{\mathcal{S}}$ is obtained by universally quantifying over $\mathit{trap\text{-}pred}_{\mathcal{S}} \wedge \mathit{initially\text{-}marked}_{\mathcal{S}}$, one would expect the automaton for the former to be much larger than the one for the latter, at least in some cases. But this does not happen: In fact, the automaton for $\mathit{trap\text{-}invariant}_{\mathcal{S}}$ is almost always smaller. Similarly, there is no blowup from $\mathit{1\text{-}pred}_{\mathcal{S}}$ to $\mathit{1\text{-}invariant}_{\mathcal{S}}$. A possible explanation could be that the exponential blowup caused by universal quantification in $\mathsf{WS}\kappa\mathsf{S}$ manifests only on theoretical corner cases, which do not occur in our examples.

6 Conclusions

We have shown that the trap technique used in [11,28,13] for the verification of single systems can be extended to parameterized systems with sophisticated communication structures, like pipelines, token rings and trees. Our extension constructs a parameterized trap invariant, a formula of $\mathsf{WS}\kappa\mathsf{S}$ satisfied by the reachable global states of all instances of the system. The core of the approach is a purely syntactic, automatic derivation of the trap invariant from the interaction formula describing the possible transitions of the system. When the set of safe global states can also be expressed in $\mathsf{WS}\kappa\mathsf{S}$, which is usually the case, we check using the Mona tool whether the trap invariant implies the safety formula. The technique proves correctness of systems that do not produce well-structured transition systems in the sense of [1,29], and of systems with broadcast communication, for which, to the best of our knowledge, cut-off results have not been obtained yet.

Our experiments demonstrate that trap invariants can be very effective in finding proofs of correctness (inductive invariants) of common benchmark examples. In practice, the technique is very cheap, since it avoids costly fixpoint computations. This suggests incorporating it into other verifiers as a preprocessing step.

Data Availability Statement and Acknowledgements. The work of the second and fifth author has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367 (PaVeS).
The tool `ostrich` and associated files are available in the Zenodo repository: https://zenodo.org/record/3676940

Benchmark	size #st. / #cls.	Properties		time (s.)	trap #st. / #tr.	trap-inv #st. / #tr.	flow #st. / #tr.	flow-inv #st. / #tr.
Berkeley	4 / 8	deadlock-freedom consistency properties	✓ ✓	0.37	18 / 95	18 / 78	12 / 50	7 / 18
Dragon	5 / 24	deadlock-freedom consistency properties	✓ ✓	1.63	39 / 433	32 / 159	54 / 537	11 / 57
Firefly	5 / 13	deadlock-freedom consistency properties	✓ ✓	0.53	55 / 409	36 / 200	38 / 309	11 / 44
Illinois	4 / 13	deadlock-freedom consistency properties	✓ ✓	0.46	14 / 83	11 / 32	16 / 95	9 / 38
MESI	4 / 8	deadlock-freedom consistency properties	✓ ✓	0.36	12 / 62	11 / 32	12 / 49	7 / 18
MOESI	5 / 10	deadlock-freedom consistency properties	✓ ✓	0.56	20 / 150	16 / 64	12 / 57	7 / 20
Synapse	3 / 6	deadlock-freedom consistency properties	✓ ✓	0.32	12 / 44	11 / 30	12 / 42	7 / 16
Dijkstra-Scholten	4 / 6	deadlock-freedom	✓	0.25	13 / 48	11 / 31	10 / 35	9 / 31
Bakery	3 / 4	deadlock-freedom mutual exclusion	✓ ✓	0.26	10 / 27	10 / 24	8 / 23	7 / 20
Burns	6 / 12	deadlock-freedom mutual exclusion	✓ ✓	0.30	10 / 64	9 / 30	8 / 35	7 / 32
Dijkstra	12 / 14	deadlock-freedom mutual exclusion	✓ ✓	11.86	375 / 11840	106 / 2887	13 / 148	10 / 110
Broadcast MutEx	2 / 2	deadlock-freedom mutual exclusion	✓ ✓	0.23	9 / 22	9 / 21	8 / 19	7 / 16
Preemptive (high)	5 / 4	deadlock-freedom mutual exclusion	✓ ✓	0.26	41 / 279	17 / 71	16 / 88	10 / 46
Preemptive	5 / 4	deadlock-freedom mutual exclusion	✓ ✓	0.25	28 / 173	22 / 97	20 / 113	12 / 62
Preemptive (uninitialized)	3 / 3	deadlock-freedom mutual exclusion	✓ ×	0.25	20 / 80	11 / 37	8 / 23	7 / 20
Semaphore	4 / 2	deadlock-freedom mutual exclusion	✓ ✓	0.23	14 / 39	9 / 32	10 / 36	8 / 30
Szymanski	15 / 31	deadlock-freedom mutual exclusion	n.a. n.a.	19.35	495 / 34990	n.a.	8 / 84	7 / 66
Dijkstra (ring)	10 / 9	deadlock-freedom mutual exclusion	✓ ✓	75.63	703 / 10160	1149 / 23195	20 / 247	20 / 387
Dining Cryptographers	7 / 15	deadlock-freedom correctness	× ✓	1.54	250 / 3232	288 / 2484	10 / 72	9 / 60
Dining Philosophers (global)	5 / 3	deadlock-freedom	✓	0.23	32 / 145	19 / 98	15 / 77	11 / 56
Herman (linear)	3 / 3	deadlock-freedom no token loss	× ✓	0.25	19 / 70	14 / 42	10 / 33	9 / 27
Herman (ring)	3 / 4	deadlock-freedom no token loss	✓ ✓	0.26	19 / 71	14 / 42	10 / 33	9 / 27
Israeli-Jalfon	3 / 5	deadlock-freedom no token loss	✓ ✓	0.25	43 / 187	14 / 42	8 / 23	7 / 20
Dining Philosophers (lefty)	5 / 5	deadlock-freedom	✓	0.25	37 / 219	27 / 167	12 / 63	11 / 57
Dining Philosophers (lefty, rem. forks)	6 / 5	deadlock-freedom	✓	0.26	37 / 270	21 / 119	14 / 101	11 / 66
Lehmann-Rabin	6 / 7	deadlock-freedom	✓	0.26	39 / 361	23 / 211	11 / 67	11 / 73

Table 1: Experimental results of `ostrich`.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: LICS. pp. 313–321. IEEE Computer Society (1996)
2. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 721–736 (2007)
3. Abdulla, P.A., Haziza, F., Holík, L.: Parameterized verification through view abstraction. STTT 18(5), 495–516 (2016)
4. Alberti, F., Ghilardi, S., Sharygina, N.: A framework for the verification of parameterized infinite-state systems. CEUR Workshop Proceedings 1195, 302–308 (01 2014)
5. Aminof, B., Kotek, T., Rubin, S., Spegni, F., Veith, H.: Parameterized model checking of rendezvous systems. Distributed Computing 31(3), 187–222 (Jun 2018)
6. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters 22(6), 307 – 309 (1986)
7. Außerlechner, S., Jacobs, S., Khalimov, A.: Tight cutoffs for guarded protocols with fairness. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 476–494. Springer (2016)
8. Barkaoui, K., Lemaire, B.: An effective characterization of minimal deadlocks and traps in Petri nets based on graph theory. In: 10th Int. Conf. on Application and Theory of Petri Nets ICATPN'89. pp. 1–21 (1989)
9. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
10. Baukus, K., Bensalem, S., Lakhnech, Y., Stahl, K.: Abstracting WS1S systems to verify parameterized networks. In: Graf, S., Schwartzbach, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 188–203 (2000)
11. Bensalem, S., Bozga, M., Nguyen, T., Sifakis, J.: D-Finder: A tool for compositional deadlock detection and verification. In: CAV'09 Proceedings. LNCS, vol. 5643, pp. 614–619 (2009)
12. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015)
13. Blondin, M., Finkel, A., Haase, C., Haddad, S.: Approaching the coverability problem continuously. In: TACAS. Lecture Notes in Computer Science, vol. 9636, pp. 480–496. Springer (2016)
14. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification. pp. 372–386 (2004)
15. Bozga, M., Esparza, J., Iosif, R., Sifakis, J., Welzel, C.: ostrich (Feb 2020), https://doi.org/10.5281/zenodo.3676940
16. Bozga, M., Esparza, J., Iosif, R., Sifakis, J., Welzel, C.: Structural invariants for the verification of systems with parameterized architectures (2020)
17. Bozga, M., Iosif, R., Sifakis, J.: Checking deadlock-freedom of parametric component-based systems. In: 25th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2019)
18. Browne, M., Clarke, E., Grumberg, O.: Reasoning about networks with many identical finite state processes. Information and Computation 81(1), 13 – 31 (1989)
19. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 76–83 (2017)

20. Clarke, E., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 126–141 (2006)
21. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification. pp. 718–724 (2012)
22. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 269–282. ACM Press, New York, NY, San Antonio, Texas (1979)
23. Dams, D., Lakhnech, Y., Steffen, M.: Iterating transducers. The Journal of Logic and Algebraic Programming 52-53, 109 – 127 (2002)
24. Delzanno, G.: Automatic verification of parameterized cache coherence protocols. In: Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. pp. 53–68 (2000)
25. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Inf. Process. Lett. 11(1), 1–4 (1980)
26. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: CADE. Lecture Notes in Computer Science, vol. 1831, pp. 236–254. Springer (2000)
27. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL'95 Proceedings. pp. 85–94 (1995)
28. Esparza, J., Ledesma-Garza, R., Majumdar, R., Meyer, P.J., Niksic, F.: An smt-based approach to coverability analysis. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 603–619. Springer (2014)
29. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
30. Fribourg, L., Olsén, H.: Reachability sets of parameterized rings as regular languages. Electr. Notes Theor. Comput. Sci. 9, 40 (1997)
31. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
32. Hartmanns, A., Seidl, M.: tacas20ae.ova (10 2019), https://figshare.com/articles/tacas20ae_ova/9699839
33. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019 (1995)
34. Jacobs, S., Sakr, M.: Analyzing guarded protocols: Better cutoffs, more systems, more expressivity. In: VMCAI. Lecture Notes in Computer Science, vol. 10747, pp. 247–268. Springer (2018)
35. Jensen, H.E., Lynch, N.A.: A proof of Burns n-process mutual exclusion algorithm using abstraction. In: TACAS (1998)
36. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theoretical Computer Science 256(1), 93 – 112 (2001)
37. Khoussainov, B., Nerode, A.: Automata Theory and Its Applications. Birkhauser Boston, Inc. (2001)
38. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 82–97 (2001)
39. Sifakis, J.: Structural properties of petri nets. In: Winkowski, J. (ed.) Mathematical Foundations of Computer Science 1978. pp. 474–483 (1978)

Automated Verification of Parallel Nested DFS

Wytse Oortwijn[1]⋆, Marieke Huisman[2], Sebastiaan J. C. Joosten[3]⋆, and Jaco van de Pol[2,4]

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland
wytse.oortwijn@inf.ethz.ch

[2] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
m.huisman@utwente.nl

[3] Dartmouth College, Hanover NH, USA
sebastiaan.joosten@dartmouth.edu

[4] Department of Computer Science, Aarhus University, Aarhus, Denmark
jaco@cs.au.dk

Abstract. Model checking algorithms are typically complex graph algorithms, whose correctness is crucial for the usability of a model checker. However, establishing the correctness of such algorithms can be challenging and is often done manually. Mechanising the verification process is crucially important, because model checking algorithms are often parallelised for efficiency reasons, which makes them even more error-prone.

This paper shows how the VerCors concurrency verifier is used to mechanically verify the parallel nested depth-first search (NDFS) graph algorithm of Laarman et al. [25]. We also demonstrate how having a mechanised proof supports the easy verification of various optimisations of parallel NDFS. As far as we are aware, this is the first automated deductive verification of a multi-core model checking algorithm.

1 Introduction

Model checking is an automated procedure for verifying behavioural properties of reactive systems. To avoid a false sense of safety, it is essential that model checkers are themselves correct. However, model checkers use ever more ingenious algorithms [12] and even parallel implementations [2] to be able to combat the large state spaces of critical industrial systems, which makes it increasingly difficult to guarantee their correctness.

This paper focusses on the mechanical verification of a *multi-core* model checking algorithm for detecting accepting cycles in automata, called nested depth-first search (NDFS). This algorithm solves the model checking problem for Linear-time Temporal Logic (LTL), a widely used logic for specifying reactive systems. Multi-core NDFS is developed by Laarman et al. in 2011 [25] and is currently deployed in the high-performance model checker LTSmin [23].

The mechanical verification of parallel NDFS is carried out in VerCors [6], a verifier based on concurrent separation logic that targets real-world concurrent

⋆ This research has been performed while working at the University of Twente.

© The Author(s) 2020
A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 247–265, 2020.
https://doi.org/10.1007/978-3-030-45190-5_14

and parallel programs. The presented verification is inspired by a previous mechanical verification of *sequential* NDFS [37] that was carried out in Dafny [30].

This paper demonstrates the feasibility of mechanical program verification of parallel graph algorithms, like multi-core NDFS. To the best of our knowledge we present the first mechanical verification of a parallel graph algorithm. Our formalisation provides reusable components that can be used to verify variations of parallel NDFS, as well as other algorithms for parallel model checking.

Before listing our contributions (§1.3) we first provide more background on model checking algorithms (§1.1) and related work on their verification (§1.2).

1.1 Background on Model Checking

Pnueli introduced the Linear-time Temporal Logic (LTL) [36] to specify properties of reactive systems. The model checking problem [12] decides whether a transition system satisfies a given LTL property. The automata-based approach [45] reduces the model checking problem to the graph-theoretic problem of checking the reachability of accepting cycles. Reachability of accepting cycles in directed graphs can be checked in linear time, with the nested depth-first search (NDFS) algorithm [13,19,41], which forms the basis of the Spin model checker [17].

Several distributed and parallel model checking algorithms have been proposed, to allocate more memory and processors to the problem [2]. NDFS is based on depth-first search, which is considered hard (impossible) to parallelise efficiently [39]. For distributed approaches, the best strategy is to turn to BFS algorithms [3], which are straightforward to parallelise but at the cost of increasing the amount of work beyond linear time. For the shared-memory setting, swarm verification was proposed [18], where each worker runs its own instance of NDFS. Various DFS-based multi-core algorithms for full LTL model checking have been devised for this strategy [14,15,25]. This paper considers the version by Laarman et al. [25], which is a parallel version of improved sequential NDFS [41].

The correctness of parallel NDFS is quite subtle. In particular, parallel DFS does not fully respect a global depth-first ordering, since each worker maintains its own search stack, yet the correctness of NDFS depends on the search order. Also, to realise speedups, the implementation avoids locking shared data structures by using atomics. This raises the question whether the implementation of a parallel model checker, meant to verify the correctness of safety-critical systems, is itself correct. For this reason the original paper [25] contains a detailed *pen-and-paper* correctness proof, which is based on a number of invariants.

1.2 Related Work

To raise the level of confidence in model checkers, one approach is to certify each of their individual runs. Obviously, the counterexample returned by a model checker is itself a certificate that can easily be verified independently. However, double-checking the absence of errors is harder. Namjoshi [33] proposed to instrument a μ-calculus model checker, to generate a deductive proof that can be checked independently, also in case the property holds. Recently, an IC3-style symbolic LTL model checker has been extended with deductive proofs as well [16]. However, these approaches do not prove correctness of the model checking algorithm, but only validate its outcome for each specific use.

Alternatively, one can formalise the model checking algorithm and its correctness proof in an interactive theorem prover. An early example of this approach was the verification of a model checker for the modal μ-calculus in Coq [43]. A framework for verifying sequential depth-first search algorithms was developed in Isabelle [27,28], and applied to the verification of NDFS with partial order reduction [9] as well as a model checker for timed automata [47]. The recent formalisations of Tarjan's SCC algorithm [10] fit in the same line of research. These approaches require to model and verify the algorithm in an interactive theorem prover, allowing one to use the full power of the theorem prover.

If one wishes to verify the code of the algorithm directly, yet another approach is to model the algorithm and its specification in a (semi-)automated program verifier, where the code is enriched with sufficient annotations to prove its correctness. This approach was followed for several standard sequential graph algorithms in Why3 [46] and for sequential NDFS in Dafny [37]. However, there is hardly any work on automated verification of parallel graph algorithms. Raad et al. [38] verified four concurrent graph algorithms in the context of CoLoSL, but the proofs have not been automated. Sergey et al. [42] verified a concurrent spanning tree algorithm, but interactively, through an embedding in Coq.

To support the verification of shared-memory parallel software, program verifiers typically use concurrent separation logic. VeriFast [20] aims at sequential and multi-threaded C and Java programs. VerCors [6] verifies concurrent programs in Java and OpenCL, by applying a correctness-preserving translation into a sequential imperative language, delegating the generation of the verification conditions to Viper [32] and their verification ultimately to Z3 [31].

1.3 Contributions and Outline

This paper discusses the mechanical verification of the *parallel* NDFS algorithm of Laarman et al. [25] using VerCors. To the best of our knowledge, this is the first mechanical verification of a parallel graph (and model checking) algorithm.

Section 2 recalls both sequential and parallel NDFS (§2.1–2.2), and gives preliminaries on concurrency verification with VerCors (§2.3). It also explains that parallel NDFS uses various colour markings on the input graph to administer the status of the nested searches of workers. Some of these colours are local to a single worker, while other colours are globally shared among all workers.

Section 3.1 presents our new (informal) correctness proof of parallel NDFS, that is based on a number of global invariants on the possible colour configurations. The main challenge lies in proving completeness, which is particularly difficult since workers can delegate the detection of accepting cycles to other workers. To be able to mechanise our completeness proof, we contribute a new invariant (Lemma 4) that guarantees the preservation of so-called *special paths*. This allows to circumvent using the complicated inductive argument used by [25].

Section 3.2 discusses how parallel NDFS is specified in VerCors. In particular, this requires the specification of permissions, to verify data race-free access to shared data structures. Moreover, we encode the colour maps and the transition relation of the input automaton as matrices, which greatly contribute to the feasibility of proof checking. We also explain how atomic updates are specified, which was left implicit in the high-level pseudo code. Similarly, we implement asymmetric termination detection: if one worker finds a counterexample, all workers can terminate immediately; if, on the other hand, all workers have completely finished their exploration, only then may one conclude that the model is correct.

Section 3.3 explains the techniques to formalise the full functional correctness proof in VerCors. In particular, this requires the distribution of permissions and invariants over threads and locks, and the introduction of auxiliary ghost state to track the precise progress of the various nested search phases of all workers.

Section 4 demonstrates how our verification is reused to verify optimisations to the algorithm. In particular, we check the optimisation "early cycle detection" that, for weak LTL properties, detects all cycles in the outer search instead of the nested inner search. We also propose and verify a repair to the "all-red" extension, by inserting an extra check that was missing in [25]. This extension improves the speedup of parallel NDFS by sharing more global information.

Finally, Section 5 concludes with a perspective on reusing our techniques for verifying other parallel graph algorithms.

2 Preliminaries

Section 2.1 recalls the standard sequential NDFS algorithm for finding reachable accepting cycles in automata. We verified a parallel version of NDFS, which is introduced in Section 2.2. The verification has been performed with VerCors; Section 2.3 gives prerequisites on concurrency verification and separation logic.

Before discussing the NDFS algorithms, let us first recall the basic definitions of automata and accepting cycles. An *automaton* G is a quadruple $(\mathcal{S}, s_I, \mathsf{succ}, \mathcal{A})$ consisting of a finite set $\mathcal{S}$ of *states*, an *initial state* $s_I \in \mathcal{S}$, a *next-state relation* $\mathsf{succ} : \mathcal{S} \to 2^{\mathcal{S}}$ and a set $\mathcal{A} \subseteq \mathcal{S}$ of *accepting states*. A *path* in G is a sequence $P = s_0, \ldots, s_{n+1}$ of $\mathcal{S}$-states so that $s_{i+1} \in \mathsf{succ}(s_i)$ for every $0 \leq i \leq n$. The notation $|P| \triangleq n + 2$ denotes the *length of* P, $P[i] \triangleq s_i$ the *ith state on* P, and $P[i..]$ the *subpath* $s_i, \ldots, s_{n+1}$. Any state s is defined to be *reachable* (in G) if there exists an (s_I, s)-path. Any path P is a *cycle* whenever $P[0] = P[|P| - 1]$ and $1 < |P|$. Finally, any cycle P is *accepting* if $P[i] \in \mathcal{A}$ for some $0 \leq i < |P|$.

2.1 Nested Depth-First Search

Figure 1 presents a standard, sequential implementation of NDFS, consisting of two nested DFS searches: `dfsblue` and `dfsred`. The *blue search* processes successors recursively in DFS order, marking them blue when done on line 8. The colour cyan indicates a partially explored state, i.e., not all of its successors have been visited yet by the blue search. Just before backtracking from an accepting

```
1  void dfsblue($s$)
2      $s.color1$ := cyan;
3      for $t \in$ succ($s$) do
4          if $t.color1$ = white then
5              dfsblue($t$);
6      if $s \in \mathcal{A}$ then
7          dfsred($s$);
8      $s.color1$ := blue;

9  void dfsred($s$)
10     $s.color2$ := pink;
11     for $t \in$ succ($s$) do
12         if $t.color1$ = cyan then
13             report cycle; exit;
14         if $t.color2$ = white then
15             dfsred($t$);
16     $s.color2$ := red;
```

Fig. 1: A standard sequential implementation of nested DFS.

state, `dfsblue` calls the *red search* on line 7, to report any accepting cycle. This colours a state red after processing its successors recursively on line 16. The pink colour denotes states that are only partially explored by `dfsred`[5].

It is straightforward to see that NDFS is *sound*, meaning that it only reports true accepting cycles. To see that NDFS is also *complete*, i.e., finds an accepting cycle if one exists, observe that `dfsred` will indeed be started from every accepting state. This in itself is not enough: the red search ignores states marked red in a previous call. It is essential that `dfsred` explores accepting states in the right order. The crucial insight is that `dfsred` only visits cyan and blue states and that accepting states coloured blue cannot be part of any accepting cycle.

The correctness of NDFS has been verified with Dafny [37]. We ported this correctness proof to VerCors as the basis for the verification of parallel NDFS.

2.2 Parallel Nested Depth-First Search

A naive strategy for parallelising NDFS is *swarming* [18]: running several instances of NDFS in parallel, each working on a private set of colours. Swarmed NDFS tends to find accepting cycles faster, since its workers are expected to explore different parts of the input graph. The correctness of swarmed NDFS with respect to sequential NDFS is almost immediate, except for termination handling: workers only share information about the exit condition. We also verified swarmed NDFS in VerCors, as a stepping stone for verifying parallel NDFS.

Laarman et al. improve on the swarming algorithm by sharing information of the red search in the backtrack phase. Figure 2 presents the improved algorithm. Here every line of code is supposed to be executed atomically. The entry point is `pndfs`(s_I, n), which spawns n parallel instances of `dfsblue`(s_I, *tid*) in the fashion of swarming. However, the red colourings are shared now, by which workers can guarantee that certain states are, or will be, sufficiently explored. So the red states can now be skipped in both the red search (line 19) *and* the blue search (line 4). PNDFS thus improves performance, since workers prune each other's search space. At the same time this significantly complicates the correctness argument, since workers may now prevent each other from finding accepting

[5] In the sequential algorithm, pink and red do not need to be distinguished, but having the distinction here makes the parallel version easier to explain.

```
1  void dfsblue($s$, $tid$)
2      $s.color[tid]$ := cyan;
3      for $t \in$ succ($s$) do
4          if $t.color[tid]$ = white $\wedge \neg t.red$ then
5              dfsblue($t$, $tid$);
6      if $s.acc \wedge \neg s.red$ then
7          $s.count := s.count + 1$;
8          dfsred($s$, $tid$);
9      $s.color[tid]$ := blue;
10 void pndfs($s$, $nthreads$)
11     par $tid = 0$ to $nthreads$ do
12         dfsblue($s$, $tid$);
13     report no cycle;
14 void dfsred($s$, $tid$)
15     $s.pink[tid]$ := true;
16     for $t \in$ succ($s$) do
17         if $t.color[tid]$ = cyan then
18             report cycle; exit all;
19         if $\neg t.pink[tid] \wedge \neg t.red$ then
20             dfsred($t$, $tid$);
21     if $s.acc$ then
22         $s.count := s.count - 1$;
23         await $s.count = 0$;
24     $s.pink[tid]$ := false, $s.red$ := true;
```

Fig. 2: An implementation of parallel NDFS, where the red colours are shared.

cycles. Moreover, if multiple workers initiated dfsred from the same accepting state s, they must now finish their red search simultaneously for the algorithm to be correct. The **await** synchroniser on line 23 ensures this, by blocking thread execution until $s.count$—the number of workers in dfsred(s, $\cdot$)—reaches 0.

The original correctness argument of Laarman et al. relies on a complicated inductive invariant stating that not all accepting cycles can be missed due to pruning. However, this invariant is unsuitable for use in a (semi-)automated verifier. Section 3 discusses the verification of pndfs and provides a new invariant on the red colours that allows its correctness to be proven mechanically. It also discusses how our verification handles concurrency and thread synchronisation.

2.3 Concurrency Verification with VerCors

Before discussing the actual verification, let us first briefly introduce VerCors, an automated program verifier for parallel programs. VerCors uses concurrent separation logic with permissions as its logical foundation. Its annotation language contains *fractional permission predicates* of the form $\mathsf{Perm}(s, \pi)$, in the style of Boyland [7], that capture the notion of ownership enforced by separation logic, where s is a shared memory location (e.g., a class field) and $\pi \in (0, 1]_{\mathbb{Q}}$ a *fractional value*. The fractional permissions denote access rights: if $\pi = 1$ it denotes *write access* to s, whereas $\pi < 1$ denotes a *read access* to s. Sometimes $\mathsf{Perm}(s)$ is written as shorthand for $\exists \pi : \mathsf{Perm}(s, \pi)$, to indicate *some* ownership of s. Soundness of the underlying logic ensures that the total sum of permissions for any shared memory location does not exceed 1, which implies data race freedom.

In addition to ownership predicates, the annotation language supports the $\ast\!\!\ast$ connective, which is the *separating conjunction* of separation logic. The assertion $P \ast\!\!\ast Q$ expresses that the ownerships captured by P and Q are *disjoint*, e.g., it is disallowed that both express write access to the same shared location. Ownership predicates can be *split* into disjoint parts and be *combined* as follows:

$$\mathsf{Perm}(s, \pi_1 + \pi_2) \iff \mathsf{Perm}(s, \pi_1) \ast\!\!\ast\ \mathsf{Perm}(s, \pi_2)$$

A standard pattern in concurrency verification is to split and distribute the ownership of all shared memory over *threads* and *locks*. Clarifying the latter; in case multiple threads need to write to a common footprint of shared memory, the ownerships to this footprint are typically protected by a *resource invariant*. Threads can then only use the resources protected by this invariant when they execute *atomic* instructions (i.e., when no other threads can interfere). For more details we refer to the standard papers on concurrent separation logic [34,8,44].

3 Automated Verification of Parallel NDFS

This section elaborates on the verification of `pndfs` with VerCors [35]. Section 3.1 presents and discusses our new correctness argument for `pndfs`, which includes the new invariant on the red colours and a proof of its correctness. Sections 3.2 and 3.3 discuss the mechanisation of this proof in VerCors.

3.1 Correctness of `pndfs`

The soundness proof of `pndfs` is not very different from the soundness argument of sequential NDFS: every time **report cycle** is executed, a witness cycle can be found. The main challenge lies in proving completeness, i.e., proving that if there exists any accepting cycle, `pndfs` will report it. This is difficult since workers can *obstruct* each other's red searches and thereby prevent the detection of accepting cycles. This section proposes a new key invariant and completeness proof that is suitable for deductive verification.

We start by introducing a number of low-level invariants on the local configurations of colours that can arise during a run of `pndfs`. Let Cyan_{tid} be the set of cyan-coloured states $\{s \in \mathcal{S} \mid s.color[tid] = \mathsf{cyan}\}$ private to worker *tid*, and likewise for White_{tid}, Blue_{tid} and Pink_{tid}. Moreover, let *Red* be the set of globally red states, and $\mathsf{succ}(X) \triangleq \bigcup_{s \in X} \mathsf{succ}(s)$ the *successor set* of a given set $X \subseteq \mathcal{S}$.

Lemma 1. `pndfs` *maintains the following global invariants during execution:*

1.1. $\forall tid : \mathsf{succ}(\mathit{Blue}_{tid} \cup \mathit{Pink}_{tid}) \subseteq \mathit{Blue}_{tid} \cup \mathit{Cyan}_{tid} \cup \mathit{Red}$
1.2. $\mathsf{succ}(\mathit{Red}) \subseteq \mathit{Red} \cup \bigcup_{tid}(\mathit{Pink}_{tid} \setminus \mathit{Cyan}_{tid})$
1.3. $\forall tid : \mathcal{A} \cap \mathit{Blue}_{tid} \subseteq \mathit{Red}$
1.4. $\forall tid : \mathcal{A} \cap \mathit{Pink}_{tid} \subseteq \mathit{Cyan}_{tid}$
1.5. $\forall tid : \mathit{Pink}_{tid} \subseteq \mathit{Blue}_{tid} \cup \mathit{Cyan}_{tid}$
1.6. $\forall tid : |\mathcal{A} \cap \mathit{Pink}_{tid}| \leq 1$

Proof. The proof basically checks their preservation by each line of the program.

Invariants *1.1–1.5* are reused from [25], whereas *1.6* is new and needed for the new completeness proof. Proving completeness amounts to proving that not all reachable accepting cycles can be missed due to search space pruning. To help proving this, we identify a new class of paths, which we call *tid-special paths*.

Definition 1 (Special path). *Any path $P = s_0, \ldots, s_{n+1}$ is defined to be tid-*special *if $s_0 \in \mathit{Pink}_{tid}$, $s_{n+1} \in \mathit{Cyan}_{tid}$, and none of the states on P are red, i.e., $s_k \notin \mathit{Red}$ for every k such that $0 \leq k \leq n+1$.*

Any path P is *special* if P is *tid*-special for some worker *tid*. Intuitively, the existence of a *tid*-special path during execution of `pndfs` means that (i) worker *tid* is doing a red search, since it has pink states, and (ii) this worker will eventually find an accepting cycle, unless other workers obstruct this path. Thus the above definition allows to formally define obstruction: a worker *tid* is *obstructed* (will miss an accepting cycle) if any state on a *tid*-special path is coloured red.

Our main strategy for proving completeness involves showing that every time a worker gets obstructed, a new special path can be found. A direct consequence of this is that not all accepting cycles can be missed: upon termination of `pndfs`, there are no more cyan or pink states. To help prove this, we use the following property (taken from [25], but rephrased to handle our special paths), that allows to find special paths by using the colouring invariants.

Lemma 2. *If invariants 1.1–1.6 are satisfied, then every path $P = s_0, \ldots, s_{n+1}$ with $s_0 \in \mathit{Red}$ and $s_{n+1} \in \mathcal{A} \setminus \mathit{Red}$ contains a special subpath.*

Proof. The original handwritten proof from [25] shows that this lemma follows from invariants *1.1–1.6*, by induction on P. □

The original completeness proof of [25] performs induction on the number of obstructed accepting cycles, to show the absence of such cycles upon termination as a result of Lemma 2. However, such an argument is out of reach for Hoare-style reasoning, since it is not an *inductive* invariant. We propose a new invariant that *is* inductive, which builds on the insight that, under certain colouring conditions, new special paths can always be found when workers get obstructed, as is shown by Lemma 3. In particular, `pndfs` guarantees that if there exists a special path before executing line 24, then there also exists a special path after its execution.

Lemma 3. *For any non-red state $r \in \mathcal{S} \setminus \mathit{Red}$ that is on a tid-special path, if:*

i. $r \in \mathcal{A} \implies \mathsf{succ}(r) \subseteq \mathit{Red}$, *and*
ii. $r \in \mathcal{A} \cap \mathit{Pink}_{tid} \implies \mathit{Pink}_{tid} = \{r\}$,

then there still exists a special path after adding r to Red.

Proof. Let $P = s_0 \ldots s_{n+1}$ be a *tid*-special path and assume that r is on P, so that $r = s_\ell$ for some ℓ such that $0 \leq \ell \leq n+1$. Since $\mathit{Pink}_{tid} \neq \emptyset$, worker *tid* is performing `dfsred` that was started from some accepting state $a \in \mathcal{A} \cap \mathit{Pink}_{tid}$. Then $a \neq r$, as otherwise $s_0 = a$ due to *ii.*, which by *i.* would contradict that P is special. Moreover, since $s_{n+1} \in \mathit{Cyan}_{tid}$ there exists a (s_{n+1}, a)-path Q (this is a standard property of `dfsblue`; the path Q must be on the recursive call stack). Then Lemma 2 applies on the path $s_\ell, \ldots, s_{n+1}, Q[1..]$ and gives a new special path when considering $\mathit{Red} \cup \{r\}$ as the new set of red states. □

Lemma 3 implies that every time an accepting cycle is missed due to pruning, there is always another accepting cycle that will eventually be reported. This is enough to establish completeness of `pndfs`, via the following key invariant.

Lemma 4. *The* `pndfs` *algorithm maintains the global invariant that either:*

4.1. All reachable accepting cycles contain an accepting state that is not red; or
4.2. There exists a special path.

Proof. The interesting case is showing that this invariant remains preserved after making a non-red state $s \in \mathit{Pink}_{tid} \setminus \mathit{Red}$ red (on line 24 of Fig. 2), by some worker tid that is doing a red search from some accepting state $a \in \mathcal{A} \cap \mathit{Pink}_{tid}$.

– Suppose $s \notin \mathcal{A}$. If s is on a special path, then Invariant *4.2* is reestablished due to Lemma 3, and otherwise the key invariant remains preserved.
– Suppose $s \in \mathcal{A}$. Then $s = a$ by Invariant *1.6*. Since worker tid is about to finish its red exploration, we have that (†) $\mathit{Pink}_{tid} = \{s\}$ (i.e., all other pink states have been fully explored) and consequently that (‡) $\mathsf{succ}(s) \subseteq \mathit{Red}$. Furthermore, due to the **await** s instruction on line 23 we have that (†) and (‡) hold for *all* workers that are doing a red exploration that involves s. If s is on a special path, then Invariant *4.2* is reestablished due to Lemma 3. So now suppose that s is on an accepting cycle P. Without loss of generality, assume that $P[0] = s$. Then (‡) implies that $1 < |P|$ and that $P[1] \in \mathit{Red}$. Thus Lemma 3 applies on the path $P[1..]$ to establish Invariant *4.2*. □

The next theorem shows how Lemma 4 allows deriving completeness of parallel NDFS. In particular, it shows that no accepting cycles can exist when all threads have terminated, in which case all the theorem's premises are fulfilled.

Theorem 1. *If for every worker tid it holds that* $\mathit{Pink}_{tid} = \emptyset$, $\mathit{Cyan}_{tid} = \emptyset$ *and* $s_I \in \mathit{Blue}_{tid}$*, then there does not exist a reachable accepting cycle.*

Proof. Towards a contradiction, suppose that there exists an accepting cycle P that is reachable via an $(s_I, P[0])$-path Q. Due to the theorem's premises no special paths can exist, and therefore by Lemma 4 there is an accepting state on P that is not red. Without loss of generality, assume that (†) $P[0] \in \mathcal{A} \setminus \mathit{Red}$. Since $Q[0] \in \mathit{Blue}_0$ (since there is at least one worker), by induction on Q together with Lemma 1 we have that $P[0] \in \mathit{Red}$, which contradicts (†). □

All the above invariants and proof steps have been encoded in VerCors, which was highly non-trivial. While mechanising the proofs, many implicit proof steps had to be made explicit. Section 3.3 further details the proof mechanisation.

3.2 Encoding of `pndfs` in VerCors

Graph structures are notoriously difficult to handle in separation logics, as they usually rely on pointer aliasing, which complicates ownership handling and prevents easy use of the frame rule [38]. However, since automata have a fixed and finite set of states, we can overcome this limitation by representing the input automata as an $|\mathcal{S}| \times |\mathcal{S}|$ *adjacency matrix*. This does not impose serious restrictions: other automata encodings can be transformed at the specification level to an adjacency matrix, e.g., via model fields in the style of JML [11,29]. The suitability of adjacency matrices for deductive verification is confirmed by [24].

```
 1 enum Color {white, cyan, blue};
 2 int $N$; // the number of automata states (equal to $|\mathcal{S}|$)
 3 int $nthreads$; // the total number of participating workers
 4 bool[$N$][$N$] $G$; // adjacency matrix representation of the input automaton
 5 bool[$N$] $acc$; // the encoding of the set of accepting states
 6 Color[$nthreads$][$N$] $color$; // the colour sets for dfsblue (one for each thread)
 7 bool[$nthreads$][$N$] $pink$; // the pink colour sets for dfsred (one per thread)
 8 bool[$N$] $red$; // the global set of red colourings
 9 bool $abort$; // global termination flag
10
11 resource resource_invariant $\triangleq \cdots$; // full definition is deferred to Fig. 4.
12
13 bool Path(int $s$, int $t$, seq⟨int⟩ $P$) $\triangleq$ // the encoding of $(s,t)$-paths in $G$
14   $0 \leq s, t < N \land 0 < |P| \land P[0] = s \land P[|P|-1] = t \land$
15     $(\forall i : 0 \leq i < |P| \Rightarrow 0 \leq P[i] < N) \land (\forall i : 0 \leq i < |P|-1 \Rightarrow G[P[i]][P[i+1]])$;
16 bool Path(seq⟨int⟩ $P$) $\triangleq 0 < |P| \land \mathsf{Path}(P[0], P[|P|-1], P)$;
17 bool ExPath(int $s$, int $t$, int $n$) $\triangleq \exists P : n \leq |P| \land \mathsf{Path}(s, t, P)$;
18 bool SpecialPath(seq⟨int⟩ $P$, int $tid$) $\triangleq$ // the encoding of $tid$-special paths
19   $pink[tid][P[0]] \land color[tid][P[|P|-1]] = \mathsf{cyan} \land \forall i : 0 \leq i < |P| \Rightarrow \neg red[P[i]]$;
20 bool ExSpecialPath(int $tid$) $\triangleq \exists P : 1 < |P| \land \mathsf{Path}(P) \land \mathsf{SpecialPath}(P, tid)$;
21
22 /* An excerpt of the top-level contract (further discussed in Section 3.3). */
23 ensures \result $\Rightarrow (\exists a : 0 \leq a < N \land acc[a] \land \mathsf{ExPath}(s_I, a, 1) \land \mathsf{ExPath}(a, a, 2))$;
24 ensures $(\exists a : 0 \leq a < N \land acc[a] \land \mathsf{ExPath}(s_I, a, 1) \land \mathsf{ExPath}(a, a, 2)) \Rightarrow$ \result;
25 bool pndfs(int $s_I$);
```

Fig. 3: The automata representation and an excerpt of `pndfs`'s top-level contract.

Figure 3 shows the encoding of the input automaton G in VerCors. The thread-local colour sets are represented as matrices of dimension $nthreads \times |\mathcal{S}|$, so that each thread tid uses $color[tid][\cdot]$ and $pink[tid][\cdot]$ to administrate their (local) status of exploration. The sets of red and accepting states are shared between threads and thus encoded as $|\mathcal{S}|$-sized Boolean arrays. The succ function can now be defined such that $t \in \mathsf{succ}(s)$ whenever $G[s][t]$ is true for every $0 \leq s, t < N$.

This encoding of automata, together with an encoding of the definition of paths (on lines 13–17) is sufficient to express the main correctness property that is proven by VerCors. More specifically, line 23 expresses *soundness*: a positive return value indicates the existence of an accepting cycle. Line 24 expresses *completeness*: if there exists an accepting cycle, then `pndfs` returns positively.

Atomic operations. The handwritten correctness argument of [25] for Figure 2 assumes that all program lines are executed atomically. This is reflected in the VerCors encoding: all updates to shared memory are made within atomic operations, which specification-wise all give access to the same shared resources. For example, the assignment $s.pink[tid] := true$ on line 15 (Fig. 2) is implemented as the atomic operation "**atomic** { $pink[tid][s] := true$ }". On the specification level, the atomic sub-program receives all the missing access rights required for the assignment, which are otherwise protected by the resource invariant declared on line 11 (Fig. 3). The exact definition of `resource_invariant` is deferred to §3.3, and the type **resource** is the type of separation logic assertions. Moreover, the **await** instruction on line 23 (Fig. 2) is implemented as a busy while-loop that only stops when $s.count = 0$, which is checked atomically in every iteration.

Termination handling. The pseudocode in Figure 2 uses an "**exit all**" command to terminate all threads when an accepting cycle has been found. However, this mechanism was left implicit. Our formalisation in VerCors makes the termination system explicit: it consists primarily of a global *abort* flag that is declared on line 9 in Figure 3. All workers regularly poll this flag to determine whether they continue or not. The *abort* flag is set to true by the main thread—the thread that started `pndfs` and spawned all worker threads on line 11 of Fig. 2—as soon as one of the workers returns with an accepting cycle.

3.3 Verification of `pndfs` in VerCors

One major challenge of concurrency verification is finding a proper distribution of shared-memory ownership, that allows proving memory safety as well as any functional properties of interest. This section starts by discussing how we distribute the ownership of the input automaton over threads and the resource invariant, in such a way that Invariants *1.1–1.6* and *4.1–4.2* can be encoded.

To prove the preservation of these invariants after every computation step, auxiliary bookkeeping is needed on the specification level. For example, to mechanise the proof of Lemmas 3 and 4 we need to make explicit that all workers *tid* with $Pink_{tid} \neq \emptyset$ are doing a red search that was started from some root state $a \in \mathcal{A} \cap Pink_{tid}$. This auxiliary bookkeeping is maintained in the resource invariant, via *auxiliary ghost state*, which is explained later. Finally, we give the fully annotated version of `pndfs` and explain how completeness is proven from Lemma 4, by applying the VerCors encoding of Theorem 1.

Ownership distribution. We start by explaining how the ownership of the automaton encoding (lines 2–8 in Fig. 3) is distributed among workers and the resource invariant. First observe that all colouring invariants express *global properties* that span over (*i*) the shared *red* colourings, as well as (*ii*) the local configurations *color*[*tid*] and *pink*[*tid*] of every worker *tid*. To define the ownership distribution for (*i*), observe that the only way to distribute the access rights to *red* to enable all threads to regain write access, is to let the resource invariant protect full ownership of *red*. The resource invariant therefore fully captures the properties about red states expressed in Lemmas 1 and 4. However, to be able to specify that, it also requires partial ownership of all thread-local colourings.

Figure 4 presents the full resource invariant, that includes: access rights to both global and thread-local colourings on lines 2–4; the encoding of Lemma 1 on lines 10–17 and 22; and the encoding of Lemma 4 on lines 30–32. In addition, the resource invariant holds partial ownership of the *abort* flag on line 8, to ensure that global termination is only announced when an accepting cycle is found.

```
resource resource_invariant $\triangleq$
    $\mathsf{Perm}(N) \ast\!\ast\ \mathsf{Perm}(nthreads) \ast\!\ast\ \mathsf{Perm}(G) \ast\!\ast\ \mathsf{Perm}(acc) \ast\!\ast$
    $(\forall tid, s : \mathsf{Perm}(color[tid][s], \frac{1}{2}) \ast\!\ast\ \mathsf{Perm}(pink[tid][s], \frac{1}{2}))\ \ast\!\ast$
    $(\forall s : \mathsf{Perm}(red[s], 1))\ \ast\!\ast$
    termination() $\ast\!\ast$ colourings() $\ast\!\ast$ dfsred_status() $\ast\!\ast$ keyinvariant();

resource termination() $\triangleq$ // Resources for termination handling.
    $\mathsf{Perm}(abort, \frac{1}{2}) \ast\!\ast\ abort \Rightarrow \exists s : acc[s] \wedge \mathsf{ExPath}(s_I, s, 1) \wedge \mathsf{ExPath}(s, s, 2)$;

resource colourings() $\triangleq$ // The low-level colouring invariant encodings.
    $\forall tid, s : (color[tid][s] = \mathsf{blue} \vee pink[tid][s]) \Rightarrow \forall s' \in \mathsf{succ}(s) :$
        $color[tid][s'] = \mathsf{blue} \vee color[tid][s'] = \mathsf{cyan} \vee red[s']\ \ast\!\ast$ // Inv. 1.1
    $\forall s : red[s] \Rightarrow \forall s' \in \mathsf{succ}(s) :$
        $red[s'] \vee \exists tid : pink[tid][s'] \wedge color[tid][s'] \neq \mathsf{cyan}\ \ast\!\ast$ // Inv. 1.2
    $\forall tid, s : (acc[s] \wedge color[tid][s] = \mathsf{blue}) \Rightarrow red[s]\ \ast\!\ast$ // Inv. 1.3
    $\forall tid, s : (acc[s] \wedge pink[tid][s]) \Rightarrow color[tid][s] = \mathsf{cyan}\ \ast\!\ast$ // Inv. 1.4
    $\forall tid, s : pink[tid][s] \Rightarrow (color[tid][s] = \mathsf{cyan} \vee color[tid][s] = \mathsf{blue})$; // 1.5

/* Auxiliary ghost state for proving Lemma 3 and preserving Inv. 4. */
resource dfsred_status() $\triangleq \forall tid :$ (
    $\mathsf{Perm}(exploringred[tid], \frac{1}{2}) \ast\!\ast\ \mathsf{Perm}(redroot[tid], \frac{1}{2}) \ast\!\ast\ \mathsf{Perm}(waiting[tid], \frac{1}{2}) \ast\!\ast$
    $\forall s : pink[tid][s] \Rightarrow (exploringred[tid] \wedge (acc[s] \Rightarrow s = redroot[tid]))\ \ast\!\ast$ // 1.6
    $exploringred[tid] \Rightarrow acc[redroot[tid]]\ \wedge$
        $(\forall s : pink[tid][s] \Rightarrow \mathsf{ExPath}(redroot[tid], s, 1))\ \wedge$
        $(\forall s : color[tid][s] = \mathsf{cyan} \Rightarrow \mathsf{ExPath}(s, redroot[tid], 1))\ \wedge$
        $(\neg waiting[tid] \Rightarrow \neg red[redroot[tid]])\ \wedge$
        $(waiting[tid] \Rightarrow \forall s : pink[tid][s] \Leftrightarrow s = redroot[tid])$)

/* The encoding of Lemma 4, from which completeness of pndfs follows. */
resource keyinvariant() $\triangleq$
    $(\forall s : acc[s] \wedge \mathsf{ExPath}(s_I, s, 1) \wedge \mathsf{ExPath}(s, s, 2) \Rightarrow \neg red[s])\ \vee$
    $(\exists tid : \mathsf{ExSpecialPath}(tid))$;
```

Fig. 4: The full definition of the resource invariant. Several bound checks have been omitted for presentational clarity.

Observe that the resource invariant holds a lot of quantified information. As a result, we experienced that proving the reestablishment of `resource_invariant` after finishing **atomic**s is expensive performance-wise. To make verification more efficient, we extracted all atomic operations (e.g., colour updates) into separate methods and prove their contracts in a function-modular way. This improves performance, as it cuts the problem of verifying `dfsred` and `dfsblue` into smaller sub-problems that are individually more manageable for the SMT solver.

Finally, Figure 5 presents an excerpt of the contract of `dfsblue`, which shows the ownership pattern of all threads. Notably, every thread *tid* receives the remaining ownership of $color[tid]$ and $pink[tid]$ on line 4. Thus threads can always read from their thread-local colour fields, and may write to them while doing so

```
1  context $\mathsf{Perm}(N) ** \mathsf{Perm}(nthreads) ** \mathsf{Perm}(G) ** \mathsf{Perm}(acc)$;
2  context $0 \leq s < N$;
3  context $0 \leq tid < nthreads$;
4  context $\forall t : 0 \leq t < N \Rightarrow \mathsf{Perm}(color[tid][t], \frac{1}{2}) ** \mathsf{Perm}(pink[tid][t], \frac{1}{2})$;
5  requires $color[tid][s] = \mathsf{white}$;
6  requires $\forall t : (0 \leq t < N \land color[tid][t] = \mathsf{cyan}) \Rightarrow \mathsf{ExPath}(t, s, 1)$;
7  ensures $\backslash\mathbf{result} \Rightarrow \exists a : 0 \leq a < N \land acc[a] \land \mathsf{ExPath}(s_I, a, 1) \land \mathsf{ExPath}(a, a, 2)$;
8  ensures $\neg\backslash\mathbf{result} \Rightarrow \forall t : color[tid][t] = \mathsf{cyan} \Leftrightarrow \backslash\mathbf{old}(color[tid][t]) = \mathsf{cyan}$;
9  ensures $\neg\backslash\mathbf{result} \Rightarrow pink[tid] = \backslash\mathbf{old}(pink[tid]) \land color[tid][s] = \mathsf{blue}$;
10 bool dfsblue($s$, $tid$)
11   ···
```

Fig. 5: The ownership specification in the contract `dfsblue` for thread *tid*. Annotations of the form **context** P abbreviate **requires** P; **ensures** P.

atomically. This distribution of ownership matches with the encoding of atomic operations discussed earlier. Line 7 expresses soundness of `dfsblue`, captured in the resource invariant (line 8 of Fig. 4) on global termination. This allows to deduce soundness of `pndfs` from the resource invariant, after all threads have terminated as result of the detection of an accepting cycle.

Auxiliary ghost state. As mentioned earlier, to prove that `pndfs` also *preserves* the (encodings of) Invariants *1.1–1.6* and *4.1–4.2* after every computation step, additional ghost state needs to be maintained. In particular, we need to make explicit that every worker *tid* with $Pink_{tid} \neq \emptyset$ is doing a `dfsred` search that was started from some root state $a \in \mathcal{A} \cap Pink_{tid}$. In addition, the proof of Lemma 3 needs that there exists an (s, a)-path for every $s \in Cyan_{tid}$. To prove the preservation of Lemma 4 we also need that, if worker *tid* is not yet executing the **await** instruction, we have that $a \notin Red$, and otherwise that $Pink_{tid} = \{a\}$.

This extra information is encoded in the loop invariant on lines 20–27 (Figure 4), via three *ghost arrays*, named *exploringred*, *redroot* and *waiting*. Firstly, *exploringred* administrates which workers are doing a red search. For verification purposes we added ghost code to the program, to set *exploringred*$[tid]$ to true whenever `dfsred(`a, tid`)` is invoked by worker *tid* from a blue search, and back to false whenever `dfsred(`a, tid`)` returns. Secondly, *redroot* stores the root state on which `dfsred` was invoked. Finally, *waiting* administrates which workers are executing an **await** instruction. These three ghost arrays together are closely related to the $s.count$ fields in the program of Figure 2, via the following invariant: $\forall s : s.count = |\{tid \mid exploringred[tid] \land redroot[tid] = s \land \neg waiting[tid]\}|$.

Establishing that `pndfs` adheres to the invariants in Lemmas 1 and 4 was highly non-trivial and required various complex auxiliary lemmas to be encoded and proven. These are all encoded in VerCors as *ghost methods*: side-effect-free helper methods on which the lemma is encoded in the method's contract [21,22]. Induction proofs, for example, are encoded using either loop invariants or recursion. Application of a lemma then translates to a function call on the specification level. The proofs in Section 3.1 are all encoded and applied in this way.

```
1  context $\mathsf{Perm}(N) \ast\!\!\ast \mathsf{Perm}(nthreads) \ast\!\!\ast \mathsf{Perm}(G) \ast\!\!\ast \mathsf{Perm}(acc) \ast\!\!\ast \mathsf{Perm}(abort, \frac{1}{2})$;
2  context $\forall tid, s : \mathsf{Perm}(color[tid][s], \frac{1}{2}) \ast\!\!\ast \mathsf{Perm}(pink[tid][s], \frac{1}{2})$;
3  context $\forall tid : \mathsf{Perm}(exploringred[tid], \frac{1}{2}) \ast\!\!\ast \mathsf{Perm}(redroot[tid], \frac{1}{2})$;
4  context $\forall tid : \mathsf{Perm}(waiting[tid], \frac{1}{2})$;
5  context $0 \leq s_I < N$;
6  requires $\forall tid, s : \neg exploringred[tid] \land color[tid][s] = \mathsf{white} \land \neg pink[tid][s]$;
7  ensures \result $\Rightarrow (\exists a : acc[a] \land \mathsf{ExPath}(s_I, a, 1) \land \mathsf{ExPath}(a, a, 2))$;
8  ensures $(\exists a : acc[a] \land \mathsf{ExPath}(s_I, a, 1) \land \mathsf{ExPath}(a, a, 2)) \Rightarrow$ \result;
9  bool pndfs($s_I$)
10   par $tid = 0$ to $nthreads$
11     context $\mathsf{Perm}(N) \ast\!\!\ast \mathsf{Perm}(nthreads) \ast\!\!\ast \mathsf{Perm}(G) \ast\!\!\ast \mathsf{Perm}(acc)$;
12     context $\forall s : \mathsf{Perm}(color[tid][s], \frac{1}{2}) \ast\!\!\ast \mathsf{Perm}(pink[tid][s], \frac{1}{2})$;
13     context $\mathsf{Perm}(term[tid], \frac{1}{2}) \ast\!\!\ast \mathsf{Perm}(exploringred[tid], \frac{1}{2})$;
14     context $\mathsf{Perm}(redroot[tid], \frac{1}{2}) \ast\!\!\ast \mathsf{Perm}(waiting[tid], \frac{1}{2})$;
15     requires $\neg exploringred[tid] \land \forall s : color[tid][s] = \mathsf{white} \land \neg pink[tid][s]$;
16     ensures $\neg abort \Rightarrow \forall s : color[tid][s] \neq \mathsf{cyan} \land \neg pink[tid][s]$;
17     ensures $\neg abort \Rightarrow color[tid][s_I] = \mathsf{blue}$;
18    do
19     bool $found$ := dfsblue($s_I$, $tid$);
20     if $found$ then
21       atomic { $abort$ := $true$; } // initiate global termination.
22   atomic { if $\neg abort$ then theorem_one() }; // apply Thm. 1's encoding.
23   return $abort$;
```

Fig. 6: The annotated version of pndfs, extending the excerpt given in Figure 3.

Correctness of pndfs. Figure 6 gives the annotated version of pndfs[6] that extends the excerpt given earlier, in lines 23–25 of Figure 3. The main thread requires partial ownership of all thread-local colour fields on line 2 and distributes these over the appropriate threads on line 12. The contract associated to the parallel block (lines 11–17) is called an *iteration contract* and assigns pre- and postconditions to every parallel instance. For more details on iteration contracts we refer to [5]. Most importantly, the iteration contract of each thread holds enough resources to satisfy all the preconditions of dfsblue, on line 19.

Soundness of pndfs (line 7) is proven as follows. Suppose that all threads have terminated and *abort* has been set to *true*. In that case, the resource invariant states that an accepting cycle has been found. This information can be retrieved by briefly obtaining the resource invariant in *ghost code* on line 22, which directly allows to deduce soundness. Note that this information is not lost upon releasing the resource invariant, as it is a Boolean property and thus duplicable.

To prove completeness, suppose that *abort* is still false when all workers have terminated. This implies that $Pink_{tid} = \emptyset$ and $Cyan_{tid} = \emptyset$ for every worker *tid* (line 16), as well as $s_I \in Blue_{tid}$ (line 17), since all threads started their blue

[6] Observe that every thread reads *abort* in their contract on lines 16–17, even though they do not have the required access rights to do so. This is resolved by adding some auxiliary ghost state, but this is omitted for presentational clarity.

```
1  void dfsblue($s$, $tid$)
2    $s.color[tid]$ := cyan;
3    for $t \in$ succ($s$) do
4      if $t.color[tid]$ = cyan then
5        if $s.acc \lor t.acc$ then
6          report cycle; exit all;
7      if $t.color[tid]$ = white then
8        if $\neg t.red$ then
9          dfsblue($t$, $tid$);
10   if $s.acc \land \neg s.red$ then
11     $s.count := s.count + 1$;
12     dfsred($s$, $tid$);
13   $s.color[tid]$ := blue;
```

(a) The "early cycle detection" extension

```
1  void dfsblue($s$, $tid$)
2    $allred := true$;
3    $s.color[tid]$ := cyan;
4    for $t \in$ succ($s$) do
5      if $t.color[tid]$ = white then
6        if $\neg t.red$ then
           dfsblue($t$, $tid$);
7      if $\neg t.red$ then $allred := false$;
8    if $allred$ then
9      await $s.count = 0$;
10     $s.red := true$;
11   else if $s.acc \land \neg s.red$ then
12     $s.count := s.count + 1$;
13     dfsred($s$, $tid$);
14   $s.color[tid]$ := blue;
```

(b) The "all-red" extension

Fig. 7: Two extensions (highlighted grey) to `dfsblue` that improve work sharing.

search from s_I. Combining this information with the information in the resource invariant allows one to prove all the premises of Theorem 1. Therefore its ghost method encoding can be applied on line 22, from which completeness is derived.

The encoding of parallel NDFS in VerCors [35] comprises roughly 2500 lines of code (of which ∼85% is proof overhead), which includes the mechanisation of all proof steps described in §3.1. The verification time is about 140s, measured on a Macbook with an Intel Core i5 CPU with 2,9 GHz, and 8Gb memory.

4 Optimisations

One major benefit of mechanically verified code is that optimisations can be applied with full confidence. Without verification, changes to critical code are often avoided, to ensure that no errors are introduced. A verified algorithm allows to apply optimisations easily, as these often do not change the outer contract, at most requiring only minor adaptions to the invariants. We illustrate this with two optimisations, for which [25] experimentally demonstrated improved speedup.

"Early cycle detection" checks already in the blue search if an accepting cycle is closed, cf. lines 4–6 in Figure 7a. It is known that for weak LTL properties, all accepting cycles will be found in the blue search when applying early cycle detection. To show that this optimisation indeed preserves all invariants, we simply inserted these 3 lines in the VerCors specification. The proof introduces a case distinction on whether s or t is accepting and constructs a witness path. This adds another 10 lines: two for the case distinction and four in each branch to show that a witness accepting cycle exists. Collectively, these extra 13 lines constitute indeed very little effort to prove this particular optimisation correct.

The second optimisation, called "all-red", checks if all successors of s became red during the blue search (lines 2 and 7 in Figure 7b). If so, we can mark $s.red$

early (lines 8–10). This optimisation is important, since it allows the global red colour to spread even in portions of the graph that are not under an accepting state, thereby allowing more pruning. However, this optimisation only preserves the invariants if we wait until $s.count = 0$ (on line 9). This test was erroneously omitted in [25][7]. Fortunately, the version in Figure 7b is correct, which has now been checked in VerCors in a straightforward manner.

5 Conclusion

This paper presents the first automated deductive verification of a parallel graph algorithm: we verified soundness and completeness of parallel nested depth-first search using VerCors. We also show that this mechanisation is helpful in quickly discovering whether optimisations of the algorithm preserve its correctness.

Many of the presented verification techniques, e.g., the use of separate contracts for single statements, the way we handle termination, and the construction of explicit witnesses through ghost variables, will be useful for the verification of other similar algorithms. Moreover, our encoding of parallel nested DFS closely resembles the implementation of such an algorithm in mainstream programming languages like C++ and Java. It would be interesting to investigate how our VerCors encoding can be automatically deployed on multi-core architectures, for example to enable comparing its performance and scalability with LTSmin.

There are many possibilities to extend the line of research on the verification of parallel model checking algorithms initiated in this paper. First, one may consider to extend the scope of this verification closer towards the actual efficient C-implementation in LTSmin. This would involve verifying the underlying concurrent hash table to store visited states (a simplified version of which has been verified before with VerCors [1]), the encoding of the colours as "bits" in the hash table buckets, and the use of CAS to manipulate these bits.

One might consider alternative parallel NDFS versions, notably [15], which shares the blue colour, invoking a repair procedure when the depth-first order is violated. Both algorithms have been reconciled in [14], sharing both blue and red. This work could be extended to a wealth of other optimisations like partial-order reduction, or other parallel model checking algorithms, for example [26,4,40].

Our work can be considered as a first step towards a library for the verification of graph-based (multi-core) model checking algorithms. It will be an interesting line of future work to continue this: developing a full-fledged verification library for common subtasks, like graph manipulations and termination detection.

Acknowledgments and data availability statement. This work is partially supported by the NWO VICI 639.023.710 Mercedes project and by the NWO TOP 612.001.403 VerDi project. The datasets for this case study are available at: https://doi.org/10.4121/uuid:36c00955-5574-44d9-9b26-340f7a1ea03b.

[7] Wan Fokkink and his students Stefan Vijzelaar and Pieter Hijma already found in 2012 that the "all-red" extension required an extra check '**await** $s.count = 0$' in [25], and wondered whether '**await** $s.count \leq 1$' would be sufficient. Independently, Akos Hajdu reported this omission in 2015.

References

1. A. Amighi, S. Blom, and M. Huisman. Resource Protection Using Atomics - Patterns and Verification. In *APLAS*, pages 255–274, 2014. doi:10.1007/978-3-319-12736-1_14.
2. J. Barnat, V. Bloemen, A. Duret-Lutz, A. Laarman, L. Petrucci, J. van de Pol, and E. Renault. Parallel Model Checking Algorithms for Linear-Time Temporal Logic. In *Handbook of Parallel Constraint Reasoning*, pages 457–507. Springer, 2018. doi:10.1007/978-3-319-63516-3_12.
3. J. Barnat and I. Cerná. Distributed breadth-first search LTL model checking. *Formal Methods in System Design*, 29(2):117–134, 2006. doi:10.1007/s10703-006-0009-y.
4. V. Bloemen, A. Laarman, and J. van de Pol. Multi-core On-the-fly SCC Decomposition. In *PPoPP*, pages 1–12. ACM, 2016. doi:10.1145/2851141.2851161.
5. S. Blom, S. Darabi, and M. Huisman. Verification of Loop Parallelisations. In *FASE*, pages 202–217. Springer, 2015. doi:10.1007/978-3-662-46675-9_14.
6. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *iFM*, LNCS, pages 102–110. Springer, 2017. doi:10.1007/978-3-319-66845-1_7.
7. J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, LNCS, pages 55–72. Springer, 2003. doi:10.1007/3-540-44898-5_4.
8. S. Brookes. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science*, 375(1–3):227–270, 2007. doi:10.1016/j.tcs.2006.12.034.
9. J. Brunner and P. Lammich. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. *Journal of Automated Reasoning*, 60(1):3–21, 2018. doi:10.1007/s10817-017-9418-4.
10. R. Chen, C. Cohen, J. Lévy, S. Merz, and L. Théry. Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle. *CoRR*, 2018. URL: http://arxiv.org/abs/1810.11979.
11. Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model Variables: Cleanly Supporting Abstraction in Design by Contract: Research Articles. *Software–Practice and Experience*, 35(6):583–599, 2005. doi:10.1002/spe.v35:6.
12. E. Clarke, T. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018. doi:10.1007/978-3-319-10575-8.
13. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2–3):275–288, 1992. doi:10.1007/BF00121128.
14. S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved Multi-Core Nested Depth-First Search. In *ATVA*, LNCS, pages 269–283. Springer, 2012. doi:10.1007/978-3-642-33386-6_22.
15. S. Evangelista, L. Petrucci, and S. Youcef. Parallel Nested Depth-First Searches for LTL Model Checking. In *ATVA*, LNCS, pages 381–396. Springer, 2011. doi:10.1007/978-3-642-24372-1_27.
16. A. Griggio, M. Roveri, and S. Tonetta. Certifying Proofs for LTL Model Checking. In *FMCAD*, pages 225–233, 2018. doi:10.23919/FMCAD.2018.8603022.
17. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. doi:10.1109/32.588521.
18. G. Holzmann, R. Joshi, and A. Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011. doi:10.1109/TSE.2010.110.

19. G. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In *The Spin Verification System*, volume 32 of *DIMACS*, pages 23–32, 1996. doi:10.1090/dimacs/032/03.
20. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM*, 2011. doi:10.1007/978-3-642-20398-5_4.
21. B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative Programs as Proofs. In *VS-Tools workshop at VSTTE*, 2010.
22. S. Joosten, W. Oortwijn, M. Safari, and M. Huisman. An Exercise in Verifying Sequential Programs with VerCors. In *FTfJP*, pages 40–45, 2018. doi:10.1145/3236454.3236479.
23. G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *TACAS*, pages 692–707. Springer, 2015. doi:10.1007/978-3-662-46681-0_61.
24. J. Kübler. Comparing Deductive Program Verification of Graph Data-Structures. *Bachelor's thesis, KIT*, 2018.
25. A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-core Nested Depth-First Search. In *ATVA*, LNCS, pages 321–335. Springer, 2011. doi:10.1007/978-3-642-24372-1_23.
26. A. Laarman, M. Olesen, A. Dalsgaard, K. Larsen, and J. van de Pol. Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction. In *CAV*, pages 968–983. Springer, 2013. doi:10.1007/978-3-642-39799-8_69.
27. P. Lammich and R. Neumann. A Framework for Verifying Depth-First Search Algorithms. In *CPP*, pages 137–146. ACM, 2015. doi:10.1145/2676724.2693165.
28. P. Lammich and S. Wimmer. IMP2 – Simple Program Verification in Isabelle/HOL. *Archive of Formal Proofs*, 2019. http://isa-afp.org/entries/IMP2.html, Formal proof development.
29. K.R.M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA*, pages 144–153. ACM, 1998. doi:10.1145/286942.286953.
30. K.R.M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.
31. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008. doi:10.1007/978-3-540-78800-3_24.
32. P. Müller, M. Schwerhoff, and A. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.
33. K. Namjoshi. Certifying Model Checkers. In *CAV*, LNCS, pages 2–13. Springer, 2001. doi:10.1007/3-540-44585-4_2.
34. P. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.
35. W. Oortwijn, M. Huisman, S. Joosten, and J. van de Pol. Artifact for Automated Verification of Parallel Nested DFS, TACAS 2020. 4TU.ResearchData. doi:10.4121/uuid:36c00955-5574-44d9-9b26-340f7a1ea03b.
36. A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.
37. J. van de Pol. Automated Verification of Nested DFS. In *FMICS*, LNCS, pages 181–197. Springer, 2015. doi:10.1007/978-3-319-19458-5_12.
38. A. Raad, A. Hobor, J. Villard, and P. Gardner. Verifying Concurrent Graph Algorithms. In *Programming Languages and Systems*, pages 314–334. Springer, 2016. doi:10.1007/978-3-319-47958-3_17.

39. J. Reif. Depth-First Search is Inherently Sequential. *Information Processing Letters*, 20(5):229–234, 1985. doi:10.1016/0020-0190(85)90024-9.
40. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on Parallel Explicit Emptiness Checks for Generalized Büchi Automata. *STTT*, 19(6):653–673, 2017. doi:10.1007/s10009-016-0422-5.
41. S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *TACAS*, LNCS, pages 174–190. Springer, 2005. doi:10.1007/978-3-540-31980-1_12.
42. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized Verification of Fine-Grained Concurrent Programs. In *PLDI*, pages 77–87. ACM, 2015. doi:10.1145/2813885.2737964.
43. C. Sprenger. A Verified Model Checker for the Modal μ-calculus in Coq. In *TACAS*, LNCS, pages 167–183. Springer, 1998. doi:10.1007/bfb0054171.
44. V. Vafeiadis. Concurrent Separation Logic and Operational Semantics. In *MFPS*, ENTCS, pages 335–351, 2011. doi:10.1016/j.entcs.2011.09.029.
45. M. Vardi and P. Wolper. Automata-Theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences*, 32(2):183–221, 1986. doi:10.1016/0022-0000(86)90026-7.
46. Why3 gallery of formally verified programs. http://toccata.lri.fr/gallery/graph.en.html *(accessed on February 2020)*.
47. S. Wimmer and P. Lammich. Verified Model Checking of Timed Automata. In *TACAS*, LNCS, pages 61–78. Springer, 2018. doi:10.1007/978-3-319-89960-2_4.

Discourje: Runtime Verification of Communication Protocols in Clojure

Ruben Hamers[1] and Sung-Shik Jongmans[1,2]

[1] Open University, Heerlen, the Netherlands
[2] CWI, Amsterdam, the Netherlands

TACAS Artifact Evaluation 2020 Accepted

Abstract. This paper presents Discourje: a runtime verification framework for communication protocols in Clojure. Discourje guarantees safety of protocol implementations relative to specifications, based on an expressive new version of multiparty session types. The framework has a formal foundation and is itself implemented in Clojure to offer a seamless specification–implementation experience. Benchmarks show Discourje's overhead can be less than 5% for real/existing concurrent programs.

1 Introduction

Background. To take advantage of today's and tomorrow's multi-core processors, shared-memory concurrent programming—a notoriously complex enterprise—is becoming increasingly important. To alleviate some of the complexities, in addition to low-level *synchronization primitives*, several modern programming languages have started to offer core support for higher-level *communication primitives* as well, in the guise of message passing through *channels* (e.g., Go [25], Rust [42], Clojure [17]). The idea is that, beyond their usage in distributed computing, channels can also serve as a *programming abstraction* for shared memory, supposedly less prone to concurrency bugs than locks, semaphores, and the like. However, in a recent study of 171 concurrency bugs in popular open source Go programs [48], Tu et al. found that "message passing does not necessarily make multi-threaded programs less error-prone than shared memory."

From a programmer's perspective, a key problem is this: if we already know which *roles* (threads), *infrastructure* (channels between threads), and *protocols* (communications through channels) our program should consist of, then how can we ensure our implementation is indeed *safe* relative to our specification? Safety means "bad" channel actions never occur: if a send, receive, or close happens in the implementation, then it is allowed by the protocol in the specification. For instance, typical protocols rule out common message-passing concurrency bugs [48], such as sends without receives, receives without sends, and type mismatches (actual type sent $\neq$ expected type received). Essentially, thus, we face a classical verification problem, with classical ingredients: an implementation language $\mathcal{I}$, a specification language $\mathcal{S}$, and an inclusion relation $\preceq$.

Over the past years, a significant body of research in this area has been based on *multiparty session types* (MPST) [27]. The idea is to specify protocols as behavioral types [1,30] against which threads are subsequently type-checked; the

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 266–284, 2020.
https://doi.org/10.1007/978-3-030-45190-5_15

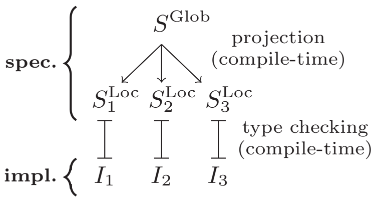


Fig. 1. MPST

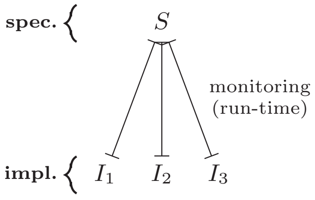


Fig. 2. This paper

theory guarantees that static well-typedness of threads at compile-time implies dynamic safety of their channel actions at run-time. Originally [27], $\mathcal{I}$ was a dialect of pi-calculus, $\mathcal{S}$ was a calculus of behavioral types, and $\preceq$ was defined through formal typing rules, but more recently, practical implementations were developed as well [14,28,29,37,38,44], where $\mathcal{I}$ is an existing *general-purpose language* (GPL; Erlang, F#, Go, Java, Scala), $\mathcal{S}$ is a new *domain-specific language* (DSL; Scribble), and $\preceq$ encodes behavioral types in $\mathcal{S}$ as non-behavioral types in $\mathcal{I}$ (e.g., through custom communication API generation [29]). These works highlight two key strengths of the MPST methodology, namely it supports:

#1 fully automated verification of *concrete programs* (vs. abstract models);
#2 user-friendly *programming language-based notation* to write specifications of protocols (vs. dynamic logic or temporal logic).

Problem. One of the key open problems of MPST concerns *expressiveness*. For instance, suppose we need to write a program in which messages are repeatedly communicated from threads I_1 and I_2 to thread I_3, non-deterministically ordered (i.e., standard producers–consumer); this protocol is not supported by MPST.

We identify two reasons why expressiveness is limited.

First, MPST were originally developed for distributed computing (service choreographies [10,11]); accordingly, *decoupled* verification of roles (per-service type-checking) has always been a key requirement [14]. This is reflected in the MPST workflow (Fig. 1): first, the programmer writes a *global* protocol specification; then, an MPST tool *projects* it onto every role to infer *local* protocol specifications; then, the implemented threads are type-checked. However, role-based decomposition of global behavior into equivalent local behaviors often cannot be done statically (e.g., [12]), so expressiveness is limited by "projectability".

Second, MPST prescribes static type-checking, which limits expressiveness, because: (a) type-checking is sound, but not complete, so the static MPST approach rejects implementations that are conservatively ill-typed but actually safe; (b) protocols whose execution relies on value-dependent control flow are supported only in limited circumstances. To alleviate (b), value-dependent type constructors can be added to $\mathcal{S}$ [20,47], but this raises practical issues (i.e., dependent types are only scarcely supported by mainstream GPLs).

Contributions. To simplify shared-memory concurrent programming in languages with channels, we aim to consolidate strengths #1 and #2 (page 2), but alleviate MPST's expressiveness issues. Specifically, this paper is founded on two tenets that depart from existing work in significant ways (Fig. 2).

First, we exploit the fact that in our context, channels serve "merely" as programming abstractions for shared memory; there is no distribution whatsoever. Thus, whereas MPST-based verification for distributed computing requires projection, this is not the case in our setting, opening the door to fully automated *projection-free* MPST and eliminating a significant source of restrictions.

Second, instead of adopting MPST-based verification through static type-checking at compile-time, we explore MPST-based verification through *dynamic monitoring at run-time*. This enables soundness *and completeness*, while it also supports *value-dependent protocols* in a generally implementable way (i.e., we are not aware of a mainstream GPL that does not support our monitoring approach).

In this paper, we present our practical embodiment of these ideas: *Discourje* (pronounced "discourse"), a runtime verification framework for communication protocols in *Clojure* [17,26]. Discourje consists of two components: a DSL to specify protocols and construct monitors, and an API to implement protocols (supplementing Clojure) and add instrumentation. While we could have developed this framework for any language with channel-based programming abstractions, including Go and Rust, Clojure is particularly interesting, because: (1) Clojure has a powerful macro system that enabled us to develop the Discourje DSL as an extension to Clojure, thereby offering programmers a seamless specification–implementation experience; (2) contrasting Go and Rust, Clojure is not a systems language but an applications language, so runtime verification overheads might be more tolerable. We summarize our contributions as follows:

- **Overview** (Sect. 2): Discourje guarantees *safety* of protocol implementations, it provides *freedom from data races* in pure Clojure, and it is *more expressive* than existing MPST tools, as demonstrated through examples.
- **Design** (Sect. 3): We developed *core calculi*, including operational semantics, for Clojure and the Discourje DSL as a theoretical foundation.
- **Implementation** (Sect. 4): We implemented Discourje fully in Clojure. The Discourje DSL comprises Clojure *macros*, while the Discourje API is a wrapper around Clojure functions to add instrumentation, *non-invasively*.
- **Evaluation** (Sect. 5): Through benchmarks, we show that Discourje's overhead can be less than 5% for real/existing concurrent programs.

Our artifact is available at https://github.com/discourje.

2 Overview

Clojure (in a nutshell). Clojure [17,26] is a general-purpose, impure functional language that compiles to Java bytecode. As a dialect of Lisp, Clojure follows the code-as-data philosophy, provides a powerful macro system, and adopts parenthesized prefix notation. Clojure offers asynchronous channel-based programming abstractions through core library `clojure.core.async` [16]. In the annual Clojure survey [15], Clojure programmers indicate "ease of development" is more important than "runtime performance"; this makes Clojure an interesting target for runtime verification (viz. overheads).

To introduce the core features of Clojure relevant to this paper, Fig. 3 shows a channel-based concurrent Tic-Tac-Toe program in Clojure[3] while Fig. 4 summarizes the meaning of every primitive (";;" indicates comments). Lines 1–9 define constants (`blank`, `cross`, `nought`, `initial-grid`) and functions (`get-blank`, `add`, `not-final?`) to represent Tic-Tac-Toe concepts. Lines 11-12 define two channels (`a->b` and `b->a`) that implement the infrastructure through which players Alice and Bob communicate. Channels in Clojure are bounded: sends/receives block until a channel is not full/empty. Lines 14–24 and 25–35 define threads that implement Alice and Bob. Both players execute a loop, starting with a blank grid. In each iteration, Alice first gets the index of some blank space on the grid, then plays a cross in that space, then sends a message to Bob to communicate the index, then awaits a message from Bob, and then updates the grid accordingly; Bob acts symmetrically. After every grid update, Alice or Bob checks if it has reached a final configuration; if so, the loop is exited and channels are closed.

Every Clojure data structure, including the vector that implements the grid, is *persistent*, and therefore, effectively *immutable*. This means that every operation on an existing data structure leaves it intact, and instead, it returns a new data structure. Thus, Alice and Bob initially share the same initial grid, but because it cannot be modified in-place, modifications need to be explicitly communicated. Persistence of Clojure data structures is also why we can guarantee freedom from data races in pure Clojure (= Clojure without Java objects): if users communicate only Clojure data through channels, race freedom is guaranteed (if Java objects are communicated, the user is responsible to avoid races).

Basic Discourje: Tic-Tac-Toe. A basic Discourje specification of the Tic-Tac-Toe protocol for Alice and Bob is shown in Fig. 5. We typeset Discourje "keywords" (which are actually just Clojure functions and macros) **`bold violet`**.

Lines 1–2 define two roles (**`role`**) to represent Alice and Bob. Lines 4–6 define an auxiliary specification, inserted twice into the main specification (**`ins`**); it states that the channels between Alice and Bob are closed (**`-##`**), in parallel (**`par`**). Lines 7–13 define the main specification; it states that recursively (**`fix`**), first a message of type `Long` (the index of a grid) is communicated from Alice to Bob (**`-->`**), and then from Bob to Alice, unless the channels are closed (the game is done). Square brackets are used to build lists of sub-specifications (sequencing).

The Tic-Tac-Toe protocol depends on value-dependent control flow, as Alice and Bob close the channels only once the grid has reached a final configuration. This is a non-protocol-related property that no existing MPST tool supports.

[3] Tic-Tac-Toe is a two-player game played on a 3x3 grid. Players take turns to fill the initially blank spaces of the grid with crosses ("X") and noughts ("O"). The first player to fill three adjacent spaces, in any direction, with the same symbol wins.

```
(def blank " ") (def cross "x") (def nought "o")

(def initial-grid [blank blank blank   ;; an initial 3x3 grid of blank spaces,
                   blank blank blank   ;;   implemented as a vector of length 9
                   blank blank blank]) ;;   (persistent data structure)

(def get-blank  (fn [g]          ...)) ;; returns a blank space in g
(def add        (fn [g i x-or-o] ...)) ;; returns g, but with i set to x-or-o
(def not-final? (fn [g]          ...)) ;; returns true iff g is not final

(def a->b (chan 1)) (def b<-a a->b) ;; b<-a is an alias of a->b
(def b->a (chan 1)) (def a<-b b->a) ;; a<-b is an alias of b->a

(thread ;; alice                     (thread ;; bob
  (loop [g initial-grid]               (loop [g initial-grid]
    (let [i (get-blank g)                (let [i (<!! b<-a)
          g (set g i cross)]                   g (set g i cross)]
      (>!! a->b i)                         (if (not-final? g)
      (if (not-final? g)                     (let [i (get-blank g)
        (let [i (<!! a<-b)                         g (set g i nought)]
              g (set g i nought)]              (>!! b->a i)
          (if (not-final? g)                   (if (not-final? g)
            (recur g))))))                       (recur g))))))
  (close! a->b))                       (close! b->a))
```

Fig. 3. Clojure implementation of Tic-Tac-Toe (dashed arrows: matching send/receive)

Library `clojure.core` (basic):

- **(def** x e**)**: first evaluates e to v; then binds x to v in the global environment.
- **(fn** [x_1 ... x_n] e_1 ... e_m**)**: evaluates to a function with parameters x_1, ..., x_n and creates a recursion point; then, when applied to arguments v_1, ..., v_n, sequentially evaluates e_1, ..., e_m with x_1, ..., x_n bound to v_1, ..., v_n.
- **(let** [x_1 e_1 ... x_n e_n] e**)**: first evaluates e_1 to v_1; then evaluates e_2 to v_2 with x_1 bound to v_1; ...; then evaluates e_n to v_n with x_1, ..., x_{n-1} bound to v_1, ..., v_{n-1}; then evaluates e with x_1, ..., x_n bound to v_1, ..., v_n.
- **(loop** [x_1 e_1 ... x_n e_n] e**)**: same as **let**, but also creates a recursion point.
- **(recur** e_1 ... e_n**)**: first evaluates e_1, ..., e_n to v_1, ..., v_n; then evaluates the nearest recursion point with x_1, ..., x_n bound to v_1, ..., v_n.
- **(if** e_1 e_2 e_3**)**: first evaluates e_1; if `true`, evaluates e_2; else, evaluates e_3.

Library `clojure.core.async` (concurrency):

- **(>!!** c e**)**: first evaluates e to v; then sends v through channel c.
- **(<!!** c**)**: receives a value through channel c.
- **(close!** c**)**: closes channel c.
- **(chan** n**)**: evaluates to a channel with a buffer of size n.
- **(thread** e**)**: creates a new thread that evaluates e

Fig. 4. Clojure primitives

```
(def alice (role "alice")) ;; roles
(def bob   (role "bob"))

(def ttt-close (dsl ;; auxiliary spec
  (par (-## alice bob)
       (-## bob alice))))
```

```
(def ttt (dsl ;; main spec
  (fix :X
    [(--> alice bob Long)
     (alt (ins ttt-close)
          [(--> bob alice Long)
           (alt (ins ttt-close)
                (fix :X))])])))
```

Fig. 5. Discourje specification of Tic-Tac-Toe

```
10 (def m (moni (spec ttt)))
11 (def a->b (chan 1 alice bob m)) (def b<-a a->b)
12 (def b->a (chan 1 bob alice m)) (def a<-b b->a)
```

Fig. 6. Changes to Fig. 3 to monitor Alice and Bob against the specification in Fig. 5

To monitor the implementations of Alice and Bob against this specification, first, we need to load library `discourje.core.async` instead of `clojure.core.async` (implicitly loaded in Fig. 3). All other code modifications are shown in Fig. 6: on line 10, the specification is evaluated to an internal form (**`spec`**) and wrapped in a new monitor (**`moni`**), while on lines 11–12, we associate the intended sender, receiver, and monitor with the channels. *No other changes are needed:* notably, the code for Alice (Fig. 3, lines 14–24) and Bob (lines 25–35) is unaffected; Discourje is non-invasive to start using. Running the monitor alongside the implementation guarantees safety: if a non-compliant channel action were to be attempted, the monitor prevents it from happening and throws an exception.

The implementation in Fig 3 can *indeed* violate the specification in Fig. 5: the specification states channels are allowed to be closed only *after* (the receive of) the previous communication is done, but in the implementation, Alice or Bob could attempt to close already *before*. In our artifact, we have a solution where we mix channels with *barrier synchronization* from the standard `java.util.concurrent` library (readily usable in Clojure), to let Alice and Bob first await each other and then close. Thus, channel-based programming abstractions monitored through Discourje can be mixed seamlessly with other concurrency libraries, which happens regularly in message passing programs [46,48].

Advanced Discourje: common patterns. Discourje specifications of common patterns of communication are shown in Fig. 7; they make use of Discourje's *role indexing* and finite *repetition* (**`rep`**) features.

Imagine we have a sequence of worker threads, organized in a pipeline (i.e., the i-th worker receives from its predecessor, $i-1$, and sends to it successor, $i+1$). Lines 1–2 define the specification of a communication from a worker to its successor. Intuitively, `succ` is a function that maps three parameters to a specification. For instance, `(ins succ bob 5 Turn)` inserts `(--> (bob 5) (bob 6) Turn)`, where `(bob 5)` and `(bob 6)` are indexed roles. We note that every role created with **`role`** allows indexing (with arbitrary types), and that specifications can be

```
 1 (def succ (dsl :w :i :t
 2   (--> (:w :i) (:w (inc :i)) t)))
 3
 4 (def pipe (dsl :w :k :t
 5   (rep seq [:i (range (dec :k))]
 6     (ins succ :w :i :t))))
 7
 8 (def ring (dsl :w :k :t
 9   [(ins pipe :w :k :t)
10    (--> (:w (dec :k)) (:w 0) :t)]))
11 (def one-one-one (dsl :m :w :k :t :u
12   (rep alt [:i (range :k)]
13     [(--> :m (:w :i) :t)
14      (--> (:w :i) :m :u)]))
15
16 (def one-all-one (dsl :m :w :k :t :u
17   (rep par [:i (range :k)]
18     [(--> :m (:w :i) :t)
19      (--> (:w :i) :m :u)]))
```

Fig. 7. Discourje specification of common patterns

parametrized by roles (`:w`), indices (`:i`), and/or types (`:t`). We also note that *any* Clojure function can be used in specifications (e.g., `inc`, to manipulate indices).

Lines 4–6 define the specification of a pipeline communication pattern; it states that specification `(ins succ :w :i :t)` is repeated for each value `:i` from `0` to `k-1`, and the iterations are composed sequentially (`seq`). Lines 8–10 extend the pipeline to a ring, where the last worker also communicates with the first.

Lines 11–14 define the specification of a communication from a "master" to *one* of k workers, and back. Similarly, lines 16–19 define the specification of a communication from a master to *all* of k workers, and back. In these specifications, loop iterations are composed alternatively (`alt`) and in parallel (`par`).

3 Design

Implementation calculus. To formalize our verification problem, we first define a calculus to model Clojure implementations. Let ℓ range over heap locations, x over variables, v over values, and I over implementations. The calculus is generated by the following grammar:

$$\begin{aligned} v &::= \mathsf{nil} \mid \ell \mid \mathsf{fn}\; x\; I \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid 2 \mid \ldots \\ I &::= v \mid I_1\; I_2 \mid x \mid \mathsf{def}\; x\; I \mid \mathsf{let}\; x\; I_1\; I_2 \mid \mathsf{loop}\; x\; I_1\; I_2 \mid \mathsf{recur}\; I \mid \\ &\qquad \mathsf{if}\; I_1\; I_2\; I_3 \mid I_1 \cdot I_2 \mid \mathsf{send}\; I_1\; I_2 \mid \mathsf{recv}\; I \mid \mathsf{close}\; I \mid \mathsf{chan}\; I \mid I_1 \parallel I_2 \end{aligned}$$

Calculus notation corresponds closely with Clojure notation (Fig. 4), with the exception of application ($I_1\; I_2$), sequencing ($I_1 \cdot I_2$), and threading ($I_1 \parallel I_2$).

The operational semantics of the calculus is defined in terms of labeled reductions of triples $(I, \mathcal{E}, \mathcal{H})$: I is an implementation, $\mathcal{E}$ is a global environment (from variables to values), and $\mathcal{H}$ is a heap (from heap locations to channel states). Channel states are represented as pairs $(\boldsymbol{w}, n)$, where $\boldsymbol{w}$ is a list of values (messages in transit, from left to right), and n the buffer size. Labels, ranged over by α, are of the form $\ell!v$ (send), $\ell?v$ (receive), $\ell\#$ (close), and τ (anything else; we verify only channel actions). The reduction rules are shown in Fig. 8.

Rule [I-Ctxt] executes the first step of implementation I in context $\mathcal{C}$: it first substitutes I for $\Box$ in $\mathcal{C}$ (notation: $\mathcal{C}[I]$), and then executes the first step.

$$\frac{(I,\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I',\mathcal{E}',\mathcal{H}')}{(\mathcal{C}[I],\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (\mathcal{C}[I'],\mathcal{E}',\mathcal{H}')}\,[\text{I-Ctxt}] \qquad \frac{I[v/x] \xrightarrow{\alpha} I'}{((\mathsf{fn}\ x\ I)\ v,\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I',\mathcal{E},\mathcal{H})}\,[\text{I-App}]$$

$$\frac{\mathcal{E}(x) = v}{(x,\mathcal{E},\mathcal{H}) \xrightarrow{\tau} (v,\mathcal{E},\mathcal{H})}\,[\text{I-Var}] \qquad \frac{}{(\mathsf{def}\ x\ v,\mathcal{E},\mathcal{H}) \xrightarrow{\tau} (\mathsf{nil},\mathcal{E}[x \mapsto v],\mathcal{H})}\,[\text{I-Def}]$$

$$\frac{I[v/x] \xrightarrow{\alpha} I'}{(\mathsf{let}\ x\ v\ I,\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I',\mathcal{E},\mathcal{H})}\,[\text{I-Let}] \qquad \frac{I[v/x][(\mathsf{fn}\ x_\mathrm{r}\ (\mathsf{loop}\ x\ x_\mathrm{r}\ I))/\mathsf{recur}\ ] \xrightarrow{\alpha} I'}{(\mathsf{loop}\ x\ v\ I,\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I',\mathcal{E},\mathcal{H})}\,[\text{I-Loop}]$$

$$\frac{v \in \{\mathsf{true},\mathsf{false}\} \qquad (I_v,\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I'_v,\mathcal{E},\mathcal{H})}{(\mathsf{if}\ v\ I_{\mathsf{true}}\ I_{\mathsf{false}},\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I'_v,\mathcal{E},\mathcal{H})}\,[\text{I-If}] \qquad \frac{(I,\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I',\mathcal{E},\mathcal{H})}{(v \cdot I,\mathcal{E},\mathcal{H}) \xrightarrow{\alpha} (I',\mathcal{E},\mathcal{H})}\,[\text{I-Seq}]$$

$$\frac{\mathcal{H}(\ell) = (\boldsymbol{w},n)\ \textbf{and}\ |\boldsymbol{w}| < n}{\begin{array}{c}(\mathsf{send}\ \ell\ v,\mathcal{E},\mathcal{H}) \xrightarrow{\ell!v}\\ (\mathsf{nil},\mathcal{E},\mathcal{H}[\ell \mapsto (v\cdot\boldsymbol{w},n)])\end{array}}\,[\text{I-Send}] \qquad \frac{\mathcal{H}(\ell) = (\boldsymbol{w}\cdot v,n)}{\begin{array}{c}(\mathsf{recv}\ \ell,\mathcal{E},\mathcal{H}) \xrightarrow{\ell?v}\\ (v,\mathcal{E},\mathcal{H}[\ell \mapsto (\boldsymbol{w},n)])\end{array}}\,[\text{I-Recv}]$$

$$\frac{\mathcal{H}(\ell) = (\boldsymbol{w},n)\ \textbf{and}\ n > 0}{\begin{array}{c}(\mathsf{close}\ \ell,\mathcal{E},\mathcal{H}) \xrightarrow{\ell\#}\\ (\mathsf{nil},\mathcal{E},\mathcal{H}[\ell \mapsto (\boldsymbol{w},0)])\end{array}}\,[\text{I-Close}] \qquad \frac{\mathcal{H}(\ell) = \bot\ \textbf{and}\ [\![v]\!] > 0}{\begin{array}{c}(\mathsf{chan}\ v,\mathcal{E},\mathcal{H}) \xrightarrow{\tau}\\ (\ell,\mathcal{E},\mathcal{H}[\ell \mapsto (\epsilon,[\![v]\!])])\end{array}}\,[\text{I-Chan}]$$

Fig. 8. Operational semantics of the implementation calculus

Contexts are generated by the following grammar:

$$\begin{array}{l}\mathcal{C} ::= \Box \mid \mathcal{C}\ I \mid (\mathsf{fn}\ x\ I)\ \mathcal{C} \mid \mathsf{def}\ x\ \mathcal{C} \mid \mathsf{let}\ x\ \mathcal{C}\ I \mid \mathsf{loop}\ x\ \mathcal{C}\ I \mid \mathsf{if}\ \mathcal{C}\ I_\mathrm{t}\ I_\mathrm{f} \mid \mathcal{C} \cdot I \mid \\ \qquad \mathsf{send}\ \mathcal{C}\ I \mid \mathsf{send}\ \ell\ \mathcal{C} \mid \mathsf{recv}\ \mathcal{C} \mid \mathsf{close}\ \mathcal{C} \mid \mathsf{chan}\ \mathcal{C} \mid \mathcal{C} \parallel I \mid I \parallel \mathcal{C}\end{array}$$

Rule [I-App] executes the first step of a function: it first substitutes value v for variable x in body I (notation: $I[v/x]$), and then executes the first step. Rule [I-Var] executes a read in the global environment. Rule [I-Def] executes a write to the global environment (notation: $\mathcal{E}[x \mapsto v]$). Rule [I-Let] executes the first step of a let binder, similar to rule [I-App]. Rule [I-Loop] executes the first step of a loop: it first substitutes value v for variable x (the loop parameter) in body I, then substitutes the loop itself (wrapped in a function to rebind x in the loop's next iteration) for recur, and then executes the first step. Rule [I-If] executes the first step of a branch of a conditional, if the condition is boolean. Rule [I-Seq] executes the first step of the suffix of a sequence, after the prefix has been executed using rule [I-Ctxt]. Rule [I-Send] executes the send through a channel, if that channel exists and is not full. Rule [I-Recv] executes the receive through a channel, if that channel exists and is not empty. Rule [I-Close] executes the close of a channel, if that channel exists and is not yet closed. Rule [I-Chan] executes the creation of a new channel.

Specification calculus. Next, we define a calculus to model Discourje specifications. Let p, q range over roles, f over boolean functions (from the implementation calculus), n, m over number expressions (from the implementation calculus),

$$\frac{}{1{\downarrow}}\,[\text{S}{\downarrow}\text{-One}] \quad \frac{S_{i\in\{1,2\}}{\downarrow}}{S_1 + S_2{\downarrow}}\,[\text{S}{\downarrow}\text{-Alt}] \quad \frac{S_1{\downarrow} \textbf{ and } S_2{\downarrow}}{S_1 \cdot S_2{\downarrow}}\,[\text{S}{\downarrow}\text{-Seq}] \quad \frac{S_1{\downarrow} \textbf{ and } S_2{\downarrow}}{S_1 \parallel S_2{\downarrow}}\,[\text{S}{\downarrow}\text{-Par}]$$

Fig. 9. Operational semantics of the specification calculus (termination)

$$\frac{(f\ v, \emptyset, \emptyset) \xrightarrow{\tau}{}^{*} (\mathsf{true}, \emptyset, \emptyset)}{p[n] \rightarrowtriangle q[m] : f \xrightarrow{p[n]q[m]!v} p[n]q[m]?v \xrightarrow{p[n]q[m]?v} 1}\,[\text{S-Com}] \quad \frac{S = p[n] \not\rightarrowtriangle q[m]}{S \xrightarrow{p[n]q[m]\#} 1}\,[\text{S-Cls}]$$

$$\frac{S_{i\in\{1,2\}} \xrightarrow{\beta} S'}{S_1 + S_2 \xrightarrow{\beta} S'}\,[\text{S-Alt}] \quad \frac{S_1 \xrightarrow{\beta} S_1'}{S_1 \cdot S_2 \xrightarrow{\beta} S_1' \cdot S_2}\,[\text{S-Seq1}] \quad \frac{S_1{\downarrow} \textbf{ and } S_2 \xrightarrow{\beta} S_2'}{S_1 \cdot S_2 \xrightarrow{\beta} S_2'}\,[\text{S-Seq2}]$$

$$\frac{S[\text{fix}\, X\, S/X] \xrightarrow{\beta} S'}{\text{fix}\, X\, S \xrightarrow{\beta} S'}\,[\text{S-Rec}] \quad \frac{S_1 \xrightarrow{\beta} S_1'}{S_1 \parallel S_2 \xrightarrow{\beta} S_1' \parallel S_2}\,[\text{S-Par1}] \quad \frac{S_2 \xrightarrow{\beta} S_2'}{S_1 \parallel S_2 \xrightarrow{\beta} S_1 \parallel S_2'}\,[\text{S-Par2}]$$

$$\frac{S[n/x] \otimes (\ldots \otimes (S[n'{-}1/x] \otimes S[n'/x])) \xrightarrow{\beta} S'}{\bigotimes^{\otimes}_{n \le x \le n'} S \xrightarrow{\beta} S'}\,[\text{S-Rep}]$$

Fig. 10. Operational semantics of the specification calculus (reduction)

and $\otimes$ over $\{+, \cdot, \parallel\}$. The calculus is generated by the following grammar:

$$S ::= \boxed{1} \mid p[n] \rightarrowtriangle q[m] : f \mid \boxed{p[n]q[m]?v} \mid p[n] \not\rightarrowtriangle q[m] \mid S_1 + S_2 \mid S_1 \cdot S_2 \mid$$
$$S_1 \parallel S_2 \mid \text{fix}\, X\, S \mid X \mid \bigotimes^{\otimes}_{n \le x \le n'} S$$

Calculus notation corresponds with Discourje notation (Sect. 2): $p[n] \rightarrowtriangle q[m] : f$ specifies communication of a value that satisfies f from $p[n]$ to $p[m]$; $p[n] \not\rightarrowtriangle q[m]$ specifies closing of the channel from $p[n]$ to $q[m]$; $S_1 \otimes S_2$ specifies the alternative, sequential, and parallel composition of S_1 and S_2; $\text{fix}\, X\, S$ and X specify recursion; and $\bigotimes^{\otimes}_{n \le x \le n'} S$ specifies repetition of S for every value x between n and n', where iterations are composed using $\otimes$. "Boxed" specifications (1 and $p[n]q[m]?v$; the box is not part of the syntax) are *auxiliary* in the sense they are used in defining the operational semantics (below), but they are not written directly in specifications by programmers: 1 specifies a skip; $p[n]q[m]?v$ specifies a receive of v by $q[m]$, previously sent by $p[n]$.

The operational semantics of the calculus is defined in terms of termination predicate $\downarrow$ and labeled reduction relation $\rightarrow$. Labels, ranged over by β, are of the form $p[n]q[m]!v$ (send), $p[n]q[m]?v$ (receive), and $p[n]q[m]\#$ (close). The termination and reduction rules are shown in Figs. 9–10. (This operational semantics coincides with Basic Process Algebra [22], plus free merge, recursion, and repetition.) Rule [S-Com] induces two reductions (first a send, then a receive), via auxiliary specification $p[n]q[m]?v$. We note that the specification calculus has no τ-reductions (which are not monitored; we verify only channel actions). We also note that it can express some, but not all, context-free languages: it can count (using $\bigotimes$), but it cannot encode a stack.

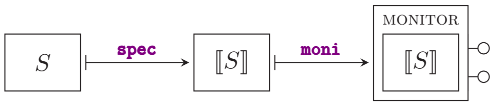


Fig. 11. DSL workflow

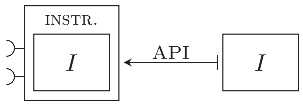


Fig. 12. API workflow

Inclusion relation. Finally, we define a relation to decide if the behavior of an implementation I is included in the behavior of a specification S.

First, let † range over functions from heap locations to sender–receiver pairs; informally, † establishes a correspondence between channel references in the implementation (characterized by their heap locations) and channel references in the specification (characterized by the roles that use them as sender/receiver).

Next, let $\rightarrow_I \subseteq \rightarrow$. We call $\rightarrow_I$ an *execution* of I if it satisfies these conditions:

- $(I, \emptyset, \emptyset) \xrightarrow{\alpha}_I (\hat{I}', \mathcal{E}', \mathcal{H}')$;
- if $(\hat{I}, \mathcal{E}, \mathcal{H}) \xrightarrow{\alpha}_I (\hat{I}', \mathcal{E}', \mathcal{H}')$, then $(\hat{I}', \mathcal{E}', \mathcal{H}') \xrightarrow{\alpha'}_I (\hat{I}'', \mathcal{E}'', \mathcal{H}'')$ or $\hat{I}'$ is a value;
- if $(\hat{I}, \mathcal{E}, \mathcal{H}) \xrightarrow{\alpha_1}_I (\hat{I}'_1, \mathcal{E}'_1, \mathcal{H}'_1)$ and $(\hat{I}, \mathcal{E}, \mathcal{H}) \xrightarrow{\alpha_2}_I (\hat{I}'_2, \mathcal{E}'_2, \mathcal{H}'_2)$, then $\alpha_1 = \alpha_2$ and $(\hat{I}'_1, \mathcal{E}'_1, \mathcal{H}'_1) = (\hat{I}'_2, \mathcal{E}'_2, \mathcal{H}'_2)$.

Finally, a $(\dagger, \rightarrow_I)$-*simulation* $\mathbf{R}$ is a binary relation such that if $(\hat{I}, \mathcal{E}, \mathcal{H}) \xrightarrow{\alpha}_I (\hat{I}', \mathcal{E}', \mathcal{H}')$ and $(\hat{I}, \mathcal{E}, \mathcal{H})\ \mathbf{R}\ S$, then for some S':

- if $\alpha \in \{\ell!v, \ell?v, \ell\#\}$ for some ℓ, v, then $S \xrightarrow{\alpha[\dagger(\ell)/\ell]} S'$ and $(\hat{I}', \mathcal{E}', \mathcal{H}')\ \mathbf{R}\ S'$;
- if $\alpha = \tau$, then $(\hat{I}', \mathcal{E}', \mathcal{H}')\ \mathbf{R}\ S$.

In words, $(\hat{I}, \mathcal{E}, \mathcal{H})\ \mathbf{R}\ S$ iff whenever $\hat{I}$ can reduce to $\hat{I}'$, S can reduce accordingly to S' (and $\hat{I}'$ and S' are again related by $\mathbf{R}$), up to τ-reductions ($\mathbf{R}$ is weak [24]).

Implementation I is *safe* relative to specification S, denoted as $I \preceq S$, if for every execution $\rightarrow_I$ of I, there is a $(\dagger, \rightarrow_I)$-simulation $\mathbf{R}$ such that $(I, \emptyset, \emptyset)\ \mathbf{R}\ S$.

4 Implementation

The DSL. The DSL consists of: Clojure macros to write specifications (cf. syntax of the specification calculus; Sect. 3); Clojure data structures to represent specifications as state machines (cf. operational semantics of the specification calculus); Clojure functions to instantiate these data structures and construct monitors. The workflow is shown in Fig. 11: first, the programmer writes a specification S using the macros; then, at run-time, function **`spec`** is applied to S to expand and evaluate the macros to a data structure $[\![S]\!]$; then, function **`moni`** is applied to $[\![S]\!]$ to construct a monitor.

Essentially, the monitor provides two operations, depicted as "lollipops" in Fig. 11: *checking* if a given channel action α is allowed by $[\![S]\!]$ (formally: $S \xrightarrow{\alpha} S'$ for some S'), and subsequently *updating* $[\![S]\!]$ to its successor. In this way, effectively, the monitor incrementally builds a formal simulation to ensure safety (Sect. 3, page 10). We note that checking/updating is protected by lock-free synchronization (compare-and-set): an α reduction happens only if it was already checked if α is allowed, *and* the state has not yet been updated after that check.

The API. The API consists of Clojure functions that act as proxies for Clojure's own functions to send, receive, close channels, and construct channels. The workflow is shown in Fig. 12: first, the programmer writes an implementation I using Clojure's own functions; then, by loading library `discourje.core.async` instead of `clojure.core.async`, the programmer adds instrumentation to the implementation that allows channel actions to be monitored. As the signatures of Discourje's send, receive, and close functions are identical to Clojure's, adding instrumentation in this way is non-invasive and nearly effortless; the only changes needed, pertain to channel creation (Sect. 2, Fig. 6), since we require the programmer also to specify which roles will use the channel and associate a monitor (this is the practical embodiment of function † in Sect. 3, page 10).[4]

Discourje's send function works as follows. When invoked, first, it waits until the underlying channel c is not full (recall channels in Clojure are bounded and blocking). Then, at time t_1, it calls the monitor associated with c to check if the send is allowed. If yes, at time t_2, it calls the monitor to update accordingly and the "actual send" happens through c; if no, only an exception is thrown. If, between t_1 and t_2, multiple threads call the monitor to update, only one will succeed; the others need to retry from the start. Discourje's receive and close functions work similarly. In this way, Discourje detects safety violations in a way that is both sound (if an exception is thrown, the violating action really was not allowed) and complete (if no exception is thrown, all actions were really allowed).

Extensions. We implemented a number of extensions to the basic framework:

– **Multi-cast:** Adding to Clojure's send, receive, and close functions, the API also contains a multi-cast function to send the same value through $n{>}1$ channels, along with special monitoring support in the DSL (more efficient than monitoring individual communications). Also, the API contains a "multi-receive" function that optionally synchronizes all receivers of a multi-cast.
– **Java interoperability:** Clojure compiles to Java bytecode and runs on the JVM; this enabled us to extend Discourje to Java. Specifically, we wrote a thin Java wrapper around Discourje, so Java programmers can easily construct and use Discourje channels, write specifications, and have them monitored from inside their Java programs, regardless of the threading mechanism (e.g., classical Java threads, thread pools, and parallel streams can be used).

5 Evaluation

General setup. We developed Discourje for two primary usage types:

[4] We currently support the following main channel operations of `clojure.core.async`: sending, receiving, and closing. Discourje works out-of-the-box for all Clojure programs, except those that use unsupported `clojure.core.async` features; mixing Discourje with other concurrency libraries is fine (Sect. 2).

An interesting next step is to also support `clojure.core.async`'s *transducers* (operations on data-in-transit): to our knowledge, no existing work on MPST supports transducers, so supporting those requires significant new theoretical work.

A. as a *testing/debugging tool* for concurrent programs in development, to reliably find/diagnose communication-related concurrency bugs;
B. as a *fail-safe mechanism* for concurrent programs in production, to prevent propagation of spurious results caused by concurrency bugs to end-users (i.e., it is better to throw a runtime error, cf. `ArrayIndexOutOfBoundsException`.)

A key factor that determines Discourje's fitness for purpose is *overhead.* We therefore conducted two kinds of benchmarks: microbenchmarks to study the *scalability* of Discourje and whole-program benchmarks to study the *slowdown* it inflicts relative to unmonitored code.

We used two different hardware configurations to run our benchmarks: VM2 is an instance of the TACAS'20 Artifact Evaluation Virtual Machine for VirtualBox, configured with 2 virtual cores and 8 GB of virtual memory; LISA is a high-end machine with 16 physical cores (Intel Xeon 6130 processor; hyper-threading disabled) and 96 GB of physical memory (far more than needed for our benchmarks). We hosted VM2 on a machine with 4 physical cores (Intel Core i7-8569U; hyper-threading enabled) and 16 GB of physical memory.

Microbenchmarks. In the microbenchmarks, we studied Discourje's scalability under extreme circumstances where threads perform *only* sends/receives and no real computations; this is the worst-case scenario for the lock-free algorithm to synchronize monitor access, as it gives rise to maximal thread contention.

We considered three specifications to investigate the core features/operators offered by the Discourje DSL in isolation, using our built-in common patterns (Fig. 7): `ring` for sequential composition, `one-one-one` (OOO) for alternative composition, and `one-all-one` (OAO) for parallel composition. Each pattern was recursively repeated (i.e., wrapped in `(fix :X [... (fix :X)])`). For Ring and OAO, a *round* consists of 1000 repetitions; for OOO, a round consists of $1000{\cdot}n$ repetitions, where n is the number of worker threads.

For each implementation $I \in \{\text{Ring}, \text{OOO}, \text{OAO}\}$ with $n \in \{2, 4, 6, 8, 10, 12, 14, 16\}$ worker threads,[5] we recorded the mean round latency μ_n^I in eight hours of execution on LISA, the standard deviation σ_n^I, and the coefficient of variation $c_n^I{=}\frac{\mu_n^I}{\sigma_n^I}$. We found $c_n^I \leq 6\%$ for all I and n, except $c_6^{\text{OOO}} = 14\%$ and $c_8^{\text{OOO}} = 8\%$.

As a measure of scalability, we computed normalized means $|\mu_n^I| = \frac{\mu_n^I}{0.5 \cdot n \cdot \mu_2^I}$: this metric is a dimensionless number that indicates the extent to which implementations scale linearly in the number of worker threads, relative to $n = 2$. For instance, if $|\mu_{16}^I| = 1$, I with 16 workers threads is exactly $8\times$ as slow as I with 2 worker threads; this is reasonable, because the worker threads perform $8\times$ more sends and receives in each round (due to the adversarial microbenchmark conditions, the sends and receives are effectively linearized by the monitor, which can check and update at most one channel action at a time).

The normalized means are shown in Fig. 13; our raw data (including standard deviations) are included in our artifact. We summarize the findings:

[5] For Ring, the total number of threads is n; for OOO and OAO, the total number of threads is $n{+}1$ (the master thread).

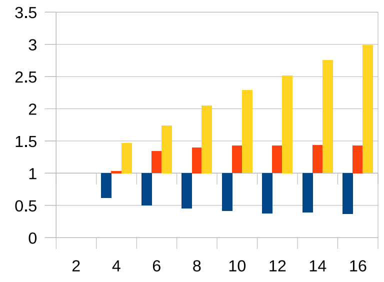


Fig. 13. Microbenchmarks on LISA: Ring (blue), OOO (red), and OAO (yellow); number of threads (x-axis) vs. scalability relative to $n = 2$ (y-axis)

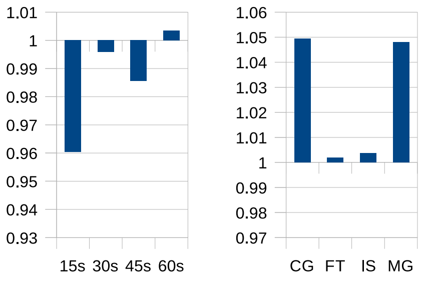


Fig. 14. Whole-program benchmarks on VM2: Chess (left) and NPB (right); play time (x-axis, left) and program (x-axis, right) vs. monitoring slowdown (y-axis)

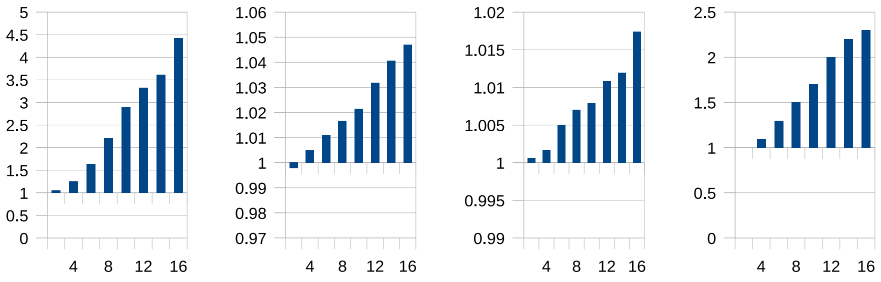


Fig. 15. Whole-program benchmarks on LISA: CG, FT, IS, and MG (from left to right); number of threads (x-axis) vs. monitoring slowdown (y-axis)

– Ring (blue) scales sub-linearly. This is because at any point in time, only one worker thread contends for monitor access (the current receiver or sender; the others are blocked, waiting for incoming channels to become non-empty).
– OOO (red) scales linearly, stabilizing around a constant factor of 1.4. This is because the number of branches in the monitor's internal state machine grows linearly in the number of worker threads. Thus, the cost of using the monitor grows proportionately, but the factor is constant.
– OAO (yellow) scales super-linearly, getting progressively worse as the number of worker threads increases. This is because all worker threads contend for monitor access all the time, *and* the number of branches in the monitor's state machine increases linearly.

To conclude, Ring (which exercises sequential composition) enjoys excellent scalability, while OOO (which exercises alternative composition) enjoys decent scalability, even under the adversarial microbenchmark conditions. Scalability of OAO (parallel composition) can be improved; we discuss one avenue in Sect. 7.

Whole-program benchmarks. In our whole-program benchmarks, we studied Discourje's possible slowdown in five real(istic)/existing concurrent programs:

- **Chess:** Simulates a game of chess between two player threads.
- **Conjugate Gradient (CG-n):** Computes an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros, using the conjugate gradient algorithm, with n worker threads.
- **Fourier Transform (FT-n):** Computes the solution of a partial differential equation, using the forward and inverse Fast Fourier Transform algorithm, with $2{\cdot}n$ worker threads.
- **Integer Sort (IS-n):** Computes a sorted list of uniformly distributed integer keys, using histogram-based integer sorting, with n worker threads.
- **Multi-Grid (MG-n):** Computes an approximate solution u to the discrete Poisson problem $\nabla^2 u = v$, using the V-cycle multigrid algorithm, with $4{\cdot}n$ worker threads.

For Chess, we used Clojure code similar to threads Alice and Bob in Tic-Tac-Toe (Fig. 3), combined with invocations of the open source chess engine Stockfish 10 (https://stockfishchess.org) to compute moves. For CG, FT, IS, and MG, we adapted existing Java implementations from the *NAS Parallel Benchmarks* (NPB) [23] suite, which consists of computational fluid dynamics kernels, by taking advantage of our Java interoperability wrapper (Sect. 4) to replace the monitor-based synchronization used in the original versions.

We also wrote specifications for these implementations in the Discourje DSL. For Chess, the specification is the same as the Tic-Tac-Toe specification (Sect. 2); for CG, FT, IS, and MG, the specifications consist of recursively repeated choices among various instances of the `one-all-one` pattern (each of which involves different subsets of worker threads and message types); the key difference between the specifications, then, is the *frequency* in which repetitions occur.

We recorded execution times of each of the implementations without and with monitoring enabled, using existing/standardized workloads. For Chess, the workload is controlled by the total amount of time each player has to compute its moves during the entire game; we used the four smallest such workloads supported by the open source chess server Lichess (https://lichess.org), namely $\{15, 30, 45, 60\}$ seconds, and we limited games to a maximum of 40 turns per player (*UltraBullet chess*).[6] For CG, FT, IS, and MG, the workload is controlled by the input size; we used the standardized inputs that are predefined by NPB.

We ran Chess on VM2; we ran CG-n, FT-n, IS-n, and MG-n on VM2 for $n = 2$ and on LISA for $n \in \{2, 4, 6, 8, 10, 12, 14, 16\}$. We repeated each of the runs 50 times to smooth out variability; the resulting coefficients of variation are below 5% for CG, FT, IS, and MG, and between 19%–22% for Chess (because moves are not computed deterministically, which affects the number of turns per game). As a measure of slowdown, we computed normalized means of execution times with monitoring, μ_{w}, against those without monitoring, μ_{wo} (i.e., $\frac{\mu_{\mathrm{w}}}{\mu_{\mathrm{wo}}}$): this

[6] We allow concurrent “ponder” computations during opponents' turns.

metric is a dimensionless number that indicates the factor by which monitoring slows down the implementation.

The normalized means are shown in Figs. 14-15; the raw data (including standard deviations) are included in our artifact. We summarize the findings:

- For Chess, for three workloads, slowdowns are <1. As the number of instructions per channel action is, objectively, higher with monitoring than without, we suspect these observed *speedups* might be an artifact of the variability in the measurements. That said, the general trend suggests both usage types of Discourje (page 12) are very well possible for Chess.
- For FT and IS, the slowdowns are low: less than 5% and 2% respectively. This seems low enough not only for Discourje's usage type A (testing/debugging in development), but even usage type B (fail-safe mechanism in production).
- For CG and MG, the slowdowns are higher: less than $5\times$ and $2.5\times$ respectively. Although this might be too much for Discourje's usage type B, it seems low enough for usage type A (cf. the industrial-strength Valgrind tool for memory debugging [35], which typically inflicts a $\geq 10\times$ slowdown).
  The difference in performance between {FT, IS} and {CG, MG} may be explained by the fact the latter are more communication-intensive than the former, so the overhead of monitoring communications is more pronounced.
- For CG, FT, IS, and MG, the slowdowns grow only linearly as the number of threads increases. This shows that the super-linear scalability we observed under the adversarial microbenchmark conditions for the `one-all-one` pattern, does not manifest in these real programs.

To conclude, we believe it is encouraging to see that *even* (extended versions of) the specification that scaled poorest in our microbenchmarks, can give well enough performance in real concurrent programs for both usage types A and B.

6 Related Work

Expressiveness issues of multiparty session types (MPST) have received some attention, but efforts have primarily been geared towards adding more advanced features (e.g., time [5,36], security [7,8,9,13], and parametrisation [14,20,39]); in contrast, restrictions on the usage of core features like choice and interleaving have remained, even though they limit MPST's applicability in practice (e.g., our Tic-Tac-Toe specification cannot be expressed; Fig. 5). Recently, work has been done to improve MPST's expressiveness in this regard using static techniques [31], but our specification language in this paper is still more expressive.

Closest to our work, then, are hybrid MPST approaches that combine static type-checking with a form of distributed runtime monitoring and/or assertion checking [3,4,19,36,37]. In contrast to this paper, however, these dynamic techniques still rely on projection, which limits expressiveness (Sect. 1); none of the specifications in this paper are supported.

Projection-free MPST has also been explored by López et al. [34,43]. Their idea is to specify MPI communication protocols in an MPI-tailored DSL, inspired by MPST, and verify the implementation against the specification using deductive verification tools (VCC [18] and Why3 [21]). However, this approach does not support push-button verification: considerable manual effort is required. In contrast, our approach is fully automated.

We are aware of only two other works that use formal techniques to reason about Clojure programs: Bonnaire-Sergeant et al. [6] formalized the optional type system for Clojure and proved soundness, while Pinzaru et al. [41] developed a translation from Clojure to Boogie [2] to verify Clojure programs annotated with pre/post-conditions. Ours is the first paper that targets concurrency in Clojure.

Verification of shared-memory concurrency with channels has received attention in the context of Go [40,32,33,45]. However, emphasis in these works is on checking deadlock-freedom, liveness, and generic safety properties, while we focus on program-specific protocol compliance. Castro et al. [14] also consider protocol compliance, but their specification language (of global types) is less expressive than ours and does not support this paper's examples.

7 Conclusion

We presented Discourje: a runtime verification framework for channel-based communication protocols in Clojure. Discourje is based on a projection-free interpretation of multiparty session types, trading static type-checking for dynamic runtime monitoring to alleviate expressiveness issues. A key design principle of Discourje has been ergonomics: we aim to make Discourje's use as comfortable as possible. Specifically, programmers can decide to start using Discourje at any stage of development (and doing so requires little effort); Discourje is itself implemented in Clojure (so no need to use a different IDE, learn completely new syntax, or install special compilers); and Discourje can be used seamlessly alongside other concurrency libraries. The framework has a formal foundation, and benchmarks indicate that monitoring overhead can be less than 5% for real/existing concurrent programs. This makes Discourje suitable both as a testing/debugging tool in development, and as a fail-safe mechanism in production.

We list two interesting avenues for future work. First, we want to refine our lock-free synchronization algorithm to enhance the way parallel composition is handled. Second, a much more profound extension pertains to *feedback* and *recovery*. Specifically, we want to explore the idea that whenever a monitor detects a violation, instead of throwing an exception, it should simply *delay* the violating action as a corrective measure, in an attempt to steer the implementation toward safe behavior. When done naively, such delays can easily yield deadlocks, so our plan is to combine this with runtime model-checking/reachability analysis to check if *eventually*, the violating action is allowed (if yes, delay; if no, throw).

Acknowledgments. Funded by the Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103. This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

References

1. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2-3), 95–230 (2016)
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005)
3. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theor. Comput. Sci. **669**, 33–58 (2017)
4. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 162–176. Springer (2010)
5. Bocchi, L., Yang, W., Yoshida, N.: Timed multiparty session types. In: CONCUR. Lecture Notes in Computer Science, vol. 8704, pp. 419–434. Springer (2014)
6. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for clojure. In: ESOP. Lecture Notes in Computer Science, vol. 9632, pp. 68–94. Springer (2016)
7. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Typing access control and secure information flow in sessions. Inf. Comput. **238**, 68–105 (2014)
8. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Information flow safety in multiparty sessions. Mathematical Structures in Computer Science **26**(8), 1352–1394 (2016)
9. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M., Rezk, T.: Session types for access and information flow control. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 237–252. Springer (2010)
10. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: ESOP. Lecture Notes in Computer Science, vol. 4421, pp. 2–17. Springer (2007)
11. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1–8:78 (2012)
12. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multiparty session. Logical Methods in Computer Science **8**(1) (2012)
13. Castellani, I., Dezani-Ciancaglini, M., Pérez, J.A.: Self-adaptation and secure information flow in multiparty communications. Formal Asp. Comput. **28**(4), 669–696 (2016)
14. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. PACMPL **3**(POPL), 29:1–29:30 (2019)
15. Clojure Team: Clojure - State of Clojure 2019 Results (04-02-2019), Accessed 1 September 2019, https://clojure.org/news/2019/02/04/state-of-clojure-2019
16. Clojure Team: Clojure - Clojure core.async Channels (28-06-2013), Accessed 1 September 2019, https://clojure.org/news/2013/06/28/clojure-clore-async-channels
17. Clojure Team: Clojure (nd), Accessed 1 September 2019, https://clojure.org

18. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009)
19. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. Formal Methods in System Design **46**(3), 197–225 (2015)
20. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Logical Methods in Computer Science **8**(4) (2012)
21. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
22. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series, Springer (2000)
23. Frumkin, M.A., Schultz, M.G., Jin, H., Yan, J.C.: Performance and scalability of the NAS parallel benchmarks in java. In: IPDPS. p. 139. IEEE Computer Society (2003)
24. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)
25. Go Team: The Go Programming Language (nd), Accessed 1 September 2019, https://golang.org
26. Hickey, R.: The clojure programming language. In: DLS. p. 1. ACM (2008)
27. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
28. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. Lecture Notes in Computer Science, vol. 9633, pp. 401–418. Springer (2016)
29. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. Lecture Notes in Computer Science, vol. 10202, pp. 116–133. Springer (2017)
30. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
31. Jongmans, S.S., Yoshida, N.: Exploring type-level bisimilarity towards more expressive multiparty session types. In: ESOP 2020 (in press)
32. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: liveness and safety for channel-based programming. In: POPL. pp. 748–761. ACM (2017)
33. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: ICSE. pp. 1137–1148. ACM (2018)
34. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: OOPSLA. pp. 280–298. ACM (2015)
35. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI. pp. 89–100. ACM (2007)
36. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. **29**(5), 877–910 (2017)
37. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: CC. pp. 128–138. ACM (2018)
38. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC. pp. 98–108. ACM (2017)

39. Ng, N., Yoshida, N.: Pabble: parameterised scribble. Service Oriented Computing and Applications **9**(3-4), 269–284 (2015)
40. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: CC. pp. 174–184. ACM (2016)
41. Pinzaru, G., Rivera, V.: Towards static verification of clojure contract-based programs. In: TOOLS. Lecture Notes in Computer Science, vol. 11771, pp. 73–80. Springer (2019)
42. Rust Team: Rust Programming Language (nd), Accessed 1 September 2019, https://rust-lang.org
43. Santos, C., Martins, F., Vasconcelos, V.T.: Deductive verification of parallel programs using why3. In: ICE. EPTCS, vol. 189, pp. 128–142 (2015)
44. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
45. Stadtmüller, K., Sulzmann, M., Thiemann, P.: Static trace-based deadlock analysis for synchronous mini-go. In: APLAS. Lecture Notes in Computer Science, vol. 10017, pp. 116–136 (2016)
46. Tasharofi, S., Dinges, P., Johnson, R.E.: Why do scala developers mix the actor model with other concurrency models? In: ECOOP. Lecture Notes in Computer Science, vol. 7920, pp. 302–326. Springer (2013)
47. Toninho, B., Yoshida, N.: Certifying data in multiparty session types. J. Log. Algebr. Meth. Program. **90**, 61–83 (2017)
48. Tu, T., Liu, X., Song, L., Zhang, Y.: Understanding real-world concurrency bugs in go. In: ASPLOS. pp. 865–878. ACM (2019)

Probabilistic Systems

Scenario-Based Verification of Uncertain MDPs*

Murat Cubuktepe[1], Nils Jansen[2], Sebastian Junges[3], Joost-Pieter Katoen[3], Ufuk Topcu[1]

[1] The University of Texas at Austin, Austin, USA
[2] Radboud University Nijmegen, Nijmegen, The Netherlands
[3] RWTH Aachen University, Aachen, Germany

Abstract. We consider Markov decision processes (MDPs) in which the transition probabilities and rewards belong to an uncertainty set parametrized by a collection of random variables. The probability distributions for these random parameters are unknown. The problem is to compute the probability to satisfy a temporal logic specification within any MDP that corresponds to a sample from these unknown distributions. In general, this problem is undecidable, and we resort to techniques from so-called scenario optimization. Based on a finite number of samples of the uncertain parameters, each of which induces an MDP, the proposed method estimates the probability of satisfying the specification by solving a finite-dimensional convex optimization problem. The number of samples required to obtain a high confidence on this estimate is independent from the number of states and the number of random parameters. Experiments on a large set of benchmarks show that a few thousand samples suffice to obtain high-quality confidence bounds with a high probability.

Keywords: MDP, Uncertainty, Verification, Scenario optimisation

1 Introduction

MDPs. Markov decision processes (MDPs) model sequential decision-making problems in stochastic dynamic environments [51]. They are widely used in areas like planning [52], reinforcement learning [53], formal verification [48], and robotics [24]. Mature model checking tools like PRISM [21] and Storm [35] employ efficient algorithms to verify the correctness of MDPs against temporal logic specifications [2] provided all transition probabilities and cost functions are exactly known. In many applications, however, this assumption may be unrealistic, as certain system parameters are typically not exactly known and under control by external sources.

* Supported by the grants ARL # ACC-APG-RTP W911NF, NASA # 80NSSC19K0209, NSF # 1646522, and NSF # 1652113.

Uncertain MDPs. A common approach to deal with unknown system parameters is to let transition probabilities and cost functions of an MDP belong to uncertainty sets, resulting in so-called *uncertain* MDPs [25,14,28], which generalize *interval* MDPs [20,27,7]. However, solution approaches, e.g., in [25,14,28], usually rely on the potentially limiting assumption that the uncertainty sets at different states of the MDP are independent from each other.

Consider a simple motion planning scenario where an unmanned aerial vehicle (UAV) is tasked to transport a certain payload to a target location. The problem is to compute a policy for the UAV to successfully deliver the payload while taking into account the weather conditions. External factors like wind strength or direction may affect the movement of the UAV. The assumption that such weather conditions are independent between the different possible states of UAV is unrealistic, and does not adequately model the scenario at hand.

For settings in which the uncertainties at different states depend on each other, an option is to account for all possible–albeit infinitely many–values in the uncertainty sets. The policy synthesis problem can be formulated as a so-called semi-infinite convex optimization problem, which includes finitely many variables but infinitely many constraints [28]. This problem, however, is NP-hard [28,18]. Furthermore, it fails to exploit additional information that may be available as random variables over the uncertainty sets [36], and may be very conservative. For instance, weather-data in the form of probability distributions may provide additional information on potential changes during the mission.

In this paper, we study a setting in which the fact the uncertain parameters are random variables and the dependencies between them are accounted for explicitly. Furthermore, each random parameter follows an unknown probability distribution from which we can sample the parameter values.

Problem statement. Compute the probability with which there exists a policy such that a reachability or an expected-cost specification is satisfied for any randomly drawn parameter value.

We call this probability the *satisfaction probability.* The intuition is that the question of whether all (or some) parameter values satisfy a specification—as is often done in parameter synthesis [46]—is replaced by the question of *how much we expect the (sampled) model to satisfy a specification.* For example, a satisfaction probability of 80% tells that, if we randomly sample the parameters, with a probability of 80% there exists a policy for the resulting MDP that satisfies the specification. Computing the satisfaction probability is in general undecidable, even for known probability distributions over the parameter values [37].

Scenario-based verification. Therefore, we resort to *sampling-based* algorithms that yield a confidence (probability) on the bounds of the satisfaction probability. Referring back to the UAV example, we want to compute a confidence probability in the probability that there exists a policy for the UAV to successfully finish the mission. As a first step, we take the aforementioned semi-infinite optimization problem that accounts for all possible parameter values as a basis. Each concrete parameter value is referred to as a *scenario* in the convex optimization literature [15]. For specific problems where a distribution over individual scenarios is present, a technique called *scenario-based optimization* provides guarantees on the satisfaction probability via efficient sampling techniques [15,16]. The basic idea is to consider a finite set of samples from the distribution over the scenarios and restrict the semi-infinite problem to these samples. The resulting convex optimization problem with finitely many constraints can be solved efficiently [50].

For our setting, we first sample a finite number of parameter instantiations each of which induces a concrete MDP. We can solve the synthesis problem for this MDP efficiently using, e.g., a probabilistic model checker. Based on the results, we compute a satisfaction probability and an estimate of its potential error. For example, a 90% estimate in a satisfaction probability of 80%, means that the error is at most 10%. We show that the error in the estimate diminishes to zero exponentially rapidly with increasing number of samples. Moreover, we show that the number of required samples does neither depend on the size of the state space nor the number of random parameters. We validate the theoretical results using several MDPs that have different sizes of state and parameter spaces and demonstrate experimentally that the required number of samples is indeed not sensitive to the dimension of the state and parameter space. In addition, we show the effectiveness of our method with a new dedicated case study based on the aforementioned UAV example which incorporates 2 500 random parameters.

Related work. The so-called *parameter synthesis* problem is concerned with computing parameter values such that there exists a policy in the induced non-parametric MDP that satisfies the specifications. Most of the work in parameter synthesis focus on finding one parameter value that satisfies the specification. The approaches involve computing a rational function of the reachability probabilities [11,17,41], utilizing convex optimization [34,40], and sampling-based methods [26,29]. The problem of whether there exists a value in the parameter space that satisfies a reachability specification is ETR-complete[4] [47], and finding a satisfying parameter value is exponential in the number of parameters.

The work in [45] considers the analysis of Markov models in the presence of uncertain rewards, utilizing statistical methods to reason about the probability of a parametric MDP satisfying an expected cost specification. This approach is restricted to reward parameters and does not explicitly compute confidence bounds. [43] computes bounds on the long-run probability of satisfying a specification with probabilistic uncertainty for Markov chains. Other related techniques include multi-objective model checking to maximize the average performance with probabilistic uncertainty sets [36], sampling-based methods which minimize the *regret* with uncertainty sets [33], and Bayesian reasoning to compute parameter values that satisfy a metric temporal logic specification on a continuous-time Markov chain [38]. [37] considers a variant of the problem in this paper where

[4] The ETR satisfiability problem is to decide if there exists a satisfying assignment to the real variables in a Boolean combination of a set of polynomial inequalities. It is known that NP $\subseteq$ ETR $\subseteq$ PSPACE.

the probability distribution of the uncertainty sets is assumed to be known. The paper formulates the policy synthesis problem as an (undecidable [30]) partially observable Markov decision process (POMDP) synthesis problem and use off-the-shelf point-based POMDP methods [10,6]. The work in [27,25] consider the verification of MDPs with convex uncertainties. However, the uncertainty sets for different states in an MDP are restricted to be independent, which does not hold in our problem setting where we have parameter dependencies.

Uncertainties in MDPs have received quite some attention in the artificial intelligence and planning literatures. Interval MDPs [27,7] use probability intervals in the transition probabilities. Dynamic programming, robust value iteration and robust policy iteration have been developed for MDPs with uncertain transition probabilities whose parameters are statistically independent, also referred to as rectangular, to find a policy ensuring the highest expected total reward at a given confidence level [14,25]. The work in [28] relaxes this independence assumption a bit and determines a policy that satisfies a given performance with a pre-defined confidence provided an observation history of the MDP is given by using conic programming. State-of-the art exact methods can handle models of up to a few hundred of states [42]. Multi-model MDPs [44] treat distributions over probability and cost parameters and aim at finding a single policy maximizing a weighted value function. For deterministic policies this problem is NP-hard, and it is PSPACE-hard for history-dependent policies.

2 Preliminaries

A *probability distribution* over a finite set X is a function $\mu\colon X \to [0,\, 1] \subseteq \mathbb{R}$ with $\sum_{x\in X} \mu(x) = 1$. The set of all distributions on X is denoted by $\mathit{Distr}(X)$. Let $V = \{x_1, \dots, x_n\}$ be a finite set of *parameters* over $\mathbb{R}^n$. The set of polynomials over V is denoted by $\mathbb{Q}[V]$. We denote the cardinality of a set $\mathcal{U}$ by $|\mathcal{U}|$.

2.1 Parametric Models

Definition 1 (pMDP). *A* parametric Markov decision process *(pMDP) $\mathcal{M}$ is a tuple $\mathcal{M} = (S, \mathit{Act}, s_I, V, \mathcal{P})$ with a finite set S of* states*, a finite set Act of* actions*, an* initial state *$s_I \in S$, a finite set V of real-valued variables* (parameters) *and a* transition function *$\mathcal{P}\colon S \times \mathit{Act} \times S \to \mathbb{Q}[V]$.*

For $s \in S$, $\mathit{ActS}(s) = \{\alpha \in \mathit{Act} \mid \exists s' \in S, \mathcal{P}(s,\, \alpha,\, s') \neq 0\}$ is the set of *enabled* actions at s. Without loss of generality, we require $\mathit{ActS}(s) \neq \emptyset$ for $s \in S$. If $|\mathit{ActS}(s)| = 1$ for all $s \in S$, $\mathcal{M}$ is a *parametric discrete-time Markov chain (pMC)*. We denote the transition function for pMCs by $\mathcal{P}(s, s')$.

A pMDP $\mathcal{M}$ is a *Markov decision process (MDP)* if the transition function yields *well-defined* probability distributions, i.e., $\mathcal{P}\colon S \times \mathit{Act} \times S \to [0, 1]$ and $\sum_{s'\in S} \mathcal{P}(s, \alpha, s') = 1$ for all $s \in S$ and $\alpha \in \mathit{ActS}(s)$. We denote the *parameter space of* $\mathcal{M}$ by $\mathcal{V}_\mathcal{M}$. Applying an *instantiation* $u \in \mathcal{V}_\mathcal{M}$ to a pMDP $\mathcal{M}$ yields the *instantiated* MDP $\mathcal{M}[u]$ by replacing each $f \in \mathbb{Q}[V]$ in $\mathcal{M}$ by $f[u]$. An instantiation u is *well-defined* for $\mathcal{M}$ if the resulting model $\mathcal{M}[u]$ is an MDP. We assume that all parameter instantiations in $\mathcal{V}_{\mathcal{M}}$ yield well-defined MDPs. We call u *graph-preserving* if for all $s, s' \in S$ and $\alpha \in \mathit{Act}$ it holds that $\mathcal{P}(s, \alpha, s') \neq 0 \Rightarrow \mathcal{P}(s, \alpha, s')[u] \in (0, 1]$. If $\mathcal{P}(s, \alpha, s') \in \{p, 1-p \mid p \in V\} \cup \mathbb{Q}$, then the parameter space $\mathcal{V}_{\mathcal{M}}$ is given by the rectangle $[0, 1]^{|V|}$. We also consider a state-action cost function $c\colon S \times \mathit{Act} \to \mathbb{Q}[V]$. We denote the set of cost parameters as $\mathcal{W}$.

To define measures on MDPs, nondeterministic choices are resolved by a so-called *policy* $\sigma\colon S \to \mathit{Act}$ with $\sigma(s) \in \mathit{ActS}(s)$. The set of all policies over $\mathcal{M}$ is $\mathit{Str}^{\mathcal{M}}$. For the specifications that we consider in this paper, memoryless deterministic policies are sufficient [48]. Applying a policy to an MDP yields an *induced Markov chain* where all nondeterminism is resolved.

For an MC $\mathcal{D}$, the *reachability specification* $\varphi_r = \mathbf{P}_{\leq \lambda}(\lozenge T)$ asserts that a set $T \subseteq S$ of *target states* is reached with probability at most $\lambda \in [0, 1]$[5]. If φ_r holds for $\mathcal{D}$, we write $\mathcal{D} \models \varphi_r$. Model checking for the more general PCTL [4] or ω-regular specifications is often reducible to checking reachability specifications [48]. For an MDP $\mathcal{M}$, φ_r holds if for all $\sigma \in \mathit{Str}^{\mathcal{M}}$ such that the induced MC $\mathcal{D}$ by the policy σ reaches the set T with a probability of at most λ. For an *expected cost specification* $\varphi_c = \mathrm{EC}_{\leq \kappa}(\lozenge G)$, it holds that $\mathcal{D} \models \varphi_c$ if and only if the expected cost of reaching a set $G \subseteq S$ is at most $\kappa \in \mathbb{R}$. The expected cost of reaching G is well-defined if and only if $\mathbf{P}(\lozenge T) = 1$ for all policies in an MDP.

2.2 Uncertain MDPs

We now introduce the setting that we study in this paper. Specifically, we use parameters to define the uncertainty in the transition probabilities and cost functions of an MDP. Each random parameter follows an unknown probability distribution from which we can sample the parameter values.

Definition 2 (uMDP). *An* uncertain Markov decision process $\mathcal{M}_{\mathbb{P}}$ *(uMDP) is a tuple $\mathcal{M}_{\mathbb{P}} = (\mathcal{M}, \mathbb{P})$ where $\mathcal{M}$ is a pMDP, and $\mathbb{P}$ is a probability distribution over the parameter space $\mathcal{V}_{\mathcal{M}}$. If $\mathcal{M}$ is a pMC, then we call $\mathcal{M}_{\mathbb{P}}$ a uMC.*

Intuitively, a uMDP is a pMDP with an associated distribution over possible (graph-preserving) parameter instantiations. That is, a realization of $\mathbb{P}$ yields a concrete MDP $\mathcal{M}[u]$ with the respective instantiation $u \in \mathcal{V}_{\mathcal{M}}$ (and $\mathbb{P}(u) > 0$).

Remark 1. In a uMDP, we distinguish *controllable* and *uncontrollable* parameters. The uncontrollable parameters follow the probability distribution $\mathbb{P}$. In contrast, we can actively *instantiate* the controllable parameters. In the following, we specifically allow cost parameters to be controllable.

Definition 3 (Satisfaction Probability). *Let $\mathcal{M}_{\mathbb{P}} = (\mathcal{M}, \mathbb{P})$ be a uMDP and φ a specification. The (weighted)* satisfaction probability *of φ is*

$$F(\mathcal{M}_{\mathbb{P}}, \varphi) = \int_{\mathcal{V}_{\mathcal{M}}} I_{\varphi}(u) \, d\,\mathbb{P}(u)$$

[5] The theory also applies to lower bounded properties.

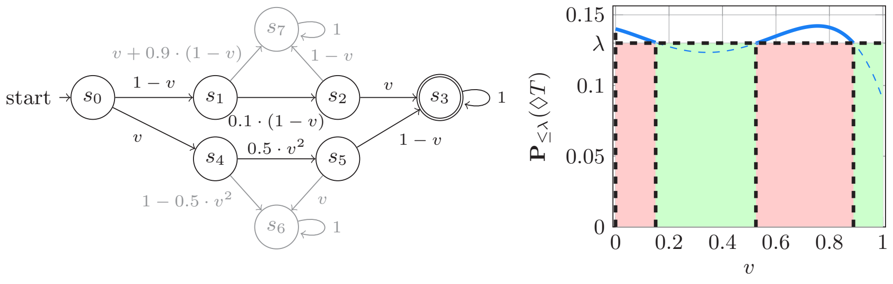


Fig. 1. Left: A uMC with parameter v. Right: The probability of satisfying the reachability specification $\varphi_r = \mathbf{P}_{\leq\lambda}(\Diamond T)$ versus the value of the parameter v. Intervals that satisfy φ_r are green, intervals that violate φ_r are red.

with $u \in \mathcal{V}_\mathcal{M}$ and $I_\varphi\colon \mathcal{V}_\mathcal{M} \to \{0,1\}$ is the indicator for φ, i.e. $I_\varphi(u) = 1$ iff $\mathcal{M}[u] \models \varphi$.

Note that I_φ is measurable, as $\mathcal{V}_\mathcal{M}$ is the finite union of semi-algebraic sets [49]. Moreover, we have that $F(\mathcal{M}_\mathbb{P}, \varphi) \in [0,1]$ and $F(\mathcal{M}_\mathbb{P}, \varphi) + F(\mathcal{M}_\mathbb{P}, \neg\varphi) = 1$.

Example 1. Consider the uMC in the left figure of Fig. 1 with the uncontrollable parameter set $V = \{v\}$, initial state s_0, target set $T = \{s_3\}$ and an uniform distribution for the parameter v over the interval $[0,1]$. We plot the probability of satisfying the specification $\varphi_r = \mathbf{P}_{\leq\lambda}(\Diamond T)$ as a function of v in the right figure of Fig. 1. We also show the satisfying region and its complementary as green and red regions. The satisfying region is given by the union of the intervals $[0.13, 0.525]$ and $[0.89, 1.0]$, and the satisfaction probability $F(\mathcal{M}_\mathbb{P}, \varphi_r)$ is $0.395 + 0.11 = 0.505$.

3 Problem Statement

In this section, we state the problem that we study in this paper. We seek to compute the satisfaction probability of the parameter space for a reachability or an expected cost specification φ on a uMDP. Intuitively, we seek the probability that a randomly sampled instantiation from the parameter space induces an MDP which satisfies φ. Formally: Given a uMDP $\mathcal{M}_\mathbb{P} = (\mathcal{M}, \mathbb{P})$, and a specification φ, compute the satisfaction probability $F(\mathcal{M}_\mathbb{P}, \varphi)$. However, as mentioned, the problem is in general undecidable [37]. Therefore, we consider an approximation of computing the satisfaction probability:

Problem 1. Given a uMDP $\mathcal{M}_\mathbb{P} = (\mathcal{M}, \mathbb{P})$, a reachability specification $\varphi_r = \mathbf{P}_{\leq\lambda}(\Diamond T)$, and a *tolerance probability* ν, compute a confidence probability α_ν such that $F(\mathcal{M}_\mathbb{P}, \varphi_r) \geq 1 - \nu$ holds with a probability of at least $1 - \alpha_\nu$.

We illustrate the problem statement with the following example.

Example 2. For the UAV motion planning example, consider the question "What is the probability on a given day such that there exists a policy for the UAV to successfully finish the mission." A possible result is, e.g., 0.78 (confidence probability: 0.99) and 0.81 (confidence probability: 0.95). Then, with a confidence probability of 0.99, the actual satisfaction probability is indeed greater than 0.78, and with a (slightly lower) confidence probability of 0.95 it is greater than 0.81. Such a result shows that it is quite likely that the UAV will finish the mission successfully with a probability that is at least 81%.

Similar to Problem 1, we also consider expected cost specifications.

Problem 2. Given a uMDP $\mathcal{M}_{\mathbb{P}} = (\mathcal{M}, \mathbb{P})$, and an expected cost specification $\varphi_c = \mathrm{EC}_{\leq \kappa}(\lozenge G)$, a tolerance probability ν, and a confidence probability α_ν determine if there exists an instantiation to the cost parameters such that $F(\mathcal{M}_{\mathbb{P}}, \varphi_c) \geq 1 - \nu$ holds with a probability of at least $1 - \alpha_\nu$.

Remark 2. The main difference between Problem 1 and Problem 2 is that we consider *controllable* cost parameters. We seek to compute an instantiation to these parameters such that the satisfaction probability is greater than $1 - \nu$ with high confidence.

4 Scenario-Based Verification

In this section, we present our approach to solving Problem 1 and 2, that is, to approximate the satisfaction probability with respect to a specification. We first consider the robust policy synthesis problem that accounts *for all possible values* in the uncertainty set, potentially leading to a very pessimistic result. This problem can be formulated as a semi-infinite convex optimization problem, which is NP-hard [28]. Here, we exploit the structure of this problem, which includes finitely many variables but infinitely many constraints. Our approach is based on *scenario optimization* [15,16]: We sample a finite number of parameter values and restrict the semi-infinite problem to these samples. The resulting *finite-dimensional* convex optimization problem can be solved efficiently [50]. Based on the solution of the optimization problem, we compute high confidence in the estimate of the satisfaction probability. The estimate also generalizes to the samples from the probability distribution that are not in the sample set.

Remark 3. For ease of presentation, we focus on uncertain Markov chains (uMCs). Our results and methods generalize to uncertain MDPs (uMDPs).

We first develop the main results for the simple setting where *all sampled* instantiated MCs from the parameter space $\mathcal{V}_\mathcal{D}$ satisfy the reachability specification φ_r. This assumption does not imply that *all* instantiated MCs satisfy φ_r: The sample set does not contain an MC that violates φ_r even though there exists such an MC in the parameter space. In Section 4.2, we drop this assumption and allow sampled points in $\mathcal{V}_\mathcal{D}$ to violate φ_r. This completes our treatment of Problem 1. In Section 4.3, we show how our results generalize to expected cost specifications φ_c, to solve Problem 2.

4.1 Restriction to Satisfying Samples

In this section, we assume that all instantiated MCs satisfy φ_r. We then generalize our method to any values of ν. We want to check if a uMC $\mathcal{D}$ satisfies a reachability specification $\varphi_r = \mathbf{P}_{\leq\lambda}(\lozenge T)$ for all instantiations in the sample set $\mathcal{U}$. For each instantiation, we can formulate a linear program (LP) that is feasible if and only if φ_r is satisfied [51]. For a subset $\mathcal{U} \subseteq \mathcal{V}_\mathcal{D}$ of the parameter space $\mathcal{V}_\mathcal{D}$ of the uMC $\mathcal{D}$, we can then write the conjunction of these LPs. We assume that $|\mathcal{U}|$ is finite and sampled from the probability distribution $\mathbb{P}$ over the parameter space $\mathcal{V}_\mathcal{D}$.

For each instantiation $u \in \mathcal{U}$, we introduce a set of linear constraints that are parametrized by u[6]. We use the following variables. For $s \in S$ and $u \in \mathcal{U}$, the variable $p_s^u \in [0, 1]$ represents the probability of reaching the target set $T \subseteq S$ from state s. The variable τ represents an upper bound on the probability of satisfying φ_r for all instantiations in $\mathcal{U}$. Note that τ is a variable in our formulation, whereas λ is the threshold of the reachability specification, and thus constant. The set $\neg\exists\lozenge T$ represents the set of states which cannot reach the target set T. The probability of reaching T from these states is zero, and the set $\neg\exists\lozenge T$ does not change for different graph-preserving instantiations [17]. The set $\neg\exists\lozenge T$ can be found in polynomial time in the size of a uMC by using standard graph-based search algorithms [48]. We solve the following LP $\mathcal{L}_r(\mathcal{U})$, which is parametrized by each instantiation u in $\mathcal{U}$,

$$\text{minimize } \tau \tag{1}$$

$$\text{subject to} \quad \forall u \in \mathcal{U},$$

$$p_{s_I}^u \leq \tau, \tag{2}$$

$$p_{s_I}^u \leq \lambda, \tag{3}$$

$$p_s^u = 1, \quad \forall s \in T, \tag{4}$$

$$p_s^u = 0, \quad \forall s \in \neg\exists\lozenge T, \tag{5}$$

$$p_s^u = \sum\nolimits_{s' \in S} \mathcal{P}(s, s')[u] \cdot p_{s'}^u, \quad \forall s \in S \setminus (T \cup \neg\exists\lozenge T)\,. \tag{6}$$

The objective (1) minimizes the maximal probability that can be achieved by all MCs induced by $\mathcal{U}$. The constraint (2) represents an upper bound on the reachability probability for all instantiations. We minimize the upper bound to compute the maximal probability of satisfying φ_r for all instantiated MCs. The constraint (3) ensures that the probability of reaching T from the initial state s_I is below the threshold λ. The constraint (4) sets the probability to reach a state in T from T to 1. The constraint (5) sets the reachability probabilities from the states in $\neg\exists\lozenge T$ to zero. The constraint (6) computes the probability of satisfying the specification for each non-target state $s \in S$ in the standard way.

There are infinitely many constraints in the semi-infinite LP $\mathcal{L}_r(\mathcal{V}_\mathcal{D})$ as the cardinality of $(\mathcal{V}_\mathcal{D})$ is infinite and $\mathcal{L}_r(\mathcal{V}_\mathcal{D})$ has infinitely many constraints in the form of (2)–(6). Our approach is based on *scenario optimization* [13,15,16], where

[6] we assume that each sample has a unique index

we instantiate the parameters $u \in \mathcal{V}_{\mathcal{D}}$ by sampling the probability distribution $\mathbb{P}$. Then, for a given violation probability $\nu \in (0,1)$, we compute a solution that violates the constraints in the LP $\mathcal{L}_r(\mathcal{V}_{\mathcal{D}})$ with a probability that is not larger than ν. We first give some properties of the LP $\mathcal{L}_r(\mathcal{U})$.

Theorem 1. *Let uMC $\mathcal{D}$ and the sample sets $\mathcal{U} \subseteq \mathcal{V}_{\mathcal{D}}$ with $K = |\mathcal{U}| \geq 2$. Assume for all $u \in \mathcal{U}$, $\mathcal{D}[u] \models \varphi_r$. For a given* tolerance probability $\nu \in [0,1)$, *let the associated* confidence probability

$$\alpha_\nu = \sum\nolimits_{i=0}^{1} \binom{K}{i} (1-\nu)^{K-i} \nu^i. \tag{7}$$

Then, with a probability of at least $1-\alpha_\nu$, we have

$$F(\mathcal{D}_{\mathbb{P}}, \varphi_r) \geq 1-\nu. \tag{8}$$

Proof. The key idea of the proof is to relate the finite LP $\mathcal{L}_r(\mathcal{U})$ induced by a sampled set $\mathcal{U}$ to the semi-infinite LP $\mathcal{L}_r(\mathcal{V}_{\mathcal{D}})$. Then, we use the results given in [16, Theorem 1] to obtain the lower bound $1-\alpha_\nu$. Let the convex set $C_{\mathcal{U}}^{\mathcal{D}_{\mathbb{P}}}(\lambda,\tau)$ be generated by the set $\mathcal{U}$ according to the probability distribution $\mathbb{P}$ over $\mathcal{V}_{\mathcal{D}}$ as

$$C_{\mathcal{U}}^{\mathcal{D}_{\mathbb{P}}}(\lambda,\tau) = \{(\lambda,\tau) \mid \ \forall u \in \mathcal{U} \text{ satisfying } (2)-(6)\}. \tag{$\star$}$$

The convex set $C_{\mathcal{U}}^{\mathcal{D}_{\mathbb{P}}}(\lambda,\tau)$ constitutes the set of feasible instantiations to the LP $\mathcal{L}_r(\mathcal{U})$ and is exactly in the form of Equation 5 in [16]. Using $C_{\mathcal{U}}^{\mathcal{D}_{\mathbb{P}}}(\lambda,\tau)$, we reformulate $\mathcal{L}_r(\mathcal{U})$ as the convex program

$$\begin{aligned} &\text{minimize } \tau \\ &\text{subject to } (\lambda,\tau) \in \mathcal{C}_{\mathcal{U}}^{\mathcal{D}_{\mathbb{P}}}(\lambda,\tau), \end{aligned} \tag{9}$$

where the last constraint denotes that for a given (λ,τ), the feasible set of $C_{\mathcal{U}}^{\mathcal{D}_{\mathbb{P}}}(\lambda,\tau)$ is not empty, i.e., there exists a feasible solution pair (λ,τ) to the scenario problem $\mathcal{L}_r(\mathcal{U})$. This convex program asserts that all MCs in $\mathcal{U}$ should induce a reachability probability that is less than τ, satisfying the specification φ_r. Moreover, the convex program constitutes a scenario approximation to the so-called *chance-constrained problem* [1]. Such an optimization problem states that the probability of satisfying a (chance) constraint is above a certain threshold:

$$\begin{aligned} &\text{minimize } \tau \\ &\text{subject to } (\lambda,\tau) \in \mathbb{R} \times \mathbb{R}, \\ &\qquad \mathbb{P}\Big((\lambda,\tau) \in \mathcal{C}_{\mathcal{V}_{\mathcal{D}}}^{\mathcal{D}_{\mathbb{P}}}(\lambda,\tau)\Big) \geq 1-\nu. \end{aligned} \tag{10}$$

The chance constraint in (10) ensures that the probability that an instantiation—obtained via distribution $\mathbb{P}$—satisfies the specification φ_r is at least $1-\nu$. Theorem 1 in [16] shows that any feasible solution to the problem in (9) is feasible to the problem in (10) with a confidence probability of $1-\alpha_\nu$, which shows that the violation probability of the solution is at most ν. In our case, the probability of violation is exactly the probability that the instantiated MCs do not satisfy the specification φ_r. Thus, the claim follows. □

Remark 4 (Independence to model size). The confidence probability in Theorem 1 is in fact independent from the number of states, transitions, or random parameters of the uMC. From a practical perspective, the number of samples that are needed for a certain confidence does not depend on the model size.

Finally, Theorem 1 asserts that with a probability of at least $1 - \alpha_\nu$, the next sampled point from $\mathcal{V}_\mathcal{D}$ will satisfy the specification with a probability of at least $1 - \nu$. Note that α_ν is the tail probability of a binomial distribution. It converges exponentially rapidly to 0 in $|\mathcal{U}|$ [16].

4.2 Satisfaction Probability by Treating Violating Samples

Theorem 1 assumes that all sampled points, that is, the induced MCs, satisfy the specification φ_r. This is a severe assumption in general. To lift this assumption, we consider the *discarding approach* from [19]. Specifically, after sampling a set of instantiations $\mathcal{U}$ from $\mathcal{V}_\mathcal{D}$ according to the probability distribution $\mathbb{P}$, we remove the constraints for the MCs that violate the specification φ_r from the LP. We construct the set $\mathcal{R} = \mathcal{U} \setminus \mathcal{Q}$, where $\mathcal{Q}$ denotes the set of samples that induce MCs violating the specification φ_r. Therefore, the set $\mathcal{R}$ denotes the set of sampled MCs that satisfy the specification φ_r. We then solve the LP $\mathcal{L}_r(\mathcal{R})$

$$\begin{aligned} &\text{minimize} \;\; \tau \\ &\text{subject to} \quad \forall u \in \mathcal{R}, \\ &(2) - (6), \end{aligned} \tag{11}$$

where for $u \in \mathcal{R}$ and $s \in S$, p_s^u gives the probability of satisfying the reachability specification of the instantiated MC $\mathcal{D}[u]$ at state s. The other constraints in the optimization problem in LP $\mathcal{L}_r(\mathcal{R})$ are identical to the LP $\mathcal{L}_r(\mathcal{U})$. We give the main result of this section.

Theorem 2. *Let uMC $\mathcal{D}$ and the sample sets $\mathcal{U}, \mathcal{Q} \subseteq \mathcal{V}_\mathcal{D}$, with $K = |\mathcal{U}| \geq 2$ and $L = |\mathcal{Q}|$. For a given* tolerance probability $\nu \in [0, 1)$*, the associated* confidence probability *is*

$$\alpha_\nu = \binom{L+1}{L} \sum\nolimits_{i=0}^{L+1} \binom{K}{i} (1-\nu)^{K-i} \nu^i. \tag{12}$$

Then, with a probability of at least $1 - \alpha_\nu$, we have

$$F(\mathcal{D}_\mathbb{P}, \varphi_r) \geq 1 - \nu. \tag{13}$$

Proof. Similar to the proof of Theorem 1, the main idea is to relate the LP $\mathcal{L}_r(\mathcal{R})$ to the chance-constrained convex problem in (10). Then, we invoke the results from [19, Theorem 1] to get the desired result. Let the convex set $C_\mathcal{R}^{\mathcal{D}_\mathbb{P}}(\lambda, \tau)$, which is generated by the samples in $\mathcal{R}$, be defined by

$$C_\mathcal{R}^{\mathcal{D}_\mathbb{P}}(\lambda, \tau) = \{(\lambda, \tau) \mid \forall u \in \mathcal{R} \text{ such that } (*) \text{ is satisfied}\}.$$

The set $C^{\mathcal{D}_{\mathbb{P}}}_{\mathcal{R}}(\lambda, \tau)$ is in the form of the Definition 2.1 in [16]. We reformulate the LP $\mathcal{L}_r(\mathcal{R})$ as the convex program

$$\begin{aligned} &\text{minimize } \tau \\ &\text{subject to } (\lambda, \tau) \in C^{\mathcal{D}_{\mathbb{P}}}_{\mathcal{R}}(\lambda, \tau). \end{aligned} \tag{14}$$

where the last constraint denotes that the instantiated MCs from the parameter values of the set $\mathcal{R}$ should induce a reachability probability less than τ, and thus, satisfy the specification φ_r. The problem in (14) is a scenario approximation to the problem in (10). Theorem 2.1 in [19] asserts that with a probability of α_ν, the violation probability of the solution is at most ν, which is the probability of violating the specification for the next sample. Similar to Theorem 1, the violation probability ν is the probability that an instantiated MC does not satisfy the specification φ_r. Thus, the claim follows. □

4.3 Expected Cost Specifications

So far, we have focused on parameters that were uncontrollable, and assumed to be random. Now, we consider the case where the cost function c is parametric and the cost parameters are *controllable*. Therefore, the parameters in the cost function are now variables that we can optimize over to satisfy an expected cost specification $\varphi_c = \mathrm{EC}_{\leq\kappa}(\lozenge G)$ for the instantiated MCs. Similar to the previous sections, we assume that we sample a set of instantiations $\mathcal{U}_c$ from the probability distribution $\mathbb{P}$ over the parameter space $\mathcal{V}_\mathcal{D}$. In this case, we modify the LP $\mathcal{L}_r(\mathcal{U})$ to obtain the following LP, which we denote by $\mathcal{L}_c(\mathcal{U}_c)$,

$$\begin{aligned} &\text{minimize} \quad \tau \\ &\text{subject to} \quad \forall u \in \mathcal{U}_c, \\ &c^u_{s_I} \leq \tau, \\ &c^u_{s_I} \leq \kappa, \\ &c^u_s = 0 \quad \forall s \in G, \\ &c^u_s = c(s) + \sum\nolimits_{s' \in S} \mathcal{P}(s, s')[u]\, c^u_{s'} \quad \forall s \in S \setminus G, \end{aligned} \tag{15}$$

where for $s \in S$, $c(s) \in \mathbb{R}^{|\mathcal{W}|}_{\geq 0}$ is the cost function at state s, $|\mathcal{W}|$ is the number of the cost parameters, and for $u \in \mathcal{U}_c$, c^k_s gives the expected cost of reaching the target G of the instantiated MC $\mathcal{D}[k]$ at state s. Note that the cost parameters $\mathcal{W}$ are in the LP $\mathcal{L}_c(\mathcal{U}_c)$ as variables for the parametric cost function,. In the scenario problem (15), we optimize over $c(s)$ and c^k_s to minimize the maximal induced cost of the instantiated MCs. If c is an *affine* function, then the optimization problem $\mathcal{L}_c(\mathcal{U}_c)$ is convex. In this case, the probabilistic properties of the scenario problem are given by the following theorem.

Theorem 3. *Let uMC $\mathcal{D}$ and the sample set $\mathcal{U}_c \subseteq \mathcal{V}_\mathcal{D}$ with $W = |\mathcal{W}|$, and $K = |\mathcal{U}_c| \geq W + 1$. Assume for all $u \in \mathcal{U}_c$, $\mathcal{D}[k] \models \varphi_c$. For a given* tolerance probability *$\nu \in [0, 1)$, let the associated* confidence probability

$$\alpha_\nu = \sum\nolimits_{i=0}^{W+1} \binom{K}{i} (1-\nu)^{K-i} \nu^i. \tag{16}$$

Then, with a probability of at least $1-\alpha_\nu$, *we have* $F(\mathcal{M}_\mathbb{P}, \varphi_c) \geq 1-\nu$.

Proof. Following the proof of Theorem 1, we define the convex set

$$\begin{aligned} C_{\mathcal{U}_c}^{\mathcal{D}_\mathbb{P}}(\kappa,\tau,c) = \Big\{(\kappa,\tau,c) \mid \quad & \forall u \in \mathcal{U}_c \text{ such that} \\ & c_{s_I}^u \leq \tau, \\ & c_{s_I}^u \leq \kappa, \\ & c_s^u = 0 \quad \forall s \in G, \\ & c_s^u = c(s) + \sum\nolimits_{s' \in S} \mathcal{P}(s,s')[u]\; c_{s'}^u \;\; \forall s \in S \setminus G \Big\} \end{aligned}$$

The main difference compared to the proof of Theorem 1 is that we have cost parameters in c as the decision variables and we consider an expected cost specification instead of a reachability specification. Similarly to the proof of Theorem 1, we reformulate the LP $\mathcal{L}_c(\mathcal{U}_c)$ as the following convex problem

$$\begin{aligned} &\text{minimize } \tau \\ &\text{subject to } (\kappa,\tau,c) \in \mathbb{R}\times\mathbb{R}\times\mathbb{R}^{|\mathcal{W}|}, \\ &\qquad\qquad (\kappa,\tau,c) \in C_{\mathcal{U}_c}^{\mathcal{D}_\mathbb{P}}(\kappa,\tau,c). \end{aligned} \tag{17}$$

This convex problem is a scenario approximation to the chance constrained problem given by

$$\begin{aligned} &\text{minimize } \tau \\ &\text{subject to } (\kappa,\tau,c) \in \mathbb{R}\times\mathbb{R}\times\mathbb{R}^{|\mathcal{W}|}, \\ &\qquad\qquad \mathbb{P}\left((\kappa,\tau,c) \in C_{\mathcal{V}_\mathcal{D}}^{\mathcal{D}_\mathbb{P}}(\kappa,\tau,c)\right) \geq 1-\nu. \end{aligned} \tag{18}$$

Therefore, similar to the Theorem 1, we obtain the desired claim. □

We now consider the case that we compute an instantiation of the cost variables, and some of the instantiated MCs satisfy the expected cost specification. We construct the set $\mathcal{R}_c = \mathcal{U}_c \setminus \mathcal{Q}_c$, where $\mathcal{Q}_c$ denotes the set of samples that induce MCs which violate the specification φ_c. For this case, we obtain:

Theorem 4. *Let uMC* $\mathcal{D}$ *and the sample sets* $\mathcal{U}_c, \mathcal{Q}_c \subseteq \mathcal{V}_\mathcal{D}$, *with* $W = |\mathcal{W}|$, $K = |\mathcal{U}_c| \geq 2$ *and* $L = |\mathcal{Q}|$. *For a given* tolerance probability $\nu \in [0,1)$, *let the associated* confidence probability

$$\alpha_\nu = \binom{l+W+1}{l} \sum\nolimits_{i=0}^{l+W+1} \binom{K}{i} (1-\nu)^{K-i} \nu^i. \tag{19}$$

Then, with a probability of at least $1-\alpha_\nu$, *we have* $F(\mathcal{M}_\mathbb{P}, \varphi_c) \geq 1-\nu$.

Proof. The proof is similar to the proofs of Theorem 2 and 3, and omitted. □

4.4 Building Scenario-Based Algorithms

The question remains how we leverage the theoretical results to compute an estimate on the satisfaction probability to solve Problems 1 and 2. For instance, let ν be a violation probability and $\mathcal{U}$ the sample set. Then, we can use Theorem 2 or 4 to compute the confidence probability α_ν by using the discarding approach from [19]. Similarly, for a the sample set $\mathcal{U}$ and a threshold on the confidence probability α_ν we do a *bisection* on ν. Specifically, we repeatedly apply Theorem 2 or 4 for different values of $\nu \in (0, 1)$, to see if the corresponding confidence probability α_ν is below the threshold. We then approximate the lower and upper bounds on ν.

The correctness of the approach is based on scenario-based optimization. However, it also applies to an obtained solution by any procedure [39]. For instance, for any obtained value for the controlled parameters, we can construct a scenario program by sampling from random parameters. We can then apply Theorem 2 or 4 to compute the confidence probability α_ν or the violation probability ν.

Generalization to uMDPs. Recall that we want to compute the satisfaction probability for a uMDP. The probability that for any sampled MDP we are able to synthesize a policy that satisfies the specification φ_r. To generalize our results to uMDPs, we can modify the constraint (6) in the LP $\mathcal{L}_r(\mathcal{U})$ as

$$p_s^u \leq \sum\nolimits_{s' \in S} \mathcal{P}(s, \alpha, s')[u] \cdot p_{s'}^u, \quad \forall s \in S \setminus (T \cup \neg\exists\Diamond T)\,, \;\; \forall \alpha \in ActS(s), \quad (20)$$

asserting that, for each non-target state $s \in S$ and action $\alpha \in ActS(s)$, the probability induced by the *minimizing policy* is an upper bound to the probability variables p_s^u. The reachability specification φ_r is satisfied if and only if the reachability probability at the initial state induced by the minimizing policy is less than λ. We can assert if φ_r is satisfied by combining the constraints (20) with the constraints (2)–(5). Then, our theoretical results apply to the uMDPs.

5 Numerical Examples

We implemented the approach from Section 4 using the model checker Storm [35] to construct and analyze samples of MDPs. To solve the scenario optimization problems with cost parameters, we used the SCS solver [31]. All computations ran on a computer with 8 2.2 GHz cores, and 32 GB of RAM.

We report on a set of well-known benchmarks used in parameter synthesis [46] that are, for instance, available on the website of the tools PARAM [17] or part of the PRISM benchmark suite [23]. Moreover, we created a dedicated case study that is based on the aforementioned UAV example.

5.1 Parameter Synthesis Benchmarks

Setup. In our first set of benchmarks, we adopt parametric MDPs and MCs from [32]. Essentially, the technique from that paper allows to approximate the percentage of instantiations that satisfy (or do not satisfy) a specification. We assume a uniform distribution over the parameter space and set ν equal to the

Table 1. Information for the benchmark instances taken from [32].

				Model Information		Satisfaction Probability	
benchmark	instance	φ	#pars	#states	#trans	sat $(1-\nu)$	unsat (ν)
brp	(256,5)	$\mathbb{P}$	2	19 720	26 627	0.055	0.898
	(16,5)	$\mathbb{E}$	4	1 304	1 731	0.275	0.676
	(32,5)	$\mathbb{E}$	4	2 600	3 459	0.232	0.718
crowds	(10,5)	$\mathbb{P}$	2	104 512	246 082	0.537	0.413
	(20,7)	$\mathbb{P}$	2	45 421 597	164 432 797	0.416	0.534
nand	(10,5)	$\mathbb{P}$	2	35 112	52 647	0.218	0.733
	(25,5)	$\mathbb{P}$	2	865 592	1 347 047	0.206	0.744
consensus	(2,2)	$\mathbb{P}$	2	272	492	0.280	0.669
	(4,2)	$\mathbb{P}$	4	22 656	75 232	0.063	0.888

Table 2. Confidence probabilities α_ν for different numbers of samples.

Samples		100		1,000		10,000		
benchmark	instance	α_ν, sat	α_ν, unsat	α_ν, sat	α_ν, unsat	α_ν, sat	α_ν, unsat	Time (s)
brp	(256,5)	$9.99 \cdot 10^{-2}$	$7.02 \cdot 10^{-1}$	$1.60 \cdot 10^{-2}$	$7.77 \cdot 10^{-2}$	$1.12 \cdot 10^{-6}$	$3.55 \cdot 10^{-6}$	1761.45
	(16,5)	$2.72 \cdot 10^{-1}$	$1.97 \cdot 10^{-1}$	$1.14 \cdot 10^{-1}$	$3.36 \cdot 10^{-2}$	$5.52 \cdot 10^{-6}$	$1.80 \cdot 10^{-8}$	39.76
	(32,5)	$4.01 \cdot 10^{-1}$	$2.95 \cdot 10^{-1}$	$1.39 \cdot 10^{-1}$	$7.76 \cdot 10^{-2}$	$1.24 \cdot 10^{-6}$	$2.63 \cdot 10^{-6}$	78.17
crowds	(10,5)	$2.57 \cdot 10^{-1}$	$3.72 \cdot 10^{-1}$	$1.65 \cdot 10^{-1}$	$1.16 \cdot 10^{-1}$	$9.33 \cdot 10^{-7}$	$8.22 \cdot 10^{-4}$	0.19
	(20,7)	$4.18 \cdot 10^{-1}$	$1.38 \cdot 10^{-1}$	$2.41 \cdot 10^{-1}$	$9.48 \cdot 10^{-2}$	$5.81 \cdot 10^{-5}$	$2.83 \cdot 10^{-5}$	0.45
nand	(10,5)	$3.48 \cdot 10^{-1}$	$2.95 \cdot 10^{-1}$	$3.64 \cdot 10^{-2}$	$3.41 \cdot 10^{-1}$	$2.64 \cdot 10^{-9}$	$1.48 \cdot 10^{-4}$	144.26
	(25,5)	$4.42 \cdot 10^{-1}$	$3.71 \cdot 10^{-1}$	$4.12 \cdot 10^{-2}$	$3.78 \cdot 10^{-1}$	$3.49 \cdot 10^{-6}$	$2.91 \cdot 10^{-4}$	5327.82
consensus	(2,2)	$3.38 \cdot 10^{-1}$	$3.56 \cdot 10^{-1}$	$1.32 \cdot 10^{-1}$	$1.32 \cdot 10^{-1}$	$5.67 \cdot 10^{-7}$	$8.37 \cdot 10^{-4}$	0.72
	(4,2)	$1.79 \cdot 10^{-1}$	$1.41 \cdot 10^{-1}$	$6.51 \cdot 10^{-2}$	$4.75 \cdot 10^{-3}$	$4.26 \cdot 10^{-5}$	$9.29 \cdot 10^{-8}$	300.21

percentage of instantiations that do not satisfy the specification (and vice versa for $1-\nu$). We solve Problem 1 and show that the satisfaction probability is with confidence α_ν as least as high as the approximate satisfaction percentages from [32]. We adapt the *Consensus* protocol [3] and the *Bounded Retransmission Protocol* (brp) [5] to uMDPs; the *Crowds Protocol* (crowds) [12] and the *NAND Multiplexing* benchmark (nand) [8] become uMCs. In Table 1 we list the type of specification checked (φ) and the number of parameters, states, and transitions. We also list the satisfaction probability (as obtained in [32]) for satisfying (sat) and falsifying (unsat) the specification φ.

Results. Table 2 shows the confidence probability α_ν for each benchmark to satisfy and falsify the specification after 100, 1 000 and 10 000 samples from the parameter space. In particular, for each number of samples, we report the average α_ν after running 10 full iterations of the same benchmark. Furthermore, we list the time to solve 1 000 samples for each instance (Time (s)).

The results in Table 2 show that for some benchmarks we get a high confidence probability already after 1 000 samples. For other benchmarks, the confidence probability is still considerably low, for instance considering nand and falsifying the specification. After 10 000 samples, we get a very high confidence in the satisfaction probability for all benchmarks. These results demonstrate that we

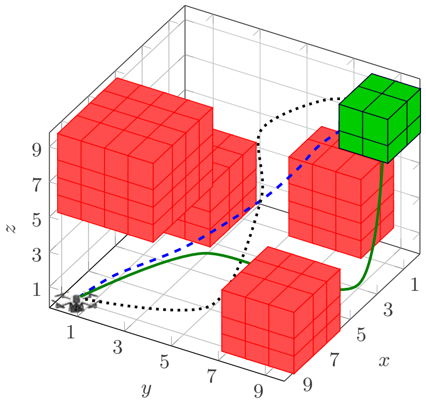


Fig. 2. An example of a 3D UAV benchmark with obstacles and a target area.

can efficiently compute a high confidence in the satisfaction probability. In particular, for the same number of samples, the obtained confidence probabilities are consistent for varying number of states and parameters of the underlying models. Therefore, no dependence on the size of models is shown (see Remark 4).

5.2 UAV Motion Planning

In our second benchmark, we consider the previously mentioned UAV motion planning example to model a realistic problem with a high number of random parameters. We model the problem as a uMDP, where the parameters represent how the weather conditions affect the movement of the UAV, and how the weather may change. In particular, different wind conditions induce specific satisfaction probabilities. We assume that the planning area is a certain valley where we have historic weather data which provide distributions over parameter values. The mission of the UAV is to transport a payload to a specific location and return safely to its initial position. The problem is to compute the satisfaction probability, that is, the probability that for any sampled MDP for this scenario we are able to synthesize a UAV policy that satisfies the specification.

We model the problem as follows: States encode the position of the UAV, the current weather situation, and the general wind direction in the valley. Parameters describe how the weather affects the position of the UAV for different zones in the valley, and how the weather/wind may change during the day. Fig. 2 shows an example environment with zones to avoid (red) and a target zone (green). We define four different weather conditions that each induce certain probability distributions over the eight different wind directions. The parameters of the model determine the probabilities of transitioning between different weather and wind conditions at each time step. The specification is to reach the target zone safely with a probability of at least 0.9. The number of states in our example is 266 880, and the number of parameters is 2 500.

For the distributions over parameter values, that is, over weather conditions, we consider the following cases. First, we assume a uniform distribution over the different weather conditions in each zone. Second, the probability for a weather condition inducing a wind direction that pushes the UAV into the positive y-direction is five times more likely than others. Similarly, in the third case, it is five times more likely to push the UAV into the negative x-direction. We depict some example trajectories of the UAV for three different conditions in Fig. 2. The trajectory given by the blue dashed line represents the expected trajectory for the first case, taking a direct route to reach the target area. Similarly, the trajectories given by the black dotted and solid green lines represent the expected trajectories for the second and third cases. For the second case, we observe that the UAV tries to avoid to get closer to the obstacles in x direction as the wind may push the UAV to the obstacles. For the third case, the UAV avoids the obstacle at the bottom and then reaches the target area.

We sample 1 000 parameters for each case and approximate the maximal satisfaction probability with a confidence probability of at least $1 - \alpha_\nu$, with $\alpha_\nu = 10^{-6}$. The highest satisfaction probability is given by the first weather condition with 0.86, and the other conditions have a satisfaction probability of 0.78 and 0.75, showing that it may be harder to navigate around the obstacles with non-uniform probability distributions. The average time to compute the satisfaction probabilities is 1 341 seconds.

Finally, we introduce costs to a 2-dimensional example, where hitting an obstacle causes (1) a cost of 100 and (2) the UAV to return to the initial position. Specifically, we introduce cost parameters for transitions that steer the UAV towards x or y-directions. We minimize the maximal possible expected cost (under all parameter values) to reach the target location. The specification asserts that the resulting expected cost should be less than 20.

We uniformly sample 1 000 parameter values for weather conditions and note that the UAV policies favor on average transitioning to y-direction more compared to the x-direction to minimize the cost while ensuring that the probability of hitting an obstacle is minimized. The average expected cost of the induced MDPs is 7.41 and the satisfaction probability is 0.71. The solving time for this example is 2 274 seconds.

6 Conclusion

We presented a new sampling-based approach to uncertain Markov models. Theoretically, we showed how to effectively and efficiently approximate the probability that any randomly drawn sample satisfies a temporal logic specification. Furthermore, we showed the computational tractability of our approaches by means of well-known benchmarks and a new, dedicated case study.

In the future, we plan to exploit our approaches for more involved models such as parametric extensions to continuous-time Markov chains [9] or Markov automata [22]. Another line of future work will be a closer integration with a parameter synthesis framework.

References

1. Abraham Charnes and William W Cooper. Chance-Constrained Programming. *Management science*, 6(1):73–79, 1959.
2. Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977.
3. James Aspnes and Maurice Herlihy. Fast Randomized Consensus Using Shared Memory. *Journal of Algorithms*, 15(1):441–460, 1990.
4. Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
5. L. Helmink, M. Sellink, and F. Vaandrager. Proof-Checking a Data Link Protocol. In *TYPES*, volume 806 of *LNCS*, pages 127–165. Springer, 1994.
6. Anthony Cassandra, Michael L Littman, and Nevin L Zhang. Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes. In *UAI*, pages 54–61, 1997.
7. Robert Givan, Sonia Leach, and Thomas Dean. Bounded-Parameter Markov Decision Processes. *Artificial Intelligence*, 122(1-2):71–109, 2000.
8. Jie Han and Pieter Jonker. A System Architecture Solution for Unreliable Nanoelectronic Devices. *IEEE Transactions on Nanotechnology*, 1:201–208, 2002.
9. Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.
10. Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-Based Value Iteration: an Anytime Algorithm for POMDPs. In *IJCAI*, pages 1025–1030, 2003.
11. Conrado Daws. Symbolic and Parametric Model Checking of Discrete-Time Markov chains. In *ICTAC*, volume 3407 of *LNCS*, pages 280–294. Springer, 2004.
12. Vitaly Shmatikov. Probabilistic Analysis of an Anonymity System. *Journal of Computer Security*, 12(3-4):355–377, 2004.
13. G.C. Calafiore and M.C. Campi. Uncertain Convex Programs: Randomized Solutions and Confidence Levels. *Mathematical Programming*, 102(1):25–46, 2005.
14. Arnab Nilim and Laurent El Ghaoui. Robust Control of Markov Decision Processes with Uncertain Transition Matrices. *Operations Research*, 53(5):780–798, 2005.
15. Giuseppe C. Calafiore and Marco C. Campi. The Scenario Approach to Robust Control Design. *IEEE Trans. Automat. Contr.*, 51(5):742–753, 2006.
16. Marco C. Campi and Simone Garatti. The Exact Feasibility of Randomized Solutions of Uncertain Convex Programs. *SIAM Journal on Optimization*, 19(3):1211–1230, 2008.
17. Ernst Moritz Hahn, Holger Hermanns, and Lijun Zhang. Probabilistic Reachability for Parametric Markov Models. *STTT*, 13(1):3–19, 2010.
18. Dimitris Bertsimas, David B Brown, and Constantine Caramanis. Theory and Applications of Robust Optimization. *SIAM review*, 53(3):464–501, 2011.
19. Marco C Campi and Simone Garatti. A Sampling-and-Discarding Approach to Chance-Constrained Optimization: Feasibility and Optimality. *Journal of Optimization Theory and Applications*, 148(2):257–280, 2011.
20. Benoît Delahaye, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, and Andrzej Wasowski. Decision problems for interval Markov chains. In *LATA*, volume 6638 of *LNCS*, pages 274–285. Springer, 2011.
21. Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

22. Hassan Hatefi and Holger Hermanns. Model Checking Algorithms for Markov Automata. *ECEASST*, 53, 2012.
23. Marta Kwiatkowska, Gethin Norman, and David Parker. The PRISM Benchmark Suite. In *QEST*, pages 203–204. IEEE CS, 2012.
24. Rowan McAllister, Thierry Peynot, Robert Fitch, and Salah Sukkarieh. Motion Planning and Stochastic Control with Experimental Validation on a Planetary Rover. In *IROS*, pages 4716–4723. IEEE, 2012.
25. Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Robust Control of Uncertain Markov Decision Processes with Temporal Logic Specifications. In *CDC*, pages 3372–3379, 2012.
26. Taolue Chen, Ernst Moritz Hahn, Tingting Han, Marta Kwiatkowska, Hongyang Qu, and Lijun Zhang. Model Repair for Markov Decision Processes. In *TASE*, pages 85–92. IEEE CS, 2013.
27. Alberto Puggelli, Wenchao Li, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Polynomial-Time Verification of PCTL Properties of MDPs with Convex Uncertainties. In *CAV*, pages 527–542. Springer, 2013.
28. Wolfram Wiesemann, Daniel Kuhn, and Berç Rustem. Robust Markov Decision Processes. *Mathematics of Operations Research*, 38(1):153–183, 2013.
29. Indika Meedeniya, Irene Moser, Aldeida Aleti, and Lars Grunske. Evaluating Probabilistic Models with Uncertain Model Parameters. *Software & Systems Modeling*, 13(4):1395–1415, 2014.
30. Krishnendu Chatterjee, Martin Chmelík, and Mathieu Tracol. What is Decidable about Partially Observable Markov Decision Processes with ω-Regular Objectives. *Journal of Computer and System Sciences*, 82(5):878–911, 2016.
31. B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, June 2016.
32. Tim Quatmann, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Parameter Synthesis for Markov Models: Faster Than Ever. In *ATVA*, volume 9938 of *LNCS*, pages 50–67, 2016.
33. Asrar Ahmed, Pradeep Varakantham, Meghna Lowalekar, Yossiri Adulyasak, and Patrick Jaillet. Sampling Based Approaches for Minimizing Regret in Uncertain Markov Decision Processes (MDPs). *J. Artif. Intell. Res.*, 59:229–264, 2017.
34. Murat Cubuktepe, Nils Jansen, Sebastian Junges, Joost-Pieter Katoen, Ivan Papusha, Hasan A. Poonawala, and Ufuk Topcu. Sequential Convex Programming for the Efficient Verification of Parametric MDPs. In *TACAS (2)*, volume 10206 of *LNCS*, pages 133–150, 2017.
35. Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is Coming: A Modern Probabilistic Model Checker. In *CAV (2)*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.
36. Dimitri Scheftelowitsch, Peter Buchholz, Vahid Hashemi, and Holger Hermanns. Multi-Objective Approaches to Markov Decision Processes with Uncertain Transition Parameters. In *VALUETOOLS*, pages 44–51, 2017.
37. Sebastian Arming, Ezio Bartocci, Krishnendu Chatterjee, Joost-Pieter Katoen, and Ana Sokolova. Parameter-Independent Strategies for pMDPs via POMDPs. In *QEST*, pages 53–70. Springer, 2018.
38. Luca Bortolussi and Simone Silvetti. Bayesian Statistical Parameter Synthesis for Linear Temporal Properties of Stochastic Models. In *TACAS*, pages 396–413, 2018.
39. Marco Claudio Campi, Simone Garatti, and Federico Alessandro Ramponi. A General Scenario Theory for Nonconvex Optimization and Decision Making. *IEEE Trans. Automat. Contr.*, 63(12):4067–4078, 2018.

40. Murat Cubuktepe, Nils Jansen, Sebastian Junges, Joost-Pieter Katoen, and Ufuk Topcu. Synthesis in pmdps: A tale of 1001 parameters. In *ATVA*, volume 11138 of *LNCS*, pages 160–176. Springer, 2018.
41. Paul Gainer, Ernst Moritz Hahn, and Sven Schewe. Incremental Verification of Parametric and Reconfigurable Markov Chains. *CoRR*, abs/1804.01872, 2018.
42. Chin Pang Ho and Marek Petrik. Fast Bellman Updates for Robust MDPs. In *ICML*, 2018.
43. Yamilet R. Serrano Llerena, Marcel Böhme, Marc Brünink, Guoxin Su, and David S. Rosenblum. Verifying the Long-run Behavior of Probabilistic System Models in the Presence of Uncertainty. In *ESEC/SIGSOFT FSE*, pages 587–597. ACM, 2018.
44. Lauren N Steimle, David L Kaufman, and Brian T Denton. Multi-Model Markov Decision Processes. *Optimization Online*, 2018.
45. Giovanni Bacci, Mikkel Hansen, and Kim Guldstrand Larsen. Model Checking Constrained Markov Reward Models with Uncertainties. In *QEST*, pages 37–51, 2019.
46. Sebastian Junges, Erika Ábrahám, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. Parameter Synthesis for Markov Models. *CoRR*, abs/1903.07993, 2019.
47. Tobias Winkler, Sebastian Junges, Guillermo A. Pérez, and Joost-Pieter Katoen. On the complexity of reachability in parametric markov decision processes. In *CONCUR*, volume 140 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
48. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
49. S. Basu, R. Pollack, and M.F. Roy. *Algorithms in Real Algebraic Geometry*. Springer, 2010.
50. Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.
51. Martin L Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.
52. Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2016.
53. Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

Good-for-MDPs Automata for Probabilistic Analysis and Reinforcement Learning*

Ernst Moritz Hahn[1,2], Mateo Perez[3], Sven Schewe[4], Fabio Somenzi[3], Ashutosh Trivedi[3], and Dominik Wojtczak[4]

[1] School of EEECS, Queen's University Belfast, UK
[2] State Key Laboratory of Computer Science, Institute of Software, CAS, PRC
[3] University of Colorado Boulder, USA
[4] University of Liverpool, UK

Abstract. We characterize the class of nondeterministic ω-automata that can be used for the analysis of finite Markov decision processes (MDPs). We call these automata 'good-for-MDPs' (GFM). We show that GFM automata are closed under classic simulation as well as under more powerful simulation relations that leverage properties of optimal control strategies for MDPs. This closure enables us to exploit state-space reduction techniques, such as those based on direct and delayed simulation, that guarantee simulation equivalence. We demonstrate the promise of GFM automata by defining a new class of automata with favorable properties—they are Büchi automata with low branching degree obtained through a simple construction—and show that going beyond limit-deterministic automata may significantly benefit reinforcement learning.

1 Introduction

System specifications are often captured in the form of finite automata over infinite words (ω-automata), which are then used for model checking, synthesis, and learning. Of the commonly-used types of ω-automata, Büchi automata have the simplest acceptance condition, but require nondeterminism to recognize all ω-regular languages. Nondeterministic machines can use unbounded look-ahead to resolve nondeterministic choices. However, important applications—like reactive synthesis or model checking and reinforcement learning (RL) for Markov Decision Process (MDPs [23])—have a game setting, which restrict the resolution of nondeterminism to be based on the past.

Being forced to resolve nondeterminism on the fly, an automaton may end up rejecting words it should accept, so that using it can lead to incorrect results. Due to this difficulty, initial solutions to these problems have been based on deterministic automata—usually with Rabin or parity acceptance conditions. For two-player games, Henzinger and Piterman proposed the notion of *good-for-games (GFG)* automata [15]. These are nondeterministic automata that simulate [21,14,9] a deterministic automaton that recognizes the same language. The existence of a simulation strategy means that nondeterministic choices can be resolved without look-ahead.

* This work has been supported by the National Natural Science Foundation of China (Grant Nr. 61532019), EPSRC grants EP/M027287/1 and EP/P020909/1, and a CU Boulder RIO grant.

The situation is better in the case of probabilistic model checking, because the game for which a strategy is sought is played on an MDP against "blind nature," rather than against a strategic opponent who may take advantage of the automaton's inability to resolve nondeterminism on the fly. As early as 1985, Vardi noted that probabilistic model checking can be performed with Büchi automata endowed with a limited form of nondeterminism [34]. *Limit deterministic Büchi automata (LDBA)* [4,11,29] perform no nondeterministic choice after seeing an accepting transition. Still, they recognize all ω-regular languages and are, under mild restrictions [29], *suitable* for probabilistic model checking.

Related Work. The production of deterministic and limit deterministic automata for model checking has been intensively studied [24,22,1,26,33,32,27,29,8,30,20], and several tools are available to produce different types of automata, incl. MoChiBA/Owl [29,30,20], LTL3BA [1], GOAL [33,32], SPOT [8], Rabinizer [19], and Büchifier [16].

So far, only deterministic and a (slightly restricted [29]) class of limit deterministic automata have been considered for probabilistic model checking [34,4,11,29]. Thus, while there have been advances in the efficient production of such automata [11,29,30,20], the consideration of suitable LDBAs by Courcoubetis and Yannakakis in 1988 [3] has been the last time when a fundamental change in the automata foundation of MDP model checking has occurred.

Contribution. The simple but effective observation that simulation preserves the suitability for MDPs (for both traditional simulation and the AEC simulation we introduce) extends the class of automata that can be used in the analysis of MDPs. This provides us with three advantages: The first advantage is that we can now use a wealth of simulation based statespace reduction techniques [7,31,10,9] on an automaton $\mathcal{A}$ (e.g. an SLDBA) that we would otherwise use for MDP model checking. The second advantage is that we can use $\mathcal{A}$ to check if a different language equivalent automaton, such as an NBA $\mathcal{B}$ (e.g. an NBA from which $\mathcal{A}$ is derived) simulates $\mathcal{A}$. For this second advantage, we can dip into the more powerful class of AEC simulation we define in Section 4 that use properties of winning strategies on finite MDPs. While this is not a complete method for identifying GFM automata, our experimental results indicate that the GFM property is quite frequent for NBAs constructed from random formulas, and can often be established efficiently, while providing a significant statespace reduction and thus offering a significant advantage for model checking.

A third advantage is that we can use the additional flexibility to tailor automata for different applications than model checking, for which specialized automata classes have not yet been developed. We demonstrate this for model-free reinforcement learning (RL). We argue that RL benefits from three properties that are less important in model checking: The first—easy to measure—property is a small number of successors, the second and third, are *cautiousness*, the scope for making wrong decisions, and *forgiveness*, the resilience against making wrong decisions, respectively.

A small number of successors is a simple and natural goal for RL, as the lack of an explicit model means that the product space of a model and an automaton cannot be evaluated backwards. In a forward analysis, it matters that nondeterministic choices have to be modeled by enriching the decisions in the MDPs with the choices made by the automaton. For LDBAs constructed from NBAs, this means guessing a suitable subset of the reachable states when progressing to the deterministic part of the automaton, meaning a number of choices that is exponential in the NBA. We show that we can instead use *slim automata* in Section 3.2 as a first example of NBAs that are good-for-MDPs, but not limit deterministic. They have the appealing property that their branching degree is at most two, while keeping the Büchi acceptance mechanism that works well with RL [12]. (Slim automata can also be used for model checking, but they don't provide similar advantages over suitable LDBAs there, because the backwards analysis used in model checking makes selecting the correct successor trivial.)

Cautiousness and forgiveness are further properties, which are—while harder to quantify—very desirable for RL: LDBAs, for example, suffer from having to make a *correct* choice when moving into the deterministic part of the automaton, and they have to make this correct choice from a very large set of nondeterministic transitions. While this is unproblematic for standard model checking algorithms that are based on backwards analysis, applications like RL that rely on forward analysis can be badly affected when more (wrong) choices are offered, and when wrong choices cannot be rectified. Cautiousness and forgiveness are a references to this: an automaton is more *cautious* if it has less scope for making wrong decisions and more *forgiving* if it allows for correcting previously made decisions (cf. Figure 5 for an example). Our experiments (cf. Section 5) indicate that cautiousness and forgiveness are beneficial for RL.

Organization of the Paper. After the preliminaries, we introduce the "good-for-MDP" property (Section 3) and show that it is preserved by simulation, which enables all minimization techniques that offer the simulation property (Section 3.1). In Section 3.2 we use this observation to construct slim automata—NBAs with a branching degree of 2 that are neither limit deterministic nor good-for-games—as an example of a class of automata that becomes available for MDP model checking and RL. We then introduce a more powerful simulation relation, AEC simulation, that suffices to establish that an automaton is good-for-MDPs (Section 4). In Section 5, we evaluate the impact of the contributions of the paper on model checking and reinforcement learning algorithms.

2 Preliminaries

A *nondeterministic Büchi automaton* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \Delta, \Gamma \rangle$, where Σ is a finite *alphabet*, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $\Delta \subseteq Q \times \Sigma \times Q$ are transitions, and $\Gamma \subseteq Q \times \Sigma \times Q$ is the transition-based *acceptance condition*.

A *run* r of $\mathcal{A}$ on $w \in \Sigma^\omega$ is an ω-word $r_0, w_0, r_1, w_1, \ldots$ in $(Q \times \Sigma)^\omega$ such that $r_0 = q_0$ and, for $i > 0$, it is $(r_{i-1}, w_{i-1}, r_i) \in \Delta$. We write $\inf(r)$ for the set of transitions that appear infinitely often in the run r. A run r of $\mathcal{A}$ is *accepting* if $\inf(r) \cap \Gamma \neq \emptyset$.

The *language*, $L_{\mathcal{A}}$, of $\mathcal{A}$ (or, *recognized* by $\mathcal{A}$) is the subset of words in Σ^ω that have accepting runs in $\mathcal{A}$. A language is *ω-regular* if it is accepted by a Büchi automaton. An automaton $\mathcal{A} = \langle \Sigma, Q, Q_0, \Delta, \Gamma \rangle$ is *deterministic* if $(q, \sigma, q'), (q, \sigma, q'') \in \Delta$ implies $q' = q''$. $\mathcal{A}$ is *complete* if, for all $\sigma \in \Sigma$ and all $q \in Q$, there is a transition $(q, \sigma, q') \in \Delta$. A word in Σ^ω has exactly one run in a deterministic, complete automaton.

A *Markov decision process (MDP)* $\mathcal{M}$ is a tuple (S, A, T, Σ, L) where S is a finite set of states, A is a finite set of *actions*, $T : S \times A \rightharpoondown \mathcal{D}(S)$, where $\mathcal{D}(S)$ is the set of probability distributions over S, is the *probabilistic transition (partial) function*, Σ is an alphabet, and $L : S \times A \times S \to \Sigma$ is the *labeling function* of the set of transitions. For a state $s \in S$, $A(s)$ denotes the set of actions available in s. For states $s, s' \in S$ and $a \in A(s)$, we have that $T(s,a)(s')$ equals $\Pr(s'|s,a)$.

A *run* of $\mathcal{M}$ is an ω-word $s_0, a_1, \ldots \in S \times (A \times S)^\omega$ such that $\Pr(s_{i+1}|s_i, a_{i+1}) > 0$ for all $i \geq 0$. A finite run is a finite such sequence. For a *run* $r = s_0, a_1, s_1, \ldots$ we define the corresponding labeled run as $L(r) = L(s_0, a_1, s_1), L(s_1, a_2, s_2), \ldots \in \Sigma^\omega$. We write $\Omega(\mathcal{M})$ (Paths($\mathcal{M}$)) for the set of runs (finite runs) of $\mathcal{M}$ and $\Omega_s(\mathcal{M})$ (Paths$_s(\mathcal{M})$) for the set of runs (finite runs) of $\mathcal{M}$ starting from state s. When the MDP is clear from the context we drop the argument $\mathcal{M}$.

A strategy in $\mathcal{M}$ is a function $\mu : \text{Paths} \to \mathcal{D}(A)$ such that $\text{supp}(\mu(r)) \subseteq A(\text{last}(r))$, where $\text{supp}(d)$ is the support of d and $\text{last}(r)$ is the last state of r. Let $\Omega_\mu^{\mathcal{M}}(s)$ denote the subset of runs $\Omega^{\mathcal{M}}(s)$ that correspond to strategy μ and initial state s. Let $\Pi_{\mathcal{M}}$ be the set of all strategies. We say that a strategy μ is *pure* if $\mu(r)$ is a point distribution for all runs $r \in \text{Paths}$ and we say that μ is *positional* if $\text{last}(r) = \text{last}(r')$ implies $\mu(r) = \mu(r')$ for all runs $r, r' \in \text{Paths}$. The behavior of an MDP $\mathcal{M}$ under a strategy μ with starting state s is defined on a probability space $(\Omega_s^\mu, \mathcal{F}_s^\mu, \Pr_s^\mu)$ over the set of infinite runs of μ from s.

3 Good-for-MDP (GFM) Automata

Given an MDP $\mathcal{M}$ and an automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \Delta, \Gamma \rangle$, we want to compute an optimal strategy satisfying the objective that the run of $\mathcal{M}$ is in the language of $\mathcal{A}$. We define the semantic satisfaction probability for $\mathcal{A}$ and a strategy μ from state s as:

$$\mathsf{PSem}_{\mathcal{A}}^{\mathcal{M}}(s,\mu) = \Pr{}_s^\mu\{r \in \Omega_s^\mu : L(r) \in L_{\mathcal{A}}\} \text{ and } \mathsf{PSem}_{\mathcal{A}}^{\mathcal{M}}(s) = \sup_{\mu \in \Pi_{\mathcal{M}}} \left(\mathsf{PSem}_{\mathcal{A}}^{\mathcal{M}}(s,\mu)\right).$$

When using automata for the analysis of MDPs, we need a syntactic variant of the acceptance condition. Given an MDP $\mathcal{M} = (S, A, T, \Sigma, L)$ with initial state $s_0 \in S$ and automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \Delta, \Gamma \rangle$, the *product* $\mathcal{M} \times \mathcal{A} = (S \times Q, (s_0, q_0), A \times Q, T^\times, \Gamma^\times)$ is an MDP [17] augmented with an initial state (s_0, q_0) and accepting transitions $\Gamma^\times$. The (partial) function $T^\times : (S \times Q) \times (A \times Q) \rightharpoonup \mathcal{D}(S \times Q)$ is defined by

$$T^\times((s,q),(a,q'))((s',q')) = \begin{cases} T(s,a)(s') & \text{if } (q, L(s,a,s'), q') \in \Delta \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Finally, $\Gamma^\times \subseteq (S \times Q) \times (A \times Q) \times (S \times Q)$ is defined by $((s,q),(a,q'),(s',q')) \in \Gamma^\times$ if, and only if, $(q, L(s,a,s'), q') \in \Gamma$ and $T(s,a)(s') > 0$. A strategy μ on the MDP defines a strategy $\mu^\times$ on the product, and vice versa. We define the syntactic satisfaction probabilities as

$$\mathsf{PSyn}_{\mathcal{A}}^{\mathcal{M}}((s,q),\mu^\times) = \Pr{}_s^\mu\{r \in \Omega_{(s,q)}^{\mu^\times}(\mathcal{M} \times \mathcal{A}) : \inf(r) \cap \Gamma^\times \neq \emptyset\}, \quad \text{and}$$
$$\mathsf{PSyn}_{\mathcal{A}}^{\mathcal{M}}(s) = \sup_{\mu^\times \in \Pi_{\mathcal{M}\times\mathcal{A}}} \left(\mathsf{PSyn}_{\mathcal{A}}^{\mathcal{M}}((s,q_0),\mu^\times)\right).$$

Note that $\mathsf{PSyn}_{\mathcal{A}}^{\mathcal{M}}(s) = \mathsf{PSem}_{\mathcal{A}}^{\mathcal{M}}(s)$ holds for a deterministic $\mathcal{A}$. In general, $\mathsf{PSyn}_{\mathcal{A}}^{\mathcal{M}}(s) \leq \mathsf{PSem}_{\mathcal{A}}^{\mathcal{M}}(s)$ holds, but equality is not guaranteed because the optimal resolution of nondeterministic choices may require access to future events (see Figure 1).

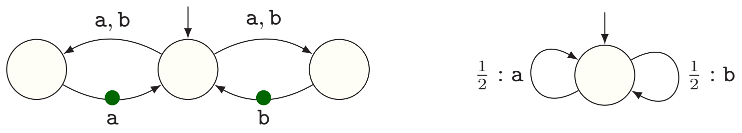


Fig. 1. An NBA, which accepts all words over the alphabet $\{a, b\}$, that is not good for MDPs. The dotted transitions are accepting. For the Markov chain on the right where the probability of a and b is $\frac{1}{2}$, the chance that the automaton makes infinitely many correct predictions is 0

Definition 1 (GFM automata). *An automaton $\mathcal{A}$ is* good for MDPs *if, for all MDPs $\mathcal{M}$, $\mathsf{PSyn}_{\mathcal{A}}^{\mathcal{M}}(s_0) = \mathsf{PSem}_{\mathcal{A}}^{\mathcal{M}}(s_0)$ holds, where s_0 is the initial state of $\mathcal{M}$.*

For an automaton to match $\mathsf{PSem}_{\mathcal{A}}^{\mathcal{M}}(s_0)$, its nondeterminism is restricted not to rely heavily on the future; rather, it must possible to resolve the nondeterminism on-the-fly. For example, the Büchi automaton presented on the left of Figure 1, which has to guess whether the next symbol is a or b, is not good for MDPs, because the simple Markov chain on the right of Figure 1 does not allow resolution of its nondeterminism on-the-fly.

There are three families of automata that are known to be good for MDPs: (1) deterministic automata, (2) good for games automata [15,18], and (3) limit deterministic automata that satisfy a few side constraints [4,11,29].

A *limit-deterministic* Büchi automaton (LDBA) is a nondeterministic Büchi automaton (NBA) $\mathcal{A} = \langle \Sigma, Q_i \cup Q_f, q_0, \Delta, \Gamma \rangle$ such that $Q_i \cap Q_f = \emptyset$; $q_0 \in Q_i$; $\Gamma \subseteq Q_f \times \Sigma \times Q_f$; $(q, \sigma, q'), (q, \sigma, q'') \in \Delta$ and $q, q' \in Q_f$ implies $q' = q''$; and $(q, \sigma, q') \in \Delta$ and $q \in Q_f$ implies $q' \in Q_f$. An LDBA behaves deterministically once it has seen an accepting transition. Usual LDBA constructions [11,29] produce GFM automata. We refer to LDBAs with this property as *suitable* (SLDBAs), cf. Theorem 1.

In the context of RL, techniques based on SLDBAs are particularly useful, because these automata use the Büchi acceptance condition, which can be translated to reachability goals. Good for games and deterministic automata require more complex acceptance conditions, like parity, that do not have a natural translation into rewards [12].

Using SLDBA [4,11,29] has the drawback that they naturally have a high branching degree in the initial part, as they naturally allow for many different transitions to the accepting part of the LDBA. This can be avoided, but to the cost of a blow-up and a more complex construction and data structure [29]. We therefore propose an automata construction that produces NBAs with a small branching degree—it never produces more than two successors. We call these automata *slim*. The resulting automata are not (normally) limit deterministic, but we show that they are good for MDPs.

Due to technical dependencies we start with presenting a second observation, namely that automata that *simulate* language equivalent GFM automata are GFM. As a side result, we observe that the same holds for good-for-games automata. The side result is not surprising, as good-for-games automata were defined through simulation of deterministic automata [15]. But, to the best of our knowledge, the observation from Corollary 1 has not been made yet for good-for-games automata.

3.1 Simulating GFM

An automaton $\mathcal{A}$ *simulates* an automaton $\mathcal{B}$ if the duplicator wins the *simulation game*. The simulation game is played between a duplicator and a spoiler, who each control a pebble, which they move along the edges of $\mathcal{A}$ and $\mathcal{B}$, respectively. The game is started by the spoiler, who places her pebble on an initial state of $\mathcal{B}$. Next, the duplicator puts his pebble on an initial state of $\mathcal{A}$. The two players then take turns, always starting with the spoiler choosing an input letter and a transition for that letter in $\mathcal{B}$, followed by the duplicator choosing a transition for the same letter in $\mathcal{A}$. This way, both players produce an infinite run of their respective automaton. The duplicator has two ways to win a play of the game: if the run of $\mathcal{A}$ he constructs is accepting, and if the run the spoiler constructs on $\mathcal{B}$ is rejecting. The duplicator wins this game if he has a winning strategy, i.e., a recipe to move his pebble that guarantees that he wins. Such a winning strategy is "good-for-games," as it can only rely on the past. It can be used to transform winning strategies of $\mathcal{B}$, so that, if they were witnessing a good for games property or were good for an MDP, then the resulting strategy for $\mathcal{A}$ has the same property.

Lemma 1 (Simulation Properties). *For ω-automata $\mathcal{A}$ and $\mathcal{B}$ the following holds.*

1. *If $\mathcal{A}$ simulates $\mathcal{B}$ then $\mathcal{L}(\mathcal{A}) \supseteq \mathcal{L}(\mathcal{B})$.*
2. *If $\mathcal{A}$ simulates $\mathcal{B}$ and $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.*
3. *If $\mathcal{A}$ simulates $\mathcal{B}$, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$, and $\mathcal{B}$ is GFG, then $\mathcal{A}$ is GFG.*
4. *If $\mathcal{A}$ simulates $\mathcal{B}$, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$, and $\mathcal{B}$ is GFM, then $\mathcal{A}$ is GFM.*

Proof. Facts (1) and (2) are well known observations. Fact (1) holds because an accepting run of $\mathcal{B}$ on a word α can be translated into an accepting run of $\mathcal{A}$ on α by using the winning strategy of $\mathcal{A}$ in the simulation game. Fact (2) follows immediately from Fact (1). Facts (3) and (4) follow by simulating the behaviour of $\mathcal{B}$ on each run. □

This observation allows us to use a family of state-space reduction techniques, in particular those based on language preserving translations for Büchi automata based on simulation relation [7,31,10,9]. This requires stronger notions of simulations, like direct and delayed simulation [9]. For the deterministic part of an LDBA, one can also use space reduction techniques for DBAs like [25].

Corollary 1. *All statespace reduction techniques that turn an NBA $\mathcal{A}$ into an NBA $\mathcal{B}$ that simulates $\mathcal{A}$ preserve GFG and GFM: if $\mathcal{A}$ is GFG or GFM, then $\mathcal{B}$ is GFG or GFM, respectively.*

3.2 Constructing Slim GFM Automata

Let us fix Büchi automaton $\mathcal{B} = \langle \Sigma, Q, Q_0, \Delta, \Gamma \rangle$. We can write Δ as a function $\hat{\delta}\colon Q \times \Sigma \to 2^Q$ with $\hat{\delta}\colon (q,\sigma) \mapsto \{q' \in Q \mid (q,\sigma,q') \in \Delta\}$, which can be lifted to sets, using the deterministic transition function $\delta\colon 2^Q \times \Sigma \to 2^Q$ with $\delta\colon (S,\sigma) \mapsto \bigcup_{q \in S} \hat{\delta}(q,\sigma)$. We also define an operator, ndet, that translates deterministic transition functions $\delta\colon R \times \Sigma \to R$ to relations, using

$$\mathsf{ndet}\colon (R \times \Sigma \to R) \to 2^{R \times \Sigma \times R} \quad \text{with} \quad \mathsf{ndet}\colon \delta \mapsto \big\{(q,\sigma,q') \mid q' \in \delta(\{q\},\sigma)\big\}.$$

This is just an easy means to move back and forth between functions and relations, and helps one to visualize the maximal number of successors. We next define the variations of subset and breakpoint constructions that are used to define the well-known limit deterministic GFM automata—which we use in our proofs—and the slim GFM automata we construct. Let $3^Q := \{(S,S') \mid S' \subsetneq S \subseteq Q\}$ and $3^Q_+ := \{(S,S') \mid S' \subseteq S \subseteq Q\}$. We define the subset notation for the transitions and accepting transitions as $\delta_S, \gamma_S \colon 2^Q \times \Sigma \to 2^Q$ with

$$\delta_S \colon (S,\sigma) \mapsto \{q' \in Q \mid \exists q \in S.\ (q,\sigma,q') \in \Delta\} \text{ and}$$
$$\gamma_S \colon (S,\sigma) \mapsto \{q' \in Q \mid \exists q \in S.\ (q,\sigma,q') \in \Gamma\}.$$

We define the raw breakpoint transitions $\delta_R \colon 3^Q \times \Sigma \to 3^Q_+$ as $((S,S'),\sigma) \mapsto (\delta_S(S,\sigma), \delta_S(S',\sigma) \cup \gamma_S(S,\sigma))$. In this construction, we follow the set of reachable states (first set) and the states that are reachable while passing at least one of the accepting transitions (second set). To turn this into a breakpoint automaton, we reset the second set to the empty set when it equals the first; the transitions where we reset the second set are exactly the accepting ones. The breakpoint automaton $\mathcal{D} = \langle \Sigma, 3^Q, (Q_0, \emptyset), \delta_B, \gamma_B \rangle$ is defined such that, when $\delta_R \colon ((S,S'),\sigma) \mapsto (R,R')$, then there are three cases:

1. if $R = \emptyset$, then $\delta_B((S,S'))$ is undefined (or, if a complete automaton is preferred, maps to a rejecting sink),
2. else, if $R \neq R'$, then $\delta_B \colon ((S,S'),\sigma) \mapsto (R,R')$ is a non-accepting transition,
3. otherwise $\delta_B, \gamma_B \colon ((S,S'),\sigma) \mapsto (R', \emptyset)$ is an accepting transition.

Finally, we define transitions $\Delta_{SB} \subseteq 2^Q \times \Sigma \times 3^Q$ that lead from a subset to a breakpoint construction, and $\gamma_{2,1} \colon 3^Q \times \Sigma \to 3^Q$ that promote the second set of a breakpoint construction to the first set as follows.

1. $\Delta_{SB} = \Big\{ (S, \sigma, (S', \emptyset)) \mid \emptyset \neq S' \subseteq \delta_S(S,\sigma) \Big\}$ are non-accepting transitions,
2. if $\delta_S(S',\sigma) = \gamma_S(S,\sigma) = \emptyset$, then $\gamma_{2,1}((S,S'),\sigma)$ is undefined, and
3. otherwise $\gamma_{2,1} \colon ((S,S'),\sigma) \mapsto (\delta_S(S',\sigma) \cup \gamma_S(S,\sigma), \emptyset)$ is an accepting transition.

We can now define standard limit deterministic good for MDP automata.

Theorem 1. *[11] $\mathcal{A} = \langle \Sigma, 2^Q \cup 3^Q, Q_0, \mathsf{ndet}(\delta_S) \cup \Delta_{SB} \cup \mathsf{ndet}(\delta_B), \mathsf{ndet}(\gamma_B) \rangle$ recognizes the same language as $\mathcal{B}$. It is limit deterministic and good for MDPs.*

We now show how to construct a slim GFM Büchi automaton.

Theorem 2 (Slim GFM Büchi Automaton). *The automaton*

$$\mathcal{S} = \langle \Sigma, 3^Q, (Q_0, \emptyset), \mathsf{ndet}(\delta_B) \cup \mathsf{ndet}(\gamma_{2,1}), \mathsf{ndet}(\gamma_B) \cup \mathsf{ndet}(\gamma_{2,1}) \rangle$$

simulates $\mathcal{A}$. $\mathcal{S}$ is slim, language equivalent to $\mathcal{B}$, and good for MDPs.

Proof. $\mathcal{S}$ is slim: its set of transitions is the union of two sets of deterministic transitions. We show that $\mathcal{S}$ simulates $\mathcal{A}$ by defining a strategy in the simulation game, which ensures that, if the spoiler produces a run $S_0 \ldots S_{j-1}(S_j, S'_j)(S_{j+1}, S'_{j+1}) \ldots$ for $\mathcal{A}$, then the duplicator produces a run $(T_0, T'_0) \dots (T_{j-1}, T'_{j-1})(T_j, T'_j)(T_{j+1}, T'_{j-1}) \dots$ for $\mathcal{S}$, such that (1) $S_i \subseteq T_i$ holds for all $i \in \omega$, and (2) if there are two accepting transitions $\big((S_{k-1}, S'_{k-1}), \sigma_k, (S_k, S'_k)\big)$ and $\big((S_{l-1}, S'_{l-1}), \sigma_l, (S_l, S'_l)\big)$ with $k < l$, there is an $k < m \leq l$, such that $\big((T_{m-1}, T'_{m-1}), \sigma_m(T_m, T'_m)\big)$ is accepting.

To obtain this, we describe a winning strategy for the duplicator while arguing inductively that it mainains (1). Note that (1) holds initially ($T_0 = S_0$, induction basis).

Initial Phase: Every move of the spoiler—with some letter σ—that uses a transition from δ_S—the subset part of $\mathcal{A}$—is followed by a move from δ_B with the same letter σ. When the duplicator follows this strategy the following holds: when, after a pair of moves, the pebble of the spoiler is on state $S \subseteq Q$, then the pebble of the duplicator is on some state (S, S'). In particular, (1) is preserved during this phase (induction step).

Transition Phase: The one spoiler move—with some letter σ—that uses a transition from Δ_{SB}—the transition to the breakpoint part of $\mathcal{A}$—is followed by a move from δ_B with the same letter σ. When the duplicator follows this strategy, and when, after the pair of moves, the pebble of the spoiler is on state $(S, \emptyset)$, then the pebble of the duplicator is on some state (T, T') with $S \subseteq T$. In particular, (1) is preserved (induction step).

Final Phase: When the spoiler moves from some state (S, S')—with some letter σ—that uses a transition from δ_B—the breakpoint part of $\mathcal{A}$—to $(\bar{S}, \bar{S}')$, and when the duplicator is in some state (T, T'), then the duplicator does the following. He calculates $(\bar{T}, \emptyset) = \gamma_{2,1}\big((T, T'), \sigma\big)$ and checks if $\bar{S} \subseteq \bar{T}$ holds. If $\bar{S} \subseteq \bar{T}$ holds, he plays this transition from $\gamma_{2,1}$ (with the same letter σ). Otherwise, he plays the transition from δ_B (with the same letter σ). In either case (1) is preserved (induction step), which closes the inductive argument for (1).

Note that no accepting transition of $\mathcal{A}$ is passed in the initial or tansition phase, so the two accepting transitions from (2) must both fall into the final phase.

To show (2), we first observe that $S'_k = \emptyset$, and thus $S'_k \subseteq T'_k$ holds. Assuming for contradition that all transitions of $\mathcal{S}$ for $\sigma_{k+1} \dots \sigma_{l-1}$ are non-accepting, we obtain—using (1)—by a straightforward inductive argument that $S'_i \subseteq T'_i$ for all i with $k \leq i < l$. (Note that transitions in δ_B are accepting when they are also be in γ_B.)

Using that $S_l = \delta_S(S'_{l-1}, \sigma_l) \cup \gamma_S(S_{l-1}, \sigma_l) \subseteq \delta_S(T'_{l-1}, \sigma_l) \cup \gamma_S(T_{l-1}, \sigma_l)$ holds, the spoiler uses an accepting transition from $\gamma_{2,1}$ in this step.

Using Lemma 1, it now suffices to show that the language of $\mathcal{S}$ is included in the language of $\mathcal{B}$. To show this, we simply argue that an accepting run $\rho = (Q_0, Q'_0), (Q_1, Q'_1), (Q_2, Q'_2), (Q_3, Q'_3), \dots$ of $\mathcal{S}$ on an input word $\alpha = \sigma_0, \sigma_1, \sigma_2, \dots$ can be interpreted as a forest of finitely many finitely branching trees of overall infinite size, where all infinite branches are accepting runs of $\mathcal{B}$. Kőnig's Lemma then proves the existence of an accepting run of $\mathcal{B}$.

This forest is the usual one. The nodes are labeled by states of $\mathcal{B}$, and the roots (level 0) are the initial states of $\mathcal{B}$. Let $I = \big\{i \in \mathbb{N} \mid \big((Q_{i-1}, Q'_{i-1}), \sigma_{i-1}, (Q_i, Q'_i)\big) \in \Gamma := \mathsf{ndet}(\gamma_B) \cup \mathsf{ndet}(\gamma_{2,1})\big\}$ be the set of positions after accepting transitions in ρ. We define the predecessor function $\mathsf{pred}\colon \mathbb{N} \to I \cup \{0\}$ with $\mathsf{pred}\colon i \mapsto \max\big\{j \in I \cup \{0\} \mid j < i\big\}$.

We call a node with label q_l on level l an end-point if one of the following applies: (1) $q_l \notin Q_l$ or (2) $l \in I$ and for all j such that $\mathsf{pred}(l) \leq j < l$, where q_j is the label of the ancestor of this node on level j, we have $(q_j, \sigma_j, q_{j+1}) \notin \Gamma$.

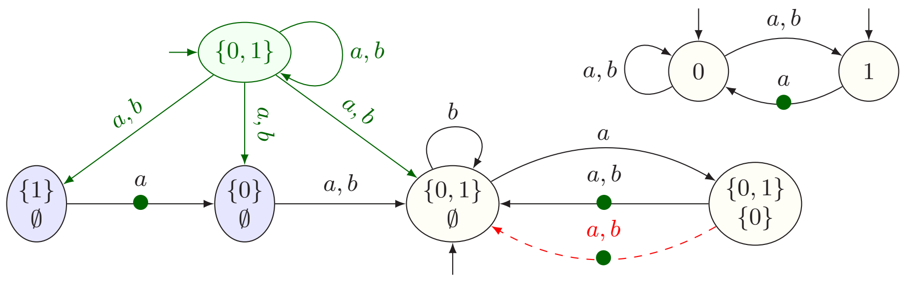


Fig. 2. An NBA for $\mathsf{G}\,\mathsf{F}\,a$ (in the upper right corner) together with an SLDBA and a slim NBA constructed from it. The SLDBA and the slim NBA are shown sharing their common part. State $\{0,1\}$, produced by the subset construction, is the initial state of the SLDBA, while state $(\{0,1\},\emptyset)$—the initial state of the breakpoint construction—is the initial state of the slim NBA. States $(\{1\},\emptyset)$ and $(\{0\},\emptyset)$ are states of the breakpoint construction that only belong to the SLDBA because they are not reachable from $(\{0,1\},\emptyset)$. The transitions out of $\{0,1\}$, except the self loop, belong to Δ_{SB}. The dashed-line transition from $(\{0,1\},\{0\})$ belongs to $\gamma_{2,1}$

(1) may only happen after a transition from $\gamma_{2,1}$ has been taken, and the q_l is not among the states that is traced henceforth. (2) identifies parts of the run tree that do not contain an accepting transition.

A node labeled with q_l on level l that is not an endpoint has $\left|\delta_S(q_l,\sigma_l)\right|$ children, labeled with the different elements of $\delta_S(q_l,\sigma_l)$. It is now easy to show by induction over i that the following holds.

1. For all $q \in Q_i$, there is a node on level i labeled with q.
2. For $i \notin I$ and $q \in Q'_i$, there is a node labeled q on level i, a j with $\mathsf{pred}(i) \leq j < i$, and ancestors on level j and $j+1$ labeled q_j and q_{j+1}, such that $(q_j,\sigma_j,q_{j+1}) \in \Gamma$. (The 'ancestor' on level $j+1$ might be the state itself.)
   For $i \in I$ and $q \in Q'_i$, there is a node labeled q on level i, which is not an end point.

Consequently, the forest is infinite, finitely branching, and finitely rooted, and thus contains an infinite path. By construction, this path is an accepting run of $\mathcal{B}$. □

The resulting automata are simple in structure and enable symbolic implementation (See Fig. 2). It cannot be expected that there are much smaller good for MDP automata, as its explicit construction is the only non-polynomial part in model checking MDPs.

Theorem 3. *Constructing a GFM Büchi automaton G that recognizes the models of an LTL formula φ requires time doubly exponential in φ, and constructing a GFM Büchi automaton G that recognizes the language of an NBA $\mathcal{B}$ requires time exponential in $\mathcal{B}$.*

Proof. As resulting automata are GFM, they can be used to model check MDPs $\mathcal{M}$ against this property, with cost polynomial in product of $\mathcal{M}$ and $\mathcal{G}$. If $\mathcal{G}$ could be produced faster (and if they could, consequently be smaller) than claimed, it will contradict the 2-EXPTIME- and EXPTIME-hardness [4] of these model checking problems. □

4 Accepting End-Component Simulation

An *end-component* [5,2] of an MDP $\mathcal{M}$ is a sub-MDP $\mathcal{M}'$ of $\mathcal{M}$ such that its underlying graph is strongly connected. A *maximal* end-component is maximal under set-inclusion. Every state of an MDP belongs to at most one maximal end-component.

Theorem 4 (End-Component Properties. Theorem 3.1 and Theorem 4.2 of [5]). *Once an end-component C of an MDP is entered, there is a strategy that visits every state-action combination in C infinitely often with probability* 1 *and stays in C forever.*

For a product MDP, an accepting end-component *(AEC) is an end-component that contains some transition in $\Gamma^\times$. There is a positional pure strategy for an AEC C that surely stays in C and almost surely visits a transition in $\Gamma^\times$ infinitely often.*

For a product MDP, there is a set of disjoint accepting end-components such that, from every state, the maximal probability to reach the union of these accepting end-components is the same as the maximal probability to satisfy $\Gamma^\times$. Moreover, this probability can be realized by combining a positional pure (reachability) strategy outside of this union with the aforementioned positional pure strategies for the individual AECs.

Lemma 1 shows that the GFM property is preserved by simulation: For language-equivalent automata $\mathcal{A}$ and $\mathcal{B}$, if $\mathcal{A}$ simulates $\mathcal{B}$ and $\mathcal{B}$ is GFM, then $\mathcal{A}$ is also GFM. However, a GFM automaton may not simulate a language-equivalent GFM automaton. (See Figure 3.) Therefore we introduce a coarser preorder, Accepting End-Component (AEC) simulation, that exploits the finiteness of the MDP $\mathcal{M}$. We rely on Theorem 4 to focus on positional pure strategies for $\mathcal{M} \times \mathcal{B}$. Under such strategies, $\mathcal{M} \times \mathcal{B}$ becomes a Markov chain [2] such that almost all its runs have the following properties:

- They will eventually reach a leaf strongly connected component (LSCC) in the Markov chain.
- If they have reached a LSCC L, then, for all $\ell \in \mathbb{N}$, all sequences of transitions of length ℓ in L occur infinitely often, and no other sequence of length ℓ occurs.

With this in mind, we can intuitively ask the spoiler to pick a run through this Markov chain, and to disclose information about this run. Specifically, we can ask her to signal when she has reached an accepting LSCC[5] in the Markov chain, and to provide information about this LSCC, in particular information entailed by the full list of sequences of transitions of some fixed length ℓ described above. Runs that can be identified to either not reach an accepting LSCC, to visit transitions not in this list, or to visit only a subset of sequences from this list, form a 0 set. In the simulation game we define below, we make use of this observation to discard such runs.

A simulation game can only use the syntactic material of the automata—-neither the MDP nor the strategy are available. The information the spoiler may provide cannot explicitly refer to them. What the spoiler may be asked to provide is information on when she has entered an accepting LSCC, and, once she has signaled this, which sequences of length l of *automata* transitions of $\mathcal{B}$ occur in the LSCC. The sequences of automata transitions are simply the projections on the automata transitions from the

[5] There is nothing to show when a non-accepting LSCC is reached—if $\mathcal{B}$ rejects, then $\mathcal{A}$ may reject too—nor when no LSCC is reached, as this occurs with probability 0.

sequences of transitions of length ℓ that occur in the LSCC L. We call this information a *gold-brim accepting end-component claim* of length ℓ, ℓ-GAEC claim for short.

The term "gold-brim" in the definition indicates that this is a powerful approach, but not one that can be implemented efficiently. We will define weaker, efficiently implementable notions of accepting end-component claims (AEC claims) later.

The AEC simulation game is very similar to the simulation game of Section 3.1. Both players produce an infinite run of their respective automata. If the spoiler makes an AEC claim, e.g., an ℓ-GAEC claim, we say that her run *complies* with it if, starting with the transition when the AEC claim is made, all states, transitions, or sequences of transitions in the claim appear infinitely often, and all states, transitions, and sequences of transitions the claim excludes do not appear. For an ℓ-GAEC claim, this means that all of the sequences of transitions of length ℓ in the claim occur infinitely often, and no other sequence of length ℓ occurs henceforth.

Thus, like a classic simulation game, an ℓ-GAEC simulation game is started by the spoiler, who places her pebble on an initial state of $\mathcal{B}$. Next, the duplicator puts his pebble on an initial state of $\mathcal{A}$. The two players then take turns, always starting with the spoiler choosing an input letter and an according transition from $\mathcal{B}$, followed by the duplicator choosing a transition for the same letter in $\mathcal{A}$.

Different from the classic simulation game, in an ℓ-GAEC simulation game, the spoiler has an additional move that she can (and, in order to win, has to) perform once in the game: In addition to choosing a letter and a transition, she can claim that she has reached an accepting end-component, and provide a complete list of sequences of automata transitions of length ℓ that can henceforth occur. This store is maintained, and never updated. It has no further effect on the rules of the game: Both players produce an infinite run of their respective automata. The duplicator has four ways to win:

1. if the spoiler never makes an AEC claim,
2. if the run of $\mathcal{A}$ he constructs is accepting,
3. if the run the spoiler constructs on $\mathcal{B}$ does not comply with the AEC claim, and
4. if the run that the spoiler produces is not accepting.

For ℓ-GAEC claims, (4) simply means that the set of transitions defined by the sequences does not satisfy the Büchi, parity, or Rabin acceptance condition.

Theorem 5. *[ℓ-GAEC Simulation] If $\mathcal{A}$ and $\mathcal{B}$ are language equivalent automata, $\mathcal{B}$ is GFM, and there exists an ℓ such that $\mathcal{A}$ ℓ-GAEC simulates $\mathcal{B}$, then $\mathcal{A}$ is GFM.*

For the proof, we use an arbitrary (but fixed) MDP $\mathcal{M}$, and an arbitrary (but fixed) pure optimal positional strategy μ for $\mathcal{M} \times \mathcal{B}$, resulting in the Markov chain $(\mathcal{M} \times \mathcal{B})_\mu$. We assume w.l.o.g. that the accepting LSCCs in $(\mathcal{M} \times \mathcal{B})_\mu$ are identified, e.g., by a bit.

Let τ be a winning strategy of the duplicator in an ℓ-GAEC simulation game. Abusing notation, we let $\tau \circ \mu$ denote the finite-memory strategy[6] obtained from μ and τ for $\mathcal{M} \times \mathcal{A}$, where τ is acting only on the automata part of $(\mathcal{M} \times \mathcal{B})$, and where the spoiler

[6] The strategy τ consists of one sub-strategy to be used before the AEC claim is made and one sub-strategy for each possible ℓ-GAEC claim. The memory of $\tau \circ \mu$ tracks the position in $(\mathcal{M} \times \mathcal{B})_\mu$. When an accepting LSCC is detected (via the marker bit) analysis of $(\mathcal{M} \times \mathcal{B})_\mu$ reveals the only possible ℓ-GAEC claim. This claim is used to select the right entry from τ.

makes the move to the end-component when she is in some LSCC B of $(\mathcal{M} \times \mathcal{B})_\mu$ and gives the full list of sequences of transitions of length ℓ that occur in B.

Proof. As $\mathcal{B}$ is good for MDPs, we only have to show that the chance of winning in $(\mathcal{M} \times \mathcal{A})_{\tau\circ\mu}$ is at least the chance of winning in $(\mathcal{M} \times \mathcal{B})_\mu$. The chance of winning in $(\mathcal{M} \times \mathcal{B})_\mu$ is the chance of reaching an accepting LSCC in $(\mathcal{M} \times \mathcal{B})_\mu$. It is also the chance of reaching an accepting LSCC $L \in (\mathcal{M} \times \mathcal{B})_\mu$ *and*, after reaching L, to see exactly the sequences of transitions of length ℓ that occur in L, and to see all of them infinitely often.

By construction, $\tau \circ \mu$ will translate those runs into accepting runs of $(\mathcal{M} \times \mathcal{A})_{\tau\circ\mu}$, such that the chance of an accepting run of $(\mathcal{M} \times \mathcal{A})_{\tau\circ\mu}$ is at least the chance of an accepting run of $(\mathcal{M} \times \mathcal{B})_\mu$. As μ is optimal, the chance of winning in $\mathcal{M} \times \mathcal{A}$ is at least the chance of winning in $\mathcal{M} \times \mathcal{B}$. As $\mathcal{B}$ is GFM, this is the chance of $\mathcal{M}$ producing a run accepted by $\mathcal{B}$ (and thus $\mathcal{A}$) when controlled optimally, which is an upper bound on the chance of winning in $\mathcal{M} \times \mathcal{A}$. $\square$

An ℓ-GAEC simulation, especially for large ℓ, results in very large state spaces, because the spoiler has to list all sequences of transitions of $\mathcal{B}$ of length ℓ that will appear infinitely often. No other sequence of length ℓ may then appear in the run[7]. This can, of course, be prohibitively expensive.

As a compromise, one can use coarser-grained information at the cost of reducing the duplicator's ability of winning the game. E.g., the spoiler could be asked to only reveal a transition that is repeated infinitely often, plus (when using more powerful acceptance conditions than Büchi), some acceptance information, say the dominating priority in a parity game or a winning Rabin pair. This type of coarse-grained claim can be refined slightly by allowing the *duplicator* to change at any time the transition that is to appear infinitely often to the transition just used by the spoiler. Generally, we say that an AEC simulation game is any simulation game, where

- the spoiler provides a list of states, transitions, or sequences of transitions that will occur infinitely often and a list of states, transitions, or sequences of transitions that will not occur in the future when making her AEC claim, and
- the duplicator may be able to update this list based on his observations,
- there exists some ℓ-GAEC simulation game such that a winning strategy of the spoiler translates into a winning strategy of the spoiler in the AEC simulation game.

The requirement that a winning spoiler strategy translates into a winning spoiler strategy in an ℓ-GAEC game entails that AEC simulation games can prove the GFM property.

Corollary 2. *[AEC Simulation] If $\mathcal{A}$ and $\mathcal{B}$ are language equivalent automata, $\mathcal{B}$ is good for MDPs, and $\mathcal{A}$ AEC-simulates $\mathcal{B}$, then $\mathcal{A}$ is good for MDPs.*

[7] The AEC claim provides information about the accepting LSCC in the product under the chosen pure positional strategy. When the AEC claim requires the exclusion of states, transitions, or sequences of transitions, then they are therefore surely excluded, whereas when it requires inclusion of, and thus inclusion of infinitely many occurrances of, states, trasitions, or sequences of transitions, then they (only) occur almost surely infinitely often. Yet, runs that do not contain them all infinitely often form a zero set, and can thus be ignored.

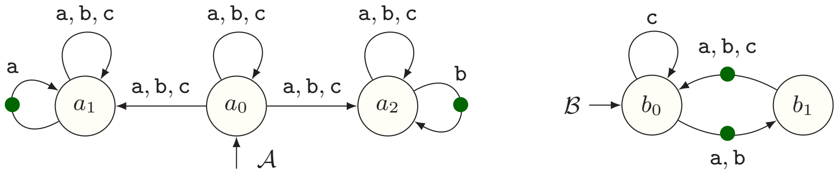


Fig. 3. Automata $\mathcal{A}$ (left) and $\mathcal{B}$ (right) for $\varphi = (\mathsf{G}\,\mathsf{F}\,\mathtt{a}) \vee (\mathsf{G}\,\mathsf{F}\,\mathtt{b})$. The dotted transitions are accepting. The NBA $\mathcal{A}$ does not simulate the DBA $\mathcal{B}$: $\mathcal{B}$ can play a's until $\mathcal{A}$ moves to either the state on the left, or the state on the right. $\mathcal{B}$ then wins by henceforth playing only b's or only a's. However, $\mathcal{A}$ is good for MDPs. It wins the AEC simulation game by waiting until an AEC is reached (by $\mathcal{B}$), and then check if a or b occurs infinitely often in this AEC. Based on this knowledge, $\mathcal{A}$ can make its decision. This can be shown by AEC simulation if $\mathcal{B}$ has to provide sufficient information, such as a list of transitions—or even a list of letters—that occur infinitely often. The amount of information the spoiler has to provide determines the strength of the AEC simulation used. If, e.g., $\mathcal{B}$ only has to reveal one accepting transition of the end-component, then it can select an end-component where the revealed transition is (b_1, c, b_0), which does not provide sufficient information. Whereas, if the duplicator is allowed to update the transition, then the duplicator wins by updating the recorded transition to the next a or b transition

Of course, for every AEC simulation, one first has to prove that winning strategies for the spoiler translate. We have used two simple variations of the AEC simulation games:

accepting transition: the spoiler may only make her AEC claim when taking an accepting transition; this transition—and no other information—is stored, and the spoiler commits to—and commits only to—seeing this transition infinitely often;

accepting transition with update: different to the *accepting transition* AEC simulation game, the duplicator can—but does not have to—update the stored accepting transition whenever the spoiler passes by an accepting transition.

Theorem 6. *Both, the* accepted transition *and the* accepted transition with update *AEC simulation, can be used to establish the good for MDPs property.*

To show this, we describe the strategy translations in accordance with Corollary 2.

Proof. In both cases, the translation of a winning strategy of the spoiler for the 1-GAEC simulation game are straightforward: The spoiler essentially follows her winning strategy from the 1-GAEC simulation game, with the extra rule that she will make her AEC claim to the duplicator on the first accepting transition on or after her AEC claim in the 1-GAEC claim. If the duplicator is allowed to update the transition, this information is ignored by the spoiler—she plays according to her winning strategy from the 1-GAEC simulation game. Naturally, the resulting play will comply with her 1-GAEC claim, and will thus also be winning for the—weaker—AEC claim made to the duplicator. □

We use AEC simulation to identify GFM automata among the automata produced (e.g., by SPOT [8]) at the beginning of the transformation. Figure 3 shows an example for which the duplicator wins the AEC simulation game, but loses the ordinary simulation game. Candidates for automata to simulate are, e.g., the slim GFM Büchi automata and the limit deterministic Büchi automata discussed above.

5 Evaluation

5.1 Size of General Büchi Automata for Probabilistic Model Checking

As discussed, automata that simulate slim automata or SLDBAs are good for MDPs. This fact can be used to allow Büchi automata produced from general-purpose tools such as SPOT's [8] ltl2tgba rather than using specialized automata types. Automata produced by such tools are often smaller because such general-purpose tools are highly optimized and not restricted to producing slim or limit deterministic automata. Thus, one produces an arbitrary Büchi automaton using any available method, then transforms this automaton into a slim or limit deterministic automaton, and finally checks whether the original automaton simulates the generated one.

We have evaluated this idea on random LTL formulas produced by SPOT's tool randltl. We have set the tree size, which influences the size of the formulas, to 50, and have produced 1000 formulas with 4 atomic propositions each. We left the other values to their defaults. We have then used SPOT's ltl2tgba (version 2.7) to turn these formulas into non-generalized Büchi automata using default options. Finally, for each automaton, we have used our tool to check whether the automaton simulates a limit deterministic automaton that we produce from this automaton. For comparison, we have also used Owl's [29] tool ltl2ldba (version 19.06.03) to compute limit deterministic non-generalized Buchi automata. We have also used the option of this tool to compute Büchi automata with a nondeterministic initial part. We used 10 minute timeouts.

Of these 1000 formulas, 315 can be transformed to deterministic Büchi automata. For an additional 103 other automata generated, standard simulation sufficed to show that they are GFM. For a further 11 of them, the simplest AEC simulation (the spoiler chooses an accepting transition to occur infinitely often) sufficed, and another 1 could be classed GFM by allowing the duplicator to update the transition. 501 automata turned out to be nonsimulatable and for 69 we did not get a decision due to a timeout.

For the LTL formulas for which ltl2tgba could not produce deterministic automata, but for which simulation could be shown, the number of states in the generated automata was often lower than the number of states in the automata produced by Owl's tools. On average, the number of states per automaton was ≈15.21 for SPOT's ltl2tgba; while for Owl's ltl2ldba it was ≈46.35. The extended version of this paper [13] contains more details about the evaluation.

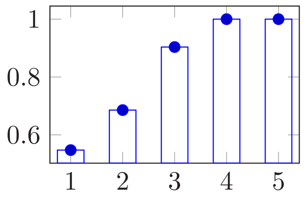


Fig. 4. Deciles ratio ltl2tgba /semi-deterministic automata

Let us consider the ratio between the size of automata produced by ltl2tgba and the size of semi-deterministic automata produced by Owl. The average of this number for all automata that are not deterministic and that can be simulated in some way is ≈ 1.0335. This means that on average, for these automata, the semi-deterministic automata are slightly smaller. If we take a look at the first 5 deciles depicted in Fig. 4, we see that there is a large number of formulas for which ltl2tgba and Owl produce automata of the same size. For around 24.3478% of the cases, automata by SPOT are smaller than those produced by Owl (ratio < 1).

5.2 GFM Automata and Reinforcement Learning

SLDBAs have been used in [12] for model-free reinforcement learning of ω-regular objectives. While the Büchi acceptance condition allows for a faithful translation of the objective to a scalar reward, the agent has to learn how to control the automaton's nondeterministic choices; that is, the agent has to learn when the SLDBA should cross from the initial component to the accepting component to produce a successful run of a behavior that satisfies the given objective.

Any GFM automaton with a Büchi acceptance condition can be used instead of an SLDBA in the approach of [12]. While in many cases SLDBAs work well, GFM automata that are not limit-deterministic may provide a significant advantage.

Early during training, the agent relies on uniform random choices to discover policies that lead to successful episodes. This includes randomly resolving the automaton nondeterminism. If random choices are unlikely to produce successful runs of the automaton in case of behaviors that should be accepted, learning is hampered because good behaviors are not rewarded. Therefore, GFM automata that are more likely to accept under random choices will result in the agent learning more quickly. We have found the following properties of GFM automata to affect the agent's learning ability.

Low branching degree. A low branching degree presents the agent with fewer alternatives, reducing the expected number of trials before the agent finds a good combination of choices. Consider an MDP and an automaton that require a specific sequence of k nondeterministic choices in order for the automaton to accept. If at each choice there are b equiprobable options, the correct sequence is obtained with probability b^{-k}.

Cautiousness. An automaton that enables fewer nondeterministic choices for the same finite input word gives the agent fewer chances to choose wrong. The slim automata construction has the interesting property of "collecting hints of acceptance" before a nondeterministic choice is enabled because S' has to be nonempty for a $\gamma_{2,1}$ transition to be present and that requires going through at least one accepting transition.

Forgiveness. Mistakes made in resolving nondeterminism may be irrecoverable. This is often true of SLDBAs meant for model checking, in which jumps are made to select a subformula to be eventually satisfied. However, general GFM automata, thanks also to their less constrained structure, may be constructed to "forgive mistakes" by giving more chances of picking a successful run.

Figure 5 compares a typical SLDBA to an automaton that is not limit-deterministic and is not produced by the breakpoint construction, but is proved GFM by AEC simulation. This latter automaton has a nondeterministic choice in state q_0 on letter $x \wedge \neg y$ that can be made an unbounded number of times. The agent may choose q_1 repeatedly even if eventually $\mathsf{F}\,\mathsf{G}\,x$ is false and $\mathsf{G}\,\mathsf{F}\,y$ is true. With the SLDBA, on the other hand, there is no room for error.

A Case Study. We compared the effectiveness in learning to control a cart-pole model of three automata for the property $\big((\mathsf{F}\,\mathsf{G}\,x) \vee (\mathsf{G}\,\mathsf{F}\,y)\big) \wedge \mathsf{G}\,\mathtt{safe}$. The safety component of the objective is to keep the pole balanced and the cart on the track. The left two thirds of the track alternate between x and y at each step. The right third is always labeled y, but in order to reach it, the cart has to cross a barrier, with probability $1/3$ of failing.

The three automata are an SLDBA (4 states), a slim automaton (8 states), and a handcrafted forgiving automaton (4 states) similar to the one of Fig. 5.

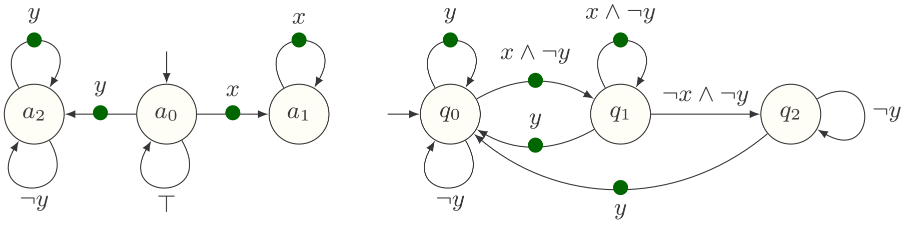


Fig. 5. Two GFM automata for $(\mathsf{F}\,\mathsf{G}\,x) \vee (\mathsf{G}\,\mathsf{F}\,y)$. SLDBA (left), and forgiving (right)

Training of the continuous-statespace model employed PPO [28] as implemented in OpenAI Baselines [6]. Figure 6 shows the learning curves for the three automata averaged over ten runs. They underline the importance of choosing the right automaton in RL. Training parameters, more details on the model, and additional examples can be found in the extended version of this paper [13].

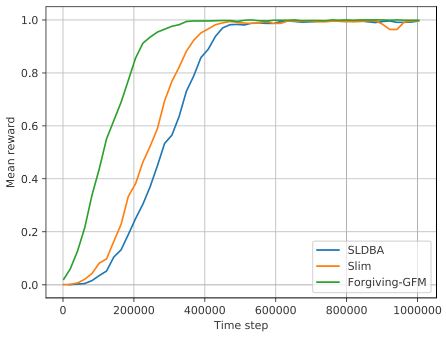


Fig. 6. Learning curves

6 Conclusion

We have defined the class of automata that are *good for MDPs*—nondeterministic automata that can be used for the analysis of MDPs—and shown it to be closed under different simulation relations. This has multiple favorable implications for model checking and reinforcement learning. Closure under classic simulation opens a rich toolbox of statespace reduction techniques that come in handy to push the boundary of analysis techniques, while the more powerful (and more expensive) AEC simulation has promise to identify source automata that happen to be good for MDPs.

The wider class of GFM automata also shows promise: the slim automata we have defined to tame the branching degree while retaining the desirable Büchi condition for reinforcement learning are able to compete even against optimized SLDBAs.

As outlined in Section 5.2, a low branching degree, cautiousness, and forgiveness make automata particularly well-suited for learning. From a practical point of view, much of the power of this new approach is in harnessing the power of simulation for learning, and forgiveness is closely related to simulation.

The natural follow-up research is to tap the full potential of simulation-based statespace reduction instead of the limited version that we have implemented. Besides using this to get the statespace small—useful for model checking—we will use simulation to construct forgiving automata, which is promising for reinforcement learning.

Datasets generated and analyzed during the current study are available at: https://doi.org/10.6084/m9.figshare.11882739 [35,36]

References

1. T. Babiak, M. Křetínský, V. Rehák, and J. Strejcek. LTL to Büchi automata translation: Fast and more deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 95–109, 2012.
2. Ch. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Foundations of Computer Science*, pages 338–345. IEEE, 1988.
4. C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–907, July 1995.
5. L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1998.
6. P. Dhariwal, Ch. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.
7. D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relations. In *Computer Aided Verification*, pages 255–265, July 1991. LNCS 575.
8. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 - A framework for LTL and ω-automata manipulation. In *Automated Technology for Verification and Analysis*, pages 122–129, 2016.
9. K. Etessami, T. Wilke, and R. A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. *SIAM J. Comput.*, 34(5):1159–1175, 2005.
10. S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In *Computer Aided Verification (CAV'02)*, pages 610–623, July 2002. LNCS 2404.
11. E. M. Hahn, G. Li, S. Schewe, A. Turrini, and L. Zhang. Lazy probabilistic model checking without determinisation. In *Concurrency Theory*, pages 354–367, 2015.
12. E. M. Hahn, M. Perez, S. Schewe, F. Somenzi, A. Trivedi, and D. Wojtczak. Omega-regular objectives in model-free reinforcement learning. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 395–412, 2019. LNCS 11427.
13. E. M. Hahn, M. Perez, F. Somenzi, A. Trivedi, S. Schewe, and D. Wojtczak. Good-for-MDPs automata. *arXiv e-prints*, abs/1909.05081, September 2019.
14. T. Henzinger, O. Kupferman, and S. Rajamani. Fair simulation. In *Concurrency Theory*, pages 273–287, 1997. LNCS 1243.
15. T. A. Henzinger and N. Piterman. Solving games without determinization. In *Computer Science Logic*, pages 394–409, September 2006. LNCS 4207.
16. D. Kini and M. Viswanathan. Optimal translation of LTL to limit deterministic automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 113–129, 2017.
17. J. Klein, D. Müller, Ch. Baier, and S. Klüppelholz. Are good-for-games automata good for probabilistic model checking? In *Language and Automata Theory and Applications*, pages 453–465. Springer, 2014.
18. J. Klein, D. Müller, Ch. Baier, and S. Klüppelholz. Are good-for-games automata good for probabilistic model checking? In *Language and Automata Theory and Applications*, pages 453–465, 2014.
19. J. Křetínský, T. Meggendorfer, S. Sickert, and Ch. Ziegler. Rabinizer 4: from LTL to your favourite deterministic automaton. In *Computer Aided Verification*, pages 567–577. Springer, 2018.
20. J. Křetínský, T. Meggendorfer, and S. Sickert. Owl: A library for ω-words, automata, and LTL. In *Automated Technology for Verification and Analysis*, pages 543–550, 2018.
21. R. Milner. An algebraic definition of simulation between programs. *Int. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

22. N. Piterman. From deterministic Büchi and Streett automata to deterministic parity automata. *Logical Methods in Computer Science*, 3(3):1–21, 2007.
23. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, NY, USA, 1994.
24. S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, March 1989.
25. S. Schewe. Beyond hyper-minimisation—minimising DBAs and DPAs is NP-complete. In *Foundations of Software Technology and Theoretical Computer Science, FSTTCS*, pages 400–411, 2010.
26. S. Schewe and T. Varghese. Tight bounds for the determinisation and complementation of generalised Büchi automata. In *Automated Technology for Verification and Analysis*, pages 42–56, 2012.
27. S. Schewe and T. Varghese. Determinising parity automata. In *Mathematical Foundations of Computer Science*, pages 486–498, 2014.
28. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
29. S. Sickert, J. Esparza, S. Jaax, and J. Křetínský. Limit-deterministic Büchi automata for linear temporal logic. In *Computer Aided Verification*, pages 312–332, 2016. LNCS 9780.
30. S. Sickert and J. Křetínský. MoChiBA: Probabilistic LTL model checking using limit-deterministic Büchi automata. In *Automated Technology for Verification and Analysis*, pages 130–137, 2016.
31. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Computer Aided Verification*, pages 248–263, July 2000. LNCS 1855.
32. M.-H. Tsai, S. Fogarty, M. Y. Vardi, and Y.-K. Tsay. State of Büchi complementation. *Logical Mehods in Computer Science*, 10(4), 2014.
33. M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. GOAL for games, omega-automata, and logics. In *Computer Aided Verification*, pages 883–889, 2013.
34. M. Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Foundations of Computer Science*, pages 327–338, 1985.
35. E. M. Hahn, M. Perez, S. Schewe, F. Somenzi, A. Trivedi, and D. Wojtczak. Good-for-MDPs Automata for Probabilistic Analysis and Reinforcement Learning Figshare (2020), https://doi.org/10.6084/m9.figshare.11882739
36. A. Hartmanns and M. Seidl. tacas20ae.ova. Figshare (2019) https://doi.org/10.6084/m9.figshare.9699839.v2

Farkas certificates and minimal witnesses for probabilistic reachability constraints

Florian Funke, Simon Jantsch, and Christel Baier

Technische Universität Dresden, Germany*
{florian.funke, simon.jantsch, christel.baier}@tu-dresden.de

Abstract. This paper introduces *Farkas certificates* for lower and upper bounds on minimal and maximal reachability probabilities in Markov decision processes (MDP), which we derive using an MDP-variant of Farkas' Lemma. The set of all such certificates is shown to form a polytope whose points correspond to witnessing subsystems of the model and the property. Using this correspondence we can translate the problem of finding minimal witnesses to the problem of finding vertices with a maximal number of zeros. While computing such vertices is computationally hard in general, we derive new heuristics from our formulations that exhibit competitive performance compared to state-of-the-art techniques. As an argument that asymptotically better algorithms cannot be hoped for, we show that the decision version of finding minimal witnesses is NP-complete even for acyclic Markov chains.

1 Introduction

The goal of program verification is to consolidate the user's trust that a given system works as intended, and if this is not the case, to provide her with useful diagnostic information. Verification tools may, however, contain bugs and so a last grain of insecurity regarding their results always remains. A widely acknowledged approach to overcome this dilemma has been made in the form of *certifying algorithms* [17, 64]. These algorithms provide every result with an accompanying *certificate*, i.e., a token that can be used to verify the result independently and with little ressources. In this way, certificates enable the user (or a third party) to quickly give a mathematically rigorous proof for the correctness of the result *irrespective* of whether the algorithm itself works correctly.

Counterexamples, i.e. certificates for the violation of a property, can often be obtained as a byproduct of verification procedures. What constitutes a counterexample is highly context-dependent. Finite executions suffice as counterexamples for safety properties and single, possibly infinite, executions are viable counterexamples for LTL [29]. Tree-like counterexamples have been considered for

* This work was funded by DFG grant 389792660 as part of TRR 248, the Cluster of Excellence EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy), DFG-projects BA-1679/11-1 and BA-1679/12-1, and the Research Training Group QuantLA (GRK 1763).

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 324–345, 2020.
https://doi.org/10.1007/978-3-030-45190-5_18

fragments of CTL [28]. For a probabilistic system $\mathcal{M}$ and a linear time property ϕ, the most prominent notion of counterexample to $\mathrm{Pr}_{\mathcal{M}}(\phi) < \lambda$ is a set of paths satisfying ϕ whose probability mass is at least λ (see [1] for a survey).

Another notion of counterexample for probabilistic systems $\mathcal{M}$ and properties of the form $\mathrm{Pr}_{\mathcal{M}}(\phi) < \lambda$ are *critical subsystems* [1]. We adopt the reverse perspective and call a subsystem $\mathcal{M}'$ of $\mathcal{M}$ a *witnessing subsystem* for the property $\mathrm{Pr}_{\mathcal{M}}(\phi) \geq \lambda$ if $\mathrm{Pr}_{\mathcal{M}'}(\phi) \geq \lambda$. Small witnessing subsystems offer an insight into what parts of the system are responsible for the satisfaction of the property. Nonetheless, witnessing subsystems can hardly be regarded as viable certificates since verifying $\mathrm{Pr}_{\mathcal{M}'}(\phi) \geq \lambda$ is as hard as checking $\mathrm{Pr}_{\mathcal{M}}(\phi) \geq \lambda$ itself.

In this paper we build a solid bridge between certificates and witnessing subsystems. The systems we consider are modeled as Markov decision processes (MDP), which contain an absorbing goal state representing a desirable outcome. This approach is motivated by the fact that numerous model checking tasks can be reduced to reachability problems [3, 31, 32, 46, 73, 74].

Using Farkas' Lemma, we introduce certificates for bounds on the minimal and maximal probability to reach the goal state. We show that the set of these certificates forms a polytope and we provide a direct translation of a certificate to a witnessing subsystems for lower bounded threshold properties. Thereby, we bridge the gap between an abstract gadget, serving solely as a proof that the result is correct, and a concrete object, containing crucial diagnostic information about *why* the result holds. Moreover, our translation reduces the computation of minimal witnessing subsystems to a purely geometric problem, for which we provide and evaluate new exact and heuristic algorithms.

All omitted proofs can be found in the full version of this paper [42].

Contributions.

- Following the concept of certificates in certifying algorithms, we introduce *Farkas certificates* for reachability problems in MDPs (Table 1).
- We give a uniform notion of *witnessing subsystem* (WS) for $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$ and $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$ (Definition 4.1). To the best of our knowledge, witnesses for $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$ have not been considered previously.
- We establish NP-completeness for finding minimal WS even for acyclic discrete time Markov chains (DTMC) (Theorem 4.5).
- Our main result establishes a strong connection between the polytopes of Farkas certificates for $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$ and $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$ and WS of the same property (Theorem 5.4). In particular, one can read off a minimal WS from a vertex of the polytope with a maximal number of zeros (Corollary 5.5).
- From our polytope characterizations we derive two algorithms for computing minimal WS: one based on vertex enumeration and one based on mixed integer linear programming (Section 6). We also introduce a linear programming based heuristic aimed at computing small WS. We evaluate our approach on DTMC and MDP benchmarks, where particularly our heuristics show competitive results compared to state-of-the-art techniques (Section 7).

Table 1: Overview of Farkas certificates for reachability properties in MDPs (where $\lesssim \in \{\leq, <\}$ and $\gtrsim \in \{\geq, >\}$).

Property	Certificate dimension	Certificate condition
$\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$	$\mathbf{z} \in \mathbb{R}^S$	$\mathbf{Az} \leq \mathbf{b} \wedge \mathbf{z}(s_0) \gtrsim \lambda$
$\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$	$\mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}$	$\mathbf{yA} \leq \delta_{s_0} \wedge \mathbf{yb} \gtrsim \lambda$
$\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\mathrm{goal}) \lesssim \lambda$	$\mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}$	$\mathbf{yA} \geq \delta_{s_0} \wedge \mathbf{yb} \lesssim \lambda$
$\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\mathrm{goal}) \lesssim \lambda$	$\mathbf{z} \in \mathbb{R}^S$	$\mathbf{Az} \geq \mathbf{b} \wedge \mathbf{z}(s_0) \lesssim \lambda$

Related work. The fundament of certifying algorithms has been surveyed in [64]. In the context of model checking, the most prominent approach for the certification of a positive result has been to construct a proof of the property in the system [15, 66, 67]. Rank-based certificates for the emptiness of a certain automaton [57] can be used to certify positive model checking results. Model checking MDPs in the presence of multiple objectives has been studied in [37, 39].

Heuristic approaches for computing small witnessing subsystems in DTMCs have been proposed in [5, 7, 49, 51, 52] and implemented in the tool Comics [50]. Witnessing subsystems in MDPs have been considered in [6, 9] and [19], which focuses on succinctly representing witnessing schedulers. The mixed integer linear programming (MILP) formulation of [77, 78] allows for an exact computation of minimal witnessing subsystems for the property $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$. NP-completeness of computing minimal witnessing subsystems in MDPs was shown in [24], but the exact complexity has, to the best of our knowledge, not been determined for DTMCs (the problem was conjectured to be NP-complete in [77]).

Minimal probabilistic counterexamples given as sets of paths can be computed by reframing the problem as a k-shortest-path problem [44, 45]. Regular expressions have been considered to succinctly represent the set of paths in [33], and extensions were proposed in [18, 76]. The tool Dipro [4] computes probabilistic counterexamples, and a translation of these to fault trees was given in [56]. Another, learning-based, approach [20] also enumerates paths and produces a witnessing subsystem as a byproduct. But none of these approaches considers state-based minimality. Probabilistic counterexamples can be used to automatically guide iterative and refinement-based model checking techniques [23–25, 27, 48, 53].

Farkas' Lemma is a well-known source of certificates for the (in)feasibility of tasks in combinatorial optimization, operations research, and economics, as presented in the detailed historical account given in [70, pp. 209–226] as well as [62, Chapter 2] and [30, 65, 75]. The lecture notes [71] contain a rich variety of applications of linear programming in general and Farkas' Lemma in particular.

2 Preliminaries

Polyhedra and Farkas' Lemma. Throughout the article we write the dot product of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ as $\mathbf{xy}$ or $\mathbf{x} \cdot \mathbf{y}$. A *halfspace* in $\mathbb{R}^n$ is a set

$H = \{\mathbf{v} \in \mathbb{R}^n \mid \mathbf{a} \cdot \mathbf{v} \leq b\}$ for some non-trivial $\mathbf{a} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. A *polyhedron* is the intersection of finitely many halfspaces, and a *polytope* is a bounded polyhedron. A *face* of a polyhedron P is a subset $F \subseteq P$ of the form $F = \{\mathbf{x} \in P \mid \mathbf{a} \cdot \mathbf{x} = \max\{\mathbf{a} \cdot \mathbf{y} \mid \mathbf{y} \in P\}\}$ for some $\mathbf{a} \in \mathbb{R}^n$. A *vertex* of P is a face consisting of only one point.

Farkas' Lemma [38] is part of the fundament of polyhedra theory and linear programming. It provides a natural source of certificates showing the infeasibility of a given system of inequalites, or in other words, the emptiness of the polyhedron described by the system. We will use it in the following version.

Lemma 2.1 (Farkas' Lemma, cf. [70, Corollary 7.1f on p. 90]). *Let* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and* $\mathbf{b} \in \mathbb{R}^m$*. Then there exists* $\mathbf{z} \in \mathbb{R}^n_{\geq 0}$ *with* $\mathbf{Az} \leq \mathbf{b}$ *if and only if there does* not *exist* $\mathbf{y} \in \mathbb{R}^m_{\geq 0}$ *with* $\mathbf{yA} \geq 0 \wedge \mathbf{yb} < 0$.

Markov decision processes. A *Markov decision process* (MDP) is a tuple $\mathcal{M} = (S, \mathrm{Act}, \iota, \mathbf{P})$, where S is a finite set of *states*, Act is a finite set of *actions*, ι is a probability distribution on S called the *initial distribution* of M, and $\mathbf{P}\colon S \times \mathrm{Act} \times S \to [0, 1]$ is the *transition probability function* where we require $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{0, 1\}$ for all $s \in S$ and $\alpha \in \mathrm{Act}$. An action α is *enabled* in state $s \in S$ if $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 1$. The set of enabled actions at state s are denoted by $\mathrm{Act}(s)$, and we require $\mathrm{Act}(s) \neq \varnothing$ for all $s \in S$. A *path* in an MDP $\mathcal{M}$ is an infinite sequence $s_0\alpha_0 s_1\alpha_1 ...$ such that $\mathbf{P}(s_i, \alpha_i, s_{i+1}) > 0$ for all $i \geq 0$. A finite path is a finite sequence $\pi = s_0\alpha_0 s_1\alpha_1 ... s_n$ with the same condition for all $0 \leq i \leq n-1$. In this case, we define $\mathrm{last}(\pi) = s_n$. Denote by $\mathrm{Paths}(\mathcal{M})$ and $\mathrm{Paths}_{\mathrm{fin}}(\mathcal{M})$ the set of infinite and finite paths in $\mathcal{M}$.

A *discrete-time Markov chain* (DTMC) is an MDP with a single action which is enabled at every state. If $\mathcal{M}$ is a DTMC, then $\mathrm{Paths}(\mathcal{M})$ carries a probability measure, where the associated σ-algebra is generated by the cylinder sets $\mathrm{Cyl}(\tau) = \{\pi \in \mathrm{Paths}(\mathcal{M}) \mid \pi \text{ has prefix } \tau\}$ of finite paths $\tau = s_0 s_1 ... s_n$ in $\mathcal{M}$ with probability $\Pr(\mathrm{Cyl}(\tau)) = \iota(s_0) \cdot \prod_{0 \leq i < n} \mathbf{P}(s_i, s_{i+1})$ (fore more details see [13, Section 10.1]). In the following we denote for a finite set X the set of probability distributions on X by $\mathrm{Dist}(X)$. Given $\mu \in \mathrm{Dist}(X)$ let the *support* of μ be $\mathrm{supp}(\mu) = \{x \in X \mid \mu(x) > 0\}$.

A *deterministic scheduler* is a function $\mathfrak{S}\colon \mathrm{Paths}_{\mathrm{fin}}(\mathcal{M}) \to \mathrm{Act}$ such that $\mathfrak{S}(\pi) \in \mathrm{Act}(\mathrm{last}(\pi))$ and a *randomized scheduler* is a function $\mathfrak{S}\colon \mathrm{Paths}_{\mathrm{fin}}(\mathcal{M}) \to \mathrm{Dist}(\mathrm{Act})$ such that $\mathrm{supp}(\mathfrak{S}(\pi)) \subseteq \mathrm{Act}(\mathrm{last}(\pi))$ for all $\pi \in \mathrm{Paths}_{\mathrm{fin}}(\mathcal{M})$. Given a deterministic (or randomized) scheduler $\mathfrak{S}$, a path $\pi = s_0\alpha_0 s_1\alpha_1 ...$ in $\mathcal{M}$ is an $\mathfrak{S}$*-path* if $\alpha_i = \mathfrak{S}(s_0\alpha_0 ... s_i)$ (or $\alpha_i \in \mathrm{supp}(\mathfrak{S}(s_0\alpha_0 ... s_i))$) for all $i \geq 0$.

We denote by $\Pr^{\mathfrak{S}}$ the probability measure on infinite $\mathfrak{S}$-paths (see [13, Definition 10.92 on page 843] for more details). If we replace ι with the distribution concentrated on state s, then we obtain a probability measure $\Pr^{\mathfrak{S}}_{\mathcal{M},s}$ or short $\Pr^{\mathfrak{S}}_s$ on infinite $\mathfrak{S}$-paths starting in s. The scheduler is *memoryless* if $\mathfrak{S}(\pi) = \mathfrak{S}(\mathrm{last}(\pi))$ for all $\pi \in \mathrm{Paths}_{\mathrm{fin}}(\mathcal{M})$. We abbreviate memoryless deterministic schedulers as *MD-schedulers* and memoryless randomized schedulers as *MR-schedulers*.

Given a state $t \in S$, we let

$$\mathbf{Pr}_s^{\max}(\lozenge t) = \sup_{\mathfrak{S}} \mathrm{Pr}_s^{\mathfrak{S}}(\lozenge t) \quad \text{and} \quad \mathbf{Pr}_s^{\min}(\lozenge t) = \inf_{\mathfrak{S}} \mathrm{Pr}_s^{\mathfrak{S}}(\lozenge t)$$

denote the maximal and minimal probability to reach t eventually when starting in s and set $\mathbf{Pr}^{\min}(\lozenge t) = (\mathbf{Pr}_s^{\min}(\lozenge t))_{s \in S}$ and $\mathbf{Pr}^{\max}(\lozenge t) = (\mathbf{Pr}_s^{\max}(\lozenge t))_{s \in S}$. The supremum and infimum is indeed attained by an MD-scheduler [13, Lemmata 10.102 and 10.113], thus justifying the superscripts.

Setting 2.2. Henceforth we will assume that $\mathcal{M} = (S_{\text{all}}, \text{Act}, \iota, \mathbf{P})$ has a unique initial state $s_0 \in S$ and two distinguished absorbing states fail and goal $\in S_{\text{all}}$, i.e., $\mathbf{P}(\text{goal}, \alpha, s) = 0$ for all $\alpha \in \text{Act}$ and $s \in S_{\text{all}}$ with $s \neq \text{goal}$, and likewise for fail. Here goal represents a desirable outcome of the modeled system and fail an outcome that is to be avoided. We use the notation $S = S_{\text{all}} \setminus \{\text{fail}, \text{goal}\}$, we assume that every state $s \in S$ is reachable from s_0. We also assume that under every scheduler fail or goal is reachable from any state, i.e., $\mathbf{Pr}_s^{\min}(\lozenge(\text{goal} \vee \text{fail})) > 0$ for all $s \in S$. If $\mathcal{M}$ does not satisfy this condition from the start, we can apply a standard preprocessing step, which is essentially given by taking the MEC quotient of $\mathcal{M}$, see [2, 3] and also [26]. While it is often easier to verify the condition $\mathbf{Pr}_s^{\min}(\lozenge(\text{goal} \vee \text{fail})) > 0$, it is in fact equivalent to $\mathbf{Pr}_s^{\min}(\lozenge(\text{goal} \vee \text{fail})) = 1$ (see the full version [42]).

Whenever suitable, we denote by $\mathcal{M}$ also the set of enabled state-action pairs, i.e., $\mathcal{M} = \{(s, \alpha) \in S \times \text{Act} \mid \alpha \in \text{Act}(s)\}$. Let $\mathbf{A} \in \mathbb{R}^{\mathcal{M} \times S}$ be defined by

$$\mathbf{A}((s,\alpha),t) = \begin{cases} 1 - \mathbf{P}(s,\alpha,s), & \text{if } s = t \\ -\mathbf{P}(s,\alpha,t), & \text{if } s \neq t \end{cases}$$

We denote by $\mathbf{b} = (\mathbf{b}(s,\alpha))_{(s,\alpha) \in \mathcal{M}} \in \mathbb{R}^{\mathcal{M}}$ with $\mathbf{b}(s,\alpha) = \mathbf{P}(s,\alpha,\text{goal})$ and by δ_{s_0} the probability distribution that assigns 1 to s_0, and 0 to all other states.

The vectors $\mathbf{Pr}^{\min}(\lozenge\,\text{goal})$ and $\mathbf{Pr}^{\max}(\lozenge\,\text{goal})$ can be characterized using the following linear programs. Although this characterization is well-known, we give a proof in the full version [42] due to slight differences with the standard literature.

Proposition 2.3 (LP characterization, cf. [16, Lemma 8]). *Let $\mathcal{M}$ be an MDP as in Setting 2.2 and let $\boldsymbol{\delta} \in \mathbb{R}^n_{>0}$. Then the vectors $\mathbf{Pr}^{\min}(\lozenge\,\text{goal})$ and $\mathbf{Pr}^{\max}(\lozenge\,\text{goal})$ are, respectively, the* unique *solution of the LPs*

$$\max\ \boldsymbol{\delta} \cdot \mathbf{z} \ \ \textit{s.t.}\ \ \mathbf{Az} \leq \mathbf{b} \qquad \textit{and} \qquad \min\ \boldsymbol{\delta} \cdot \mathbf{z} \ \ \textit{s.t.}\ \ \mathbf{Az} \geq \mathbf{b}.$$

3 Farkas certificates for reachability in MDPs

In this section we establish certificates for the following statements:

(1) All schedulers $\mathfrak{S}$ satisfy $\mathrm{Pr}_{s_0}^{\mathfrak{S}}(\lozenge\,\text{goal}) \gtrsim \lambda$ (i.e., $\mathbf{Pr}_{s_0}^{\min}(\lozenge\,\text{goal}) \gtrsim \lambda$).
(2) Some scheduler $\mathfrak{S}$ satisfies $\mathrm{Pr}_{s_0}^{\mathfrak{S}}(\lozenge\,\text{goal}) \gtrsim \lambda$ (i.e., $\mathbf{Pr}_{s_0}^{\max}(\lozenge\,\text{goal}) \gtrsim \lambda$).
(3) All schedulers $\mathfrak{S}$ satisfy $\mathrm{Pr}_{s_0}^{\mathfrak{S}}(\lozenge\,\text{goal}) \lesssim \lambda$ (i.e., $\mathbf{Pr}_{s_0}^{\max}(\lozenge\,\text{goal}) \lesssim \lambda$).
(4) Some scheduler $\mathfrak{S}$ satisfies $\mathrm{Pr}_{s_0}^{\mathfrak{S}}(\lozenge\,\text{goal}) \lesssim \lambda$ (i.e., $\mathbf{Pr}_{s_0}^{\min}(\lozenge\,\text{goal}) \lesssim \lambda$).

where $\lesssim\, \in \{\leq, <\}$ and $\gtrsim\, \in \{\geq, >\}$. The basis of our construction is the LP characterization of the probabilities above and, crucially, Farkas' Lemma.

Certificates for universally-quantified statements. In order to deal with the cases (1) and (3), we need the following lemma proved in the full version [42].

Lemma 3.1. *For* $\mathbf{A} \in \mathbb{R}^{\mathcal{M} \times S}, \mathbf{b} \in \mathbb{R}^{\mathcal{M}}$ *as in Setting 2.2, we have for all* $\mathbf{z} \in \mathbb{R}^S$:

$$\begin{aligned}\mathbf{A}\mathbf{z} \leq \mathbf{b} &\implies \mathbf{z} \leq \mathbf{Pr}^{\min}(\Diamond\, \mathrm{goal}) \\ \mathbf{A}\mathbf{z} \geq \mathbf{b} &\implies \mathbf{z} \geq \mathbf{Pr}^{\max}(\Diamond\, \mathrm{goal})\end{aligned}$$

Corollary 3.2. *For* $\gtrsim \in \{\geq, >\}$ *and* $\lesssim \in \{\leq, <\}$ *we have*

$$\begin{aligned}\mathbf{Pr}^{\min}_{s_0}(\Diamond\, \mathrm{goal}) \gtrsim \lambda &\iff \exists \mathbf{z} \in \mathbb{R}^S.\ \mathbf{A}\mathbf{z} \leq \mathbf{b} \land \mathbf{z}(s_0) \gtrsim \lambda \\ \mathbf{Pr}^{\max}_{s_0}(\Diamond\, \mathrm{goal}) \lesssim \lambda &\iff \exists \mathbf{z} \in \mathbb{R}^S.\ \mathbf{A}\mathbf{z} \geq \mathbf{b} \land \mathbf{z}(s_0) \lesssim \lambda\end{aligned}$$

Proof. For the direction from left to right, we take $\mathbf{z}$ to be $\mathbf{Pr}^{\min}(\Diamond\, \mathrm{goal})$. The opposite direction follows from Lemma 3.1. □

The right hand sides of Corollary 3.2 provide *certifying* formulations for problems (1) and (3): to check whether the corresponding threshold statement holds, one must merely check whether $\mathbf{z}$ satisfies the inequalities, rather than checking whether $\mathbf{Pr}^{\min/\max}_{s_0}(\Diamond\, \mathrm{goal})$ was computed correctly. If the threshold condition is satisfied, then the vectors $\mathbf{Pr}^{\min/\max}_{s_0}(\Diamond\, \mathrm{goal})$ are also valid certificates.

Certificates for existentially-quantified statements. To find certificates for the cases (2) and (4), we calculate:

$$\begin{aligned}& \mathbf{Pr}^{\min}_{s_0}(\Diamond\, \mathrm{goal}) < \lambda \\ \overset{\text{Cor. 3.2}}{\iff}\ & \neg \exists \mathbf{z} \in \mathbb{R}^S_{\geq 0}.\ \ \mathbf{A}\mathbf{z} \leq \mathbf{b} \land \mathbf{z}(s_0) \geq \lambda \\ \iff\ & \neg \exists \mathbf{z} \in \mathbb{R}^S_{\geq 0}.\ \ \begin{pmatrix} \mathbf{A} \\ -1\, 0 \dots 0 \end{pmatrix} \mathbf{z} \leq \begin{pmatrix} \mathbf{b} \\ -\lambda \end{pmatrix} \\ \overset{\text{Lem. 2.1}}{\iff}\ & \exists \mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}, y^* \geq 0.\ \ (\mathbf{y}, y^*) \begin{pmatrix} \mathbf{A} \\ -1\, 0 \dots 0 \end{pmatrix} \geq 0 \land (\mathbf{y}, y^*) \begin{pmatrix} \mathbf{b} \\ -\lambda \end{pmatrix} < 0 \\ \iff\ & \exists \mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}.\ \ \mathbf{y}\mathbf{A} \geq \delta_{s_0} \land \mathbf{y}\mathbf{b} < \lambda.\end{aligned}$$

For non-strict inequalities, we apply Farkas' Lemma in the opposite direction:

$$\begin{aligned}& \mathbf{Pr}^{\min}_{s_0}(\Diamond\, \mathrm{goal}) \leq \lambda \\ \overset{\text{Cor. 3.2}}{\iff}\ & \neg \exists \mathbf{z} \in \mathbb{R}^S_{\geq 0}.\ \ \mathbf{A}\mathbf{z} \leq \mathbf{b} \land \mathbf{z}(s_0) > \lambda \\ \iff\ & \neg \exists \mathbf{z} \in \mathbb{R}^S_{\geq 0}, z^* \geq 0.\ \ \begin{pmatrix} -\mathbf{A} & \mathbf{b} \end{pmatrix} \begin{pmatrix} \mathbf{z} \\ z^* \end{pmatrix} \geq 0 \land \begin{pmatrix} -\delta_{s_0} & \lambda \end{pmatrix} \begin{pmatrix} \mathbf{z} \\ z^* \end{pmatrix} < 0 \\ \overset{\text{Lem. 2.1}}{\iff}\ & \exists \mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}.\ \ \mathbf{y} \begin{pmatrix} -\mathbf{A} & \mathbf{b} \end{pmatrix} \leq \begin{pmatrix} -\delta_{s_0} & \lambda \end{pmatrix} \\ \iff\ & \exists \mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}.\ \ \mathbf{y}\mathbf{A} \geq \delta_{s_0} \land \mathbf{y}\mathbf{b} \leq \lambda.\end{aligned}$$

The deductions for $\mathbf{Pr}^{\max}(\Diamond\, \mathrm{goal})$ are analogous, so that we get:

Proposition 3.3. *For $\gtrsim \in \{\geq, >\}$ and $\lesssim \in \{\leq, <\}$ we have*

$$\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\mathrm{goal}) \lesssim \lambda \iff \exists \mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}.\ \mathbf{yA} \geq \delta_{s_0} \wedge \mathbf{yb} \lesssim \lambda$$
$$\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda \iff \exists \mathbf{y} \in \mathbb{R}^{\mathcal{M}}_{\geq 0}.\ \mathbf{yA} \leq \delta_{s_0} \wedge \mathbf{yb} \gtrsim \lambda$$

Together, Corollary 3.2 and Proposition 3.3 give us all certificate conditions of Table 1.

4 Minimal witnesses for reachability in MDPs

In this section we consider the following problem: Given an MDP $\mathcal{M}$ that satisfies the property $\mathbf{Pr}^{\min}_{\mathcal{M},s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$ (or $\mathbf{Pr}^{\max}_{\mathcal{M},s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$), find a small subsystem $\mathcal{M}'$ of $\mathcal{M}$ that still satisfies these thresholds. Such a subsystem is a witness to the satisfaction of the property in $\mathcal{M}$. We first define subsystems and consider different measures of size which we show to be equivalent. Then we deal with the question of finding minimal witnessing subsystems.

Subsystems, witnesses and notions of minimality. Our definition of subsystem is essentially the same to the definition in [77, 78] that was used for witnessing subsystems of $\mathbf{Pr}^{\max}_{\mathcal{M},s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$. From now on we restrict our attention to properties of the form $\mathbf{Pr}^{\min\,/\,\max}_{\mathcal{M},s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$. One can deal with upper bounds by exchanging the roles of fail and goal and invoking the equality $\mathbf{Pr}^{\min}_{\mathcal{M},s_0}(\lozenge\,\mathrm{goal}) = 1 - \mathbf{Pr}^{\max}_{\mathcal{M},s_0}(\lozenge\,\mathrm{fail})$, which holds by the conditions of Setting 2.2.

Intuitively, a subsystem $\mathcal{M}'$ of $\mathcal{M}$ contains a subset of states of $\mathcal{M}$, and a transition of $\mathcal{M}$ originating in a state of $\mathcal{M}'$ remains unchanged in $\mathcal{M}'$ or is redirected to fail (instead of explicitely redirecting to fail, sub-stochastic distributions are used in [77, 78] with the same effect).

Definition 4.1 (Subsystem and witness). *Let $\mathcal{M} = (S_{\mathrm{all}}, \mathrm{Act}, s_0, \mathbf{P})$ be an MDP as in Setting 2.2. A* subsystem *$\mathcal{M}' \subseteq \mathcal{M}$ is an MDP $\mathcal{M}' = (S'_{\mathrm{all}}, \mathrm{Act}, s_0, \mathbf{P}')$ with $\mathrm{fail}, \mathrm{goal} \in S'_{\mathrm{all}} \subseteq S_{\mathrm{all}}$, $\mathrm{Act}_{\mathcal{M}'}(s) = \mathrm{Act}_{\mathcal{M}}(s)$ for all $s \in S'_{\mathrm{all}}$, and for all $s, t \in S'_{\mathrm{all}}$ with $t \neq \mathrm{fail}$ and $\alpha \in \mathrm{Act}$ we have*

$$\mathbf{P}'(s,\alpha,t) > 0 \Longrightarrow \mathbf{P}'(s,\alpha,t) = \mathbf{P}(s,\alpha,t).$$

We say that the states $S_{\mathrm{all}} \setminus S'_{\mathrm{all}}$ and the transitions (s,α,t) with $\mathbf{P}(s,\alpha,t) > 0$ and $\mathbf{P}'(s,\alpha,t) = 0$ have been deleted *in $\mathcal{M}'$. A* witness *for $\mathbf{Pr}^{\min\,/\,\max}_{\mathcal{M},s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$ is a subsystem $\mathcal{M}' \subseteq \mathcal{M}$ such that $\mathbf{Pr}^{\min\,/\,\max}_{\mathcal{M}',s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$.*

Remark 4.2. The condition $\mathrm{Act}_{\mathcal{M}'}(s) = \mathrm{Act}_{\mathcal{M}}(s)$ ensures that the probability of a deleted transition (s,α,t) is added to (s,α,fail). This is essential for witnesses for $\mathbf{Pr}^{\min}_{\mathcal{M},s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$ as one could otherwise remove entire actions causing low probabilities and obtain greater $\mathbf{Pr}^{\min}$ in $\mathcal{M}'$ than in $\mathcal{M}$ as a result. For witnesses of $\mathbf{Pr}^{\max}_{\mathcal{M}',s_0}(\lozenge\,\mathrm{goal}) \gtrsim \lambda$ one could delete this condition, thus leading to the notion of [77, 78].

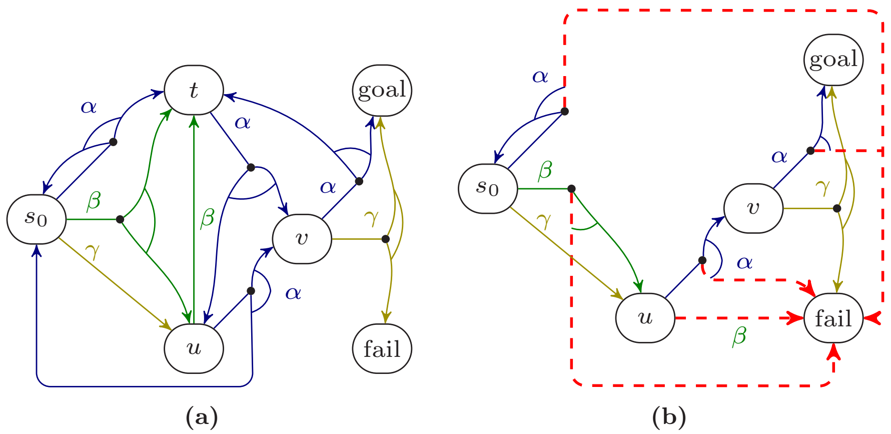


Fig. 1: An MDP (with omitted probabilities (a)) and a subsystem (b), where redirected transitions are dashed.

Example 4.3. Figure 1a depicts an MDP and Figure 1b indicates the subsystem that is obtained by deleting the state t and additionally the transition (u, α, s_0).

The following lemma ensures that we can use the subsystems as witnesses for both $\mathbf{Pr}^{\max}_{\mathcal{M},s_0}(\lozenge \, \text{goal}) \gtrsim \lambda$ and $\mathbf{Pr}^{\min}_{\mathcal{M},s_0}(\lozenge \, \text{goal}) \gtrsim \lambda$.

Lemma 4.4. *Let $\mathcal{M}$ be an MDP as in Setting 2.2 and $\mathcal{M}' \subseteq \mathcal{M}$. Then:*

$$\mathbf{Pr}^{\min}_{\mathcal{M}',s_0}(\lozenge \, \text{goal}) \leq \mathbf{Pr}^{\min}_{\mathcal{M},s_0}(\lozenge \, \text{goal}) \quad \textit{and} \quad \mathbf{Pr}^{\max}_{\mathcal{M}',s_0}(\lozenge \, \text{goal}) \leq \mathbf{Pr}^{\max}_{\mathcal{M},s_0}(\lozenge \, \text{goal})$$

We consider the following notions of minimality for subsystems:

(1) *State-minimality:* $|S'_{\text{all}}|$ is minimal.
(2) *Transition-minimality:* The number of transitions, i.e. triples (s, α, t) satisfying $\mathbf{P}'(s, \alpha, t) > 0$, is minimal;
(3) *Size-minimality:* The sum of states and transitions is minimal.

Depending on the situation, one notion might be more suitable than the others. However, in the full version [42] we show that finding transition-minimal (respectively, size-minimal) witnesses can be reduced to finding state-minimal witnesses with a linear (respectively, quadratic) blow-up. We will therefore restrict ourselves to state-minimality for the rest of this paper.

NP-completeness of finding minimal witnesses for DTMCs. In this section we determine the computational complexity of the *witness problem*: Given a DTMC $\mathcal{M}$, a positive integer k, and a rational number $\lambda \in [0, 1]$, decide whether there exists a witness $\mathcal{M}' \subseteq \mathcal{M}$ for $\Pr_{\mathcal{M},s_0}(\lozenge \, \text{goal}) \geq \lambda$ with at most k states. The corresponding problem for MDPs is known to be NP-complete [24, 78][1]. In this section we show that the witness problem is already

[1] Although the framework in [24] considers a richer logic, the hardness proof uses only probabilistic reachability formulas such as the ones we consider.

NP-complete for acyclic DTMCs, where acyclicity means that the underlying graph with $V = S$ and $E = \{(s,t) \in S \times S \mid \mathbf{P}(s,t) > 0\}$ is acyclic (as before, we take $S = S_{\text{all}} \setminus \{\text{goal}, \text{fail}\}$). This answers a conjecture of [77] in the affirmative and also shows NP-completeness of finding minimal witnesses for $\mathbf{Pr}^{\min}_{\mathcal{M},s_0}(\lozenge\, \text{goal}) \geq \lambda$.

Theorem 4.5. *The witness problem is NP-complete for acyclic DTMCs.*

Proof (Sketch). An NP-algorithm for the witness problem is given by guessing a set of states of size k and verifying in polynomial time that the corresponding subsystem satisfies $\text{Pr}_{\mathcal{M}',s_0}(\lozenge\, \text{goal}) \geq \lambda$.

For hardness, we give a reduction from the *clique problem*, which is among Karp's 21 NP-complete problems [54]. The idea is the following: Given an instance of the clique problem with graph $G = (V, E)$ and integer k, construct an acyclic Markov chain $\mathcal{M}$ with states $S = \{s_0\} \cup V \cup E \cup \{\text{goal}, \text{fail}\}$ and edges from each vertex $v \in V$ to all edges to which it is incident. Then the existence of a k-clique can be reduced to the existence of a "saturated" subsystem in $\mathcal{M}$ with k states in V. To check whether the subsystem is saturated, we require it to have more probability than a certain threshold, which depends on k and $|V|$. Details can be found in the full version [42].

Remark 4.6. NP-completeness of transition-minimal and size-minimal versions of the witness problem for acyclic DTMCs follows along the same lines, where only the sizes and thresholds for the subsystems need to be adapted.

However, DTMCs whose underlying graph is a tree permit an efficient algorithm for computing minimal witnesses (for the proof see the full version [42]).

Proposition 4.7. *Minimal witnesses in tree-shaped DTMCs can be computed in polynomial time.*

Proof (Sketch). The algorithm first transforms the DTMC at hand into a binary (tree-shaped) DTMC, and then works bottom up by storing for each state the highest probability that can be obtained with a subsystem of size k, for all k up to the size of the subtree.

5 Relating Farkas certificates and minimal witnesses

In this section we establish a strong connection between Farkas certificates on the one hand and witnesses for probabilistic reachability constraints on the other hand. We first note that the set of Farkas certificates for non-strict lower bounds forms a polytope, i.e., a bounded polyhedron.

Lemma 5.1 (Polytopes of Farkas certificates). *Let $\mathcal{M} = (S_{\text{all}}, \text{Act}, s_0, \mathbf{P})$ be an MDP as in Setting 2.2 and consider $\mathbf{A} \in \mathbb{R}^{\mathcal{M} \times S}$ and $\mathbf{b} \in \mathbb{R}^S$ introduced there. Then for every $\lambda \in [0, 1]$ the polyhedra*

$$\mathcal{P}^{\min}(\lambda) = \{\mathbf{z} \in \mathbb{R}^S \mid \mathbf{A}\mathbf{z} \leq \mathbf{b} \wedge \mathbf{z}(s_0) \geq \lambda\}$$
$$\mathcal{P}^{\max}(\lambda) = \{\mathbf{y} \in \mathbb{R}^{\mathcal{M}} \mid \mathbf{y} \geq 0 \wedge \mathbf{y}\mathbf{A} \leq \delta_{s_0} \wedge \mathbf{y}\mathbf{b} \geq \lambda\}$$

are both polytopes, called the polytopes of Farkas certificates.

Remark 5.2. For any vector $\mathbf{v} \in \mathbb{R}^n$ the support is defined as $\mathrm{supp}(\mathbf{v}) = \{i \in \{1, ..., n\} \mid \mathbf{v}_i > 0\}$, and analogously for the vector spaces $\mathbb{R}^S$ and $\mathbb{R}^{\mathcal{M}}$. As our connection between subsystems of $\mathcal{M}$ and points in $\mathcal{P}^{\min}(\lambda)$ is based on taking the support, we restrict our attention to the subpolytope $\mathcal{P}^{\min}_{\geq 0}(\lambda) = \mathcal{P}^{\min}(\lambda) \cap \mathbb{R}^S_{\geq 0}$.

Notation 5.3. Given an MDP $\mathcal{M} = (S_{\mathrm{all}}, \mathrm{Act}, s_0, \mathbf{P})$ as in Setting 2.2 and a subset $R \subseteq \mathcal{M}$, where $\mathcal{M}$ also denotes the state-action pairs (compare with Section 2). We let $\mathcal{M}_R = (S'_{\mathrm{all}}, \mathrm{Act}, s_0, \mathbf{P}')$ be the subsystem where, roughly speaking, the state-action pairs in R *remain*. More precisely, let

$$S'_{\mathrm{all}} = \{s \in S \mid \exists \alpha \in \mathrm{Act} \,.\, (s, \alpha) \in R\} \cup \{\mathrm{goal}, \mathrm{fail}\}$$

$$\mathbf{P}'(s, \alpha, t) = \begin{cases} \mathbf{P}(s, \alpha, t) & \text{if } (s, \alpha) \in R \text{ and } t \in S'_{\mathrm{all}} \setminus \{\mathrm{fail}\} \\ 1 - \sum_{t \in S'_{\mathrm{all}} \setminus \{\mathrm{fail}\}} \mathbf{P}(s, \alpha, t) & \text{if } (s, \alpha) \in R \text{ and } t = \mathrm{fail} \\ 1 & \text{if } (s, \alpha) \notin R, \alpha \in \mathrm{Act}(s) \text{ and } t = \mathrm{fail} \\ 0 & \text{else} \end{cases}$$

For $R \subseteq S$ we set $\mathcal{M}_R = \mathcal{M}_{R'}$ for $R' = \bigcup_{s \in R} \{s\} \times \mathrm{Act}(s)$.

Theorem 5.4 (Farkas certificates yield witnesses). *Let $\mathcal{M}$ be an MDP as in Setting 2.2 and $\lambda \in [0, 1]$. Then for a set $R \subseteq S$ the following statements are equivalent:*

(1) The subsystem $\mathcal{M}_R$ is a witness for $\mathbf{Pr}^{\min}_{\mathcal{M}, s_0}(\lozenge \,\mathrm{goal}) \geq \lambda$.
(2) There is a point $\mathbf{p}$ in $\mathcal{P}^{\min}_{\geq 0}(\lambda)$ such that $\mathrm{supp}(\mathbf{p}) \subseteq R$.
(3) There is a vertex $\mathbf{v}$ of $\mathcal{P}^{\min}_{\geq 0}(\lambda)$ such that $\mathrm{supp}(\mathbf{v}) \subseteq R$.

Moreover, for a set $R \subseteq \mathcal{M}$ the following statements are equivalent:

(a) The subsystem $\mathcal{M}_R$ is a witness for $\mathbf{Pr}^{\max}_{\mathcal{M}, s_0}(\lozenge \,\mathrm{goal}) \geq \lambda$.
(b) There is a point $\mathbf{p}$ in $\mathcal{P}^{\max}(\lambda)$ such that $\mathrm{supp}(\mathbf{p}) \subseteq R$.
(c) There is a vertex $\mathbf{v}$ of $\mathcal{P}^{\max}(\lambda)$ such that $\mathrm{supp}(\mathbf{v}) \subseteq R$.

One consequence of Theorem 5.4 is that every MD-scheduler $\mathfrak{S}$ with $\mathrm{Pr}^{\mathfrak{S}}_{s_0}(\lozenge \,\mathrm{goal}) \geq \lambda$ corresponds to a point in $\mathcal{P}^{\max}(\lambda)$, i.e. to a certificate for $\mathbf{Pr}^{\max}_{\mathcal{M}, s_0}(\lozenge \,\mathrm{goal}) \geq \lambda$.

Corollary 5.5 (Detecting minimal witnesses by vertices of $\mathcal{P}$). *Let $\mathcal{M} = (S_{\mathrm{all}}, \mathrm{Act}, s_0, \mathbf{P})$ be an MDP as in Setting 2.2 and $\lambda \in [0, 1]$. Then a vertex $\mathbf{v}$ of $\mathcal{P}^{\min}_{\geq 0}(\lambda)$ has a maximal number of zeros among all vertices of $\mathcal{P}^{\min}_{\geq 0}(\lambda)$ if and only if $\mathcal{M}_{\mathrm{supp}(\mathbf{v})}$ is a minimal witness for $\mathbf{Pr}^{\min}_{s_0}(\lozenge \,\mathrm{goal}) \geq \lambda$.*

Dually, a vertex $\mathbf{v}$ of $\mathcal{P}^{\max}(\lambda)$ has a maximal number of zeros among all vertices of $\mathcal{P}^{\max}(\lambda)$ if and only if all of the following hold:

(1) $\mathcal{M}_{\mathrm{supp}(\mathbf{v})} = (S'_{\mathrm{all}}, \mathrm{Act}, s_0, \mathbf{P}')$ is a minimal witness for $\mathbf{Pr}^{\max}_{s_0}(\lozenge \,\mathrm{goal}) \geq \lambda$,
(2) for every $s \in S'$ there is precisely one $\alpha \in \mathrm{Act}(s)$ with $(s, \alpha) \in \mathrm{supp}(\mathbf{v})$,
(3) the corresponding map $\mathfrak{S} \colon S' \to \mathrm{Act}$ is an MD-scheduler on $\mathcal{M}_{\mathrm{supp}(\mathbf{v})}$ with $\mathrm{Pr}^{\mathfrak{S}}_{s_0}(\lozenge \,\mathrm{goal}) \geq \lambda$.

6 Computing witnessing subsystems

In this section we use the results of Section 5 to derive two algorithms for the computation of minimal witnesses for reachability constraints in MDPs. As the problem is NP-hard, we also present a heuristic approach aimed at computing small witnessing subsystems.

Vertex enumeration. Corollary 5.5 gives rise to the following approach of computing minimal witnessing subsystems: enumerate all vertices in the corresponding polytope and choose one with a maximal amount of zeros. Vertex enumeration of polytopes has been studied extensively [11, 12, 14, 21, 22, 35, 36, 40, 41, 63, 68] and has been shown to be computationally hard [55, Corollary 2].

First experiments that we have conducted with the `SageMath`[2] toolkit which supports vertex enumeration have not scaled well in the dimension, which in our case is the number of states in the original system. Also, we found no tool support for vertex enumeration that is able to handle sparse matrices, which is essential for bigger benchmarks.

Mixed integer linear programming. An approach that computes minimal witnesses to the threshold problem $\mathbf{Pr}^{\max}_{s_0}(\Diamond\,\mathrm{goal}) \geq \lambda$ using mixed integer linear programs (MILP) was presented in [77, 78]. Using the following lemma, we can derive MILP formulations from our polytope formulations.

Lemma 6.1. *Let $\mathcal{P} = \{\mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0\} \subseteq \mathbb{R}^n$ be a polytope and $K \geq 0$ be such that for all $\mathbf{p} \in \mathcal{P}$ and $1 \leq i \leq n$ we have $\mathbf{p}(i) \leq K$. Consider the MILP*

$$\min \sum_{1\leq i\leq n} \boldsymbol{\sigma}(i) \quad s.t. \quad \mathbf{x} \in \mathcal{P}, \quad \mathbf{x} \leq K \cdot \boldsymbol{\sigma}, \quad \boldsymbol{\sigma}(i) \in \{0,1\}$$

Then a vector $(\boldsymbol{\sigma}, \mathbf{x})$ is an optimal solution of this MILP if and only if $\mathbf{x}$ is a point in $\mathcal{P}$ with a maximal number of zeros.

For $\mathcal{P}^{\min}_{\geq 0}(\lambda)$ we can use Lemma 3.1 to derive that $K = 1$ is a viable bound. By invoking again Corollary 5.5, this means that a solution $(\mathbf{z}, \boldsymbol{\sigma})$ of the MILP

$$\min \sum_{s\in S} \boldsymbol{\sigma}(s) \;\text{ s.t. }\; \mathbf{z} \in \mathcal{P}^{\min}_{\geq 0}(\lambda), \quad \mathbf{z} \leq \boldsymbol{\sigma}, \quad \boldsymbol{\sigma}(i) \in \{0,1\}$$

encodes a minimal witnessing subsystem in the integral variables $\boldsymbol{\sigma}$. This MILP was used in [77, 78] for the computation of minimal witnessing subsystems of DTMCs .

An upper bound K as in Lemma 6.1 for $\mathcal{P}^{\max}(\lambda)$ can be found in polynomial time by taking the objective value of an optimal solution to the LP

$$\max \sum_{(s,\alpha)\in\mathcal{M}} \mathbf{y}(s,\alpha) \;\text{ s.t. }\; \mathbf{y} \in \mathcal{P}^{\max}(\lambda)$$

[2] http://www.sagemath.org/

Remark 6.2. To compute minimal witnesses for $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$, [77, 78] (witnesses for $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\mathrm{goal}) \geq \lambda$ were not considered) propose the MILP with objective: min $\sum_{(s,\alpha)\in\mathcal{M}} \boldsymbol{\sigma}(s,\alpha)$, subject to the conditions

$$\forall (s,\alpha) \in \mathcal{M}.\quad \mathbf{z}(s) \leq 1 - \boldsymbol{\sigma}(s,\alpha) + \sum_{s'\in S} \mathbf{P}(s,\alpha,s')\cdot \mathbf{z}(s') + \mathbf{b}(s) \tag{6.1}$$

$$\forall s \in S.\quad \mathbf{z}(s) \leq \sum_{\alpha\in \mathrm{Act}(s)} \boldsymbol{\sigma}(s,\alpha), \quad \mathbf{z}(s_0) \geq \lambda \tag{6.2}$$

where $\boldsymbol{\sigma}(s,\alpha)$ are binary integer variables. It was implemented in the tool `ltlsubsys`. The idea is to directly encode a scheduler in the set of equations $\mathbf{Az} \leq \mathbf{b}$ using $\boldsymbol{\sigma}$. In [77, 78] a number of additional redundant constraints are given to guide the search. In contrast to [77, 78] we do not need to handle so-called *problematic states*, as our precondition $\mathbf{Pr}^{\min}_{s}(\lozenge(\mathrm{goal} \vee \mathrm{fail})) > 0$ guarantees that no such states exist.

***k*-step quotient sum (QS$_k$) heuristics.** Approximating the maximal number of zeros in a polytope is computationally hard in general [8]. We now derive a heuristic approach for this problem called *quotient sum heuristic* which is based on iteratively solving LPs over the polytope, where the objective function for each iteration depends on an optimal solution of the previous LP. More precisely, we take $\mathbf{o}_1 = (1,\ldots,1)$ and take an optimal solution QS_1 of the LP $\min \mathbf{o}_1 \cdot \mathbf{y}$ s.t. $\mathbf{y} \in \mathcal{P}^{\max}(\lambda)$. Many entries in QS_1 may be small, but still greater than zero. In order to push as many of the small values of QS_1 to zero, we define a new objective function by

$$\mathbf{o}_2(i) = \begin{cases} 1/\,\mathrm{QS}_1(i), & \text{if } \mathrm{QS}_1(i) > 0 \\ C, & \text{if } \mathrm{QS}_1(i) = 0 \end{cases} \tag{6.3}$$

where C is a value that is greater than any value $1/\,\mathrm{QS}_1(i)$. We now take a solution QS_2 of the new LP $\min \mathbf{o}_2 \cdot \mathbf{y}$ s.t. $\mathbf{y} \in \mathcal{P}^{\max}(\lambda)$ and form the next objective function $\mathbf{o}_3$ as in (6.3). Inductively this generates a sequence of objective functions $(\mathbf{o}_k)_{k\geq 1}$ and corresponding optimal solutions $(\mathrm{QS}_k)_{k\geq 1}$ in $\mathcal{P}^{\max/\min}(\lambda)$. By Theorem 5.4 we can construct a witnessing subsystem with as many states as the number of non-zero entries in QS_k.

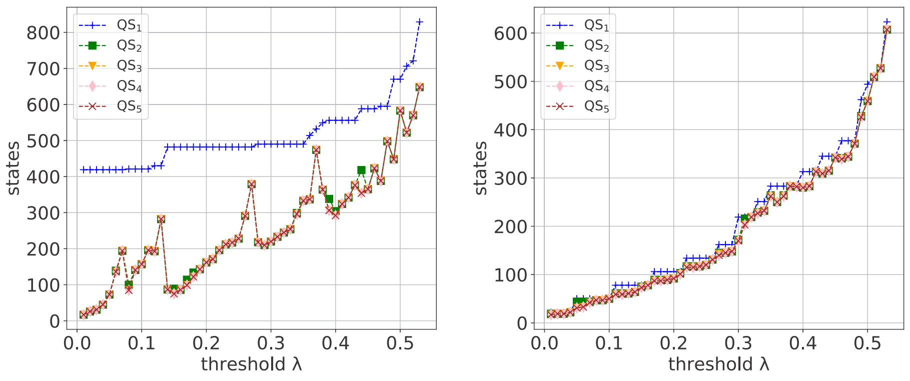
7 Experiments

In this section we evaluate our MILP formulations and heuristics on a number of DTMC and MDP benchmarks from the Prism benchmark-suite [58, 59]. We compare our results with the tool Comics [50], which implements heuristic approaches to compute small subsystems for DTMCs. It has two modes: the *local search* extends a given subsystem by short paths that carry much probability, whereas the *global search* searches for the next most probable path from the initial state to goal, and adds it to the subsystem. Both approaches iteratively extend a subsystem until it carries more probability than the given threshold and thus have to compute the probability of the subsystem at each iteration.

All computations were performed on a computer with two Intel E5-2680 8 cores at 2.70 GHz running Linux, with a time bound of 30 minutes, a memory bound of 100 GB and with each benchmark instance having access to 4 cores. For the LP and MILP instances we use the Gurobi solver, version 8.1.1 [43]. The recorded times of our computations include the construction of the LPs/MILPs and are wall clock times. Pre-processing steps, such as collapsing states that cannot reach goal, are not counted in the time consumption. For Comics, we use the time that is reported as counterexample generation time by the tool.

To validate our implementation, we used Prism to verify that the subsystems that we compute indeed satisfy the probability thresholds. We noticed that for a few instances ($< 0.5\%$) Prism reported a deviation of less than 10^{-8}, which can be explained by the fact that both Prism and the solvers that we use rely on floating-point arithmetic, which is approximate by nature.

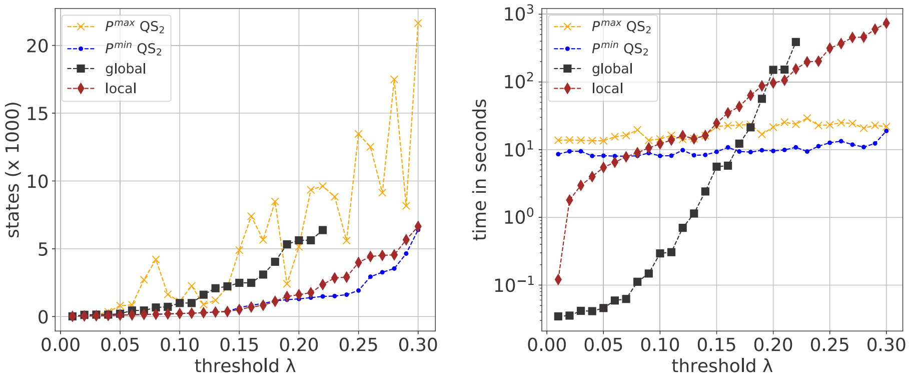
Our implementation, together with the models we use and benchmark results can be found at https://github.com/simonjantsch/farkas.



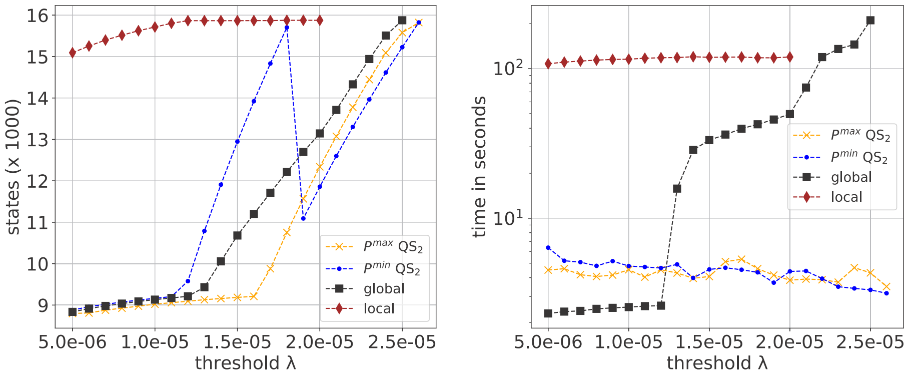
(a) QS-heuristic applied to $\mathcal{P}^{\max}(\lambda)$. **(b)** QS-heuristic applied to $\mathcal{P}^{\min}_{\geq 0}(\lambda)$.

Fig. 2: crowds-2-8: comparing QS_k for growing k.

DTMC benchmarks. As $\mathbf{Pr}^{\max}$ and $\mathbf{Pr}^{\min}$ coincide on DTMCs, we can use the heuristics and exact computations derived from either the $\mathcal{P}^{\max}$ or the $\mathcal{P}^{\min}_{\geq 0}$ polytope for DTMCs (in Comics we use the standard query $\mathrm{Pr}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$). We consider two DTMC benchmarks: a model of the crowds-N-K protocol [69, 72] for ensuring anonymous web browsing (with N members and K protocol runs) and a model of the bounded retransmission protocol [34, 47] for file transfers (where brp-N-K is the instance with N chunks and K retransmissions).



(a) crowds-5-8 (46,873 states). COMICS-global runs out of memory for $\lambda \geq 0.23$.

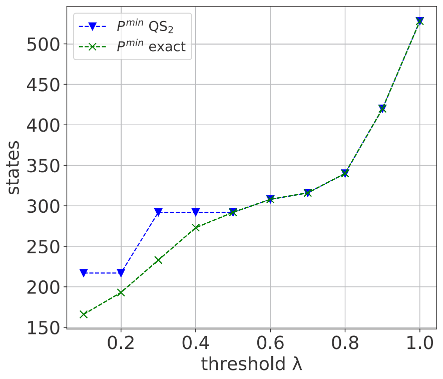


(b) brp-512-2 (15,875 states). COMICS-local reports an error for $\lambda \geq 2.1 \cdot 10^{-5}$ and COMICS-global runs out of memory for $\lambda = 2.6 \cdot 10^{-5}$.

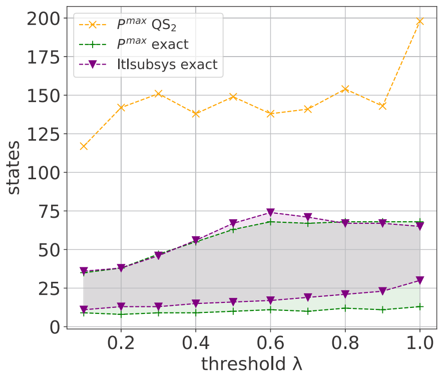
Fig. 3: Comparison of heuristic methods on DTMC benchmarks.

Figure 2 shows the effect of increasing the number of iterations of the QS-heuristic for the model crowds-2-8. While the first iteration (taking QS$_2$ instead of QS$_1$) has an impact on the number of states, more iterations do not improve the result significantly. For QS$_1$, the sizes of subsystems increase monotonically with growing λ. Starting with QS$_2$ the results may, interestingly, have "spikes": increasing λ can lead to smaller subsystems.

Figure 3 shows the results of the QS$_2$-heuristic compared to the two modes of COMICS for λ that ranges between 0 and the actual reachability probability of the model. A general observation is that the runtime of the QS-heuristic is independent of λ, whereas both modes of COMICS use significantly more time with increasing λ. The same observation can be done for memory consumption, which stayed below 200 MB for our heuristics. Also, especially for crowds-5-8, one can see that relatively small subsystems are possible even for large λ. The exact computations via MILPs hit the timeout for almost all instances.



(a) Witnesses for $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$.



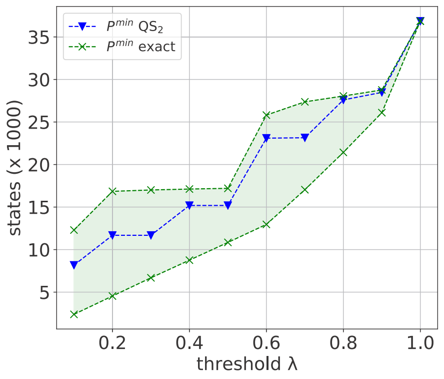
(b) Witnesses for $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$.

Fig. 4: MDP benchmark: consensus-2-4 (528 states)

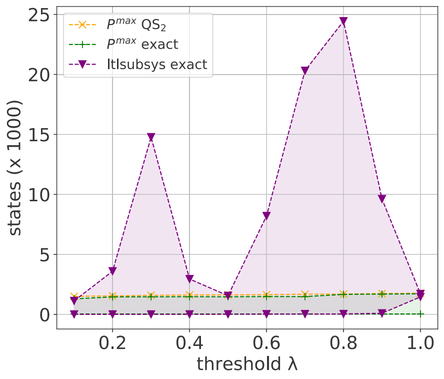
In Figure 3 it can be seen that the QS heuristics derived from the two polytopes $\mathcal{P}^{\max}$ and $\mathcal{P}^{\min}_{\geq 0}$ may produce different results. However, for both models one of them gives monotonically growing subsystems and outperforms COMICS. While QS$_2$ applied to $\mathcal{P}^{\min}_{\geq 0}$ performs better on crowds-5-8 (Figure 3a), it is the other way around on brp-512-2 (Figure 3b). In future work we intend to investigate what properties determine which of the two formulations performs better for a given DTMC.

MDP benchmarks. We consider two MDP models: the randomized consensus-N-K protocol of [10, 60] (with N processes and a bound K on the random walk) and the CSMA-N-K protocol for data channels [61] (where N is the number of stations, and K is the maximal backoff count). The results of both heuristic and exact computations can be seen in Figure 4 and Figure 5. Whereas the heuristics all needed less than 5 minutes, all MILP instances ran into the timeout except for the ones in Figure 4a. Whenever a MILP instance could not be solved optimally in 30 minutes, we plot both the found upper and lower bound, with the region in between shaded. It should be noted that the condition $\mathbf{Pr}^{\min}(\lozenge(\text{goal} \vee \text{fail}))$ holds for the instances of these models, and reachability properties, that we consider.

The comparison between the MILP formulation that we derived from $\mathcal{P}^{\max}(\lambda)$ and the one presented in [77, 78] (labeled by `ltlsubsys`, see also Section 6) shows that both compute comparable upper and lower bounds in Figure 4b, whereas `ltlsubsys` found worse upper bounds in Figure 5b. In all instances apart from Figure 4b the corresponding QS$_2$ heuristics performs well and generates subsystems that are as good, or better, than the best upper bounds computed by the MILPs in 30 minutes. As expected, the witnessing subsystems for $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$ tend to the entire state space as λ tends to the actual value $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\text{goal})$ (which is 1 in these two models). However, subsystems for $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$ may be substantially smaller even for large λ.



(a) Witnesses for $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$.



(b) Witnesses for $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$.

Fig. 5: MDP benchmark: CSMA-3-2 (36,850 states)

8 Conclusion

In this paper we brought together two a priori unrelated notions in the context of probabilistic reachability constraints: on the one hand Farkas certificates, which are vectors satisfying certain linear inequalities that we derive using MDP-specific variants of Farkas' Lemma, and on the other hand witnessing subsystems, which provide insight into which parts of the system are essential for the satisfaction of the considered property. This connection reduces the computation of minimal (respectively, small) witnessing subsystems to finding a Farkas certificate with a maximal (respectively, large) number of zeros. Furthermore, it leads to a unified notion of witnessing subsystem for $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$ and $\mathbf{Pr}^{\min}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$.

We showed that the decision version of computing minimal witnessing subsystems is NP-complete for acyclic DTMCs and introduced heuristics for the computation of small witnesses based on Farkas certificates. Experiments of the heuristics exhibited competitive results compared to the approach implemented in Comics and showed that they scale well with the system size and threshold. As expected, computing minimal subsystems using the derived MILP formulations consumed significantly more time than the heuristics and often triggered timeouts. The upper and lower bounds that were computed in the given time by the new MILP formulation for $\mathbf{Pr}^{\max}_{s_0}(\lozenge\,\text{goal}) \geq \lambda$ were comparable to known techniques.

We have considered MDPs in which the probability to reach goal or fail is positive under each scheduler. In future work, we plan to extend our techniques to weaken this assumption. Exploring how vertex enumeration techniques could be adapted to the MDP-specific form of the Farkas polytopes is another interesting line of future work. We also plan to implement a tool for working with Farkas certificates in practice, which encompasses their generation as well as their independent validation.

References

1. Ábrahám, E., Becker, B., Dehnert, C., Jansen, N., Katoen, J., Wimmer, R.: Counterexample generation for discrete-time Markov models: An introductory survey. In: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014. pp. 65–121 (2014), https://doi.org/10.1007/978-3-319-07317-0_3
2. de Alfaro, L.: Formal verification of probabilistic systems. Ph.D. thesis, Stanford University, Department of Computer Science (1997)
3. de Alfaro, L.: Temporal logics for the specification of performance and reliability. In: STACS 97. pp. 165–176. Springer, Berlin, Heidelberg (1997)
4. Aljazzar, H., Leitner-Fischer, F., Leue, S., Simeonov, D.: Dipro - A tool for probabilistic counterexample generation. In: Model Checking Software - 18th International SPIN Workshop 2011. pp. 183–187 (2011), https://doi.org/10.1007/978-3-642-22306-8_13
5. Aljazzar, H., Leue, S.: Extended directed search for probabilistic timed reachability. In: Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006. pp. 33–51 (2006), https://doi.org/10.1007/11867340_4
6. Aljazzar, H., Leue, S.: Generation of counterexamples for model checking of Markov decision processes. In: Sixth International Conference on the Quantitative Evaluation of Systems, QEST 2009. pp. 197–206 (2009), https://doi.org/10.1109/QEST.2009.10
7. Aljazzar, H., Leue, S.: Directed explicit state-space search in the generation of counterexamples for stochastic model checking. IEEE Trans. Software Eng. **36**(1), 37–60 (2010), https://doi.org/10.1109/TSE.2009.57
8. Amaldi, E., Kann, V.: On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems. Theoretical Computer Science **209**(1), 237 – 260 (1998), http://www.sciencedirect.com/science/article/pii/S0304397597001151
9. Andrés, M.E., D'Argenio, P.R., van Rossum, P.: Significant diagnostic counterexamples in probabilistic model checking. In: Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008. pp. 129–148 (2008), https://doi.org/10.1007/978-3-642-01702-5_15
10. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms **11**(3), 441–461 (1990), https://doi.org/10.1016/0196-6774(90)90021-6
11. Avis, D., Fukuda, K.: A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. Discrete & Computational Geometry **8**, 295–313 (1992), https://doi.org/10.1007/BF02293050
12. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Applied Mathematics **65**, 21–46 (1993)
13. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press, Cambridge, MA (2008)
14. Balinski, M.L.: An algorithm for finding all vertices of convex polyhedral sets. Journal of the Society for Industrial and Applied Mathematics **9**(1), 72–88 (1961), https://doi.org/10.1137/0109008
15. Bernasconi, A., Menghi, C., Spoletini, P., Zuck, L.D., Ghezzi, C.: From model checking to a temporal proof for partial models. In: Software Engineering and Formal Methods - 15th International Conference, SEFM 2017. pp. 54–69 (2017), https://doi.org/10.1007/978-3-319-66197-1_4
16. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Foundations of Software Technology and Theoretical Computer Science. pp. 499–513. Springer, Berlin, Heidelberg (1995)

17. Blum, M., Kannan, S.: Designing programs that check their work. Journal of the ACM **42**(1), 269–291 (1995), https://doi.org/10.1145/200836.200880
18. Braitling, B., Wimmer, R., Becker, B., Jansen, N., Ábrahám, E.: Counterexample generation for Markov chains using SMT-based bounded model checking. In: Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011. pp. 75–89 (2011), https://doi.org/10.1007/978-3-642-21461-5_5
19. Brázdil, T., Chatterjee, K., Chmelik, M., Fellner, A., Kretínský, J.: Counterexample explanation by learning small strategies in Markov decision processes. In: Computer Aided Verification - 27th International Conference, CAV 2015. pp. 158–177 (2015), https://doi.org/10.1007/978-3-319-21690-4_10
20. Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínský, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of Markov Decision Processes Using Learning Algorithms. In: Automated Technology for Verification and Analysis (ATVA 2014). pp. 98–114 (2014), https://doi.org/10.1007/978-3-319-11936-6_8
21. Bremner, D., Fukuda, K., Marzetta, A.: Primal–dual methods for vertex and facet enumeration. Discrete & Computational Geometry **20**(3), 333–357 (1998), https://doi.org/10.1007/PL00009389
22. Bussieck, M.R., Lübbecke, M.E.: The vertex set of a 0/1 polytope is strongly $\mathcal{P}$-enumerable. Computational Geometry Theory and Applications **11**(2), 103–109 (1998)
23. Ceska, M., Hensel, C., Junges, S., Katoen, J.: Counterexample-driven synthesis for probabilistic program sketches. In: Formal Methods - The Next 30 Years - Third World Congress, FM 2019. pp. 101–120 (2019), https://doi.org/10.1007/978-3-030-30942-8_8
24. Chadha, R., Viswanathan, M.: A counterexample-guided abstraction-refinement framework for Markov decision processes. ACM Transactions on Computational Logic **12**(1), 1:1–1:49 (2010), http://doi.acm.org/10.1145/1838552.1838553
25. Chatterjee, K., Chmelik, M., Daca, P.: CEGAR for qualitative analysis of probabilistic systems. In: Computer Aided Verification - 26th International Conference, CAV 2014. pp. 473–490 (2014), https://doi.org/10.1007/978-3-319-08867-9_31
26. Ciesinski, F., Baier, C., Größer, M., Klein, J.: Reduction techniques for model checking Markov decision processes. In: 2008 Fifth International Conference on Quantitative Evaluation of Systems. pp. 45–54 (2008). https://doi.org/10.1109/QEST.2008.45
27. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003), https://doi.org/10.1145/876638.876643
28. Clarke, E.M., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002). pp. 19–29 (2002), https://doi.org/10.1109/LICS.2002.1029814
29. Clarke, E.M., Veith, H.: Counterexamples revisited: Principles, algorithms, applications. In: Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday. pp. 208–224 (2003), https://doi.org/10.1007/978-3-540-39910-0_9
30. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Computer Aided Verification, 15th International Conference, CAV 2003. pp. 420–432 (2003)
31. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: Proceedings of the 29th Annual Symposium on Founda-

tions of Computer Science. pp. 338–345. SFCS '88, IEEE Computer Society (1988), https://doi.org/10.1109/SFCS.1988.21950

32. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM **42**(4), 857–907 (1995), http://doi.acm.org/10.1145/210332.210339
33. Damman, B., Han, T., Katoen, J.: Regular expressions for PCTL counterexamples. In: Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008). pp. 179–188 (2008), https://doi.org/10.1109/QEST.2008.11
34. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: Process Algebra and Probabilistic Methods, Performance Modeling and Verification: Joint International Workshop, PAPM-PROBMIV 2001. pp. 39–56 (2001), https://doi.org/10.1007/3-540-44804-7_3
35. Dyer, M.E.: The complexity of vertex enumeration methods. Mathematics of Operations Research **8**(3), 381–402 (1983), https://doi.org/10.1287/moor.8.3.381
36. Dyer, M.E., Proll, L.G.: An algorithm for determining all extreme points of a convex polytope. Mathematical Programming **12**(1), 81–96 (1977), https://doi.org/10.1007/BF01593771
37. Etessami, K., Kwiatkowska, M., Vardi, M.Y., Yannakakis, M.: Multi-Objective Model Checking of Markov Decision Processes. Logical Methods in Computer Science **4**(4) (2008), https://lmcs.episciences.org/990
38. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und angewandte Mathematik **124**, 1–27 (1902), http://eudml.org/doc/149129
39. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011. pp. 112–127 (2011), https://doi.org/10.1007/978-3-642-19835-9_11
40. Fukuda, K., Liebling, T.M., Margot, F.: Analysis of backtrack algorithms for listing all vertices and all faces of a convex polyhedron. Computational Geometry **8**(1), 1–12 (1997), http://www.sciencedirect.com/science/article/pii/0925772195000496
41. Fukuda, K., Prodon, A.: Double description method revisited. In: Combinatorics and Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference 1995. pp. 91–111 (1995), https://doi.org/10.1007/3-540-61576-8_77
42. Funke, F., Jantsch, S., Baier, C.: Farkas certificates and minimal witnesses for probabilistic reachability constraints (2019), https://arxiv.org/abs/1910.10636
43. Gurobi Optimization LLC, L.: Gurobi optimizer reference manual (2019), http://www.gurobi.com
44. Han, T., Katoen, J.: Counterexamples in probabilistic model checking. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007). pp. 72–86 (2007), https://doi.org/10.1007/978-3-540-71209-1_8
45. Han, T., Katoen, J., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Transactions on Software Engineering **35**(2), 241–257 (2009), https://doi.org/10.1109/TSE.2009.5
46. Hart, S., Sharir, M., Pnueli, A.: Termination of probabilistic concurrent program. ACM Transactions on Programming Languages and Systems **5**(3), 356–380 (1983), http://doi.acm.org/10.1145/2166.357214
47. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: Types for Proofs and Programs, International Workshop TYPES'93. pp. 127–165 (1993), https://doi.org/10.1007/3-540-58085-9_75
48. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Computer Aided Verification, 20th International Conference, CAV 2008. pp. 162–175 (2008), https://doi.org/10.1007/978-3-540-70545-1_16

49. Jansen, N., Ábrahám, E., Katelaan, J., Wimmer, R., Katoen, J., Becker, B.: Hierarchical counterexamples for discrete-time Markov chains. In: Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011. pp. 443–452 (2011), https://doi.org/10.1007/978-3-642-24372-1_33
50. Jansen, N., Ábrahám, E., Volk, M., Wimmer, R., Katoen, J., Becker, B.: The COMICS tool - computing minimal counterexamples for dtmcs. In: Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012. pp. 349–353 (2012), https://doi.org/10.1007/978-3-642-33386-6_27
51. Jansen, N., Ábrahám, E., Zajzon, B., Wimmer, R., Schuster, J., Katoen, J., Becker, B.: Symbolic counterexample generation for discrete-time Markov chains. In: Formal Aspects of Component Software, 9th International Symposium, FACS 2012. pp. 134–151 (2012), https://doi.org/10.1007/978-3-642-35861-6_9
52. Jansen, N., Wimmer, R., Ábrahám, E., Zajzon, B., Katoen, J., Becker, B., Schuster, J.: Symbolic counterexample generation for large discrete-time Markov chains. Science of Computer Programming **91**, 90–114 (2014), https://doi.org/10.1016/j.scico.2014.02.001
53. Jr., M.C., Jansen, N., Junges, S., Katoen, J.: Shepherding hordes of Markov chains. In: Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019. pp. 172–190 (2019), https://doi.org/10.1007/978-3-030-17465-1_10
54. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, 1972. pp. 85–103. Springer US, Boston, MA (1972)
55. Khachiyan, L., Boros, E., Borys, K., Elbassioni, K., Gurvich, V.: Generating all vertices of a polyhedron is hard. Discrete & Computational Geometry **39**(1), 174–190 (2008), https://doi.org/10.1007/s00454-008-9050-5
56. Kuntz, M., Leitner-Fischer, F., Leue, S.: From probabilistic counterexamples via causality to fault trees. In: Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security (SAFECOMP). pp. 71–84 (2011), https://doi.org/10.1007/978-3-642-24270-0_6
57. Kupferman, O., Vardi, M.Y.: From complementation to certification. In: Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004. pp. 591–606 (2004), https://doi.org/10.1007/978-3-540-24730-2_43
58. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification - 23rd International Conference, CAV 2011. pp. 585–591 (2011), https://doi.org/10.1007/978-3-642-22110-1_47
59. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012. pp. 203–204 (2012), https://doi.org/10.1109/QEST.2012.14
60. Kwiatkowska, M.Z., Norman, G., Segala, R.: Automated verification of a randomized distributed consensus protocol using cadence SMV and PRISM. In: Computer Aided Verification, 13th International Conference, CAV 2001. pp. 194–206 (2001), https://doi.org/10.1007/3-540-44585-4_17
61. Kwiatkowska, M.Z., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. Information and Computation **205**(7), 1027–1077 (2007), https://doi.org/10.1016/j.ic.2007.01.004
62. Mangasarian, O.: Nonlinear Programming. Classics in Applied Mathematics, Society for Industrial and Applied Mathematics (1994)

63. Mattheiss, T.H.: An algorithm for determining irrelevant constraints and all vertices in systems of linear inequalities. Operations Research **21**(1), 247–260 (1973), http://www.jstor.org/stable/169104
64. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011), https://doi.org/10.1016/j.cosrev.2010.09.009
65. Naiman, D.Q., Scheinerman, E.R.: Arbitrage and geometry. Preprint (2017), https://arxiv.org/abs/1709.07446
66. Namjoshi, K.S.: Certifying model checkers. In: Computer Aided Verification, 13th International Conference, CAV 2001. pp. 2–13 (2001), https://doi.org/10.1007/3-540-44585-4_2
67. Peled, D.A., Pnueli, A., Zuck, L.D.: From falsification to verification. In: FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science. pp. 292–304 (2001), https://doi.org/10.1007/3-540-45294-X_25
68. Provan, J.S.: Efficient enumeration of the vertices of polyhedra associated with network LP's. Mathematical Programming **63**(1), 47–64 (1994), https://doi.org/10.1007/BF01582058
69. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for web transactions. ACM Transactions on Information and System Security **1**(1), 66–92 (1998), https://doi.org/10.1145/290163.290168
70. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons, Inc., New York, NY, USA (1986)
71. Schrijver, A.: A course in combinatorial optimization. Lecture notes (2017), https://homepages.cwi.nl/~lex/files/dict.pdf
72. Shmatikov, V.: Probabilistic analysis of an anonymity system. Journal of Computer Security **12**(3-4), 355–377 (2004)
73. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite state programs. In: Proceedings of the 26th Annual Symposium on Foundations of Computer Science. pp. 327–338. SFCS '85, IEEE Computer Society (1985), https://doi.org/10.1109/SFCS.1985.12
74. 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 86). pp. 332–344 (1986)
75. Vohra, R.V.: The ubiquitous farkas lemma. In: Perspectives in Operations Research: Papers in Honor of Saul Gass' 80th Birthday. pp. 199–210. Springer US, Boston, MA (2006), https://doi.org/10.1007/978-0-387-39934-8_11
76. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009. pp. 366–380 (2009), https://doi.org/10.1007/978-3-540-93900-9_29
77. Wimmer, R., Jansen, N., Ábrahám, E., Becker, B., Katoen, J.: Minimal critical subsystems for discrete-time markov models. In: Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012. pp. 299–314 (2012), https://doi.org/10.1007/978-3-642-28756-5_21
78. Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J., Becker, B.: Minimal counterexamples for linear-time probabilistic verification. Theoretical Computer Science **549**, 61–100 (2014), https://doi.org/10.1016/j.tcs.2014.06.020

Simple Strategies in Multi-Objective MDPs*

Florent Delgrange[1,2,†], Joost-Pieter Katoen[1], Tim Quatmann[1], and Mickael Randour[2]

[1] RWTH Aachen University, Aachen, Germany
[2] UMONS – Université de Mons, Mons, Belgium

Abstract We consider the verification of multiple expected reward objectives at once on Markov decision processes (MDPs). This enables a trade-off analysis among multiple objectives by obtaining a Pareto front. We focus on strategies that are easy to employ and implement. That is, strategies that are pure (no randomization) and have bounded memory. We show that checking whether a point is achievable by a pure stationary strategy is NP-complete, even for two objectives, and we provide an `MILP` encoding to solve the corresponding problem. The bounded memory case is treated by a product construction. Experimental results using Storm and Gurobi show the feasibility of our algorithms.

1 Introduction

MDPs. Markov decision processes (MDPs) [4,3] are a key model in stochastic decision making. The classical setting involves a system subject to a stochastic model of its environment, and the goal is to synthesize a system controller, represented as a *strategy* for the MDP, ensuring a given level of *expected performance*. Tools such as Prism [30] and Storm [16] support MDP model checking.

Multi-objective MDPs. MDPs where the goal is to achieve a *combination* of objectives (rather than just one) are popular in e.g., AI [41] and verification [2]. This is driven by applications, where controllers have to fulfill multiple, potentially conflicting objectives, requiring a *trade-off* analysis. This includes multi-dimension MDPs [14,20,40,13] where weight vectors are aggregated at each step and MDPs where the specification mixes different views (e.g., average and worst case performance) of the same weight [11,8]. With multiple objectives, optimal strategies no longer exist in general: instead, *Pareto-optimal* strategies are considered. The Pareto front, i.e., the set of non-dominated achievable value vectors is usually non-trivial. Elaborate techniques are needed to explore it efficiently, e.g., [23,24].

Simple strategies. Another stumbling block in multi-objective MDPs is the complexity of strategies: Pareto-optimal strategies typically need both *memory* and *randomization*. A simple conjunction of reachability objectives already requires randomization and exponential memory (in the number of reachability sets) [40].

[†] currently affiliated with Vrije Universiteit Brussel.

* Research partially supported by F.R.S.-FNRS Grant n° F.4520.18 (*ManySynth*). Mickael Randour is an F.R.S.-FNRS Research Associate.

© The Author(s) 2020
A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 346–364, 2020.
https://doi.org/10.1007/978-3-030-45190-5_19

Some complex objectives even need infinite memory, e.g., [11,8]. In controller synthesis, strategies requiring randomization and/or (much) memory may not be practical. Limited-memory strategies are required on devices with limited resources [7]. Randomization is elegant and powerful from a theoretical view, but has practical limitations, e.g., it limits reproducibility which complicates debugging. Randomized strategies are also often despised for medical applications [33] and product design – all products should have the same design, not a random one. This motivates to consider the analysis of *simple strategies*, i.e., strategies using no randomization and a limited amount of memory (given as a parameter). While most works study the Pareto front among *all* strategies, we establish ways to explore efficiently the Pareto front among *simple* strategies only.

Problem statement. We consider pure (i.e., no randomization) and bounded-memory strategies and study two problems: (a) *achievability* queries – is it possible to achieve a given value vector – and (b) *approximation of the Pareto front*. Considering pure, bounded-memory strategies is natural as randomization can be traded for memory [12]: without randomization, optimal strategies may require arbitrarily large memory, (see Ex. 4). We study mixtures of *expected (accumulated) reward objectives*, covering various studied settings like reachability [20,40], shortest path [39,40,28,9] and total reward objectives [23,24].

Contributions. We first consider the achievability problem for pure stationary (i.e., memoryless) strategies and show that finding optimal strategies for multi-objective MDPs is NP-complete, even for two objectives. This contrasts the case of general strategies, where the problem is polynomial-time if the number of objectives is fixed [40]. We provide a *mixed integer linear program* (`MILP`) encoding. The crux lies in dealing with end components. The `MILP` is polynomial in the input MDP and the number of objectives. Inspired by [22], we give an alternative `MILP` encoding which is better suited for total reward objectives. To approximate the Pareto front under pure stationary strategies, we solve multiple `MILP` queries. This iteratively divides the solution space into achievable and non-achievable regions. Bounded-memory strategies are treated via a product construction. Our approach works for finite *and* infinite expected rewards.

Practical evaluation. We successfully compute Pareto fronts for 13 benchmarks using our implementation in Storm, exploiting the `MILP` solver Gurobi. Despite the hard nature of the problem, our experiments show that Pareto fronts for models with tens of thousands of states can be successfully approximated.

Related work. NP completeness for discounted rewards under pure strategies was shown in [14]. [19] claims that this generalizes to PCTL objectives but no proof is given. [42] treats multi-objective bounded MDPs whose transition probabilities are intervals. A set of Pareto optimal policies is computed using policy iteration and an efficient heuristic is exploited to compute a set of mutually non-dominated policies that are likely to be Pareto optimal. Pure stationary Pareto optimal strategies for discounted rewards are obtained in [44] using value-iteration but is restricted to small MDPs where all probabilities are 0 or 1. In [34],Tchebycheff-optimal strategies for discounted rewards are obtained via an LP approach; such strategies minimize the distance to a reference point and are not always pure.

2 Preliminaries

For a finite set Ω, let $\mathit{Dist}(\Omega) = \{\mu \colon \Omega \to [0,1] \mid \sum_{\omega \in \Omega} \mu(\omega) = 1\}$ be the set of probability distributions over Ω with support $\mathit{supp}(\mu) = \{\omega \in \Omega \mid \mu(\omega) > 0\}$. We write $\mathbb{R}_{\geq 0} = \{|x| \mid x \in \mathbb{R}\}$ and $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$ for the non-negative and extended real numbers, respectively. $\mathbf{1}^\ell = \langle 1, \dots, 1\rangle$ denotes the vector of size $\ell \in \mathbb{N}$ with all entries 1. We just write $\mathbf{1}$ if ℓ is clear. Let $\boldsymbol{p}[\![i]\!]$ denote the i^{th} entry and $\boldsymbol{p} \cdot \boldsymbol{p}'$ the dot product of $\boldsymbol{p}, \boldsymbol{p}' \in (\mathbb{R}_\infty)^\ell$. $\boldsymbol{p} \leq \boldsymbol{p}'$, $\boldsymbol{p} + \boldsymbol{p}'$, and $|\boldsymbol{p}|$ are entry-wise. For Boolean expression *cond*, let $[\mathit{cond}] = 1$ if *cond* is true and $[\mathit{cond}] = 0$ otherwise.

2.1 Markov Decision Processes, Strategies, and End Components

Definition 1 (Markov decision process [36]). *A* Markov decision process *(MDP) is a tuple $\mathcal{M} = \langle S, \mathit{Act}, \mathbf{P}, s_I\rangle$ with finite set of states S, initial state $s_I \in S$, finite set of actions Act, and transition function $\mathbf{P} \colon S \times \mathit{Act} \times S \to [0,1]$ with $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{0,1\}$ for all $s \in S$ and $\alpha \in \mathit{Act}$.*

We fix an MDP $\mathcal{M} = \langle S, \mathit{Act}, \mathbf{P}, s_I\rangle$. Intuitively, $\mathbf{P}(s, \alpha, s')$ is the probability to take a transition from s to s' when choosing action α. An infinite path in $\mathcal{M}$ is a sequence $\pi = s_0\alpha_1 s_1 \alpha_2 \dots \in (S \times \mathit{Act})^\omega$ with $\mathbf{P}(s_i, \alpha_{i+1}, s_{i+1}) > 0$ for all $i \in \mathbb{N}$. We write $\pi[i] = s_i$ for the $(i{+}1)$th state visited by π and define the length of π as $|\pi| = \infty$. A finite path is a finite prefix $\hat{\pi} = s_0 \alpha_1 \dots \alpha_n s_n$ of infinite path π, where $\mathit{last}(\hat{\pi}) = s_n \in S$, $|\hat{\pi}| = n$ and $\hat{\pi}[i] = s_i$ for $i \leq n$. The set of finite (infinite) paths in $\mathcal{M}$ is denoted by $\mathit{Paths}^{\mathcal{M}}_{\mathrm{fin}}$ ($\mathit{Paths}^{\mathcal{M}}_{\mathrm{inf}}$). The *enabled actions* at a state $s \in S$ are given by the set $\mathit{Act}(s) = \{\alpha \in \mathit{Act} \mid \exists\, s' \in S \colon \mathbf{P}(s, \alpha, s') > 0\}$. We assume $\mathit{Act}(s) \neq \emptyset$ for all s. If $|\mathit{Act}(s)| = 1$ for all $s \in S$, $\mathcal{M}$ is called a *Markov Chain (MC)*. We write $\mathcal{M}_s$ for the MDP obtained by replacing the initial state of $\mathcal{M}$ by $s \in S$. For $s \in S$ and $\alpha \in \mathit{Act}$, we define the set of successor states $\mathit{succ}(s, \alpha) = \{s' \mid \mathbf{P}(s, \alpha, s') > 0\}$. For $s' \in S$, the set of predecessor state-action pairs is given by $\mathit{pre}(s') = \{\langle s, \alpha\rangle \mid \mathbf{P}(s, \alpha, s') > 0\}$. For a set $\mathcal{E} \subseteq S \times \mathit{Act}$, we define $S[\![\mathcal{E}]\!] = \{s \in S \mid \exists\, \alpha \colon \langle s, \alpha\rangle \in \mathcal{E}\}$, $\mathit{Act}[\![\mathcal{E}]\!] = \{\alpha \in \mathit{Act} \mid \exists\, s \colon \langle s, \alpha \rangle \in \mathcal{E}\}$, and $\mathbf{P}[\![\mathcal{E}]\!](s, \alpha, s') = [\langle s, \alpha\rangle \in \mathcal{E}] \cdot [s' \in S[\![\mathcal{E}]\!]] \cdot \mathbf{P}(s, \alpha, s')$. We say $\mathcal{E}$ is *closed* for $\mathcal{M}$ if $\forall\, \langle s, \alpha\rangle \in \mathcal{E} \colon \alpha \in \mathit{Act}(s)$ and $\mathit{succ}(s, \alpha) \subseteq S[\![\mathcal{E}]\!]$.

Definition 2 (Sub-MDP). *The* sub-MDP *of $\mathcal{M}$, closed $\mathcal{E} \subseteq S \times \mathit{Act}$, and $s \in S[\![\mathcal{E}]\!]$ is given by $\mathcal{M}[\![\mathcal{E}, s]\!] = \langle S[\![\mathcal{E}]\!], \mathit{Act}[\![\mathcal{E}]\!], \mathbf{P}[\![\mathcal{E}]\!], s\rangle$. We also write $\mathcal{M}[\![\mathcal{E}]\!]$ for the sub-MDP $\mathcal{M}[\![\mathcal{E}, s]\!]$ and an arbitrary state $s \in S[\![\mathcal{E}]\!]$.*

Definition 3 (End Component). *A non-empty set $\mathcal{E} \subseteq S \times \mathit{Act}$ is an* end component (EC) *of $\mathcal{M}$ if $\mathcal{E}$ is closed for $\mathcal{M}$ and for each pair of states $s, s' \in S[\![\mathcal{E}]\!]$ there is a finite path $\hat{\pi} \in \mathit{Paths}^{\mathcal{M}[\![\mathcal{E}]\!]}_{\mathrm{fin}}$ with $\hat{\pi}[0] = s$ and $\mathit{last}(\hat{\pi}) = s'$. An EC $\mathcal{E}$ is* maximal*, if there is no other EC $\mathcal{E}'$ with $\mathcal{E} \subsetneq \mathcal{E}'$. The set of all maximal end components of $\mathcal{M}$ is $\mathit{MECS}(\mathcal{M})$.*

The maximal ECs of a Markov chain are also called *bottom strongly connected components (BSCCs)*. A strategy resolves nondeterminism in MDPs:

Definition 4 (Strategy). *A (general)* strategy *for MDP $\mathcal{M}$ is a function $\sigma\colon \mathit{Paths}_{\mathrm{fin}}^{\mathcal{M}} \to \mathit{Dist}(\mathit{Act})$ with $\mathit{supp}(\sigma(\hat{\pi})) \subseteq \mathit{Act}(\mathit{last}(\hat{\pi}))$ for all $\hat{\pi} \in \mathit{Paths}_{\mathrm{fin}}^{\mathcal{M}}$.*

Let σ be a strategy for $\mathcal{M}$. Intuitively, $\sigma(\hat{\pi})(\alpha)$ is the probability to perform action α after observing history $\hat{\pi} \in \mathit{Paths}_{\mathrm{fin}}^{\mathcal{M}}$. A strategy is *pure* if all histories are mapped to *Dirac distributions*, i.e., the support is a singleton. A strategy is *stationary* if its decisions only depend on the current state, i.e., $\forall\, \hat{\pi}, \hat{\pi}' \in \mathit{Paths}_{\mathrm{fin}}^{\mathcal{M}}\colon \mathit{last}(\hat{\pi}) = \mathit{last}(\hat{\pi}')$ implies $\sigma(\hat{\pi}) = \sigma(\hat{\pi}')$. We often assume $\sigma\colon S \to \mathit{Dist}(\mathit{Act})$ for stationary and $\sigma\colon S \to \mathit{Act}$ for pure stationary strategies σ. Let $\Sigma^{\mathcal{M}}$ and $\Sigma_{\mathrm{PS}}^{\mathcal{M}}$ be the sets of general and pure stationary strategies, respectively. A set of paths $\Pi \subseteq \mathit{Paths}_{\mathrm{inf}}^{\mathcal{M}}$ is *compliant* with $\sigma \in \Sigma^{\mathcal{M}}$ if for all $\pi = s_0\alpha_1 s_1 \cdots \in \Pi$ and prefixes $\hat{\pi}$ of π satisfy $\sigma(\hat{\pi})(\alpha_{|\hat{\pi}|+1}) > 0$. The *induced Markov chain* of $\mathcal{M}$ and $\sigma \in \Sigma_{\mathrm{PS}}^{\mathcal{M}}$ is given by $\mathcal{M}^{\sigma} = \mathcal{M}[\![\mathcal{E}^{\sigma}, s_I]\!]$ with $\mathcal{E}^{\sigma} = \{\langle s, \sigma(s)\rangle \mid s \in S\}$.

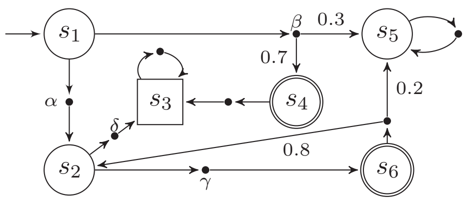
MDP $\mathcal{M}$ and strategy $\sigma \in \Sigma^{\mathcal{M}}$ induce a probability measure $\mathrm{Pr}_{\sigma}^{\mathcal{M}}$ on subsets $\Pi \subseteq \mathit{Paths}_{\mathrm{inf}}^{\mathcal{M}}$ given by a standard cylinder set construction [4,22]. The expected value of $X\colon \mathit{Paths}_{\mathrm{inf}}^{\mathcal{M}} \to \mathbb{R}_{\infty}$ is $\mathrm{E}_{\sigma}^{\mathcal{M}}(X) = \int_{\pi} X(\pi)\, d\mathrm{Pr}_{\sigma}^{\mathcal{M}}(\{\pi\})$. For $\sigma \in \Sigma_{\mathrm{PS}}^{\mathcal{M}}$, $\mathrm{Pr}_{\sigma}^{\mathcal{M}}$ and $\mathrm{E}_{\sigma}^{\mathcal{M}}$ coincide with the corresponding measures on MC $\mathcal{M}^{\sigma}$.

2.2 Objectives

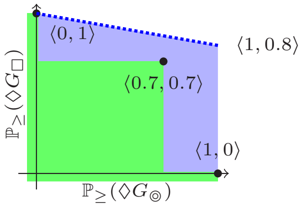
A reward structure $\mathbf{R}\colon S \times \mathit{Act} \times S \to \mathbb{R}_{\geq 0}$ assigns non-negative rewards to transitions. We accumulate rewards on (in)finite paths $\pi = s_0\alpha_1 s_1 \alpha_2 \ldots\colon \mathbf{R}(\pi) = \sum_{i=1}^{|\pi|} \mathbf{R}(s_{i-1}, \alpha_i, s_i)$. For a set of goal states $G \subseteq S$, let $\mathbf{R}\Diamond G(\pi) = \mathbf{R}(\hat{\pi})$, where $\hat{\pi}$ is the smallest prefix of π with $\mathit{last}(\hat{\pi}) \in G$ (or $\hat{\pi} = \pi$ if no such prefix exists). Intuitively, $\mathbf{R}\Diamond G(\pi)$ is the reward accumulated on π until a state in G is reached. A (reward) *objective* has the form $\mathbb{E}_{\sim}(\mathbf{R}\Diamond G)$ for $\sim\, \in \{\geq, \leq\}$. We write $\langle \mathcal{M}, \sigma, p\rangle \models \mathbb{E}_{\sim}(\mathbf{R}\Diamond G)$ iff $\mathrm{E}_{\sigma}^{\mathcal{M}}(\mathbf{R}\Diamond G) \sim p$, i.e., for $\mathcal{M}$ and σ, the expected accumulated reward until reaching G is at least (or at most) $p \in \mathbb{R}_{\infty}$. We call the objective *maximizing* if $\sim\, = \,\geq$ and *minimizing* otherwise. If $G = \emptyset$ (i.e., $\mathbf{R}\Diamond G(\pi) = \mathbf{R}(\pi)$ for all paths π), we call the objective a *total reward objective*. Let the reward structure $\mathbf{R}^G$ be given by $\mathbf{R}(s, \alpha, s') = [s' \in G]$. Then, $\mathrm{Pr}_{\sigma}^{\mathcal{M}}(\Diamond G) = \mathrm{E}_{\sigma}^{\mathcal{M}}(\mathbf{R}^G \Diamond G)$ for every $\sigma \in \Sigma^{\mathcal{M}}$, where $\Diamond G \subseteq \mathit{Paths}_{\mathrm{inf}}^{\mathcal{M}}$ denotes the set of paths that visit a state in G. We use $\mathbb{P}_{\sim}(\Diamond G)$ as a shortened for $\mathbb{E}_{\sim}(\mathbf{R}^G \Diamond G)$ and call such an objective a *reachability objective*.

Definition 5 (Multi-objective query). *For MDP $\mathcal{M}$, an ℓ-*dimensional multi-objective query *is a tuple $\mathcal{Q} = \langle \psi_1, \ldots, \psi_\ell\rangle$ of ℓ objectives $\psi_j = \mathbb{E}_{\sim_j}(\mathbf{R}_j \Diamond G_j)$.*

Each objective ψ_j considers a different reward structure $\mathbf{R}_j$. The MDP $\mathcal{M}$, strategy σ, and point $\boldsymbol{p} \in (\mathbb{R}_{\infty})^{\ell}$ satisfy a multi-objective query $\mathcal{Q} = \langle \psi_1, \ldots, \psi_\ell\rangle$ (written $\langle \mathcal{M}, \sigma, \boldsymbol{p}\rangle \models \mathcal{Q}$) iff $\forall\, j\colon\ \langle \mathcal{M}, \sigma, \boldsymbol{p}[\![j]\!]\rangle \models \psi_j$. Then, we also say σ *achieves* $\boldsymbol{p}$ and call $\boldsymbol{p}$ *achievable*. Let $\mathit{Ach}^{\mathcal{M}}(\mathcal{Q})$ ($\mathit{Ach}_{\mathrm{PS}}^{\mathcal{M}}(\mathcal{Q})$) denote the set of points achieved by a general (pure stationary) strategy. The *closure* of a set $P \subseteq (\mathbb{R}_{\infty})^{\ell}$ with respect to query $\mathcal{Q}$ is $\mathit{cl}^{\mathcal{Q}}(P) = \{\boldsymbol{p} \in (\mathbb{R}_{\infty})^{\ell} \mid \exists\, \boldsymbol{p}' \in P\colon\ \forall\, j\colon\ \boldsymbol{p}'[\![j]\!] \sim_j \boldsymbol{p}[\![j]\!]\}$. For $\boldsymbol{p}, \boldsymbol{p}' \in (\mathbb{R}_{\infty})^{\ell}$, we say that $\boldsymbol{p}$ *dominates* $\boldsymbol{p}'$ if $\boldsymbol{p}' \in \mathit{cl}^{\mathcal{Q}}(\{\boldsymbol{p}\})$. In this case, $\langle \mathcal{M}, \sigma, \boldsymbol{p}\rangle \models \mathcal{Q}$ implies $\langle \mathcal{M}, \sigma, \boldsymbol{p}'\rangle \models \mathcal{Q}$ for any $\sigma \in \Sigma^{\mathcal{M}}$. We are interested in the Pareto front, which is the set of non-dominated achievable points.



(a) MDP $\mathcal{M}$ with $G_{\circledcirc} = \{s_4, s_6\}$ and $G_{\square} = \{s_3\}$



(b) $Ach^{\mathcal{M}}(\mathcal{Q})$ and $Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$

Figure 1: An MDP and a plot of the pure stationary and general Pareto fronts.

Definition 6 (Pareto front). *The* (general) Pareto front *for* $\mathcal{M}$ *and* $\mathcal{Q}$ *is* $Pareto^{\mathcal{M}}(\mathcal{Q}) = \left\{ \boldsymbol{p} \in Ach^{\mathcal{M}}(\mathcal{Q}) \mid \forall\, \boldsymbol{p'} \in Ach^{\mathcal{M}}(\mathcal{Q})\colon\ \boldsymbol{p} \in cl^{\mathcal{Q}}(\{\boldsymbol{p'}\}) \implies \boldsymbol{p} = \boldsymbol{p'} \right\}$.

The Pareto front is the smallest set $P \subseteq (\mathbb{R}_{\infty})^{\ell}$ with $cl^{\mathcal{Q}}(P) = Ach^{\mathcal{M}}(\mathcal{Q})$. In a similar way, we define the *pure stationary* Pareto front $Pareto^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$ which only consider points in $Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$.

Example 1. Let $\mathcal{M}$ be the MDP in Fig. 1a and $\mathcal{Q} = \langle \mathbb{P}_{\geq}(\Diamond G_{\circledcirc}), \mathbb{P}_{\geq}(\Diamond G_{\square})\rangle$. A pure stationary strategy choosing β at s_1 reaches both, $s_4 \in G_{\circledcirc}$ and $s_3 \in G_{\square}$ with probability 0.7 and thus achieves $\langle 0.7, 0.7\rangle$. Similarly, $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$ are achievable by a pure stationary strategy. Point $\langle 1, 0.8\rangle$ is achievable by a non-stationary pure strategy that chooses α at s_1, γ at the first visit of s_2, and δ in all other cases. Changing this strategy by picking γ only with probability 0.5 achieves $\langle 0.5, 0.9\rangle$. Fig. 1b illustrates $Pareto^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$ (dots), $Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$ (green area), $Pareto^{\mathcal{M}}(\mathcal{Q})$ (dotted line), and $Ach^{\mathcal{M}}(\mathcal{Q})$ (blue and green area).

3 Deciding Achievability

The achievability problem asks whether a given point is achievable.

General Multi-objective Achievability Problem (GMA)

Input: MDP $\mathcal{M}$, ℓ-dimensional multi-objective query $\mathcal{Q}$, point $\boldsymbol{p} \in (\mathbb{R}_{\infty})^{\ell}$
Output: Yes iff $\boldsymbol{p} \in Ach^{\mathcal{M}}(\mathcal{Q})$ holds

For GMA, the point can be achieved by a general strategy that can potentially make use of memory and randomization. As discussed earlier, this class of strategies is not suitable for various applications. In this work, we focus on a variant of the achievability problem that only considers pure stationary strategies. Sect. 5 also addresses pure strategies that can store more information from the history, e.g., whether a goal state set has been reached already.

Pure Stationary Multi-objective Achievability Problem (PSMA)

Input: MDP $\mathcal{M}$, ℓ-dimensional multi-objective query $\mathcal{Q}$, point $\boldsymbol{p} \in (\mathbb{R}_{\infty})^{\ell}$
Output: Yes iff $\boldsymbol{p} \in Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$ holds

3.1 Complexity Results

`GMA` is PSPACE hard (already with only reachability objectives) [40] and solvable within exponential runtime [20,23]. To the best of our knowledge, a PSPACE upper bound on the complexity of `GMA` is unknown. This complexity is rooted in the dimension ℓ of the query $\mathcal{Q}$: for fixed ℓ, the algorithms of [20,23] have polynomial runtime. In contrast, `PSMA` is NP-complete, even if restricted to 2 objectives.

Lemma 1. `PSMA` *with only reachability objectives is NP-hard.*

Proof. The result follows by a reduction from the subset sum problem. Given $n \in \mathbb{N}$, $\boldsymbol{a} \in \mathbb{N}^n$ and $z \in \mathbb{N}$, the subset sum problem is to decide the existence of $\boldsymbol{v} \in \{0,1\}^n$ such that $\boldsymbol{v} \cdot \boldsymbol{a} = z$. This problem is NP-complete [25]. For a given instance of the subset sum problem, we construct the MDP $\mathcal{M}^\star = \langle S, Act, \mathbf{P}, s_I \rangle$ with state space $S = \{s_I, s_1, \dots, s_n, g_1, g_2\}$, actions $Act = \{\alpha, Y, N\}$, and for all $i \in \{1, \dots, n\}$, $\mathbf{P}(s_I, \alpha, s_i) = \frac{\boldsymbol{a}[\![i]\!]}{\mathbf{1} \cdot \boldsymbol{a}}$ and $\mathbf{P}(s_i, Y, g_1) = \mathbf{P}(s_i, N, g_2) = 1$. States g_1 and g_2 are made absorbing, i.e., $\mathbf{P}(g_1, \alpha, g_1) = \mathbf{P}(g_2, \alpha, g_2) = 1$.

We claim that the `PSMA` problem for $\mathcal{M}^\star$, $\mathcal{Q} = \langle \mathbb{P}_{\geq}(\lozenge\{g_1\}), \mathbb{P}_{\geq}(\lozenge\{g_2\}) \rangle$, and $\boldsymbol{p} = \left(\frac{z}{\mathbf{1} \cdot \boldsymbol{a}}, 1 - \frac{z}{\mathbf{1} \cdot \boldsymbol{a}}\right)$ answers "yes" iff there is a vector $\boldsymbol{v}$ satisfying the subset sum problem for n, $\boldsymbol{a}$ and z. Consider the bijection $f\colon \Sigma_{\mathrm{PS}}^{\mathcal{M}^\star} \to \{0,1\}^n$ with $f(\sigma)[\![i]\!] = [\sigma(s_i){=}Y]$ for all $\sigma \in \Sigma_{\mathrm{PS}}^{\mathcal{M}^\star}$ and $i \in \{1, \dots, n\}$. We get $\mathrm{Pr}_\sigma^{\mathcal{M}^\star}(\lozenge\{g_1\}) = \sum_{i=1}^{n} \frac{\boldsymbol{a}[\![i]\!]}{\mathbf{1} \cdot \boldsymbol{a}} [\sigma(s_i){=}Y] = \frac{f(\sigma) \cdot \boldsymbol{a}}{\mathbf{1} \cdot \boldsymbol{a}}$. Moreover, $\mathrm{Pr}_\sigma^{\mathcal{M}^\star}(\lozenge\{g_2\}) = 1 - \mathrm{Pr}_\sigma^{\mathcal{M}^\star}(\lozenge\{g_1\}) = 1 - \frac{f(\sigma) \cdot \boldsymbol{a}}{\mathbf{1} \cdot \boldsymbol{a}}$. It follows that σ achieves $\boldsymbol{p}$ iff $f(\sigma)$ is a solution to the instance of the subset sum problem. Our construction is inspired by similar ideas from [14,40].

Lemma 2 ([14]). `PSMA` *with only total reward objectives is NP-hard.*

Theorem 1. `PSMA` *is NP-complete.*

Proof. Containment follows by guessing a pure stationary strategy and evaluating it on the individual objectives. This can be done in polynomial time [4]. Hardness follows by either Lemma 1 or 2.

Proofs of Lemmas 1 and 2 only consider 2-dimensional multi-objective queries. Hence, in contrast to `GMA`, the hardness of `PSMA` is not due to the size of the query.

Corollary 1. `PSMA` *with only two objectives is NP-complete.*

3.2 A Mixed Integer Linear Programming Approach

An MDP $\mathcal{M} = \langle S, Act, \mathbf{P}, s_I \rangle$ has exactly $|\Sigma_{\mathrm{PS}}^{\mathcal{M}}| = \prod_{s \in S} |Act(s)|$ many pure stationary strategies. A simple algorithm for `PSMA` enumerates all $\sigma \in \Sigma_{\mathrm{PS}}^{\mathcal{M}}$ and checks whether $\langle \mathcal{M}, \sigma, \boldsymbol{p} \rangle \models \mathcal{Q}$ holds. In practice, however, such a brute-force approach is not feasible. For the MDPs that we consider in our experiments in Sect. 6, the number of pure stationary strategies often exceeds $10^{10\,000}$. Instead, our approach is to encode an instance for `PSMA` as an `MILP` problem.

Mixed Integer Linear Programming Problem (`MILP`)

Input: $\ell, m, n \in \mathbb{N}$, $\boldsymbol{A} \in \mathbb{Q}^{n\times(\ell+m)}$, $\boldsymbol{b} \in \mathbb{Q}^n$, $\boldsymbol{c} \in \mathbb{Q}^{\ell+m}$

Output: $\begin{cases} \boldsymbol{x} \in \arg\max_{\boldsymbol{x}\in\mathcal{X}} \boldsymbol{c}^T\boldsymbol{x} & \text{if } \mathcal{X} \neq \emptyset \\ \texttt{infeasible} & \text{if } \mathcal{X} = \emptyset \end{cases}$ with $\mathcal{X} = \{\boldsymbol{x} \in \mathbb{Z}^\ell \times \mathbb{R}^m \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$

For an `MILP` instance as above, each of the n rows of the inequation system $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ represent a *constraint* that is linear over the ℓ integral and m real-valued *variables* given by $\boldsymbol{x}$. We call the constraints *feasible* if there is a solution to the inequation system. The task is to decide whether the constraints are feasible and if so, find a solution that maximizes a linear optimization function $\boldsymbol{c}^T\boldsymbol{x}$. The optimization function can be omitted if we are only interested in feasibility. `MILP` is NP-complete [35]. However, tools such as Gurobi [27] and SCIP [26] implement practically efficient algorithms that can solve large instances.

For the rest of this section, let $\mathcal{M} = \langle S, Act, \mathbf{P}, s_I\rangle$, $\mathcal{Q} = \langle \psi_1, \dots, \psi_\ell\rangle$ with $\psi_j = \mathbb{E}_{\sim_j}(\mathbf{R}_j \lozenge G_j)$, and $\boldsymbol{p} \in (\mathbb{R}_\infty)^\ell$ be an instance for `PSMA`. We provide a translation of the `PSMA` instance to an instance for `MILP` that has a feasible solution iff $\boldsymbol{p} \in Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$. The `MILP` encoding considers integer variables to encode a pure stationary strategy $\sigma \in \Sigma^{\mathcal{M}}_{\mathrm{PS}}$. The other variables and constraints encode the expected reward for each objective on the induced MC $\mathcal{M}^\sigma$.

3.3 Unichain MDP and Finite Rewards

Restriction 1 (Unichain MDP). *MDP $\mathcal{M}$ has exactly one end component.*

Restriction 2 (Reward Finiteness). $\mathrm{E}^{\mathcal{M}_s}_\sigma(\mathbf{R}_j \lozenge G_j) < \infty$ *holds for each objective* $\psi_j = \mathbb{E}_{\sim_j}(\mathbf{R}_j \lozenge G_j)$*, state s, and pure stationary strategy σ.*

For simplicity, we first explain our encoding for unichain MDP with finite reward. Sect. 3.5 lifts Restriction 1 and Sect. 3.6 lifts Restriction 2 with more details given in [17, App. B]. Sect. 3.4 presents an alternative to the encoding of this section, which is smaller but restricted to *total* reward objectives.

Fig. 2 shows the `MILP` encoding in case Restrictions 1 and 2 hold. We assume $\forall j\colon \boldsymbol{p}[\![j]\!] \neq \infty$ for the point $\boldsymbol{p}$ since (i) $\mathrm{E}^{\mathcal{M}}_\sigma(\mathbf{R}_j \lozenge G_j) \leq \infty$ holds trivially and (ii) $\mathrm{E}^{\mathcal{M}}_\sigma(\mathbf{R}_j \lozenge G_j) \geq \infty$ will never hold due to Restriction 2. For $j \in \{1, \dots, \ell\}$, let $S^j_0 = \{s \in S \mid \forall \sigma \in \Sigma^{\mathcal{M}}\colon \mathrm{E}^{\mathcal{M}}_\sigma(\mathbf{R}_j \lozenge G_j) = 0\}$ and $S^j_? = \{s \in S \setminus S^j_0 \mid s$ can be reached from s_I without visiting a state in $S^j_0\}$. These sets can be obtained a priori by analyzing the graph structure of $\mathcal{M}$ [4]. Moreover, we consider upper bounds $U^j_s \in \mathbb{Q}$ for the expected reward at state $s \in S^j_?$ such that $U^j_s \geq \max_{\sigma\in\Sigma^{\mathcal{M}}} \mathrm{E}^{\mathcal{M}_s}_\sigma(\mathbf{R}_j \lozenge G_j)$. We compute such upper bounds using single-objective model checking techniques [4,5]. The `MILP` encoding applies the characterization of expected rewards for MCs as a *linear equation system* [4].

Lemma 3. *For every $\sigma \in \Sigma^{\mathcal{M}}_{\mathrm{PS}}$, the following equation system has a unique solution $\Phi\colon \{x_s \mid s \in S\} \to \mathbb{R}^{|S|}$ satisfying $\Phi(x_s) = \mathrm{E}^{\mathcal{M}_s}_\sigma(\mathbf{R}_j \lozenge G_j)$:*

$$\forall s \in S^j_0\colon\ x_s = 0 \qquad \forall s \in S^j_?\colon\ x_s = \sum_{s'\in S} \mathbf{P}(s, \sigma(s), s') \cdot \big(x_{s'} + \mathbf{R}(s, \sigma(s), s')\big)$$

```
$\forall\, s \in S$:                                                    $\triangleright$ *Select an action at each state*
    $\forall\, \alpha \in Act(s)$:    $a_{s,\alpha} \in \{0,1\}$                                            (1)
        $\sum_{\alpha \in Act(s)} a_{s,\alpha} = 1$                                                         (2)
$\forall\, j \in \{1,\dots,\ell\}$ :                                    $\triangleright$ *Compute expected reward values*
    $\forall\, s \in S_0^j$:    $x_s^j = 0$                                                                  (3)
    If $\psi_j$ is maximising, $\pm = +$ and $[\min] = 0$. Otherwise, $\pm = -$ and $[\min] = 1$.
    $\forall\, s \in S_?^j$:    $\pm x_s^j \in [0, U_s^j]$                                                   (4)
        $\forall\, \alpha \in Act(s)$:    $\pm x_{s,\alpha}^j \in [0, U_s^j]$                                (5)
            $x_{s,\alpha}^j \le \sum_{s' \in S} \mathbf{P}(s,\alpha,s') \cdot \big(x_{s'}^j \pm \mathbf{R}_j(s,\alpha,s')\big)$   (6)
            $x_{s,\alpha}^j \le U_s^j \cdot (a_{s,\alpha} - [\min])$                                         (7)
        $x_s^j \le \sum_{\alpha \in Act(s)} x_{s,\alpha}^j + [\min] \cdot (|Act(s)| - 1) \cdot U_s^j$       (8)
    $\pm x_{s_I}^j \sim_j \boldsymbol{p}[\![j]\!]$                     $\triangleright$ *Assert value at initial state*  (9)
```

Figure 2: `MILP` encoding for unichain MDP and finite rewards.

Proof. Since $\mathcal{M}$ is unichain and we do not collect infinite reward, the only EC of $\mathcal{M}$ (i.e., the only BSCC of $\mathcal{M}^\sigma$ for any σ) either contains a goal state or only contains transitions with zero reward. It follows that $\forall\, \sigma \in \Sigma_{\mathrm{PS}}^{\mathcal{M}}\colon\ \Pr_\sigma^{\mathcal{M}}(\Diamond S_0^j) = 1$. Lemma 3 follows by standard arguments for MCs with rewards [4, Section 10.5.1].

We discuss the intuition of each constraint in Fig. 2. Let $\Phi\colon \mathit{Var} \to \mathbb{R}$ be an assignment of the occurring variables Var to values. Φ is a solution of the constraints if all (in)equations are satisfied upon replacing all variables v by $\Phi(v)$.

Lines 1 and 2 encode a strategy $\sigma \in \Sigma_{\mathrm{PS}}^{\mathcal{M}}$ by considering a binary variable $a_{s,\alpha}$ for each state s and enabled action α such that $\sigma(s)(\alpha) = 1$ iff $\Phi(a_{s,\alpha}) = 1$ for a solution Φ. Due to Line 2, exactly one action has to be chosen at each state.

Lines 3 to 8 encode for each objective ψ_j the expected rewards obtained for the encoded strategy σ. For every $s \in S$, the variable x_s^j represents a (lower or upper) bound on the expected reward at s. Line 3 sets this value for all $s \in S_0^j$, reflecting the analogous case from Lemma 3. For $s \in S_?^j$, we distinguish maximizing ($\sim_j = \ge$) and minimizing ($\sim_j = \le$) objectives ψ_j.

For maximizing ψ_j, we have $\Phi(x_s^j) \le \mathrm{E}_\sigma^{\mathcal{M}_s}(\mathbf{R}_j \Diamond G_j)$ for every solution Φ. This is achieved by considering a variable $x_{s,\alpha}^j$ for each enabled action $\alpha \in Act(s)$. In Line 6, we use the equation system characterization from Lemma 3 to assert that the value of $x_{s,\alpha}^j$ can not be greater than the expected reward at s, given that the encoded strategy σ selects α. If σ does not select α (i.e., $\Phi(a_{s,\alpha}) = 0$), Line 7 implies $\Phi(x_{s,\alpha}^j) = 0$. Otherwise, this constraint has no effect. Line 8 ensures that every solution satisfies $\Phi(x_s^j) \le \Phi(x_{s,\alpha}^j) \le \mathrm{E}_\sigma^{\mathcal{M}_s}(\mathbf{R}_j \Diamond G_j)$ for α with $\Phi(a_{s,\alpha}) = 1$.

For minimizing ψ_j, we have $-\Phi(x_s^j) \ge \mathrm{E}_\sigma^{\mathcal{M}_s}(\mathbf{R}_j \Diamond G_j)$ for every solution Φ, i.e., we consider the negated reward values. The encoding is as for maximizing objectives. However, Line 7 yields $\Phi(x^j_{s,\alpha}) = -U^j_s$ if α is not selected. Thus, in Line 8 we add U^j_s for each of the $(|Act(s)| - 1)$ non-selected actions.

Line 9 and our observations above yield $\mathrm{E}^{\mathcal{M}_s}_{\sigma}(\mathbf{R}_j \Diamond G_j) \geq \Phi(x^j_{s_I}) \geq \boldsymbol{p}[\![j]\!]$ for maximizing and $\mathrm{E}^{\mathcal{M}_s}_{\sigma}(\mathbf{R}_j \Diamond G_j) \leq -\Phi(x^j_{s_I}) \leq \boldsymbol{p}[\![j]\!]$ for minimizing objectives. Therefore, $\boldsymbol{p}$ is achievable if a solution Φ exists. On the other hand, if $\boldsymbol{p}$ is achievable by some $\sigma \in \Sigma^{\mathcal{M}}_{\mathrm{PS}}$, the solution Φ exists with $\Phi(a_{s,\alpha}) = \sigma(s)(\alpha)$, $\Phi(x^j_s) = \Phi(x^j_{s,\alpha}) = \pm\mathrm{E}^{\mathcal{M}_s}_{\sigma}(\mathbf{R}_j \Diamond G_j)$ if $\alpha = \sigma(s)$, and $\Phi(v) = 0$ for other $v \in Var$.

Theorem 2. *For unichain $\mathcal{M}$ and finite rewards, the constraints in Fig. 2 are feasible iff $\boldsymbol{p} \in Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$.*

Proposition 1. *The* `MILP` *encoding above considers $\mathcal{O}(|S| \cdot |Act| \cdot \ell)$ variables.*

3.4 Alternative Encoding for Total Rewards

We now consider `PSMA` instances where all objectives $\psi_j = \mathbb{E}_{\sim_j}(\mathbf{R}_j \Diamond G_j)$ are expected *total* reward objectives, i.e., $G_j = \emptyset$. For such instances, we can employ an encoding from [23] (restated in Lemma 4) for `GMA`. In fact, we can often translate reachability reward objectives to total reward objectives, e.g., if the set of goal states can not be left or if all objectives consider the same goal states.

Lemma 4 ([23]). *For $S_0 \subseteq S$, let $\Phi\colon Var \to \mathbb{R}_{\geq 0}$ be an assignment of variables $Var = \{y_{s,\alpha} \mid s \in S \setminus S_0, \alpha \in Act(s)\}$ and let σ_Φ be a stationary strategy satisfying $\sigma_\Phi(s)(\alpha) = \Phi(y_{s,\alpha}) / \sum_{\beta \in Act(s)} \Phi(y_{s,\beta})$ for all $s \in S \setminus S_0$ and $\alpha \in Act(s)$ for which the denominator is non-zero. Then, Φ is a solution to the equation system*

$$\forall\, s \in S \setminus S_0\colon \quad \sum_{\alpha \in Act(s)} y_{s,\alpha} \;=\; [s = s_I] \;+ \sum_{\langle s',\alpha'\rangle \in pre(s)} \mathbf{P}(s',\alpha',s) \cdot y_{s',\alpha'}$$

$$1 \;=\; \sum_{y_{s,\alpha} \in Var} y_{s,\alpha} \cdot \sum_{s' \in S_0} \mathbf{P}(s,\alpha,s')$$

iff $\mathrm{Pr}^{\mathcal{M}}_{\sigma_\Phi}(\Diamond S_0) = 1$ and $\forall\, y_{s,\alpha} \in Var\colon \Phi(y_{s,\alpha}) = \mathrm{E}^{\mathcal{M}}_{\sigma_\Phi}(\mathbf{R}_{s,\alpha} \Diamond S_0)$ with reward structure $\mathbf{R}_{s,\alpha}$ given by $\mathbf{R}_{s,\alpha}(\hat{s},\hat{\alpha},s') = [\hat{s} = s \text{ and } \hat{\alpha} = \alpha]$.

In [23], the lemma is applied to decide achievability of multiple total reward objectives under strategies that are stationary, but not necessarily pure. Intuitively, $\mathrm{E}^{\mathcal{M}}_{\sigma_\Phi}(\mathbf{R}_{s,\alpha} \Diamond S_0)$ coincides with the expected number of times action α is taken at state s until S_0 is reached. Since this value can be infinite if $\mathrm{Pr}^{\mathcal{M}}_{\sigma_\Phi}(\Diamond S_0) < 1$, a solution Φ can only exist if it induces a strategy that almost surely reaches S_0.

The encoding for unichain MDP with finite rewards and total reward objectives is shown in Fig. 3, where $S_0 = \bigcap_j S^j_0$ and $S_? = S \setminus S_0$. We consider the constraints in conjunction with Lines 1 and 2 from Fig. 2. Let Φ be a solution and let σ be the strategy encoded by such a solution, i.e., $\sigma(s)(\alpha) = \Phi(a_{s,\alpha})$.

Lines 10 to 12 reflect the equations of Lemma 4. Since $\mathcal{M}$ is unichain and we assume finite rewards, there is just one end component in which no reward can be collected. Hence, S_0 is almost surely reached. Line 10 ensures that the

$$\forall s \in S_?, \alpha \in Act(s): \quad y_{s,\alpha} \in [0, V_s \cdot a_{s,\alpha}] \tag{10}$$

$$\sum_{\alpha \in Act(s)} y_{s,\alpha} = [s = s_I] + \sum_{\langle s', \alpha' \rangle \in pre(s)} \mathbf{P}(s', \alpha', s) \cdot y_{s',\alpha'} \tag{11}$$

$$1 = \sum_{s \in S_?} \sum_{\alpha \in Act(s)} y_{s,\alpha} \cdot \sum_{s' \in S_0} \mathbf{P}(s, \alpha, s') \tag{12}$$

$$\forall j \in \{1, \dots, \ell\}: \quad x^j_{s_I} = \sum_{s \in S_?} \sum_{\alpha \in Act(s)} y_{s,\alpha} \cdot \sum_{s' \in S} \big(\mathbf{P}(s, \alpha, s') \cdot \mathbf{R}_j(s, \alpha, s')\big) \tag{13}$$

$$x^j_{s_I} \sim_j \boldsymbol{p}[\![j]\!] \tag{14}$$

Figure 3: `MILP` encoding for total reward objectives.

strategy in Lemma 4 coincides with the encoded pure strategy σ. We write V_s for an upper bound of the value a solution can possibly assign to $y_{s,\alpha}$, i.e., $\forall \sigma \in \Sigma^{\mathcal{M}}_{\mathrm{PS}}\colon V_s \geq \mathrm{E}^{\mathcal{M}}_{\sigma}(\mathbf{R}_{s,\alpha} \Diamond S_0)$. Such an upper bound can be computed based on ideas of [5]. More details are given in [17, App. A].

With Lemma 4 we get that $\Phi(y_{s,\sigma(s)})$ is the expected number of times state s is visited under strategy σ. Therefore, in Line 13 we sum up for each state $s \in S_?$ the expected amount of reward collected at s. This yields $\Phi(x^j_{s_I}) = \mathrm{E}^{\mathcal{M}}_{\sigma}(\mathbf{R}_j \Diamond G_j)$. Finally, Line 14 asserts that the resulting values exceed the thresholds given by $\boldsymbol{p}$.

Theorem 3. *For unichain $\mathcal{M}$, finite rewards, and total reward objectives, the constraints in Fig. 3 and Lines 1 and 2 of Fig. 2 are feasible iff $\boldsymbol{p} \in Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$.*

Proposition 2. *The* `MILP` *encoding above considers $\mathcal{O}(|S| \cdot |Act| + \ell)$ variables.*

The encoding for total reward objectives considers fewer variables compared to the encoding of Sect. 3.3 (cf. Proposition 1). In practice, this often leads to faster solving times as we will see in Sect. 6.

3.5 Extension to Multichain MDP

We now lift the restriction to unichain MDP, i.e., we consider multichain MDP with finite rewards. We focus on the encoding of Sect. 3.3. Details for the approach of Sect. 3.4 are in [17, App. C]. The key challenge is that the equation system in Lemma 3 does not yield a unique solution for multichain MDP.

Example 2. For the multichain MDP in Fig. 5a with $G = \{s_1\}$ we have $S_0 = \{s_1\}$ and $S_? = \{s_0\}$ (the superscript j is omitted as there is only one objective). For σ with $\sigma(s_0) = \alpha$ we get $\mathrm{E}^{\mathcal{M}}_{\sigma}(\mathbf{R} \Diamond G) = 0$, but every $\Phi\colon \{x_{s_0}, x_{s_1}\} \to \mathbb{R} \times \{0\}$ is a solution for the equation system in Lemma 3.

For multichain MDP it can be the case that for some strategy σ the set S^j_0 is not reached with probability 1, i.e., there is a positive probability to stay in the set $S^j_?$ forever. For the induced Markov chain $\mathcal{M}^\sigma$, this means that there is a reachable BSCC consisting only of states in $S^j_?$. Since BSCCs of $\mathcal{M}^\sigma$ coincide with end components of $\mathcal{M}$, we need to inspect the ECs of $\mathcal{M}$ that only consist of

$$
\begin{array}{llr}
\forall\, j \in \{1,\ldots,\ell\}, \mathcal{E} \in \mathit{MECS}(\mathcal{M}[\![\mathcal{E}^j_?]\!]): & & \triangleright\ \textit{Detect states with zero reward} \\
\quad \forall\, s \in S[\![\mathcal{E}]\!]: & \pm x^j_s \le U^j_s \cdot (1 - e^j_s) & (15) \\
\quad \forall\, \langle s,\alpha\rangle \in \mathcal{E}: & e^j_{s,\alpha} \in \{0, a_{s,\alpha}\} & (16) \\
\quad\quad \forall\, s' \in \mathit{succ}(s,\alpha): & e^j_{s,\alpha} \le e^j_{s'} & (17) \\
\quad \forall\, s \in S[\![\mathcal{E}]\!]: & e^j_s = \sum_{\alpha\in \mathit{Act}(s)} [\langle s,\alpha\rangle \in \mathcal{E}] \cdot e^j_{s,\alpha} & (18) \\
\quad\quad \forall\, \alpha \in \mathit{Act}(s): & z^j_{s,\alpha} \in [0, V_s \cdot a_{s,\alpha}] & (19) \\
 & z^j_{s,\perp} \in [0, V_s \cdot e^j_s] & (20) \\
\quad\quad z^j_{s,\perp} + \sum_{\alpha\in \mathit{Act}(s)} z^j_{s,\alpha} = & \frac{1}{|S[\![\mathcal{E}]\!]|} + \sum_{\langle s',\alpha'\rangle \in \mathit{pre}(s)\cap\mathcal{E}} \mathbf{P}(s',\alpha',s)\cdot z^j_{s',\alpha'} & (21) \\
 & 1 = \sum_{s\in S[\![\mathcal{E}]\!]} \Big(z^{\psi_j}_{s,\perp} + \sum_{\alpha\in \mathit{Act}(s)} [\langle s,\alpha\rangle \notin \mathcal{E}] \cdot z^j_{s,\alpha} \Big) & (22)
\end{array}
$$

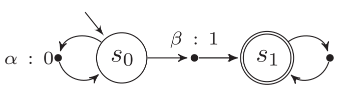
Figure 4: `MILP` encoding for detection of end components.

$S^j_?$-states. These ECs correspond to the ECs of the sub-MDP $\mathcal{M}[\![\mathcal{E}^j_?]\!]$, where $\mathcal{E}^j_?$ is the largest subset of $S^j_? \times \mathit{Act}$ that is closed for $\mathcal{M}$. For each $\mathcal{E} \in \mathit{MECS}(\mathcal{M}[\![\mathcal{E}^j_?]\!])$, we need to detect whether the encoded strategy induces a BSCC $\mathcal{E}' \subseteq \mathcal{E}$.

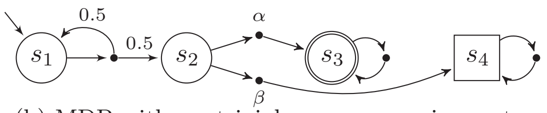
To cope with multiple ECs, we consider the constraints from Fig. 2 in conjunction with the constraints from Fig. 4. Let $\varPhi$ be a solution to these constraints and let σ be the encoded strategy σ with $\sigma(s)(\alpha) = \varPhi(a_{s,\alpha})$. For each objective ψ_j and state s, a binary variable e^j_s is set to 1 if s lies on a BSCC of the induced MC $\mathcal{M}^\sigma$. We only need to consider states $s \in S[\![\mathcal{E}]\!]$ for $\mathcal{E} \in \mathit{MECS}(\mathcal{M}[\![\mathcal{E}^j_?]\!])$.

Line 15 ensures that the value of x^j_s is set to 0 if s lies on a BSCC of $\mathcal{M}^\sigma$. Lines 16 to 18 introduce binary variables $e^j_{s,\alpha}$ for each state-action pair in the EC such that any solution $\varPhi$ satisfies $\varPhi(e^j_{s,\alpha}) = 1$ iff $\varPhi(e^j_s) = \varPhi(a_{s,\alpha}) = 1$. Line 17 yields that $\varPhi(e^j_{s,\alpha}) = 1$ implies $\varPhi(e^j_{s'}) = 1$ for all successors s' of s and the selected action α. Hence, for all s with $\varPhi(e^j_s) = 1$ and for all s' reachable from s in $\mathcal{M}^\sigma$, we have $\varPhi(e^j_{s'}) = 1$ and $\langle s', \sigma(s')\rangle \in \mathcal{E}$. Therefore, we can only set e^j_s to 1 if there is a BSCC $\mathcal{E}' \subseteq \mathcal{E}$ that either contains s or that is almost surely reached from s without leaving $\mathcal{E}$. As finite rewards are assumed, $\mathcal{E}$ can not contain a transition with positive reward, yielding $\mathrm{E}^{\mathcal{M}}_\sigma(\mathbf{R}_j \lozenge G_j) = 0$ if $\varPhi(e^j_s) = 1$.

An assignment that sets all variables e^j_s and $e^j_{s,\alpha}$ to 0 trivially satisfies the constraints in Lines 15 to 18. In Lines 19 to 22 we therefore ensure that if a BSCC $\mathcal{E}' \subseteq \mathcal{E}$ exists in $\mathcal{M}^\sigma$, $\varPhi(e^j_s) = 1$ holds for at least one $s \in S[\![\mathcal{E}']\!]$. The idea is based on the observation that if a BSCC $\mathcal{E}' \subseteq \mathcal{E}$ exists, there is a state $s \in S[\![\mathcal{E}]\!]$ that does not reach the set $S \setminus S[\![\mathcal{E}]\!]$ almost surely. We consider the MDP $\mathcal{M}^{\mathcal{E}}$, a mild extension of $\mathcal{M}[\![\mathcal{E}]\!]$ given by $\mathcal{M}^{\mathcal{E}} = (S[\![\mathcal{E}]\!] \uplus \{s^{\mathcal{E}}_I, s^{\mathcal{E}}_\perp\}, \mathit{Act} \uplus \{\alpha_I, \perp\}, \mathbf{P}^{\mathcal{E}}, s^{\mathcal{E}}_I)$, where $\mathbf{P}^{\mathcal{E}}$ extends $\mathbf{P}[\![\mathcal{E}]\!]$ such that $\mathbf{P}^{\mathcal{E}}(s^{\mathcal{E}}_\perp, \perp, s^{\mathcal{E}}_\perp) = 1$ and $\forall\, s \in S[\![\mathcal{E}]\!]$:
– $\mathbf{P}^{\mathcal{E}}(s^{\mathcal{E}}_I, \alpha_I, s) = 1/|S[\![\mathcal{E}]\!]|$, $\mathbf{P}^{\mathcal{E}}(s, \perp, s^{\mathcal{E}}_\perp) = 1$, and
– $\forall\, \alpha \in \{\hat\alpha \in \mathit{Act}(s) \mid \langle s, \hat\alpha\rangle \notin \mathcal{E}\} :\ \mathbf{P}^{\mathcal{E}}(s, \alpha, s^{\mathcal{E}}_\perp) = 1$.
Lines 21 and 22 reflect the equation system from Lemma 4 for MDP $\mathcal{M}^{\mathcal{E}}$ and $S_0 = \{s_\perp\}$. Additionally, Lines 19 and 20 exclude negative solutions and assert



(a) Multichain MDP



(b) MDP with nontrivial memory requirements

Figure 5: MDPs referenced in Examples 2 and 4.

$\Phi(z^j_{s,\alpha}) = 0$ if $\Phi(a_{s,\alpha}) = 0$ and $\Phi(z^{\psi_j}_{s,\perp}) = 0$ if $\Phi(e^j_s) = 0$ for any solution Φ. Hence, for states $s \in S[\![\mathcal{E}]\!]$ where $\Phi(e^j_s) = 0$, the strategy σ encoded by the variables $a_{s,\alpha}$ coincides with the strategy considered in Lemma 4. Assume that solution Φ yields a BSCC within the states of $\mathcal{E}$ in $\mathcal{M}^\sigma$ and therefore also a BSCC in $(\mathcal{M}^\mathcal{E})^\sigma$. Since $s^\mathcal{E}_\perp$ has to be reached almost surely in $\mathcal{M}^\mathcal{E}$ (cf. Lemma 4), the BSCC has to contain at least one state s with $\Phi(e^j_s) = 1$.

In summary, Lines 19 to 22 imply that every BSCC $\mathcal{E}' \subseteq \mathcal{E}$ of $\mathcal{M}^\sigma$ contains at least one state s with $\Phi(e^j_s) = 1$. Then, with Lines 16 to 18 we get that $\Phi(e^j_{s'}) = 1$ has to hold for *all* $s' \in S[\![\mathcal{E}']\!]$. In $\mathcal{M}^\sigma$, the set $S^j_0 \cup \{s \mid \Phi(e^j_s) = 1\}$ is therefore reached almost surely and all the states in this set get assigned value 0. In this case, the solution of the equation system from Lemma 3 becomes unique again.

Theorem 4. *For finite rewards, the constraints in Figs. 2 and 4 are feasible iff* $\boldsymbol{p} \in Ach^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$.

3.6 Extension to Infinite Rewards

Our approach can be modified to allow `PSMA` instances where infinite expected reward can be collected, i.e., where Restriction 2 does not hold. Infinite reward can be collected if we cycle through an EC of $\mathcal{M}$ that contains a transition with positive reward. Such instances are of practical interest as this often corresponds to strategies that do not accomplish a certain goal (e.g., a robot that stands still and therefore requires infinite time to finish its task).

We sketch the necessary modifications. More details are in [17, App. B]. Let S_∞ be the set of states where every pure strategy induces infinite reward for at least one minimizing objective. To ensure that the `MILP` instance has a (real-valued) solution, we consider the sub-MDP of $\mathcal{M}$ obtained by removing S_∞.

If infinite reward can be collected in an EC, it should not be considered in Fig. 4. We therefore let $\mathcal{E}$ range over maximal ECs that only consist of (a) states in $S^j_?$ and (b) transitions with reward 0.

The upper bounds U^j_s for the maximal expected rewards at each state can not be set to ∞. However, for the encoding it suffices to compute values that are sufficiently large. However, we remark that in practice our approach from [17, App. B] can lead to very large values, yielding numerical instabilities.

For maximizing objectives, we introduce one additional objective which, in a nutshell, checks that the probability to reach a 0-reward BSCC is below 1. If this is the case, there is a positive probability to reach a BSCC in which infinitely many reward can be collected.

4 Computing the Pareto Front

Our next goal is to compute the *pure stationary Pareto front* $\mathit{Pareto}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$ for MDP $\mathcal{M}$ and multi-objective query $\mathcal{Q}$. This set can be very large, in particular if the objectives are strongly conflicting with many different tradeoffs. In the worst case, every pure stationary strategy induces a point $\boldsymbol{p} \in \mathit{Pareto}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$ (e.g., for $\mathcal{Q} = \langle \mathbb{E}_{\leq}(\mathbf{R} \lozenge G), \mathbb{E}_{\geq}(\mathbf{R} \lozenge G) \rangle$). We try to find an approximation of $\mathit{Pareto}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$.

Definition 7. *Let* $\boldsymbol{\epsilon} \in (\mathbb{R}_{>0})^{\ell}$. *An* $\boldsymbol{\epsilon}$-approximation *of* $P \subseteq (\mathbb{R}_{\infty})^{\ell}$ *is a pair* $\langle L, U \rangle$ *with* $L \subseteq P \subseteq U$ *and* $\forall \boldsymbol{p} \in P \colon \exists \boldsymbol{p}' \in L \cup ((\mathbb{R}_{\infty})^{\ell} \setminus U) \colon |\boldsymbol{p} - \boldsymbol{p}'| \leq \boldsymbol{\epsilon}$.

Pure Stationary Pareto Approximation Problem ($\mathtt{PSP}^{\approx}$)

Input: MDP $\mathcal{M}$, ℓ-dimensional multi-objective query $\mathcal{Q}$, precision $\boldsymbol{\epsilon} \in (\mathbb{R}_{>0})^{\ell}$ such that $\mathit{Pareto}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q}) \subseteq \mathbb{R}^{\ell}$

Output: An $\boldsymbol{\epsilon}$-approximation of $cl^{\mathcal{Q}}(\mathit{Pareto}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q}))$

For simplicity, we only consider inputs that satisfy restriction Restriction 2, i.e., for $\psi_j = \mathbb{E}_{\sim_j}(\mathbf{R}_j \lozenge G_j)$ there is $U^j \neq \infty$ such that $\forall \sigma \in \Sigma^{\mathcal{M}}_{\mathrm{PS}} \colon U^j \geq \mathrm{E}^{\mathcal{M}}_{\sigma}(\mathbf{R}_j \lozenge G_j)$. Ideas of Sect. 3.6 can be used for some other inputs. An all-embracing treatment of infinite rewards, in particular for maximizing ψ_j, is subject to future work.

Our approach for $\mathtt{PSP}^{\approx}$ successively divides the solution space into candidate regions. For each region $\mathcal{R}$ (initially, let $\mathcal{R} = [0, U^1] \times \cdots \times [0, U^{\ell}]$), we use the $\mathtt{MILP}$ encoding from Sect. 3 with an optimization function to find a point $\boldsymbol{p} \in \mathcal{R} \cap \mathit{Pareto}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$ (or find out that no such point exists). The region $\mathcal{R}$ is divided into (i) an achievable region $\mathcal{R}_{\mathrm{A}} \subseteq \mathit{Ach}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$, (ii) an unachievable region $\mathcal{R}_{\mathrm{U}} \subseteq \mathbb{R}^{\ell} \setminus \mathit{Ach}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q})$, (iii) further candidate regions $\mathcal{R}_1, \dots, \mathcal{R}_n$ that are analyzed subsequently, and (iv) the remaining area $\mathcal{R} \setminus (\mathcal{R}_{\mathrm{A}} \cup \mathcal{R}_{\mathrm{U}} \cup \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_n)$ which does not require further analysis as we are only interested in an $\boldsymbol{\epsilon}$-approximation. The procedure stops as soon as no more candidate regions are found.

Example 3. Fig. 6 sketches the approach for an MDP $\mathcal{M}$ and a query $\mathcal{Q}$ with two maximizing objectives. We maintain a set of achievable points (light green) and a set of unachievable points (red). Initially, our candidate region corresponds to $\mathcal{R}_1 = [0, U^1] \times [0, U^2]$ given by the white area in Fig. 6a. We consider the direction vector $\boldsymbol{w}_1$ which is orthogonal to the line connecting $\langle U^1, 0 \rangle$ and $\langle 0, U^2 \rangle$. To find some point $\boldsymbol{p} \in \mathit{Pareto}^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q}) \cap \mathcal{R}_1$, we solve the $\mathtt{MILP}$ resulting from the constraints as in Sect. 3, the constraint $\langle x^1_{s_I}, x^2_{s_I} \rangle \in \mathcal{R}_1$, and the optimization function $\boldsymbol{w}_1 \cdot \langle x^1_{s_I}, x^2_{s_I} \rangle$. Fig. 6b shows the obtained point $\boldsymbol{p}_1 \in \mathcal{R}_1$. Since $\boldsymbol{p}_1$ is achievable, we know that any point in $cl^{\mathcal{Q}}(\{\boldsymbol{p}_1\})$ has to be achievable as well. Moreover, the set $\{\boldsymbol{p} \in \mathcal{R}_1 \mid \boldsymbol{w}_1 \cdot \boldsymbol{p} > \boldsymbol{w}_1 \cdot \boldsymbol{p}_1\}$ indicated by the area above the diagonal line in Fig. 6b can not contain an achievable point. The gray areas do not have to be checked in order to obtain an $\boldsymbol{\epsilon}$-approximation. We continue with $\mathcal{R}_2$ indicated by the white area and the direction vector $\boldsymbol{w}_2$, orthogonal to the line connecting $\langle 0, U^2 \rangle$ and $\boldsymbol{p}_1$. As before, we solve an $\mathtt{MILP}$ now yielding the point $\boldsymbol{p}_2$ in Fig. 6c. We find achievable points $cl^{\mathcal{Q}}(\{\boldsymbol{p}_2\})$ but no further unachievable points. The next iteration considers candidate region $\mathcal{R}_3$ and direction vector

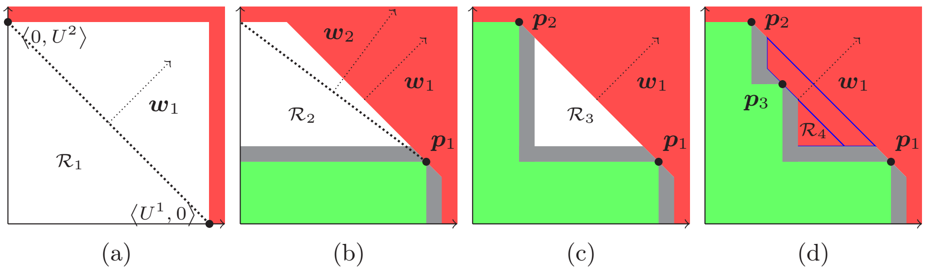


Figure 6: Example exploration of achievable points.

$\boldsymbol{w}_1$, yielding point $\boldsymbol{p}_3$ shown in Fig. 6d. The trapezoidal area is added to the unachievable points whereas $cl^{\mathcal{Q}}(\{\boldsymbol{p}_3\})$ is achievable. Finally, we check $\mathcal{R}_4$ for which the corresponding `MILP` instance is infeasible, i.e., $\mathcal{R}_4$ is unachievable.

The ideas sketched above can be lifted to $\ell > 2$ objectives. Inspired by [24, Alg. 4], we choose direction vectors that are orthogonal to the convex hull of the achievable points found so far. In fact, for total reward objectives we can apply the approach of [24] to compute the points in $Pareto^{\mathcal{M}}_{\mathrm{PS}}(\mathcal{Q}) \cap Pareto^{\mathcal{M}}(\mathcal{Q})$ first and only perform `MILP`-solving for the remaining regions. As the distance between two found points $\boldsymbol{p}, \boldsymbol{p}'$ is at least $|\boldsymbol{p} - \boldsymbol{p}'| \geq \boldsymbol{\epsilon}$, we can show that our approach terminates after finding at most $\prod_j U^j / \boldsymbol{\epsilon}[\![j]\!]$ points. Other strategies for choosing direction vectors are possible and can strongly impact performance.

5 Bounded Memory

For `GMA`, it is necessary and sufficient to consider strategies that require memory exponential in the number of objectives [20,24,40] by storing which goal state set has been reached already. In contrast, restricting to pure (but not necessarily stationary) strategies imposes nontrivial memory requirements that do not only depend on the number of objectives, but also on the point that is to be achieved.

Example 4. Let $\mathcal{M}$ be the MDP in Fig. 5b and $\mathcal{Q} = \langle \mathbb{P}_{\geq}(\lozenge G_{\circledcirc}), \mathbb{P}_{\geq}(\lozenge G_{\square})\rangle$. The point $\boldsymbol{p}_k = \langle 0.5^k, 1{-}0.5^k\rangle$ for $k \in \mathbb{N}$ is achievable by taking α with probability 0.5^k. $\boldsymbol{p}_k$ is also achievable with the *pure* strategy σ_k where $\sigma_k(\hat{\pi}) = \alpha$ iff $|\hat{\pi}| \geq k$. σ_k uses k memory states. Pure strategies with fewer memory states do not suffice.

We search for pure strategies with *bounded memory*. For an MDP $\mathcal{M}$ and $K > 0$, let $\Sigma^{\mathcal{M}}_{\mathrm{P},K}$ denote the set of pure K-memory strategies, i.e., any $\sigma \in \Sigma^{\mathcal{M}}_{\mathrm{P},K}$ can be represented by a *Mealy machine* using up to K states (c.f. [17, App. D]). For a query $\mathcal{Q}$, let $Ach^{\mathcal{M}}_{\mathrm{P},K}(\mathcal{Q})$ be the set of points achievable by some $\sigma \in \Sigma^{\mathcal{M}}_{\mathrm{P},K}$.

PURE BOUNDED MULTI-OBJECTIVE ACHIEVABILITY PROBLEM (`PBMA`)

Input: MDP $\mathcal{M}$, multi-objective query $\mathcal{Q}$, memory bound K, point $\boldsymbol{p} \in (\mathbb{R}_{\infty})^{\ell}$
Output: Yes iff $\boldsymbol{p} \in Ach^{\mathcal{M}}_{\mathrm{P},K}(\mathcal{Q})$

Table 1: Results for stationary strategies.

Bench-		Instance 1				ε=0.01		ε=0.001		Instance 2				ε=0.01		ε=0.001	
mark	ℓ	Par.	$\lvert S\rvert$	$\%\mathcal{E}$	$\overline{Act}$	Time	$\lvert P\rvert$	Time	$\lvert P\rvert$	Par.	$\lvert S\rvert$	$\%\mathcal{E}$	$\overline{Act}$	Time	$\lvert P\rvert$	Time	$\lvert P\rvert$
dpm	2^*	2	1272	32	3.2	17	37	315	377	3	1696	30	3.2	82	30	TO	
eajs	2^*	2-3	689	0	1.2	5	23	45	202	3-6	$2{\cdot}10^4$	0	1.2	201	52	3787	375
jobs	3^*	3-2	17	0	1.1	3	3	2	3	5-2	117	0	1.5	2042	76	TO	
mutex	3^*	1	1795	36	2.2	TO		TO		2	$1{\cdot}10^4$	33	2.3	TO		TO	
polling	2	2-2	233	86	1.5	6	5	23	6	3-2	990	84	1.8	299	5	TO	
rg	2^*	2-1-20	2173	14	2.9	5	5	12	5	5-2-50	$3{\cdot}10^4$	5	3.1	496	27	TO	
rover	2^*	2500	$2{\cdot}10^4$	0	1.2	110	47	417	251	5000	$4{\cdot}10^4$	0	1.2	258	47	3105	472
serv	2^*		$5{\cdot}10^4$	93	1.9	1828	38	TO									
str	2^*	30	1426	0	1.3	11	21	822	218	500	$4{\cdot}10^5$	0	1.3	2428	17	TO	
team2	2^*	2	1847	24	1.2	2	5	2	5	3	$1{\cdot}10^4$	21	1.2	18	43	MO	
team3	3^*	2	1847	24	1.2	165	15	166	15	3	$1{\cdot}10^4$	21	1.2	TO		TO	
uav	2^*	750	$2{\cdot}10^5$	29	1.6	400	39	5799	332	1000	$4{\cdot}10^5$	31	1.8	3546	36	TO	
wlan	2^*	0	2954	0	1.3	160	16	TO		2	$3{\cdot}10^4$	0	1.3	6728	23	TO	

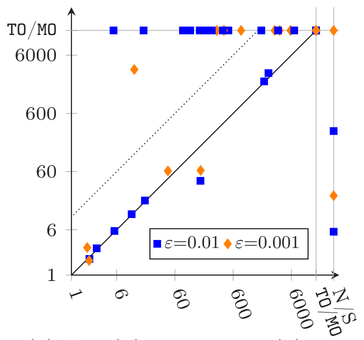
The pure bounded Pareto approximation problem is defined similarly. We reduce a PBMA instance to an instance for PSMA. The idea is to incorporate a memory structure of size K into $\mathcal{M}$ and then construct a pure stationary strategy in this product MDP (see, e.g., [29] for a similar construction). The set of strategies can be further refined by considering e.g., a memory structure that only allows counting or that only remembers visits of goal states. See [17, App. D] for details.

6 Evaluation

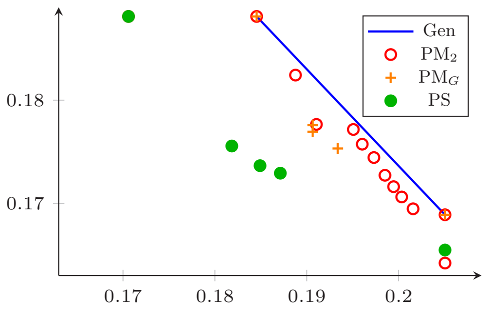
We implemented our approach for $\mathtt{PSP}^{\approx}$ in the model checker Storm [16] using Gurobi [27] as back end for MILP-solving. The implementation takes an MDP (e.g., in Prism syntax), a multi-objective query, and a precision $\varepsilon > 0$ as input and computes an $\boldsymbol{\epsilon}$-approximation of the Pareto front. Here, we set $\boldsymbol{\epsilon}[\![j]\!] = \varepsilon \cdot \delta_j$, where δ_j is the difference between the maximal and minimal achievable value for objective ψ_j. We also support reward objectives for Markov automata via [38]. The computations within Gurobi might suffer from numerical instabilities. To diminish their impact, we use the *exact* engine of Storm to confirm for each MILP solution that the encoded strategy achieves the encoded point. However, sub-optimal solutions returned by Gurobi may still yield inaccurate results.

We evaluate our approach on 13 multi-objective benchmarks from [24,28,38], each considering one or two parameter instantiations. Application areas range over scheduling (dpm [37], eajs [1], jobs [10], polling [43]), planning (rg [6], rover [28], serv [32], uav [21]), and protocols (mutex [38], str [38], team [15], wlan [31]).

The results for pure stationary strategies are summarized in Table 1. For each benchmark we denote the number of objectives ℓ and whether the alternative encoding from Sect. 3.4 has been applied (*). For each parameter instantiation (Par.), the number of states ($|S|$), the percentage of the states that are contained in an end component ($\%\mathcal{E}$), and the average number of available actions at each state ($\overline{Act}$) are given. For each precision $\varepsilon \in \{0.01, 0.001\}$, we then depict the



(a) Alt. (x) vs. original (y) encoding.



(b) Pareto fronts for restricted strategies.

Figure 7: Comparison of the two encodings (left) and impact of memory (right).

runtime of Storm and the number of points on the computed approximation of the Pareto front. `TO` denotes that the approach did not terminate within 2 hours, `MO` denotes insufficient memory (16 GB). All experiments used 8 cores of an Intel® Xeon® Platinum 8160 Processor.

Storm is often able to compute pure stationary Pareto fronts, even for models with over 100 000 states (e.g., uav). However, the model structure strongly affects the performance. For example, the second instance of jobs is challenging although it only considers 117 states, a low degree of nondeterminism, and no (non-trivial) end components. Small increments in the model size can increase runtimes significantly (e.g., dpm or uav). If a higher precision is requested, much more points need to be found, which often leads to timeouts. Similarly, for more than 2 objectives the desired accuracy can often not be achieved within the time limit. The approach can be stopped at any time to report on the current approximation, e.g., after 2 hours Storm found 65 points for Instance 1 of mutex.

For almost all benchmarks, the objectives could be transformed to total reward objectives, making the more efficient encoding form Sect. 3.4 applicable. We plot the runtimes of the two encoding in Fig. 7a. The alternative encoding is superior for almost every benchmark. In fact, the original encoding timed out for many models as indicated at the horizontal line at the top of the figure.

In Fig. 7b we plot the Pareto front for the first polling instance under general strategies (Gen), pure 2-memory strategies that can change the memory state exactly once (PM_2), pure strategies that observe which goal state set G_j has been visited already (PM_G), and pure stationary strategies (PS). Adding simple memory structures already leads to noticeable improvements in the quality of strategies. In particular, PM_2 strategies perform quite well, and even outperform PM_G strategies (which would be optimal if randomization were allowed).

Data availability. The artifact [18] accompanying this paper contains source code, benchmark files, and replication scripts for our experiments.

Acknowledgments. The authors thank Sebastian Junges for his valuable contributions during early stages of this work.

References

1. Baier, C., Daum, M., Dubslaff, C., Klein, J., Klüppelholz, S.: Energy-utility quantiles. In: NASA Formal Methods, NFM. pp. 285–299 (2014). https://doi.org/10.1007/978-3-319-06200-6_24
2. Baier, C., Dubslaff, C., Klüppelholz, S.: Trade-off analysis meets probabilistic model checking. In: CSL-LICS. pp. 1:1–1:10. ACM (2014)
3. Baier, C., Hermanns, H., Katoen, J.: The 10, 000 facets of MDP model checking. In: Computing and Software Science, LNCS, vol. 10000, pp. 420–451. Springer (2019)
4. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
5. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: CAV (1). LNCS, vol. 10426, pp. 160–180. Springer (2017)
6. Barrett, L., Narayanan, S.: Learning all optimal policies with multiple criteria. In: (ICML). pp. 41–47 (2008)
7. Benini, L., Bogliolo, A., Paleologo, G.A., De Micheli, G.: Policy optimization for dynamic power management. Trans. Comp.-Aided Des. Integ. Cir. Sys. **18**(6), 813–833 (2006). https://doi.org/10.1109/43.766730
8. Berthon, R., Randour, M., Raskin, J.: Threshold constraints with guarantees for parity objectives in Markov decision processes. In: ICALP. LIPIcs, vol. 80, pp. 121:1–121:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
9. Bouyer, P., González, M., Markey, N., Randour, M.: Multi-weighted Markov decision processes with reachability objectives. In: GandALF. EPTCS, vol. 277, pp. 250–264 (2018)
10. Bruno, J.L., Downey, P.J., Frederickson, G.N.: Sequencing tasks with exponential service times to minimize the expected flow time or makespan. J. ACM **28**(1), 100–113 (1981). https://doi.org/10.1145/322234.322242
11. Bruyère, V., Filiot, E., Randour, M., Raskin, J.: Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games. Inf. Comput. **254**, 259–295 (2017)
12. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: Trading memory for randomness. In: QEST. pp. 206–217. IEEE Computer Society (2004)
13. Chatterjee, K., Kretínská, Z., Kretínský, J.: Unifying two views on multiple mean-payoff objectives in markov decision processes. LMCS **13**(2) (2017)
14. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Markov decision processes with multiple objectives. In: STACS. LNCS, vol. 3884, pp. 325–336. Springer (2006)
15. Chen, T., Kwiatkowska, M.Z., Parker, D., Simaitis, A.: Verifying team formation protocols with probabilistic model checking. In: CLIMA. pp. 190–207 (2011)
16. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: CAV. LNCS, vol. 10427. Springer (2017)
17. Delgrange, F., Katoen, J.P., Quatmann, T., Randour, M.: Simple strategies in multi-objective MDPs (technical report). CoRR **abs//1910.11024** (2019), http://arxiv.org/abs/1910.11024
18. Delgrange, F., Katoen, J.P., Quatmann, T., Randour, M.: Evaluated artifact for this paper. figshare (2020). https://doi.org/10.6084/m9.figshare.11569485
19. von Essen, C., Giannakopoulou, D.: Probabilistic verification and synthesis of the next generation airborne collision avoidance system. STTT **18**(2), 227–243 (2016)
20. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. Logical Methods in Computer Science **4**(4) (2008). https://doi.org/10.2168/LMCS-4(4:8)2008

21. Feng, L., Wiltsche, C., Humphrey, L.R., Topcu, U.: Controller synthesis for autonomous systems interacting with human operators. In: ICCPS. pp. 70–79. ACM (2015)
22. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: SFM. LNCS, vol. 6659, pp. 53–113. Springer (2011)
23. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: TACAS. LNCS, vol. 6605, pp. 112–127. Springer (2011)
24. Forejt, V., Kwiatkowska, M.Z., Parker, D.: Pareto curves for probabilistic model checking. In: ATVA. LNCS, vol. 7561, pp. 317–332. Springer (2012)
25. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
26. Gleixner, A., Bastubbe, M., Eifler, L., Gally, T., Gamrath, G., Gottwald, R.L., Hendel, G., Hojny, C., Koch, T., Lübbecke, M.E., Maher, S.J., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schlösser, F., Schubert, C., Serrano, F., Shinano, Y., Viernickel, J.M., Walter, M., Wegscheider, F., Witt, J.T., Witzig, J.: The SCIP Optimization Suite 6.0. Technical report, Optimization Online (July 2018), http://www.optimization-online.org/DB_HTML/2018/07/6692.html
27. Gurobi Optimization, L.: Gurobi optimizer reference manual (2019), http://www.gurobi.com
28. Hartmanns, A., Junges, S., Katoen, J., Quatmann, T.: Multi-cost bounded reachability in MDP. In: TACAS (2). LNCS, vol. 10806, pp. 320–339. Springer (2018)
29. Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L., Katoen, J., Becker, B.: Finite-state controllers of POMDPs using parameter synthesis. In: UAI. pp. 519–529. AUAI Press (2018)
30. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)
31. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: QEST. pp. 203–204 (2012). https://doi.org/10.1109/QEST.2012.14
32. Lacerda, B., Parker, D., Hawes, N.: Multi-objective policy generation for mobile robots under probabilistic time-bounded guarantees. In: ICAPS. pp. 504–512. AAAI Press (2017)
33. Lizotte, D.J., Bowling, M., Murphy, S.A.: Linear fitted-Q iteration with multiple reward functions. J. Mach. Learn. Res. **13**, 3253–3295 (2012)
34. Perny, P., Weng, P.: On finding compromise solutions in multiobjective Markov decision processes. In: ECAI. FAIA, vol. 215, pp. 969–970. IOS Press (2010)
35. Pia, A.D., Dey, S.S., Molinaro, M.: Mixed-integer quadratic programming is in NP. Math. Program. **162**(1-2), 225–240 (2017)
36. Puterman, M.L.: Markov Decision Processes. John Wiley and Sons (1994)
37. Qiu, Q., Wu, Q., Pedram, M.: Stochastic modeling of a power-managed system: Construction and optimization. In: ISLPED. pp. 194–199. ACM (1999)
38. Quatmann, T., Junges, S., Katoen, J.: Markov automata with multiple objectives. In: CAV (1). LNCS, vol. 10426, pp. 140–159. Springer (2017)
39. Randour, M., Raskin, J., Sankur, O.: Variations on the stochastic shortest path problem. In: VMCAI. Lecture Notes in Computer Science, vol. 8931, pp. 1–18. Springer (2015)

40. Randour, M., Raskin, J., Sankur, O.: Percentile queries in multi-dimensional Markov decision processes. FMSD **50**(2-3), 207–248 (2017)
41. Roijers, D.M., Vamplew, P., Whiteson, S., Dazeley, R.: A survey of multi-objective sequential decision-making. JAIR **48**, 67–113 (2013)
42. Scheftelowitsch, D., Buchholz, P., Hashemi, V., Hermanns, H.: Multi-objective approaches to Markov decision processes with uncertain transition parameters. In: VALUETOOLS. pp. 44–51. ACM (2017)
43. Srinivasan, M.: Nondeterministic polling systems. Management Science **37**(6), 667–681 (1991). https://doi.org/10.1287/mnsc.37.6.667
44. Wiering, M.A., de Jong, E.D.: Computing optimal stationary policies for multi-objective Markov decision processes. In: ADPRL. pp. 158–165 (2007). https://doi.org/10.1109/ADPRL.2007.368183

Model Checking and Reachability

Partial Order Reduction for Deep Bug Finding in Synchronous Hardware [⋆]

Makai Mann and Clark Barrett

Stanford University, Stanford, CA 94305 USA

Abstract. Symbolic model checking has become an important part of the verification flow in industrial hardware design. However, its use is still limited due to scaling issues. One way to address this is to exploit the large amounts of symmetry present in many real world designs. In this paper, we adapt partial order reduction for bounded model checking of synchronous hardware and introduce a novel technique that makes partial order reduction practical in this new domain. These approaches are largely automatic, requiring only minimal manual effort. We evaluate our technique on open-source and commercial packet mover circuits – designs containing FIFOs and arbiters.

1 Introduction

Modern society relies increasingly on electronic systems, powered by hardware components that continue to grow in complexity and variety. Ensuring the functional correctness of these components is essential, as bugs and errors can have consequences ranging from undermining a company's reputation to jeopardizing human safety [1,22,25,32,33]. Most electronic designs must therefore include a significant verification effort, and this effort often consumes more time and resources than all other aspects of the design process [17,34].

Formal methods such as symbolic model checking have become a crucial part of the verification effort because of their strong guarantees and automation [24]. However, due to the *state space explosion problem* [14], model checking typically only works well for small- to medium-sized circuits with primarily control logic, limiting its potential for addressing industry verification challenges.

One approach for combating the state space explosion problem is *partial order reduction* [14]. While symbolic partial order reduction has been successfully applied for the verification of asynchronous systems [37], its use in synchronous systems has been limited. In this paper, we introduce a novel approach for adapting symbolic partial order reduction to model checking of synchronous hardware

[⋆] This work was supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported by the Defense Advanced Research Projects Agency, grant FA8650-18-2-7854. We thank the funding agencies and our corporate collaborators for their support.

and demonstrate dramatic reductions in the time to reach deep bugs on certain classes of synchronous circuits. Moreover, the technique requires only an interface-level annotation of the circuit, and when fully automated approaches fail, can be guided by the user. The paper makes the following contributions:

1. We adapt partial order reduction for synchronous hardware verification.
2. We introduce a novel technique for reducing the possible inputs to a circuit at a single time step, which is crucial for practical application of partial order reduction to synchronous hardware.
3. We provide a set of sufficient conditions, which, if proven, guarantee that the proposed techniques maintain the reachable states.
4. We introduce conservative proof techniques for verifying these conditions, which empirically work well on packet movers.
5. We evaluate our techniques on a set of open-source and commercial packet mover circuits, demonstrating dramatic speed-ups with minimal manual effort.

The rest of the paper is organized as follows. We first provide a motivating example, below. Then, in Section 2, we cover relevant background material and notation. We explain our partial order reduction in Section 3 and our interface simplification technique in Section 4. We provide an experimental evaluation in Section 5. Section 6 covers related work, and Section 7 concludes.

1.1 Motivating Example

Throughout this paper we use the running example shown in Code Snippet 1. We chose this example because: i) it is easy to understand; ii) it resembles real-world packet mover circuits; and iii) it contains a difficult to reach bug.

The system has a synchronizing clock and takes two 1-bit inputs: `inc_x` and `inc_y`. The 6-bit registers (state elements) `x` and `y` index the `valid` vector and are initialized to 0. The 64-bit registers `valid` and `data` start at 0 and 1, respectively. The 64x64 bit memory is uninitialized. If `inc_x` and `en_x` are true, the system increments the value of `x`. When `inc_y` is true, the system increments `y`, sets the valid bit at index `y`, writes `data` to the memory at location `y`, and rotates the `data` vector to the left. Notice that the `en_x` signal ensures that `x` never surpasses `y` (until all bits in `valid` are set). This incrementing pointer logic is similar to that found in a circular pointer FIFO. To ensure the asserted property, the code attempts to maintain the invariant: `data = 1 << y`.

At first, it appears that the asserted property should hold based on this invariant, but it does not. There is a bug that can first occur at cycle 65: the overflow check in the `data` update uses integers, which are assumed to be 32-bits. Since `y` is zero-extended to be 32-bits, `y+1` can never be equal to 0. Thus, when `y` has the value 63 and is incremented, `data`, which is supposed to be one-hot, is set to 0.

Although the system is small, this is a surprisingly difficult bug to reach using model checking. We believe this is due in part to the non-determinism in the

Code Snippet 1: Buggy Toy Example

```
module deep_bug(input clk, input inc_x, input inc_y);
   reg [5:0]        x = 0;
   reg [5:0]        y = 0;
   reg [63:0]       valid = 0;
   reg [63:0]       data = 1;
   reg [63:0]       mem [63:0];
   wire             en_x;
   assign en_x      = valid[x];

   always @(posedge clk) begin
      if (inc_x & en_x)
         x <= x + 1;
      if (inc_y) begin
         y <= y + 1; valid[y] <= 1'b1; mem[y] <= data;
         data <= (y+1 == 0) ? 1 : (data << 1);
      end
   end

   always @*
      assert ((mem[x] == (1 << x)) || ~valid[x]);
endmodule // deep_bug
```

update logic. In every state, there are 4 possible input combinations. As a result, there are an exponential number of execution paths. Model checkers routinely verify hardware designs with an exponential number of reachable states; however, we have observed that systems such as this which also have an exponential number of execution paths are difficult for a model checker to manage.

Specifically, all but two of the model checker configurations we tried timed out at 2 hours before reaching the bug. Since bounded model checking (BMC) is one of the best approaches for bug-finding, we focus on improvements to BMC that help reach this bug. We introduce automated, best effort techniques that reduce the time to hit this bug from over 1000 seconds to 46 seconds by safely adding temporal symmetry breaking constraints to the system.

2 Background

Before explaining our algorithm, we adapt the standard notion of synchronous transition systems and review fundamental model checking concepts below. For a more thorough introduction to model checking, we refer the reader to [14,15].

Definition 1. *A Synchronous Transition System (STS) is a tuple, $\langle S, Init, A, En, D, T \rangle$:*

- *S: a set of states*
- *$Init \subseteq S$: a set of initial states*

- *A: a finite set of atomic actions - logically distinct operations of the system*
- $En = \{en_a | a \in A\}$*: where* $en_a : S \to \mathbb{B}$ *is a state predicate that holds* iff *action a is* enabled *in a given state*
- *D: a set of data inputs to the system*
- $T \subseteq S \times (\mathcal{P}(A) \times D) \times S$*: the state transition relation, where* $\mathcal{P}$ *denotes power set*

For our purposes, an STS instruction can perform multiple atomic actions simultaneously. We define the system's *instruction set* (i.e. the set of actions that the system can perform in one transition) as $\mathcal{I} := \mathcal{P}(A)$. We then define the set of *inputs* of an STS as $\mathit{Input} := \mathcal{I} \times D$. Thus, the transition relation T is a subset of $S \times \mathit{Input} \times S$.

We denote the cardinality of an instruction i as $|i|$. For $s, s' \in S, in \in \mathit{Input}$, $T(s, in, s')$ holds *iff* it is possible to reach s' from s by applying input in. It is often convenient to reason about sequences using vector notation. Let $\boldsymbol{in} \in \mathit{Input}^n$ and $\boldsymbol{s} \in S^{n+1}$, with $n > 0$. We use subscripts to name individual elements of vectors, e.g. $\boldsymbol{s} := \langle s_0, s_1, \dots \rangle$. We use the notation $T(\boldsymbol{in}, \boldsymbol{s})$ to denote $\bigwedge_{0 \le i < n} T(s_i, in_i, s_{i+1})$. The length of a vector is given by $| \cdot |$, e.g. $|\boldsymbol{s}| = n + 1$, and prepending is represented as $\cdot : \cdot$, e.g. $\boldsymbol{s} = s_0 : \boldsymbol{s'}$ for some $s' \in S^n$. With some abuse of notation, we allow prepending both sequences and single elements. For $k > 0$, we say that $\boldsymbol{s} \in S^k$ is reachable if $\exists n \in \mathbb{N}, \boldsymbol{s'} \in S^{n+1}, \boldsymbol{in} \in \mathit{Input}^{n+k} \ . \ \mathit{Init}(s'_0) \wedge T(\boldsymbol{in}, \boldsymbol{s'} : \boldsymbol{s})$.

The set of enabledness predicates En constrain the valid states in which an action can occur. For an instruction $i \in \mathcal{I}$ and $s \in S$, let $en_i(s) := \bigwedge_{a \in i} en_a(s)$. In the remainder of the paper, we only consider transition relations T that respect the enabledness conditions. That is, we assume $\forall\, s, i.(en_i(s) \leftrightarrow \exists\, s', d.T(s, \langle i, d \rangle, s'))$. Depending on the context, this can be checked with a model checker or added as an environmental assumption. We also assume that the existence of a transition does not depend on the data input, that is, $\forall\, s, i.\, (\exists\, d, s'.\, T(s, \langle i, d \rangle, s') \implies \forall\, d.\, \exists\, s'.\, T(s, \langle i, d \rangle, s'))$.

Example 1. We can define an STS for the motivating example. Let BV_k denote the set of all bitvectors of width k. Because there is only a single clock with no negative edge behavior, we model the system without the clock, where every transition corresponds to a clock cycle. Define an STS $\langle S, \mathit{Init}, A, En, D, T \rangle$, where:

- $S = BV_6 \times BV_6 \times BV_{64} \times BV_{64} \times (BV_{64})^{64}$ is the set of values for $\langle \mathrm{x}, \mathrm{y}, \mathrm{valid}, \mathrm{data}, \mathrm{mem} \rangle$
- *Init* is the set containing all states where $\mathrm{x} = 0$, $\mathrm{y} = 0$, $\mathrm{valid} = 0$ and $\mathrm{data} = 1$
- $A = \{\mathrm{inc\_x}, \mathrm{inc\_y}\}$
- $En = \{en_{\mathrm{inc\_x}} := \mathrm{valid}[x] = 1, en_{\mathrm{inc\_y}} := \mathit{true}\}$
- $D = \{\mathit{nil}\}$ (here, *nil* is just a dummy placeholder used to ensure that T is not empty).
- T is the relation describing the next state updates in Code Snippet 1.

Model Checking. Given an STS $\mathbf{S}$, let a safety property $P \subseteq S$ be a set containing acceptable states. The *model checking problem* is to determine whether the system stays within this acceptable set for all possible execution traces. Formally, we want to check whether the following holds:

$$\forall\, n \geq 0, \boldsymbol{in} \in Input^n, \boldsymbol{s} \in S^{n+1}.\, (Init(s_0) \wedge T(\boldsymbol{in}, \boldsymbol{s})) \implies s_n \in P \qquad (1)$$

When equation (1) holds, we say that P is an invariant of $\mathbf{S}$. A number of techniques exist for solving this problem, including Binary Decision Diagram (BDD)-based [12] approaches, Interpolant-based [27] approaches, and IC3/PDR (property directed reachability) techniques [10,16]. We refer the interested reader to [15] for a more complete survey of model checking algorithms.

In this paper, we will focus on *bounded model checking* (BMC). In BMC, instead of proving (1) for all n, we prove it for all n less than some finite bound k. Though it typically cannot be used to prove properties, BMC can be quite effective at finding bugs [6] and is especially useful when full model checking is infeasible.

Symmetry. Early on in the development of model checking, researchers recognized the importance of symmetry reduction to combat the state explosion problem [13]. Existing approaches in the hardware domain perform data symmetry reduction and data type reduction through the use of bit-width reduction preprocessing passes or syntactic restrictions such as *scalarsets* [8,20,28]. There have also been abstraction-refinement loop algorithms proposed to handle memory symmetries [9]. All of these approaches are focused on symmetries present in the transition system description, such as the presence of large data types. We refer to these types of symmetries as *data* symmetries. Most of these techniques are intended to speed up proofs of true properties rather than accelerate bug-finding.

Model checking of asynchronous systems such as concurrent programs faces an orthogonal issue due to the many possible redundant interleavings of independent processes. Throughout this paper, we refer to this as *path* symmetry. Path symmetry is a temporal symmetry: it relates to executions of a system rather than just its size. Path symmetries occur when there are many distinct ways of reaching the same state in a system execution. Exploring all such paths can result in exponential case splitting.

This paper provides evidence that path symmetry can also severely hurt model checking performance in synchronous systems. One of the first techniques proposed to handle path symmetry was partial order reduction.

Partial Order Reduction. Partial order reduction was first developed in the explicit-state model checking context but was later extended to symbolic model checking [37]. The approach is named "partial order reduction" for historical reasons, but Clarke noted in [14] that "model checking using representatives" [30,31] may have been a more appropriate name. In particular, partial order reduction attempts to develop equivalence classes of behaviors so that only one representative from each class needs to be considered during model checking. Note that partial order reductions are sound only for checking state invariants. If the property of interest is temporal, the reduction could disallow input sequences that trigger the property. This can be avoided by first instantiating a monitor [15] and, if necessary, converting liveness properties to safety [5].

Partial order reduction is less natural in the synchronous setting, because synchronous transition systems do not have easily expressible independent actions. Nevertheless, these systems can still benefit from partial order reduction. Consider our motivating example: despite the huge number of system execution paths to consider, many of them are redundant. Observe that if both inputs are zero, then the state does not change. Furthermore, there is a temporal symmetry in the system execution: from any state where `en_x` is true, driving only `inc_x` followed by only `inc_y` results in the same state as driving them in the opposite order. Thus, this system has a large number of redundant interleavings, much like a multi-threaded program. To address this problem, we introduce a partial order reduction for synchronous hardware. Our goal is to remove redundant interleavings by adding constraints to the system. To maintain soundness, we provide a set of conditions which must pass before we can add constraints.

3 Synchronous Partial Order Reduction

In order to be able to apply partial order reduction to a synchronous transition system, we are interested in identifying pairs of instructions that can be reordered without affecting the resulting state. More generally, we also want to be able to find pairs that can only be reordered under certain conditions. To formalize these notions, we adapt the notation and representation of *guarded independence relations* from [37].[1]

Definition 2. *Given an STS:* $\langle S, Init, A, En, D, T\rangle$ *with instruction set* $\mathcal{I}$*, let* $G := \mathcal{P}(S)$ *be the set of predicates over the states. Let* $\langle i_0, i_1, g\rangle$ *be a* guarded independence tuple *iff for all* $d_0, d_1 \in D$ *and reachable* $\boldsymbol{s} \in S^3$*, the following condition holds:*

$$en_{i_0}(s_0) \wedge g(s_0) \wedge T(\langle\langle i_1, d_1\rangle, \langle i_0, d_0\rangle\rangle, \boldsymbol{s}) \implies \exists\, s'.T(\langle\langle i_0, d_0\rangle, \langle i_1, d_1\rangle\rangle, \langle s_0, s', s_2\rangle).$$

According to this definition, if we can prove that $\langle i_0, i_1, g\rangle$ is a guarded independence tuple, then we can reorder $\langle i_1, i_0\rangle$ instruction sequences as long as i) i_0 is enabled in the first state; ii) g holds in the first state; and iii) we also reorder the corresponding data inputs. We check only the enabledness of i_0 because $\langle i_0, i_1\rangle$ is the representative order, and we only need to be able to reorder to the representative, not from it. The guard allows us to consider partial order reductions that only hold for a subset of the reachable states. To avoid trivially overconstraining the system with conflicting reorderings, we will only consider one ordering for each pair of instructions.

The condition in Definition 2 is difficult to check automatically because of the existential quantifier. We instead check two slightly weaker conditions that

[1] The main differences are our STS formalism and that we consider reachability.

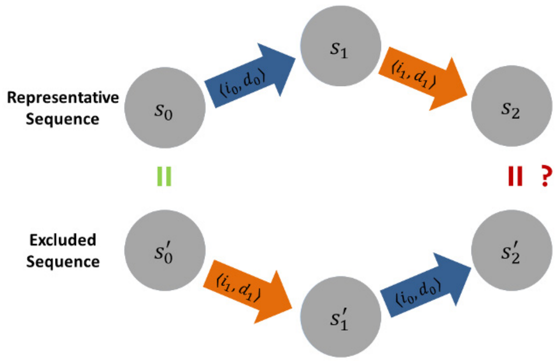

Representative Sequence
s_0
$\langle i_0, d_0\rangle$
s_1
$\langle i_1, d_1\rangle$
s_2
=
= ?
Excluded Sequence
s'_0
$\langle i_1, d_1\rangle$
s'_1
$\langle i_0, d_0\rangle$
s'_2

Fig. 1: Partial Order Reduction Condition (3) Proof Goal

imply guarded independence. These conditions are also standard in the POR literature [14,37]. The first condition states that instruction i_0 cannot disable i_1 under guard g:

$$\forall\, d \in D, \boldsymbol{s} \in S^2.\, (en_{i_1}(s_0) \wedge g(s_0) \wedge T(s_0, \langle i_0, d\rangle, s_1)) \implies en_{i_1}(s_1) \tag{2}$$

Intuitively, this condition ensures that we do not remove reachable states by disabling instructions. The second condition is that executing the instructions in either order leads to the same final state:

$$\begin{aligned}&\forall d_0, d_1 \in D, \boldsymbol{s}, \boldsymbol{s}' \in S^3 \ .\ (g(s_0) \wedge (s_0 = s'_0) \wedge \\ &\quad T(\langle\langle i_0, d_0\rangle, \langle i_1, d_1\rangle\rangle, \boldsymbol{s}) \wedge T(\langle\langle i_1, d_1\rangle, \langle i_0, d_0\rangle\rangle, \boldsymbol{s}')) \implies (s_2 = s'_2)\end{aligned} \tag{3}$$

When applying partial order reduction to concurrent programs, the standard approach is to check conservative syntactic properties which guarantee conditions (2) and (3). Synchronous systems do not typically have these syntactic properties, because there is no notion of distinct processes. Instead, we must check these conditions directly. In real circuits, it is unlikely that (2) will hold over arbitrary states. However, it is sufficient to prove that it holds for all reachable states. This can be done with a model checker.

To prove (3), we could encode it as an LTL property or build a monitor automaton and use a model checker. Alternatively, we have found that we can often use a straightforward commuting-diagram approach starting from a symbolic initial state, depicted in Fig. 1. We duplicate the system, unroll it twice, then start both copies in the same symbolic state and check that applying the instructions in either order results in the same final state. This simple approach has the disadvantage that a symbolic initial state ignores reachability which could lead to spurious counterexamples. However, notice that the initial state is constrained by enabledness assumptions. To apply an instruction it must be enabled, so both instructions must be enabled in the initial state. We have found that these enabledness assumptions often constrain the initial state enough to rule out spurious counterexamples.

If both conditions pass, then we can choose a representative order and disallow the opposite ordering for that pair of instructions. If the proof of condition (3) fails, it provides a counterexample which should either convince the user that partial order reduction does not apply for that pair of instructions (a real counterexample), or serve as a guide for the user to write guards that would remove the spurious counterexample. Other invariants of the system, either obtained automatically or manually guessed by the user, could also remove spurious counterexamples. We can now state the first theorem of synchronous partial order reduction: that these conditions guarantee guarded independence over all reachable states.

Theorem 1. *Given an STS* $\mathbf{S} := \langle S, \mathit{Init}, A, \mathit{En}, D, T\rangle$*, with instruction set* $I := \mathcal{P}(A)$*: if conditions* (2) *and* (3) *hold for instructions* $i_o, i_1 \in I$*, and guard* $g \in \mathcal{P}(S)$*, then* $\langle i_0, i_1, g\rangle$ *is a guarded independence tuple.*

Proof. Assume conditions (2) and (3) and that for some $d_0, d_1 \in D$ and reachable $\boldsymbol{s} \in S^3$, we have:

$$en_{i_0}(s_0) \wedge g(s_0) \wedge T(\langle\langle i_1, d_1\rangle, \langle i_0, d_0\rangle\rangle, \boldsymbol{s})$$

Because $en_{i_0}(s_0)$, we have $\exists s', d' \ . \ T(s_0, \langle i_0, d'\rangle, s')$ because of our enabledness assumption. Furthermore, by the data-input independence property of transition relations, it follows that for some s_1', $T(s_0, \langle i_0, d_0\rangle, s_1')$ Now, because one of our assumptions is a transition from s_0 using i_1, $en_{i_1}(s_0)$ must be true. Condition (2) implies that $en_{i_1}(s_1')$, thus $\exists\, s', d' . \, T(\langle\langle i_0, d_0\rangle, \langle i_1, d'\rangle\rangle, \langle s_0, s_1', s'\rangle)$. As before, this implies that for some s_2', we also have that $T(\langle\langle i_0, d_0\rangle, \langle i_1, d_1\rangle\rangle, \langle s_0, s_1', s_2'\rangle)$. It then follows from (3) that $s_2' = s_2$, and thus, $\langle i_0, i_1, g\rangle$ satisfies the condition from Definition 2. $\square$

Let a guarded independence relation, $R \subseteq \mathcal{I} \times \mathcal{I} \times G$, be a set of guarded independence tuples. We now describe how to apply partial order reductions, given some R. For each $\langle i_0, i_1, g\rangle \in R$, and for every $\boldsymbol{s} \in S^2, d \in D$, whenever $T(s_0, \langle i_1, d_1\rangle, s_1) \wedge en_{i_0}(s_0) \wedge g(s_0)$ holds, we remove from T every transition of the form $\langle s_1, \langle i_0, d\rangle, s\rangle$ (for any d and s). Let T^R be the result. To apply this reduction in practice, we add a constraint to the BMC encoding: $(g(s_0) \wedge en_{i_0}(s_0) \wedge i_1) \implies \neg \mathit{next}(i_0)$.

This makes it impossible for the STS system to ever execute an instruction i_0 after an instruction i_1 when starting from a state where i_0 is enabled and g holds. This effectively gives preference to i_0 as long as it is enabled. The effect of partial order reduction on a pair of instructions in a synchronous system is depicted in Fig. 2. Red X's show removed transitions, and for simplicity, we assume a trivial guard of *true*. Notice that all states are still reachable via some path from the initial state in the bottom left corner.

Theorem 2. *Given* $\mathbf{S} := \langle S, \mathit{Init}, A, \mathit{En}, D, T\rangle$*, let* R *be a guarded independence relation and let* $\mathbf{S}_R$ *be the* reduced *STS obtained by replacing* T *with* T^R *in* $\mathbf{S}$*. Then, if a property* P *is an invariant for* $\mathbf{S}_R$*, it is also an invariant for* $\mathbf{S}$.

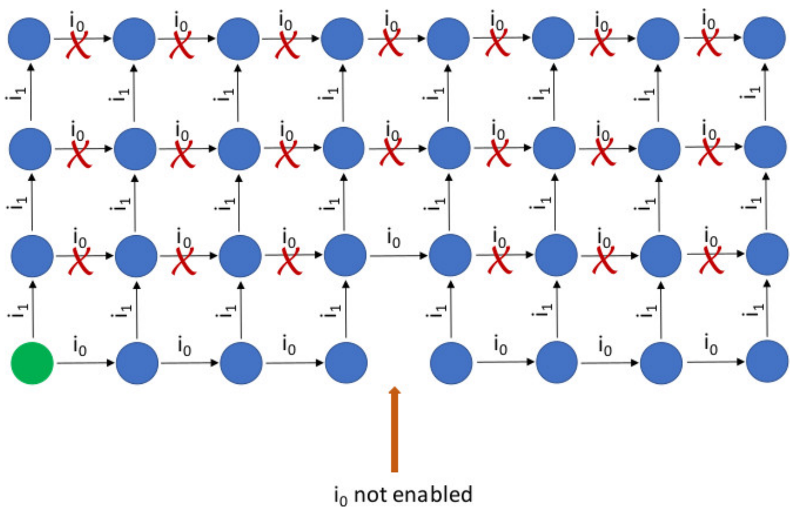


Fig. 2: Effect of Partial Order Reduction for Instructions i_0 and i_1. Initial state is green.

Proof. It suffices to show that $\mathbf{S}_R$ can reach all the same states as $\mathbf{S}$. We prove this by contradiction. Assume there is some $\boldsymbol{in}, \boldsymbol{s}$ such that $Init(s_0) \wedge T(\boldsymbol{in}, \boldsymbol{s})$ and $0 \leq j \leq |\boldsymbol{s}| - 1$ such that s_j is the first state that is unreachable in S_R. The value of j cannot be 0 or 1, because $\mathbf{S}$ and $\mathbf{S}_R$ have the same initial states and T^R only excludes sequences of length 2. Then, by the definition of T_R, $\langle in_{j-2}, in_{j-1} \rangle$ must be a sequence excluded by T^R. Conditions (2) and (3) guarantee that permuting in_{j-2} and in_{j-1} results in an enabled sequence that ends in the same state, s_j, which contradicts the assumption. Thus, there cannot be a state which is reachable in $\mathbf{S}$ but not $\mathbf{S}_R$. □

4 Reduced Instruction Sets

Now that we can apply partial order reduction to synchronous systems, our main goal is to identify a maximal guarded independence relation, R. Recall that we defined instructions as sets of atomic actions. We call an instruction containing at most one action *atomic* (this includes the instruction with *no* actions). Non-atomic instructions are *complex*. Instructions thus reflect the parallelism of synchronous hardware, and lead to natural candidates for R: pairs of atomic instructions.

Furthermore, notice that the number of instructions is exponential in the number of actions. Thus, it could be prohibitively expensive to check every pair of instructions for guarded independence. In contrast, the number of *atomic* instructions is equal to the number of actions (plus one). Furthermore, it is likely that many complex instruction pairs will not have a guarded independence relationship because they contain common actions. Our goal in this section is to disallow as many complex instructions as possible without losing any reachable states, thereby reducing the number of pairs of instructions we need to check while also making it more likely for the checks to succeed. Note that, in isolation, removing instructions might be problematic, because it could extend the bound needed to reach a property violation. However, as we will demonstrate in the experimental section, this disadvantage is more than compensated for when it is applied in combination with partial order reduction.

Given an STS with instruction set $\mathcal{I}$, we seek a reduced instruction set, $\mathcal{I}_r \subseteq \mathcal{I}$, which preserves the reachable states of the system. Let $Input_r$ be the set of inputs which only use instructions from $\mathcal{I}_r$. Given an input $in \in Input$, our goal is to prove the existence of a witness $w(in) \in Input_r^n$ (for some $n > 0$) that simulates the behavior of in using only reduced instructions. Formally, the witness function w should satisfy:

$$\begin{aligned} &\forall\, s, s' \in S, in \in Input.\, T(s, in, s') \implies \\ &\quad \exists\, n \in \mathbb{N}, \boldsymbol{s} \in S^n.\, T(w(in), s : \boldsymbol{s}) \wedge (s_{n-1} = s') \end{aligned} \tag{4}$$

In other words, we need to show that for every instruction in the original instruction set, there exists a sequence of inputs, using only instructions from the reduced instruction set (RIS), that results in the same final state. Notice that a witness function that also depended on the state would be more general, but for our purposes, it is sufficient for the witness function to depend only on the input.

4.1 Atomic instruction sets

The condition in (4) is quite general and does not provide any intuition on how to choose w. Here, we focus on a specific case where w is easy to construct: we choose $\mathcal{I}_r$ to be an *atomic instruction set*, defined as an instruction set containing only atomic instructions. We then must prove that the set of reachable states is not affected by restricting the instructions to those in $\mathcal{I}_r$.

It is sufficient to prove that for each complex instruction, we can remove one of its actions and perform that action in the next step, with the same result. For some complex instruction i containing a and some data input d, let $w^a(\langle i, d\rangle)$ be $\langle\langle i - \{a\}, d\rangle, \langle\{a\}, d\rangle\rangle$. We must show that for each input in containing a complex instruction, there exists some a where $w^a(in)$ has the equivalent effect on the system as in. Formally, the requirement is:

$$\begin{aligned} &\forall i \in \mathcal{I} \setminus \mathcal{I}_r, d \in D, \boldsymbol{s} \in S^2. \\ &\quad T(s_0, \langle i, d\rangle, s_1) \implies \exists\, a \in i, \boldsymbol{s'} \in S^3.\, T(w^a(\langle i, d\rangle), \boldsymbol{s'}) \wedge s_0 = s'_0 \wedge s_1 = s'_2 \end{aligned} \tag{5}$$

Condition (5) is still difficult to prove because of the existential quantifier. One conservative approach is to replace the existential quantifier with a universal quantifier and attempt to prove that stronger condition. For real systems, this is unlikely to hold. Instead, we propose a counterexample blocking procedure which, if it succeeds, guarantees (5). We introduce symbolic values for i, d, and a and then iteratively add constraints over them until the proof succeeds or

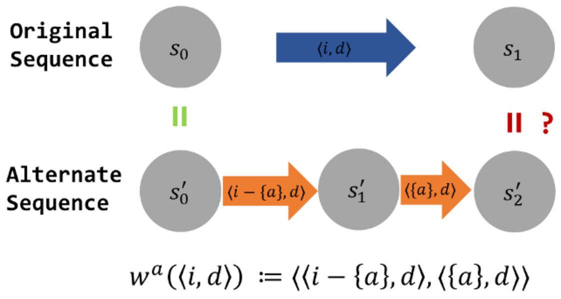

Original Sequence
s_0
$\langle i, d\rangle$
s_1
=
= ?
Alternate Sequence
s_0'
$\langle i-\{a\}, d\rangle$
s_1'
$\langle\{a\}, d\rangle$
s_2'
$w^a(\langle i,d\rangle) := \langle\langle i-\{a\},d\rangle,\langle\{a\},d\rangle\rangle$

Fig. 3: Equivalence of original and reduced instruction sequences. Circles represent states.

we have enumerated all possibilities. This algorithm is a specialized ∀∃ decision procedure that exploits the structure of (5) and additional domain knowledge about the proof goal. We use a constraint solver as an oracle.

Algorithm 1 ProveRIS(S)

```
1: $\mathbf{S}' := \langle S', Init', A', En', D', T'\rangle \leftarrow copy\_sys(\mathbf{S})$
2: $\mathcal{I} := \mathcal{P}(A)$, $\mathcal{I}' := \mathcal{P}(A')$        // instruction sets are power sets of actions
3: var $i : \mathcal{I}$, var $i' : \mathcal{I}'$, var $a : A$
4: var $\boldsymbol{s} : S^2$, var $\boldsymbol{s}' : S'^3$, var $d : D$, var $d' : D'$
5: $add\_constraint(s_0 = s_0' \wedge d = d' \wedge i' = i - \{a\})$
6: $add\_constraint(T(\langle\langle i, d\rangle\rangle, \boldsymbol{s}) \wedge T'(\langle\langle i', d'\rangle, \langle\{a\}, d'\rangle\rangle, \boldsymbol{s}'))$
7: for c = 2 ... $|A|$ do
8:     while $check\_sat(|i| = c \wedge s_1 \neq s_2')$ do
9:         $\mu \leftarrow get\_model()$
10:        $i_\mu \leftarrow assignment(\mu, i)$
11:        $a_\mu \leftarrow assignment(\mu, a)$
12:        $add\_constraint(i_\mu \subseteq i \implies a \neq a_\mu)$
13:        if $\neg check\_sat(i = i_\mu)$ then
14:            return false // exhausted all possible decompositions for this instruction
15:        end if
16:    end while
17: end for
18: return true        // every instruction can be decomposed
```

Algorithm 1 takes an STS, $\mathbf{S} := \langle S, Init, A, En, D, T\rangle$ and returns true if the instruction set can be decomposed into an atomic instruction set by delaying a single action from each instruction.[2] For simplicity, the algorithm assumes (and we check this assumption separately) that if a complex instruction i is enabled, then for each $a \in i$, executing $i - \{a\}$ results in a state where a is enabled.

[2] We also implemented a more general version of this algorithm which can drop more than one action at a time from the instruction i, but this simpler version is sufficient for the results we report in this paper.

Formally:

$$\forall\, i \in \mathcal{I} \setminus \mathcal{I}_r, d \in D, \boldsymbol{s} \in S^2, a \in i.\ en_i(s_0) \wedge T(s_0, \langle i - \{a\}, d\rangle, s_1) \implies en_a(s_1) \quad (6)$$

Note that this is only a slight generalization of the property that atomic instructions do not disable each other, a condition that we will need anyway in order to apply partial order reduction to the atomic instruction set (see condition (2)).

The algorithm first creates an identical copy of the STS in line 1. Lines 2-4 set up symbolic variables for the instructions, data, and states of each system. Line 5 adds constraints to the solver enforcing that both systems start in the same state, use the same data, and that i' is i but with symbolic action a dropped. Line 6 adds the transition relation constraint for each STS. The initial symbolic set up is depicted in Fig. 3.

The outer loop at line 7 iterates over all possible complex instruction cardinalities. The inner loop starting at line 8 attempts to show that for each cardinality c, instructions of that cardinality can be decomposed by delaying one action (symbolically represented by a). If all instructions of cardinality c have been decomposed, then the while loop condition is false and the outer loop continues. Otherwise, it gets variable assignments from the constraint solver in lines 9-11 and learns a constraint at line 13 that prevents this particular action, a_μ, from being chosen for decomposition again. To ensure that we have not blocked all possible actions, there is an additional check at line 13, which returns false in the case that no action can be delayed for the current instruction.

Importantly, the algorithm assumes that if the delay of action a_μ does not create a valid witness sequence for a given complex instruction i_μ, then the same is true *whenever* the instruction i includes i_μ. We call this a *monotonicity* assumption, and it typically holds when actions are somewhat independent. The monotonicity assumption motivated the current structure of the algorithm and can significantly reduce the number of iterations in the algorithm. We can remove this assumption by changing $i_\mu \subseteq i$ to $i_\mu = i$ in the antecedent in line 13. Note that the monotonicity assumption does not make the algorithm unsound: if it returns true, then (as we prove below) condition (5) holds. However, if the algorithm returns false, then it may be that the version without the assumption would return true. For each of our experiments, we were able to get a true result with the monotonicity assumption.

Because the algorithm does not consider state reachability and looks for a witness function that only depends on inputs, it can still return false when an equivalent sequence might exist for reachable states. In such cases, users can examine the constraint solver models and attempt to remove some of them by proving other invariants.[3]

If algorithm 1 returns true, we replace T with T_r, where T_r is the result of removing from T all transitions $\langle s, \langle i, d\rangle, s'\rangle$ where $|i| > 1$. Practically, this is

[3] This was rarely necessary in our experiments. Our implementation also extended the algorithm to support predicate abstraction, which could also rule out spurious counterexamples, but this feature was never needed in our experiments.

achieved by adding a disjunctive constraint over the possible atomic actions. We can now state the main results for reduced instruction sets.

Theorem 3. *Let S be an STS. If condition (6) holds and **ProveRIS(S)** returns true, then (5) holds.*

Proof. We maintain the loop invariant at line 8 that for every instruction i', there is some action a' such that $check\_sat(|i| = c \wedge i = i' \wedge a = a')$ is true. It's true initially for each c by condition (6). Afterwards, the check on line 14 ensures that it is maintained. Furthermore, the check on line 9 ensures that when the while loop is exited, then any satisfying assignment for $check\_sat(|i| = c)$ is such that $s_1 = s'_2$. Together, these conditions guarantee that (5) holds.

Theorem 4. *Let $\mathbf{S} \coloneqq \langle S, Init, A, En, D, T\rangle$ be an STS such that condition (6) holds and **ProveRIS(S)** returns true, and let T_r be the transition relation for the reduced instruction set. Let $\mathbf{S}_r$ be the* reduced *STS obtained by replacing T with T_r in $\mathbf{S}$. Then, safety property $P \in S$ is an invariant for $\mathbf{S}_r$ if and only if it is also an invariant for $\mathbf{S}$.*

Proof. It suffices to show that the reachable states of $\mathbf{S}$ and $\mathbf{S}_r$ are identical. *Init* does not change, so the initial states cannot be different. Furthermore, T_r is obtained by removing transitions from T, we know that S_r cannot add any reachable states. To show that it also does not remove any reachable states, consider an arbitrary trace $Init(s_0) \wedge T(\boldsymbol{in}, \boldsymbol{s})$ with $|s| = n$, we must show $\exists\, \boldsymbol{in'}, m, \boldsymbol{s'} \in S^m.\, Init(s'_0) \wedge T_r(\boldsymbol{in'}, \boldsymbol{s'}) \wedge s_{n-1} = s'_{m-1}$. We prove this by showing by induction that it holds whenever $\boldsymbol{in}$ contains instructions of cardinality at most c.

In the base case, $c = 1$, so all instructions are of size one or less. All of these are already atomic and thus we can take $\boldsymbol{in'} = \boldsymbol{in}$ and $\boldsymbol{s'} = \boldsymbol{s}$ by the definition of T_r.

For the inductive step, suppose that it holds for cardinalities up to $c-1$, and assume $Init(s_0) \wedge T(\boldsymbol{in}, \boldsymbol{s})$ with $|s| = n$. Let $in_j = \langle i, d\rangle$ be an input containing an instruction of size at most c. If $|i| < c$, there is nothing to be done. Thus we only consider the case where $|i| = c$. We know that $T(s_j, in_j, s_{j+1})$ holds. By Theorem 3 and condition (5), it follows that $T(\langle\langle i-\{a\}, d\rangle, \langle\{a\}, d\rangle\rangle, \langle s_j, s, s_{j+1}\rangle)$ holds for some a and s. We can thus replace in_j in $\boldsymbol{in}$ by $\langle i - \{a\}, d\rangle$ followed by $\langle\{a\}, d\rangle$ to obtain an input sequence $\boldsymbol{in}_c$ and insert s between s_j and s_{j+1} in $\boldsymbol{s}$ to obtain $\boldsymbol{s}_c$ with final state s_{n-1} such that $Init(s_0) \wedge T(\boldsymbol{in}_c, \boldsymbol{s}_c)$. Repeating this process for each input containing an instruction of size c yields a final $\boldsymbol{in}_c$ such that the maximum cardinality of any instruction is $c-1$. The property then holds by the inductive hypothesis. $\square$

Note that if there is some instruction $i \in \mathcal{I}$ which cannot be decomposed into atomic instructions, we could always keep this instruction in $\mathcal{I}_r$ and still benefit from removing other complex instructions. In many cases, we can also remove the empty instruction, $i_e = \emptyset$. If applying i_e cannot change the state of the system, regardless of the data input, then it is considered a *stutter step* [14]. It is straightforward to check whether i_e can be removed by comparing the state before and after applying i_e.

5 Experimental Results

We developed a prototype flow for proving the POR and RIS conditions and applying the necessary constraints. We use the IC3/PDR implementation in ABC [11], `pdr`, to prove condition (6) (which implies condition (2)). This requires manually writing a Verilog property for each atomic instruction.[4] We implemented the **ProveRIS** algorithm in our SMT-based model checker, CoSA [26], configured with boolector [29] on the `smtcomp19` branch, using CaDiCaL [4] as the underlying SAT solver.[5] We check the commuting diagram for condition (3) in CoSA as well. It tries the trivial guard `true` by default, and allows the user to provide additional candidate guards if necessary. The set up for proofs in CoSA is automated based on user-provided annotations for the actions and enable conditions. We show our best results which used an encoding leveraging the SMT theory of arrays to represent memories for proving conditions, and a pure bitvector encoding for bounded model checking.

Our flow applies the following steps: i) read in a system description in Verilog using Yosys [38] and generate AIGER [7] for ABC (or BTOR2 [29] for other tools); ii) check condition (6) for each atomic instruction; iii) run the **ProveRIS** algorithm, and if it returns true, add constraints to rule out all but atomic instructions; and iv) check POR condition (3) for each pair of atomic instructions and add constraints for each passing pair of instructions with the associated guard. Each step depends on the previous step passing successfully. In each of our experiments described below, we successfully completed every step of this flow, though in some cases guards were required in step (iv). For POR and RIS runtimes, we always include the time to check the conditions. We tried running with POR alone, but it resulted in negligible improvements in runtime and thus we omit these results. This demonstrates the importance of RIS. We ran all experiments on a 3.5GHz Intel Xeon CPU with 16GB of RAM.

5.1 Motivating Example

First, we return to our motivating example. We compare the time to reach the bug using the SAT-based ABC [11] engines `pdr` and `bmc`, and SMT-based bounded model checking using btormc [29] and CoSA. We ran the SMT-based model checkers both with and without the SMT theory of arrays for the encoding of the memory. Both btormc and CoSA without the array encoding were able to reach the bug in 1230s and 1437s, respectively, but all other approaches timed out at two hours. In particular, `pdr` times out at 2 hours on the property, but can

[4] This could be automated based on user-tagged actions and user-provided enable conditions.

[5] GitHub Commit Hashes for Tools:
Boolector/Btormc: 1989080261235f33e344cbd095e70a337c45bd16
CoSA: ff3c8cee1f0834c03167b2a8ecdd1223031312b3
PySMT: 09dc3031858121495501101 23ad266326beb1179
Yosys: a4b59de5d48a89ba5e1b46eb44877a91ceb6fa44
ABC: 5776ad07e7247993976bffed4802a5737c456782

Design	#	#Solved	#Solved PR	Time	Time PR
com 1	49	35	**47**	103.8	**20.3**
com 2	49	25	**34**	470.8	**4.9**
cp	49	35	**47**	230.3	**18.9**
sr	49	25	**33**	912.6	**5.6**
arb n=2	49	35	**42**	89.1	**20.5**
arb n=3	49	35	**42**	94.0	**21.9**
arb n=4	49	35	**42**	101.3	**35.5**
arb n=5	49	35	**42**	111.7	**31.8**

Table 1: Number Solved and Average Runtime

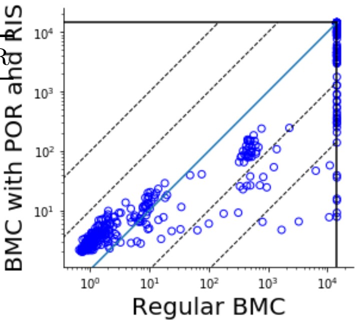


Fig. 4: Runtime Comparison

prove condition (6) for every atomic instruction in less than a second. Intuitively, this makes sense because the enabledness conditions do not involve `data` or `mem`. Thus, none of the datapath falls in the cone of influence, leaving only control logic for IC3 to reason about. The remaining conditions, (3) and (5), are proven in less than three seconds. Since all the conditions pass, we apply the POR and RIS constraints, which reduces the time to hit the bug from 1437s to 46s in CoSA, including the time to check the conditions.

5.2 Packet Movers

We now evaluate our approach on data integrity properties for a variety of packet-mover circuits. Data integrity is a safety property that ensures no packets are dropped or corrupted. In practice, data integrity is often checked by instantiating a monitor, called a *scoreboard*. It provides the necessary infrastructure for formal verification. In our case, it non-deterministically tags a *magic packet* and checks that this packet exits the system when it should. Crucially, the scoreboard is a reusable module which can check data integrity of arbitrary packet movers.

Notice that existing symmetry reduction techniques will not be very effective for this scoreboard setup. For example, consider a circular pointer FIFO which maintains two incrementing pointers that index a memory for reading and writing, respectively. We cannot use scalarsets to break symmetries in the memory addresses because the pointers index the memory and are involved in arithmetic, breaking the syntactic requirements for scalarsets [28]. Furthermore, sequential memory abstraction [9] could reduce the size of the memory, but does not address the path symmetry. In addition, both these symmetry reduction techniques are focused on proofs, not bug-finding.

We evaluate our approach on two commercial library components from a major hardware company. We also implemented simpler, open-source versions of these designs. Our open-source benchmarks include: i) a circular pointer FIFO which assumes power-of-two depth but is instantiated with a non-power-of-two depth (one greater than the provided parameter); ii) a shift register FIFO which does not properly add data to the last register in the pipeline; and iii) 2-5 correct circular pointer FIFOs in parallel with a non-deterministic arbiter and credit counters for managing data flow. The reset state of the credit counter has one too many credits, so data can be pushed to a full FIFO. The single FIFOs have two actions each: one for pushing data, and one for popping data. For the arbitrated circuits, there is a separate action for pushing data onto each FIFO as well as a single request action which is enabled whenever any FIFO is non-empty. There is an inherent symmetry in all of these designs. Consider any of the FIFOs. There are two main actions: pushing data (which is enabled if the FIFO is not full); and popping data (which is enabled if the FIFO is not empty). In a state where both are enabled, pushing data followed by popping results in the same state as popping and then pushing the same data. Furthermore, the actions can be performed simultaneously, but requiring that they are performed separately should not change the reachable states (depending on the implementation), so RIS is applicable.

Our experiments vary both the parameterizable data width and depth of the packet movers, by sweeping all powers of two between 2 and 128. All benchmarks contain injected bugs and reach the bug at a deep bound relative to the depth. We used a timeout of 4 hours. We use our prototype flow for checking the conditions and CoSA for bounded model checking.[6] For condition (3), we had to write one guard which is true whenever the scoreboard counter is greater than zero to handle an edge case. This same guard was used for every design, but an appropriate invariant relating the scoreboard counter to the internal state of the system being verified would also have worked. The open-source shift register FIFO required one more guard about the number of stored elements. We obtained both guards by observing counterexamples.

Table 1 compares the number of solved instances (49 total per row) within the timeout and the average runtime of commonly solved instances in seconds. Columns marked "PR" used the POR and RIS constraints. We additionally use the following abbreviations: "com" for commercial, "cp" for circular pointer, "sr" for shift register and "arb" for arbitrated. In Fig. 4 we plot the actual runtime on a log-scale for all the benchmarks with and without POR and RIS. The dotted lines show 10x and 100x improvements.

Analysis. There is a cluster of points in the bottom left of Fig. 4 which are solved extremely quickly by both approaches, but slightly faster without POR and RIS. These are results on benchmarks with very small parameter values, where the bug occurs at a low depth, and so the POR and RIS results are dominated by the time taken to check the conditions. However, as the parameter sizes, and runtimes, increase, it is clear that POR and RIS can result in exponential speed ups.

Recall that one concern is that RIS could extend the bound needed to reach the bug. In the shift register and arbitrated FIFO systems, it extended the bound by a few steps. However, for the bug in the open-source circular pointer FIFO, it doubled the bound needed to reach the bug. Regardless, this was more than

[6] Note: CoSA's bounded model checking performance is comparable to commercial model checkers on these benchmarks.

compensated for by the symmetry-breaking of POR, as evidenced by the faster times to reach the bug. The deepest bound was 260 which occurred at FIFO depth 129.

It is interesting to note that encoding the transition systems to SMT using the theory of arrays was always slower for bounded model checking, but was noticeably faster for checking RIS and POR conditions. Perhaps this is because the state comparison is easier for the solver to reason about using array extensionality [23].

We have demonstrated that these techniques work well for packet movers. In part, this is because packet movers are often well-constrained by their environmental assumptions, and their behavior is largely independent of incoming data values. Furthermore, we typically expect the POR and RIS conditions to hold for a correct packet-mover implementation, so a failure in a condition could identify a bug.

6 Related Work

Various techniques have been employed to accelerate bounded model checking. The authors of [19] use BDDs to accelerate BMC, and the techniques introduced in [35,36] exploit the structure of BMC queries to help the SAT solver. The authors of [18] take advantage of structural information with an SMT framework tailored for BMC. Our technique is similar in that we speed up bounded model checking by adding constraints to the transition system, but we obtain constraints using partial order reduction analysis.

Wang et al. [37] pioneered partial order reduction for symbolic software model checking, guaranteeing optimal reduction for two threads. Their follow-up paper, [21], extended this framework to find the optimal reduction for any number of threads. We adapted their symbolic POR technique for synchronous hardware model checking, and developed reduced instruction sets to improve the efficacy of POR in this new domain. Bhattacharya et al. used a SAT solver to directly check guarded independence conditions (as opposed to checking syntactic properties) for asynchronous rule-based languages [3]. We also check conditions directly, but in a synchronous setting.

The techniques developed by McMillan, *temporal case splitting* and *path splitting* [28], provide a framework for splitting on possible values at a given timestep. These approaches deal with system executions, but still rely on breaking data symmetries for performance. In contrast, our techniques focus on mitigating path symmetries.

The work of Bengtsson et al. [2] extended POR to timed automata using a local-time desynchronization of clocks, followed by resynchronization with an added global clock. Similarly, our techniques adapt POR by modifying the system. However, our approach targets a different domain, and only modifies the original system by adding constraints.

7 Conclusion

We have presented a set of conservative conditions over transition systems and automated techniques for proving these conditions. If the conditions can be proved, then constraints can be added to the system that break path symmetries. We evaluated our approach on parameterized open-source and commercial packet-mover circuits and demonstrated significant improvements in bounded model checking performance.

Some potential future work includes improvements to the **ProveRIS** procedure, investigating applications of partial order reduction to sequentially-composed packet movers, developing more targeted condition proofs by associating actions with particular data inputs, and building an interactive tool which helps the user identify and manage reduced instruction sets and partial order reductions.

8 Data Availability Statement

The experimental results and the necessary software for reproducing results in a standard Ubuntu 18.04 installation are available in the Figshare repository: https://doi.org/10.6084/m9.figshare.11874687.

References

1. Bailey, B.: When bugs escape (July 2018), https://semiengineering.com/when-bugs-escape/, [Online]
2. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR'98 Concurrency Theory. pp. 485–500. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
3. Bhattacharya, R., German, S., Gopalakrishnan, G.: Symbolic partial order reduction for rule based transition systems. In: Borrione, D., Paul, W. (eds.) Correct Hardware Design and Verification Methods. pp. 332–335. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
4. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions. vol. B-2017-1, pp. 14–15. University of Helsinki (2017)
5. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. **66**(2), 160–177 (2002)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
7. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. rep., FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
8. Bjesse, P.: A practical approach to word level model checking of industrial netlists. pp. 446–458 (07 2008)
9. Bjesse, P.: Word-level sequential memory abstraction for model checking. In: FMCAD. pp. 1–9. IEEE (2008)

10. Bradley, A.R.: Sat-based model checking without unrolling. In: VMCAI. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011)
11. Brayton, R., Mishchenko, A.: Abc: An academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
12. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 1020 states and beyond. Information and Computation **98**(2) (1992). https://doi.org/https://doi.org/10.1016/0890-5401(92)90017-A
13. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions inmodel checking. In: CAV. Lecture Notes in Computer Science, vol. 1427, pp. 147–158. Springer (1998)
14. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
15. Clarke, E., Henzinger, T., Veith, H.: Handbook of Model Checking. Springer International Publishing (2016), https://books.google.com/books?id=qxG8oAEACAAJ
16. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134. FMCAD Inc. (2011)
17. Foster, H.: Applied assertion-based verification: An industry perspective. Foundations and Trends in Electronic Design Automation **3**(1), 1–95 (2009)
18. Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: ICCAD. pp. 794–801. ACM (2006)
19. Gupta, A., Ganai, M.K., Wang, C., Yang, Z., Ashar, P.: Learning from bdds in sat-based bounded model checking. In: DAC. pp. 824–829. ACM (2003)
20. Johannsen, P.: Speeding Up Hardware Verification by Automated Data Path Scaling. Ph.D. thesis, University of Kiel (2002)
21. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: CAV. Lecture Notes in Computer Science, vol. 5643, pp. 398–413. Springer (2009)
22. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. CoRR **abs/1801.01203** (2018), http://arxiv.org/abs/1801.01203
23. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer Publishing Company, Incorporated, 1 edn. (2008)
24. Lam, W.K.: Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series). Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
25. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown. CoRR **abs/1801.01207** (2018)
26. Mattarei, C., Mann, M., Barrett, C.W., Daly, R.G., Huff, D., Hanrahan, P.: Cosa: Integrated verification for agile hardware design. In: FMCAD. pp. 1–5. IEEE (2018)
27. McMillan, K.L.: Interpolation and sat-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification. pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
28. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Sci. Comput. Program. **37**(1-3), 279–309 (2000)
29. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , btormc and boolector 3.0. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018)

30. Peled, D.: Verification for robust specification. In: Gunter, E.L., Felty, A. (eds.) Theorem Proving in Higher Order Logics. pp. 231–241. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
31. Peled, D.A., Wilke, T., Wolper, P.: An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. Theor. Comput. Sci. **195**(2), 183–203 (1998)
32. Price, D.: Pentium fdiv flaw-lessons learned. IEEE Micro **15**(2), 86–88 (April 1995). https://doi.org/10.1109/40.372360
33. Sagstetter, F., Lukasiewycz, M., Steinhorst, S., Wolf, M., Bouard, A., R. Harris, W., Jha, S., Peyrin, T., Poschmann, A., Chakraborty, S.: Security challenges in automotive hardware/software architecture design. pp. 458–463 (01 2013). https://doi.org/10.7873/DATE.2013.102
34. Shacham, O., Azizi, O., Wachs, M., Richardson, S., Horowitz, M.: Rethinking digital design: Why design must change. IEEE Micro **30**(6), 9–24 (2010)
35. Strichman, O.: Tuning SAT checkers for bounded model checking. In: CAV. Lecture Notes in Computer Science, vol. 1855, pp. 480–494. Springer (2000)
36. Strichman, O.: Accelerating bounded model checking of safety properties. Formal Methods in System Design **24**(1), 5–24 (2004)
37. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 382–396. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
38. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free Verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)

Revisiting Underapproximate Reachability for Multipushdown Systems*

S. Akshay[1], Paul Gastin[2], S Krishna[1], and Sparsa Roychowdhury[1]

[1] IIT Bombay, Mumbai, India
[2] ENS Paris-Saclay, Paris, France

Abstract Boolean programs with multiple recursive threads can be captured as pushdown automata with multiple stacks. This model is Turing complete, and hence, one is often interested in analyzing a restricted class that still captures useful behaviors. In this paper, we propose a new class of bounded underapproximations for multi-pushdown systems, which subsumes most existing classes. We develop an efficient algorithm for solving the under-approximate reachability problem, which is based on efficient fix-point computations. We implement it in our tool BHIM and illustrate its applicability by generating a set of relevant benchmarks and examining its performance. As an additional takeaway BHIM solves the binary reachability problem in pushdown automata. To show the versatility of our approach, we then extend our algorithm to the timed setting and provide the first implementation that can handle timed multi-pushdown automata with closed guards.

Keywords: Multipushdown Systems · Underapproximate Reachability · Timed pushdown automata.

1 Introduction

The reachability problem for pushdown systems with multiple stacks is known to be undecidable. However, multi-stack pushdown automata (MPDA hereafter) represent a theoretically concise and analytically useful model of multi-threaded recursive programs with shared memory. As a result, several previous works in the literature have proposed different under-approximate classes of behaviors of MPDA that can be analyzed effectively, such as *Round Bounded*, *Scope Bounded*, *Context Bounded* and *Phase Bounded* [18,19,27,14,20,28]. From a practical point of view, these underapproximations have led to efficient tools including, GetaFix [21], SPADE [23]. It has also been argued (e.g., see [24]) that such bounded underapproximations suffice to find several bugs in practice. In many such tools efficient fix-point techniques are used to speed-up computations.

We extend known fix-point based approaches by developing a new algorithm that can handle a larger class of bounded underapproximations than the well-known bounded context and bounded scope underapproximations for

* Partly supported by UMI ReLaX, DST/CEFIPRA/INRIA project EQuaVe & TCS.

multi-pushdown systems while remaining efficiently implementable. Our algorithm works for a new class of underapproximate behaviors called *hole bounded* behaviors, which subsumes context/scope bounded underapproximations, and is orthogonal to phase bounded underapproximations. A "hole" is a maximal sequence of push operations of a fixed stack, interspersed with well-nested sequences of any stack. Thus, in a sequence $\alpha = \beta\gamma$ where $\beta = [push_1(push_2 push_3 pop_3 pop_2)push_1(push_3 pop_3)]^{10}$ and $\gamma = push_2 push_1 pop_2 pop_1 (pop_1)^{20}$, β is a hole with respect to stack 1. The suffix γ has 2 holes (the $push_2$ and the $push_1$). Thus we say that α is 3-hole bounded. On the other hand, the number of context switches (and scope bound) in α is > 50. A (k-)hole bounded sequence is one such, where, at any point of the computation, the number of "open" holes are bounded at this point (by k). We show that the class of hole bounded sequences subsumes most of the previously defined classes of underapproximations and is, in fact, contained in the very generic class of tree-width bounded sequences. This immediately shows decidability of the reachability problem for our class.

Analyzing the more generic class of tree-width bounded sequences is often much more difficult; for instance, building bottom-up tree automata for this purpose does not scale very well as it explores a large (and often useless) state space. Our technique is radically different from using tree automata. Under the hole bounded assumption, we pre-compute information regarding well-nested sequences and holes using fix-point computations and use them in our algorithm. Using efficient data structures to implement this approach, we develop a tool (BHIM) for Bounded Hole reachability in Multi-stack pushdown systems.

Highlights of BHIM.

• Two significant aspects of the fix-point approach in BHIM are: (i) we efficiently solve the binary reachability problem for pushdown automata. i.e., BHIM computes all pairs of states (s, t) such that t is reachable from s with empty stacks. This allows us to go beyond reachability and handle some liveness questions; (ii) we pre-compute the set of pairs of states that are endpoints of holes. This allows us to greatly limit the search for an accepting run.

• While the fix-point approach solves (binary) reachability efficiently, it does not a priori produce a witness of reachability. We remedy this situation by proposing a backtracking algorithm, which cleverly uses the computations done in the fix-point algorithm, to generate a witness efficiently.

• BHIM is parametrized with respect to the hole bound: if non-emptiness can be checked or witnessed by a well-nested sequence (this is an easy witness and BHIM looks for easy witnesses first, then gradually increases complexity, if no easy witness is found), then it is sufficient to have the hole bound 0. Increasing this complexity measure as required to certify non-emptiness gives an efficient implementation, in the sense that we search for harder witnesses only when no easier witnesses (w.r.t this complexity measure) exist. In examples described in the experimental section, a small (less than 4) bound suffices and we expect this to be the case for most practical examples.

• Finally, we extend our approach to handle timed multi-stack pushdown systems. This shows the versatility of our approach and also requires us to solve several technical challenges which are specific to the timed setting. Implementing this approach in BHIM makes it, to the best of our knowledge, the first tool that can analyze timed multi-stack pushdown automata with closed guards.

We analyze the performance of BHIM in practice, by considering benchmarks from the literature, and generating timed variants of some of them. One of our benchmarks is a variant of the Bluetooth example [11,23], where BHIM was able to catch a known race detection error. Another interesting benchmark is a model of a parameterized multiple producer consumer example, having parameters M, N on the quantities of two items A, B produced. Here, BHIM could detect bugs by finding witnesses having just 2 holes, while, it is unlikely that existing tools working on scope/context bounded underapproximations can handle them as the number of scope/context switches is dependent on M, N (in fact, it is twice the least common multiple of M and N). In the timed setting, one of the main challenges has been the unavailability of timed benchmarks; even in the untimed setting, many benchmarks were unavailable due to their proprietary nature. Due to lack of space, proofs, technical details and parametric plots of experiments are in [4].

Related Work. Among other under-approximations, scope bounded [27] subsumes context and round bounded underapproximations, and it also paves path for GetaFix [21], a tool to analyze recursive (and multi-threaded) boolean programs. As mentioned earlier hole boundedness strictly subsumes scope boundedness. On the other hand, GetaFix uses symbolic approaches via BDDs, which is orthogonal to the improvements made in this paper. Indeed, our next step would be to build a symbolic version of BHIM which extends the hole-bounded approach to work with symbolic methods. Given that BHIM can already handle synthetic examples with 12-13 holes (see [4]), we expect this to lead to even more drastic improvements and applicability. For sequential programs, a summary-based algorithm is used in [21]; summaries are like our well-nested sequences, except that well-nested sequences admit contexts from different stacks unlike summaries. As a result, our class of bounded hole behaviors generalizes summaries. Many other different theoretical results like phase bounded [18], order bounded [8] which gives interesting underapproximations of MPDA, are subsumed in tree-width bounded behaviors, but they do not seem to have practical implementations. Adding real-time information to pushdown automata by using clocks or timed stacks has been considered, both in the discrete and dense-timed settings. Recently, there has been a flurry of theoretical results in the topic [10,1,2,5,6]. However, to the best of our knowledge none of these algorithms have been successfully implemented (except [6] which implements a tree-automata based technique for single-stack timed systems) for multi-stack systems. One reason is that these algorithms do not employ scalable fix-point based techniques, but instead depend on region automaton-based search or tree automata-based search techniques.

2 Underapproximations in MPDA

A multi-stack pushdown automaton (MPDA) is a tuple $M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, n, \Sigma, \Gamma)$ where, $\mathcal{S}$ is a finite non-empty set of locations, Δ is a finite set of transitions, $s_0 \in \mathcal{S}$ is the initial location, $\mathcal{S}_f \subseteq \mathcal{S}$ is a set of final locations, $n \in \mathbb{N}$ is the number of stacks, Σ is a finite input alphabet, and Γ is a finite stack alphabet which contains $\bot$. A transition $t \in \Delta$ can be represented as a tuple (s, op, a, s'), where, $s, s' \in \mathcal{S}$ are respectively, the source and destination locations of the transition t, $a \in \Sigma$ is the label of the transition, and op is one of the following operations (1) nop, or no stack operation, (2) $(\downarrow_i \alpha)$ which pushes $\alpha \in \Gamma$ onto stack $i \in \{1, 2, \ldots, n\}$, (3) $(\uparrow_i \alpha)$ which pops stack i if the top of stack i is $\alpha \in \Gamma$.

For a transition $t = (s, \mathsf{op}, a, s')$ we write $\mathsf{src}(t) = s, \mathsf{tgt}(t) = s'$ and $\mathsf{op}(t) = \mathsf{op}$. At the moment we ignore the action label a but this will be useful later when we go beyond reachability to model checking. A *configuration* of the MPDA is a tuple $(s, \lambda_1, \lambda_2, \ldots, \lambda_n)$ such that, $s \in \mathcal{S}$ is the current location and $\lambda_i \in \Gamma^*$ represents the current content of i^{th} stack. The semantics of the MPDA is defined as follows: a run is accepting if it starts from the initial state and reaches a final state with all stacks empty. The language accepted by a MPDA is defined as the set of words generated by the accepting runs of the MPDA. Since the reachability problem for MPDA is Turing complete, we consider under-approximate reachability.

A sequence of transitions is called **complete** if each push in that sequence has a matching pop and vice versa. A **well-nested** sequence denoted ws is defined inductively as follows: a possibly empty sequence of nop-transitions is ws, and so is the sequence $t\ ws\ t'$ where $\mathsf{op}(t) = (\downarrow_i \alpha)$ and $\mathsf{op}(t') = (\uparrow_i \alpha)$ are a matching pair of push and pop operations of stack $i, \forall i \in \{1 \ldots n\}$. Finally the concatenation of two well-nested sequences is a well-nested sequence, i.e., they are closed under concatenation. The set of all well-nested sequences defined by an MPDA is denoted WS. If we visualize this by drawing edges between pushes and their corresponding pops, well-nested sequences have no crossing edges, as in and , where we have two stacks, depicted with red and violet edges. We emphasize that a well-nested sequence can have well-nested edges from any stack. In a sequence σ, a push (pop) is called a **pending** push (pop) if its matching pop (push) is not in the same sequence σ.

Bounded Underapproximations. As mentioned in the introduction, different bounded under-approximations have been considered in the literature to get around the Turing completeness of MPDA. During a computation, a context is a sequence of transitions where only one stack or no stack is used. In *context bounded* computations the number of contexts are bounded [25]. A *round* is a sequence of (possibly empty) contexts for stacks $1, 2, \ldots, n$. *Round bounded* computations restrict the total number of rounds allowed [19,5,6]. *Scope bounded* computations generalize bounded context computations. Here, the context changes within any push and its corresponding pop is bounded [19,20,28]. A *phase* is a contiguous sequence of transitions in a computation, where we restrict pop to only one stack, but there are no restrictions on the pushes [18]. A phase bounded computation is one where the number of phase changes is bounded.

Tree-width. A generic way of looking at them is to consider classes which have a bound on the tree-width [22]. In fact, the notions of split-width/clique-width/tree-width of communicating finite state machines/timed push down systems has been explored in [3], [13]. The behaviors of the underlying system are then represented as graphs. It has been shown in these references that if the family of graphs arising from the behaviours of the underlying system (say S) have a bounded tree-width, then the reachability problem is decidable for S via, tree-automata. However, this does not immediately give rise to an efficient implementation. The tree-automata approach usually gives non-deterministic or bottom-up tree automata, which when implemented in practice (see [6]) tend to blow up in size and explore a large and useless space. Hence there is a need for efficient algorithms, which exist for more specific underapproximations such as context-bounded (leading to fix-point algorithms and their implementations [21]).

2.1 A new class of under-approximations

Our goal is to bridge the gap between having practically efficient algorithms and handling more expressive classes of under-approximations for reachability of multi-stack pushdown systems. To do so, we define a bounded approximation which is expressive enough to cover previously defined practically interesting classes (such as context bounded etc), while at the same time allowing efficient decidable reachability tests, as we will see in the next section.

Definition 1. *(Holes). Let σ be complete sequence of transitions, of length n in a* MPDA*, and let ws be a well-nested sequence.*

- *A* ***hole*** *of stack i is a maximal factor of σ of the form $(\downarrow_i ws)^+$, where $ws \in \mathsf{WS}$. The maximality of the hole of stack i follows from the fact that any possible extension ceases to be a hole of stack i; that is, the only possible events following a maximal hole of stack i are a push $\downarrow_j$ of some stack $j \neq i$, or a pop of some stack $j \neq i$. In general, whenever we speak about a hole, the underlying stack is clear.*
- *A push $\downarrow_i$ in a hole (of stack i) is called a pending push at (i.e., just before) a position $x \leq n$, if its matching pop occurs in σ at a position $z > x$.*
- *A hole (of stack i) is said to be* ***open*** *at a position $x \leq n$, if there is a pending push $\downarrow_i$ of the hole at x. Let $\#_x(\mathsf{hole})$ denote the number of open holes at position x. The* ***hole bound*** *of σ is defined as $\mathsf{max}_{1 \leq x \leq |\sigma|} \#_x(\mathsf{hole})$.*
- *A* hole segment *of stack i is a prefix of a hole of stack i, ending in a ws, while an* atomic hole segment *of stack i is just the segment of the form $\downarrow_i ws$.*

As an example, consider the sequence σ in Figure 1 of transitions of a MPDA having stacks 1,2 (denoted respectively red and blue). We use superscripts for

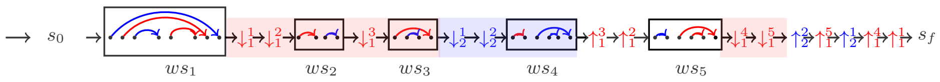


Figure 1. A run σ with 2 holes (2 red patches) of the red stack and 1 hole (one blue patch) of the blue stack.

each push, pop of each stack to distinguish the ith push, jth pop and so on of each stack. There are two holes of stack 1 (red stack) denoted by the red patches, and one hole of stack 2 (blue stack) denoted by the blue patch. The subsequence $\downarrow_1^1\downarrow_1^2 ws_2$ of the first hole is not a maximal factor, since it can be extended by $\downarrow_1^3 ws_3$ in the run σ, extending the hole. Consider the position in σ marked with $\downarrow_2^1$. At this position, there is an open hole of the red stack (the first red patch), and there is an open hole of the blue stack (the blue patch). Likewise, at the position $\uparrow_1^5$, there are 2 open holes of the red stack (2 red patches) and one open hole of the blue stack 2 (the blue patch). The hole bound of σ is 3. The green patch consisting of $\uparrow_1^3$, $\uparrow_1^2$ and ws_5 is a pop-hole of stack 1. Likewise, the pops $\uparrow_2^2$, $\uparrow_1^5$, $\uparrow_2^1$ are all pop-holes (of length 1) of stacks 2,1,2 respectively.

Definition 2. (Hole Bounded Reachability Problem) *Given a* MPDA *and $K \in \mathbb{N}$, the K-hole bounded reachability problem is the following: Does there exist a K-hole bounded accepting run of the* MPDA*?*

Proposition 1. *The tree-width of K-hole bounded* MPDA *behaviors is at most $(2K + 3)$.*

With this, from [22][5][6], decidability and complexity follow. Thus,

Corollary 1. *The K-hole bounded reachability problem for* MPDA *is decidable in $\mathcal{O}(|\mathcal{M}|^{2K+3})$ where, $\mathcal{M}$ is the size of the underlying* MPDA.

Next, we turn to the expressiveness of this class with respect to the classical underapproximations of MPDA: first, the **hole** bounded class strictly subsumes **scope** bounded which already subsumes **context** bounded and **round** bounded classes. Also **hole** bounded MPDA and **phase** bounded MPDA are orthogonal.

Proposition 2. *Consider a* MPDA *M. For any K, let L_K denote a set of sequences accepted by M which have number of rounds or number of contexts or scope bounded by K. Then there exists $K' \leq K$ such that L_K is K' hole bounded. Moreover, there exist languages which are K hole bounded for some constant K, which are not K' round or context or scope bounded for any K'. Finally, there exists a language which is accepted by phase bounded* MPDA *but not accepted by hole bounded* MPDA *and vice versa.*

Proof. We first recall that if a language L is K-round, or K-context bounded, then it is also K'-scope bounded for some $K' \leq K$ [20,19]. Hence, we only show that scope bounded systems are subsumed by hole bounded systems.

Let L be a K-scope bounded language, and let M be a MPDA accepting L. Consider a run ρ of $w \in L$ in M. Assume that at any point i in the run ρ, $\#_i(\texttt{holes}) = k'$, and towards a contradiction, let, $k' > K$. Consider the leftmost open hole in ρ which has a pending push $\downarrow_p$ whose pop $\uparrow_p$ is to the right of i. Since $k' > K$ is the number of open holes at i, there are at least $k' > K$ context changes in between $\downarrow_p$ and $\uparrow_p$. This contradicts the K-scope bounded assumption, and hence $k' \leq K$.

To show the strict containment, consider the visibly pushdown language [7] given by $L^{bh} = \{a^n b^n (a^{p_1} c^{p_1+1} b^{p'_1} d^{p'_1+1} \cdots a^{p_n} c^{p_n+1} b^{p'_n} d^{p'_n+1}) \mid n, p_1, p'_1, \ldots, p_n, p'_n \in$

$\mathbb{N}\}$. A possible word $w \in L^{bh}$ is $a^3b^3\ a^2c^3b^2d^3\ a^2c^3bd^2\ ac^2bd^2$ with a, b representing push in stack 1,2 respectively and c, d representing the corresponding matching pop from stack 1,2. A run ρ accepting the word $w \in L^{bh}$ will start with a sequence of pushes of stack 1 followed by another sequence of pushes of stack 2. Note that, the number of the pushes n is same in both stacks. Then there is a group G consisting of a well-nested sequence of stack 1 (equal a and c) followed by a pop of the stack 1 (an extra c), another well-nested sequence of stack 2 (equal b and d) and a pop of the stack 2 (an extra d), repeated n times. From the definition of the `hole`, the total number of holes required in G is 0. But, we need 1 hole for the sequence of a's and another for the sequence of b's at the beginning of the run, which creates at most 2 holes during the run. Thus, the hole bound for any accepting run ρ is 2, and the language L^{bh} is 2-hole bounded.

However, L^{bh} is not k-scope bounded for any k. Indeed, for each $m \geq 1$, consider the word $w_m = a^m b^m (ac^2bd^2)^m \in L^{bh}$. It is easy to see that w_m is $2m$-scope bounded (the matching c, d of each a, b happens $2m$ context switches later) but not k-scope bounded for $k < 2m$. It can be seen that L^{bh} is not k-phase bounded either. Finally, $L' = \{(ab)^n c^n d^n \mid n \in \mathbb{N}\}$ with a, b and c, d respectively being push and pop of stack 1,2 is not hole-bounded but 2-phase bounded. □

3 A Fix-point Algorithm for Hole Bounded Reachability

In the previous section, we showed that hole-bounded underapproximations are a decidable subclass for reachability, by showing that this class has a bounded tree-width. However, as explained in the introduction, this does not immediately give a fix-point based algorithm, which has been shown to be much more efficient for other more restricted sub-classes, e.g., context-bounded. In this section, we provide such a fix-point based algorithm for the hole-bounded class and explain its advantages. Later we discuss its versatility by showing extensions and evaluating its performance on a suite of benchmarks.

We describe the algorithm in two steps: first we give a simple fix-point based algorithm for the problem of 0-hole or *well-nested reachability*, i.e, reachability by a well-nested sequence without any holes. For the 0-hole case, our algorithm computes the *reachability relation*, also called the *binary reachability problem [15]*. That is, we accept all pairs of states (s, s') such that there is a well-nested run from s with empty stack to s' with empty stack. Subsequently, we combine this binary reachability for well-nested sequences with an efficient graph search to obtain an algorithm for K-hole bounded reachability.

Binary well-nested reachability for MPDA. Note that single stack PDA are a special case, since all runs are indeed well-nested.

1. **Transitive Closure**: Let $\mathcal{R}$ be the set of tuples of the form (s_i, s_j) representing that state s_j is reachable from state s_i via a nop discrete transition. Such a sequence from s_i to s_j is trivially *well-nested*. We take the TransitiveClosure of $\mathcal{R}$ using Floyd-Warshall algorithm [12]. The resulting set $\mathcal{R}_c$ of tuples answers the binary reachability for finite state automata (no stacks).

Algorithm 1: Algorithm for Emptiness Checking of hole bounded MPDA

```
Function IsEmpty($M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, n, \Sigma, \Gamma), K$):
    Result: True or False
    WR := WellNestedReach($M$); \\Solves binary reachability for pushdown system
    if some $(s_0, s_1) \in$ WR with $s_1 \in \mathcal{S}_f$ then
        return False;
    forall $i \in [n]$ do
        $AHS_i := \emptyset$; $Set_i := \emptyset$;
        forall $(s, \downarrow_i(\alpha), a, s_1) \in \Delta$ and $(s_1, s') \in$ WR do
            $AHS_i := AHS_i \cup \{(i, s, \alpha, s')\}$; $Set_i := Set_i \cup \{(s, s')\}$;
        $HS_i := \{(i, s, s') \mid (s, s') \in$ TransitiveClosure($Set_i$)$\}$;
    $\mu := [s_0]$; $\mu$.NumberOfHoles := 0;
    SetOfLists$_{new}$ := $\{\mu\}$; SetOfLists := $\emptyset$;
    do
        SetOfLists := SetOfLists $\cup$ SetOfLists$_{new}$;
        SetOfLists$_{todo}$ := SetOfLists$_{new}$; SetOfLists$_{new}$ := $\emptyset$;
        forall $\mu' \in$ SetOfLists$_{todo}$ do
            if $\mu'$.NumberOfHoles $< K$ then
                forall $i \in [n]$ do
                    \\ Add hole for stack i
                    SetOfLists$_h$ := AddHole$_i$($\mu', HS_i$) \ SetOfLists;
                    SetOfLists$_{new}$ := SetOfLists$_{new}$ $\cup$ SetOfLists$_h$;
            if $\mu'$.NumberOfHoles $> 0$ then
                forall $i \in [n]$ do
                    \\ Add pop for stack i
                    SetOfLists$_p$ := AddPop$_i$($\mu', M, AHS_i, HS_i$, WR) \ SetOfLists;
                    SetOfLists$_{new}$ := SetOfLists$_{new}$ $\cup$ SetOfLists$_p$;
                    forall $\mu_3 \in$ SetOfLists$_p$ do
                        if $\mu_3.last \in \mathcal{S}_f$ and $\mu_3$.NumberOfHoles $= 0$ then
                            return False; \\If reached destination state
    while SetOfLists$_{new} \neq \emptyset$;
    return True;
```

2. **Push-Pop Closure**: For stack operations, consider a push transition on some stack (say stack i) of symbol γ, enabled from a state s_1, reaching state s_2. If there is a matching pop transition from a state s_3 to s_4, which pops the same stack symbol γ from the stack i and if we have $(s_2, s_3) \in \mathcal{R}_c$, then we can add the tuple (s_1, s_4) to $\mathcal{R}_c$. The function WellNestedReach repeats this process and the transitive closure described above until a fix-point is reached. Let us denote the resulting set of tuples by WR. Thus,

Lemma 1. $(s_1, s_2) \in$ WR *iff* $\exists$ *a well-nested run in the* MPDA *from* s_1 *to* s_2*.*

Beyond well-nested reachability. A naive algorithm for K-hole bounded reachability for $K > 0$ is to start from the initial state s_0, and do a Breadth First Search (BFS), nondeterministically choosing between extending with a well-nested segment, creating hole segments (with a pending push) and closing hole segments (using pops). We accept when there are no open hole segments and reach a final state; this gives an exponential time algorithm. Given the exponential dependence on the hole-bound K (Corollary 1), this exponential blowup is unavoidable in the worst case, but we can do much better in practice. In particular, the naive algorithm makes arbitrary non-deterministic choices resulting in a blind exploration of the BFS tree.

In this section, we use the binary well-nested reachability algorithm as an efficient subroutine to limit the search in BFS to its reachable part (note that this is quite different from DFS as well since we do not just go down one path). The crux is that at any point, we create a new hole for stack i, *only* when (i) we know that we cannot reach the final state without creating this hole and (ii) we know that we can close all such holes which have been created. Checking (i) is easy, since we just use the WR relation for this. Checking (ii) blindly would correspond to doing a DFS; however, we precompute this information and simply look it up, resulting in a constant time operation after the precomputation.

Precomputing hole information. Recall that a *hole* of stack i is a maximal sequence of the form $(\downarrow_i ws)^+$, where ws is a well-nested sequence and $\downarrow_i$ represents a push of stack i. A *hole segment* of stack i is a prefix of a hole of stack i, ending in a ws, while an *atomic hole segment* of stack i is just the segment of the form $\downarrow_i ws$. A *hole-segment* of stack i which starts from state s in the MPDA and ends in state s', can be represented by the triple (i, s, s'), that we call a *hole triple*. We compute the set HS_i of all hole triples (i, s, s') such that starting at s, there is a hole segment of stack i which ends at state s', as detailed in lines (5-9) of Algorithm 1. In doing so, we also compute the set AHS_i of all atomic hole segments of stack i and store them as tuples of the form (i, s_p, α, s_q) such that s_p and s_q are the MPDA states respectively at the left and right end points of an atomic hole segment of stack i, and α is the symbol pushed on stack i ($s_p \xrightarrow{\downarrow_i(\alpha)ws} s_q$).

A guided BFS exploration. We start with a list $\mu_0 = [s_0]$ consisting of the initial state and construct a BFS exploration tree whose nodes are lists of bounded length. A list is a sequence of states and hole triples representing a K-hole bounded run in a concise form. If H_i represents a hole triple for stack i, then a list is a sequence of the form $[s, H_i, H_j, H_k, H_i, \ldots, H_\ell, s']$. The simplest kind of list is a single state s. For example, a list with 3 holes of stacks i, j, k is $\mu = [s_0, (i, s, s'), (j, r, r'), (k, t, t'), t'']$. The hole triples (in red) denote open holes in the list. The maximum number of open holes in a list is bounded, making the length of the list also bounded. Let $\mathsf{last}(\mu)$ represent the last element of the list μ. This is always a state. For a node v storing list μ in the BFS tree, if $v_1, \ldots v_k$ are its children, then the corresponding lists $\mu_1, \ldots \mu_k$ are obtained by extending the list μ by one of the following operations:

1. **Extend μ with a hole**. Assume there is a hole of some stack i, which starts at $\mathsf{last}(\mu) = s$, and ends at s'. If the list at the parent node v is $\mu = [\ldots, s]$, then for all $(i, s, s') \in HS_i$, we obtain the list $\mathsf{trunc}(\mu) \cdot \mathsf{append}[(i, s, s'), s']$ at the child node (i.e., we remove the last element s of μ, then append to this list the hole triple (i, s, s'), followed by s').
2. **Extend μ with a pop**. Suppose there is a transition $t = (s_k, \uparrow_i(\alpha), a, s'_k)$ from $\mathsf{last}(\mu) = s_k$, where μ is of the form $[s_0, \ldots, (h, u, v), (i, s, s'), (j, t, t') \ldots, s_k]$, such that there is no hole triple of stack i after (i, s, s'), we extend the run by matching this pop (with its push). However, to obtain the last pending push of stack i corresponding to this hole, just HS_i information is not

enough since we also need to match the stack content. Instead, we check if we can split the hole (i, s, s') into (1) a hole triple $(i, s, s_a) \in HS_i$, and (2) a tuple $(i, s_a, \alpha, s') \in AHS_i$. If both (1) and (2) are possible, then the pop transition t corresponds to the last pending push of the hole (i, s, s'). t indeed matches the pending push recorded in the atomic hole (i, s_a, α, s') in μ, enabling the firing of transition t from the state s_k, reaching s'_k. In this case, we add the child node with the list μ' obtained from μ as follows. We replace (i) s_k with s'_k, and (ii) (i, s, s') with (i, s, s_a), respectively signifying firing of the transition t and the "shrinking" of the hole, by shifting the end point of the hole segment to the left. When we obtain the hole triple (i, s, s) (the start and end points of the hole segment coincide), we may have uncovered the last pending push and thereby "closed" the hole segment completely. At this point, we may choose to remove (i, s, s) from the list, obtaining $[s_0, \dots, (h, u, v),\ (j, t, t') \dots, s'_k]$. For every such $\mu' = [s_0, \dots, (h, u, v), (i, s, s_a), (j, t, t'), \dots, s'_k]$ and all $(s'_k, s_m) \in WS$ we also extend μ' to $\mu'' = [s_0, \dots, (h, u, v), (i, s, s_a), (j, t, t'), \dots, s_m]$. Notice that the size of the list in the child node obtained on a pop, is either the same as the list in the parent, or is smaller.

The number of lists is bounded since the number of states and the length of the lists are bounded. The BFS exploration tree will thus terminate. Combining the above steps gives us Algorithm 1, whose correctness gives us:

Theorem 1. *Given a* MPDA *and a positive integer K, Algorithm 1 terminates and answers "false" iff there exists a K-hole bounded accepting run of the* MPDA.

Complexity of the Algorithm. The maximum number of states of the system is $|\mathcal{S}|$. The time complexity of transitive closure is $\mathcal{O}(|\mathcal{S}|^3)$, using a Floyd-Warshall implementation. The time complexity of computing WellNestedReach which uses the transitive closure, is $\mathcal{O}(|\mathcal{S}|^5) + \mathcal{O}(|\mathcal{S}|^2 \times (|\Delta| \times |\mathcal{S}|))$. To compute AHS for n stacks the time complexity is $\mathcal{O}(n \times |\Delta| \times |\mathcal{S}|^2)$ and to compute HS for n stacks the complexity is $\mathcal{O}(n \times |\mathcal{S}|^2)$. For multistack systems, each list keeps track of (i) the number of hole segments($\leq K$), and (ii) information pertaining to holes (start, end points of holes, and which stack the hole corresponds to). In the worst case, this will be $(2K + 2)$ possible states in a list, as we are keeping the states at the start and end points of all the hole segments and a stack per hole. So, there are $\leq |\mathcal{S}|^{2K+3} \times n^{K+1}$ lists. In the worst case, when there is no K-hole bounded run, we may end up generating all possible lists for a given bound K on the hole segments. The time complexity is thus bounded above by $\mathcal{O}(|\mathcal{S}|^{2K+3} \times n^{K+1} + |\mathcal{S}|^5 + |\mathcal{S}|^3 \times |\Delta|)$.

Beyond Reachability. We can solve the usual safety questions in the (bounded-hole) underapproximate setting, by checking for underapproximate reachability on the product of the given system with the complement of the safe set. Given the way Algorithm 1 is designed, the fix-point algorithm allows us to go beyond reachability. In particular, we can solve several (increasingly difficult) variants of the repeated reachability problem, without much modification.

Consider the question : For a given state s and MPDA, does there exist a run ρ starting from s_0 which visits s infinitely often? This is decidable if we can decompose ρ into a finite prefix ρ_1 and an infinite suffix ρ_2 s.t. (1) both ρ_1, ρ_2 are well-nested, or (2) ρ_1 is K-hole bounded complete (all stacks empty), and ρ_2 is well-nested, or (3) ρ_1 is K-hole bounded, and $\rho_2 = (\rho_3)^\omega$, where ρ_3 is K-hole bounded. It is easy to see that (1) is solved by two calls to WellNestedReach and choosing non-empty runs. (2) is solved by a call to Algorithm 1, modified so that we reach s, and then calling WellNestedReach. Lastly, to solve (3), first modify Algorithm 1 to check reachability to s with possibly non-empty stacks. Then run the modified algorithm twice : first start from s_0 and reach s; second start from s and reach s again.

4 Generating a Witness

We next focus on the question of generating a witness for an accepting run when our algorithm guarantees non-emptiness. This question is important to address from the point of view of applicability: if our goal is to see if bad states are reachable, i.e., non-emptiness corresponds to presence of a bug, the witness run gives the trace of how the bug came about and hence points to what can be done to fix it (e.g., designing a controller). We remark that this question is difficult in general. While there are naive algorithms which can explore for the witness (thus also solving reachability), these do not use fix-point techniques and hence are not efficient. On the other hand, since we use fix-point computations to speed up our reachability algorithm, finding a witness, i.e., an explicit run witnessing reachability, becomes non-trivial. Generation of a witness in the case of well-nested runs is simpler than the case when the run has holes, and requires us to "unroll" pairs $(s_0, s_f) \in \mathtt{WR}$ recursively and generate the sequence of transitions responsible for (s_0, s_f).

Getting Witnesses from Holes. Now we move on to the more complicated case of behaviours having holes. Recall that in BFS exploration we start from the states reachable from s_0 by well-nested sequences, and explore subsequent states obtained either from (i) a hole creation, or (ii) a pop operation on a stack. Proceeding in this manner, if we reach a final configuration (say s_f), with all holes closed (which implies empty stacks), then we declare non-emptiness. To generate a witness, we start from the final state s_f reachable in the run (a leaf node in the BFS exploration tree) and *backtrack* on the BFS exploration tree till we reach the initial state s_0. This results in generating a witness run in the reverse, from the right to the left.

• Assume that the current node of the BFS tree was obtained using a pop operation. There are two possibilities to consider here (see below) depending on whether this pop operation closed or shrunk some hole. Recall that each hole has a left end point and a right end point and is of a specific stack i, depending on the pending pushes $\downarrow_i$ it has. So, if the MPDA has k stacks, then a list in the exploration tree can have k kinds of holes. The witness algorithm uses k stacks called *witness stacks* to correctly implement the backtracking procedure, to deal with k kinds of holes. Witness stacks should not be confused with the stacks of the MPDA.

• Assume that the current pop operation is closing a hole |🟩|🟪|🟥 of kind i as in Figure 2. This hole consists of the atomic holes |🟥, |🟪 and |🟩. The atomic hole |🟩 consists of the push | and the well-nested sequence 🟩 (same for the other two atomic holes). Searching among possible push transitions, we identify the matching push | associated with the current pop, resulting in closing the hole. On backtracking, this leads to a parent node with the atomic hole |🟩 having as left end point, the push |, and the right end point as the target of the ws 🟩. We push onto the witness stack i, a barrier (a delimiter symbol #) followed by the matching push transition | and then the ws, 🟩. The barrier segregates the contents of the witness stack when we have two pop transitions of the same stack in the reverse run, closing/shrinking two different holes.

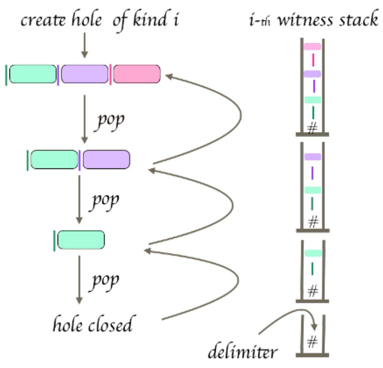


Figure 2. Backtracking to spit out the hole |🟩|🟪|🟥 in reverse. The transitions of the atomic hole |🟥 are first written in the reverse order, followed by those of |🟪 in reverse, and then of |🟩 in reverse.

• Assume that the current pop operation is shrinking a hole of kind i. The list at the present node has this hole, and its parent will have a larger hole (see Figure 2, where the parent node of |🟩 has |🟩 |🟪). As in the case above, we first identify the matching push transition, and check if it agrees with the push in the last atomic hole segment in the parent. If so, we populate the witness stack i with the rightmost atomic hole segment of the parent node (see Figure 2, |🟪 is populated in the stack). Each time we find a pop on backtracking the exploration tree, we find the rightmost atomic hole segment of the parent node, and keep pushing it on the stack, until we reach the node which is obtained as a result of a hole creation. Now we have completely recovered the entire hole information by backtracking, and fill the witness stack with the reversed atomic hole segments which constituted this hole. Notice that when we finish processing a hole of kind i, then the witness stack i has the hole reversed inside it, followed by a barrier. The next hole of the same kind i will be treated in the same manner.

• If the current node of the BFS tree is obtained by creating a hole of kind i in the fix-point algorithm, then we pop the contents of witness stack i till we reach a barrier. This spits out the atomic hole segments of the hole from the right to the left, giving us a sequence of push transitions, and the respective ws in between. The transitions constituting the ws are retrieved and added. Notice that popping the witness stack i till a barrier spits out the sequence of transitions in the correct reverse order while backtracking.

5 Adding Time to Multi-pushdown systems

In this section, we briefly describe how the algorithms described in section 3 can be extended to work in the timed setting. Due to lack of space, we focus on some of the significant challenges and advances, leaving the formal details and algorithms to the supplement [4]. A TMPDA extends a MPDA $\mathcal{S}$ with a set $\mathcal{X}$ of clock variables. Transitions check constraints which are conjunctions/disjunctions of constraints (called closed guards in the literature) of the form $x \leq c$ or $x \geq c$ for $c \in \mathbb{N}$ and x any clock from $\mathcal{X}$. Symbols pushed on stacks "age" with time elapse; that os, they store the time elapsed since they were pushed onto the stack. A pop is successful only when the age of the symbol lies within a certain interval. The acceptance condition is as in the case of MPDA.

The first main challenge in adapting the algorithms in section 3 to the timed setting was to take care of all possible time elapses along with the operations defined in Algorithm 1. The usage of closed guards in TMPDA means that it suffices to explore all runs with integral time elapses (for a proof see e.g., Lemma 4.1 in [5]). Thus configurations are pairs of states with valuations that are vectors of non-negative integers, each of which is bounded by the maximal constant in the system. Now, to check reachability we need to extend all the precomputations (transitive closure, well-nested reachability, as well as atomic and non-atomic hole segments) with the time elapse information. To do this, we use a weighted version of the Floyd-Warshall algorithm by storing time elapses during precomputations. This allows us to use this precomputed *timed* well-nested reachability information while performing the BFS tree exploration, thus ensuring that any explored state is indeed reachable by a timed run. In doing so, the most challenging part is extending the BFS tree wrt a pop. Here, we not only have to find a split of a hole into an atomic hole-segment and a hole-segment as in Algorithm 1, but also need to keep track of possible partitions of time, making the algorithm quite challenging.

Timed Witness: As in the untimed case, we generate a witness certifying non-emptiness of TMPDA. But, producing a witness for the fix-point computation as discussed earlier requires unrolling. The fix-point computation generates a pre-computed set `WRT` of tuples $((s, \nu), t, (s', \nu'))$, where s, s' are states t is time elapsed in the well-nested sequence and $\nu, \nu' \in \mathbb{N}^{|\mathcal{X}|}$ are integral valuations, i.e., integer values taken by clocks. This set of tuples does not have information about the intermediate transitions and time-elapses. To handle this, using the pre-computed information, we define a lexicographic progress measure which ensures termination of this search. The main idea is as follows: the first progress measure is to check if there a time-elapse t transition possible between (s, ν) and (s', ν') and if so, we print this out. If not, $\nu' \neq \nu + t$, and some set of clocks have been reset in the transition(s) from (s, ν) to (s', ν'). The second progress measure looks at the sequence of transitions from (s, ν) to (s', ν'), consisting of reset transitions (at most the number of clocks) that result in ν' from ν. If neither the first nor the second progress measure apply, then $\nu = \nu'$, and we are left to explore the last progress measure, by exploring at most $|\mathcal{S}|$ number of transitions from (s, ν) to (s', ν'). Using this progress measure, we can seamlessly extend the witness generation to the timed setting. The challenges involved therein, can be seen in the full version [4].

6 Implementation and Experiments

We implemented a tool BHIM (**B**ounded **H**oles **I**n **M**PDA) in C++ based on Algorithm 1, which takes an MPDA and a constant K as input and returns *True* iff there exists a K-hole bounded run from the start state to an accepting state of the MPDA. In case there is such an accepting run, BHIM generates one such, with minimal number of holes. For a given hole bound K, BHIM first tries to produce a witness with 0 holes, and iteratively tries to obtain a witness by increasing the bound on holes till K. In most cases, BHIM found the witness before reaching the bound K. Whenever BHIM's witness had K holes, it is guaranteed that there are no witnesses with a smaller number of holes.

To evaluate the performance of BHIM, we looked at some available benchmarks and modeled them as MPDA. We also added timing constraints to some examples such that they can be modeled as TMPDA. Our tests were run on a GNU/Linux system with Intel® Core^{TM} i7–4770K CPU @ 3.50GHz, and 16GB of RAM. Details of all examples here, as well as an additional example of a linux kernel bug can be found [4].

• **Bluetooth Driver [25]**. The Bluetooth device driver example [25], has an arbitrary number of threads, working with a shared memory. We model this using a 2-stack pushdown system, where a system state represents the current valuation of the global variables, and the stacks are used to maintain the call-return between different functions, as well as to keep track of context switches between threads. A known error as pointed out in [25] is a race condition between two threads where one thread tries to write to a global variable and the other thread tries to read from it. BHIM found this error, with a well-nested witness. A timed extension of this example was also considered, where, a witness was obtained again with hole bound 0.

• **Bluetooth Driver v2** [11,23]. A modified version of Bluetooth driver is considered [11,23], where a counter is maintained to count the number of threads actively using the driver. We model this with a A two stack MPDA. With a well-nested witness, BHIM found the error of interrupted I/O, where the stopping thread kills the driver while the other thread is busy with I/O.

• **A Multi-threaded Producer Consumer Problem**. The producer consumer problem (see e.g., [26]) is a classic example of concurrency and synchronization. An interesting scenario is when there are multiple producers and consumers. Assume that two ingredients called 'A' and 'B' are produced in a production line in batches (of M and N respectively). These parameters M and N are fixed for each day but may vary across days. There is another consumer machine that (1) consumes one unit of 'A' and one unit of 'B' in that order; (2) repeats this process until all ingredients are consumed. In between if one of the ingredients runs out, then we non-deterministically produce more batches of the ingredient and then continue. To avoid wastage the factory aims to consume all ingredients produced in a day, hence the problem of interest is to check if all A's and B's produced in a day are consumed. We can model this factory using a two-stack pushdown system, one stack per product, A, B, where the sizes of the batches, $M > 0$ and $N > 0$ respectively, are parameters. The production

Name	Locations	Transitions	Stacks	Holes	Time Empty (mili sec)	Time Witness (mili sec)	Memory(KB)
Bluetooth	45	89	2	0	149.3	0.241	6934
Bluetooth v2	47	134	2	0	92.2	0.176	5632
MultiProdCons(3,2)	7	11	2	2	126.529	0.281	5632
MultiProdCons(24,7)	32	34	2	2	1879.33	10.63	21836
Binary Search Tree	29	78	2	2	60.8	5.1	5143
untimed-L^{crit}	6	10	2	2	14.9	0.7	4692
untimed-Maze	9	12	2	0	8.25	0.07	5558
L^{bh} (from Sec. 2.1)	7	13	2	2	22.2	0.6	4404

Table 1. Experimental results: Time Empty and Time Witness column represents no. of milliseconds needed for emptiness checking and to generate witness respectively.

and consumption of the 'A's and 'B's are modeled using push and pop in the respective stack. For a given M and N, the language accepted by the system is non-empty iff there is a run where all the produced 'A's and 'B's are consumed. The language accepted by the two-stack pushdown system is given by $L_{M,N} = ((a^M + b^N)^+(\bar{a}\bar{b})^+)^+$, where a, b represent respectively, the push on stack 1, 2 and $\bar{a}, \bar{b}$ represent the pop on stack 1, 2 and hence must happen equal number of times.

For any $M, N > 0$, any accepting run of the two stack pushdown system cannot be well-nested. Further, in an accepting run, the minimum number of items produced (and hence its length) must be a multiple of $LCM(M, N)$. As the consumption of 'A's and 'B's happen in an order one by one i.e., in a sequence where consumption of 'A' and 'B' alternate, the minimum number of context changes (and the scope bound) required in an accepting run depends on M and N (in fact it is $O(2 \times LCM(M, N))$. On the other hand, the shortest accepting run is 2-hole bounded: at any position of the word, the open holes come from the unmatched sequences of a and b seen so far. Thus for any $M, N{>}0$, BHIM was able to check for non-emptiness of $L_{M,N}$ with a witness of hole bound 2.

• **Critical time constraints** [9]. This is one of the timed examples, where we consider the language $L^{crit} = \{a^y b^z c^y d^z \mid y, z \geq 1\}$ with time constraints between occurrences of symbols. The first c must appear after 1 time-unit of the last a, the first d must appear within 3 time-units of the last b, and the last b must appear within 2 time units from the start, and the last d must appear at 4 time units. L^{crit} is accepted by a TMPDA with two timed stacks. L^{crit} has no well-nested word, is 4-context bounded, but only 2 hole-bounded.

• **Concurrent Insertions in Binary Search Trees**. Concurrent insertions in binary search trees is a very important problem in database management systems. [17,11] proposes an algorithm to solve this problem for concurrent implementations. However, incorrect implementation of locks allows a thread to overwrite others. We modified the algorithm [17] to capture this bug, and modeled it as MPDA. BHIM found the bug with a witness of hole-bound 2.

• **Maze Example**. Finally we consider a robot navigating a maze, picking items; an extended (from single to multiple stack) version of the example from [6]. In the untimed setting, a witness for non-emptiness was obtained with hole-bound 0, while in the extension with time, the witness had a hole-bound 2.

Name	Locations	Transitions	Stacks	Clocks	$c_{\max}$	Aged(Y/N)	Holes	Time Empty(mili sec)	Time Witness (mili sec)	Memory(KB)
Bluetooth	45	89	2	0	2	Y	0	152.8	0.119	5568
L^{crit}	6	10	2	2	8	Y	2	9965.2	3.7	203396
Maze	9	12	2	2	5	Y	2	349.3	0.31	11604

Table 2. Experimental results of timed examples. The column $c_{\max}$ is defined as the maximum constant in the automaton, and Aged denotes if the stack is timed or not

Results and Discussion. The performance of BHIM is presented in Table 1 for untimed examples and in Table 2 for timed examples.

Apart from the results in the tables, to check the robustness of BHIM wrt parameters like the number of locations, transitions, stacks, holes and clocks (for TMPDA), we looked at examples with an empty language, by making accepting states non-accepting in the examples considered so far. This forces BHIM to explore all possible paths in the BFS tree, generating the lists at all nodes. The scalability of BHIM wrt all these parameters are in [4].

BHIM Vs. State of the art. What makes BHIM stand apart wrt the existing state of the art tools is that (i) none of the existing tools handle underapproximations captured by bounded holes, (ii) none of the existing tools work with multiple stacks in the timed setting (even closed guards!). The state of the art research in underapproximations wrt untimed multistack pushdown systems has produced some robust tools like GetaFix which handles multi-threaded programs with bounded context switches. While we have adapted some of the examples from GetaFix, the latest available version of GetaFix has some issues in handling those examples[3]. Likewise, SPADE, MAGIC and the counter implementation [16] are currently not maintained, resulting in a non-comparison of BHIM and these tools. Most examples handled by BHIM correspond to non-context bounded, or non-scope bounded, or timed languages which are beyond GetaFix : the 2-hole bounded witness found by BHIM for the language $L_{9,5}$ for the multi producer consumer case cannot be found by GetaFix/MAGIC/SPADE with less than 90 context switches. In the timed setting, the Maze example which has a 2 hole-bounded witness where the robot visits certain locations equal number of times is beyond [6], which can handle only single stack.

7 Future Work

As immediate future work, we are working on BHIM **v2** to be symbolic, inspired from GetaFix. The current avatar of BHIM showcases the efficiency of fix-point techniques extended to larger bounded underapproximations; indeed going symbolic will make BHIM much more robust and scalable. This version will also include a parser to handle boolean programs, allowing us to evaluate larger repositories of available benchmarks.

Acknowledgements. We would like to thank Gennaro Parlato for the discussions on GetaFix and for providing us benchmarks and anonymous reviewers for more pointers.

[3] we did get in touch with one of the authors, who confirmed this.

References

1. Abdulla, P.A., Atig, M.F., Stenman, J.: Dense-timed pushdown automata. In: Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012. p. 35–44 (2012), https://doi.org/10.1109/LICS.2012.15
2. Abdulla, P.A., Atig, M.F., Stenman, J.: The minimal cost reachability problem in priced timed pushdown systems. In: Language and Automata Theory and Applications - 6th International Conference, LATA 2012, A Coruña, Spain, March 5-9, 2012. Proceedings. pp. 58–69 (2012), https://doi.org/10.1007/978-3-642-28332-1_6
3. Akshay, S., Gastin, P., Jugé, V., Krishna, S.N.: Timed systems through the lens of logic. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–13 (2019)
4. Akshay, S., Gastin, P., Krishna, S., Roychowdhury, S.: Revisiting underapproximate reachability for multipushdown systems (2020), https://arxiv.org/abs/2002.05950
5. Akshay, S., Gastin, P., Krishna, S.N.: Analyzing Timed Systems Using Tree Automata. Logical Methods in Computer Science **Volume 14, Issue 2** (May 2018), https://lmcs.episciences.org/4489
6. Akshay, S., Gastin, P., Krishna, S.N., Sarkar, I.: Towards an efficient tree automata based technique for timed systems. In: 28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany. pp. 39:1–39:15 (2017), https://doi.org/10.4230/LIPIcs.CONCUR.2017.39
7. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing. pp. 202–211. ACM (2004)
8. Atig, M.F.: Model-Checking of Ordered Multi-Pushdown Automata. Logical Methods in Computer Science **Volume 8, Issue 3** (Sep 2012). https://doi.org/10.2168/LMCS-8(3:20)2012
9. Bhave, D., Dave, V., Krishna, S.N., Phawade, R., Trivedi, A.: A perfect class of context-sensitive timed languages. In: International Conference on Developments in Language Theory. pp. 38–50. Springer, Berlin, Heidelberg (2016)
10. Bouajjani, A., Echahed, R., Robbana, R.: On the automatic verification of systems with continuous variables and unbounded discrete data structures. In: International Hybrid Systems Workshop. pp. 64–85. Springer (1994)
11. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 334–349. Springer (2006)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
13. Cyriac, A.: Verification of communicating recursive programs via split-width. (Vérification de programmes récursifs et communicants via split-width). Ph.D. thesis, École normale supérieure de Cachan, France (2014), https://tel.archives-ouvertes.fr/tel-01015561
14. Cyriac, A., Gastin, P., Kumar, K.N.: MSO decidability of multi-pushdown systems via split-width. In: International Conference on Concurrency Theory. pp. 547–561. Springer, Berlin, Heidelberg (2012)
15. Dang, Z., Ibarra, O.H., Bultan, T., Kemmerer, R.A., Su, J.: Binary reachability analysis of discrete pushdown timed automata. In: International Conference on Computer Aided Verification. p. 69–84. Springer (2000)

16. Hague, M., Lin, A.W.: Synchronisation- and reversal-bounded analysis of multi-threaded programs with counters. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. p. 260–276 (2012), https://doi.org/10.1007/978-3-642-31424-7_22
17. Kung, H., Lehman, P.L.: Concurrent manipulation of binary search trees. ACM Transactions on Database Systems (TODS) **5**(3), 354–382 (1980)
18. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on. pp. 161–170. IEEE (2007)
19. La Torre, S., Madhusudan, P., Parlato, G.: The language theory of bounded context-switching. In: Latin American Symposium on Theoretical Informatics. pp. 96–107. Springer (2010)
20. La Torre, S., Napoli, M.: Reachability of multistack pushdown systems with scope-bounded matching relations. In: International Conference on Concurrency Theory. p. 203–218. Springer (2011)
21. La Torre, S., Parthasarathy, M., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. ACM Sigplan Notices **44**(6), 211–222 (2009)
22. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: ACM SIGPLAN Notices. vol. 46, pp. 283–294. ACM (2011)
23. Patin, G., Sighireanu, M., Touili, T.: Spade: Verification of multithreaded dynamic and recursive programs. In: International Conference on Computer Aided Verification. pp. 254–257. Springer (2007)
24. Qadeer, S.: The case for context-bounded verification of concurrent programs. In: Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings. pp. 3–6 (2008), https://doi.org/10.1007/978-3-540-85114-1_2
25. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. ACM sigplan notices **39**(6), 14–24 (2004)
26. Silberschatz, A., Gagne, G., Galvin, P.B.: Operating system concepts. Wiley (2018)
27. Torre, S.L., Napoli, M., Parlato, G.: Scope-bounded pushdown languages. International Journal of Foundations of Computer Science **27**(02), 215–233 (2016)
28. Torre, S.L., Parlato, G.: Scope-bounded Multistack Pushdown Systems: Fixed-Point, Sequentialization, and Tree-Width **18**, 173–184 (2012). https://doi.org/10.4230/LIPIcs.FSTTCS.2012.173

KReach: A Tool for Reachability in Petri Nets*

Alex Dixon and Ranko Lazić

Department of Computer Science, University of Warwick
Coventry, United Kingdom
{alexander.dixon,r.s.lazic}@warwick.ac.uk

Abstract. We present KReach, a tool for deciding reachability in general Petri nets. The tool is a full implementation of Kosaraju's original 1982 decision procedure for reachability in VASS. We believe this to be the first implementation of its kind. We include a comprehensive suite of libraries for development with Vector Addition Systems (with States) in the Haskell programming language. KReach serves as a practical tool, and acts as an effective teaching aid for the theory behind the algorithm. Preliminary tests suggest that there are some classes of Petri nets for which we can quickly show unreachability. In particular, using KReach for coverability problems, by reduction to reachability, is competitive even against state-of-the-art coverability checkers.

Keywords: Petri Nets · Kosaraju's Algorithm · Reachability · Vector Addition Systems · Coverability

1 Introduction

Petri nets [26] (equivalently, Vector Addition Systems with States [12,14]) are one of the best-known formalisms in concurrency theory. They form a highly expressive model which is applicable in a broad range of domains including software and hardware verification [5,6], chemical modelling [3], and business processes [22]. Two of the most studied decision problems on Petri nets are those of *coverability* and *reachability*.

Coverability is the central decision problem for verifying safety properties on Petri nets. The coverability problem asks, given a starting configuration $\mathbf{m_0}$ and a target $\mathbf{m}$, whether we can reach, by some sequence of valid transitions (i.e. by a *run*), any configuration $\mathbf{m'} \geq \mathbf{m}$. The problem is known to be EXPSPACE-complete [23,27]. Coverability has seen considerable study in recent years, in particular with a view towards minimising the running time of coverability decision procedures [2,11].

The reachability problem, to which coverability is easily reducible, can capture both safety and liveness properties of systems [13]. Formally, the reachability problem asks if we can, by some run, get from a starting configuration $\mathbf{m_0}$ to

* Supported by the Centre for Doctoral Training in Urban Science and Progress (EPSRC Grant EP/L016400/1).

the target configuration **m** exactly. Historically, there has been a wide gap between the upper and lower bounds, but recently these have been improved to a non-elementary lower bound [7] and an Ackermannian upper bound [21].

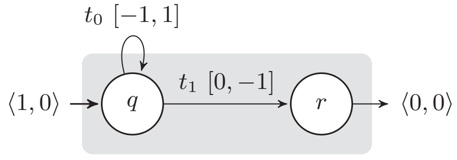
The first complete algorithm for reachability is due to Mayr [24], since further developed and simplified by Kosaraju [17] and Lambert [18]. More recently, a strikingly simple but not yet practical algorithm was obtained by Leroux [20], based on enumerating Presburger-definable invariants. In this paper we will focus on Kosaraju's algorithm. The latter is the subject of an entire book by Reutenauer [29], since translated into English [28], and more recently presented in a novel, readable format with contemporary notation by Lasota [19].

In spite of these substantial and sustained theoretical developments, the area has seen little in the way of practical implementation. We seek to address this gap by making the following contributions:

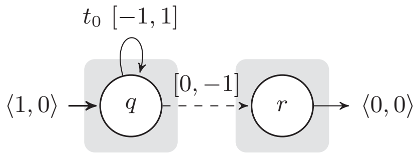
- We present a tool, KReach, which we believe to be the first complete implementation of a decision procedure for the general Petri net reachability problem.
- Noting that the reachability problem is infamous for its complexity—both in terms of its worst-case runtime, and the impenetrability of its decision procedures for newcomers—we offer an accessible implementation of Kosaraju's algorithm, which can be used as a detailed learning aid.
- We have designed the implementation in a modular and extensible way which is conducive to development of future improvements to the algorithm.
- We include a number of parameterised example nets which demonstrate correctness and performance, and can be used to assess further work.
- We provide a full suite of libraries which aid programming with Vector Addition Systems (with States) in the Haskell programming language.

2 Design and Implementation

Algorithm We will now give a brief overview of Kosaraju's classic reachability algorithm for Petri nets, and explain the translation into code. Note that Kosaraju's algorithm operates over VASS—the translation between the two is immediate, and can be performed while parsing the problem instance.



(a) We denote ⟨constraints⟩ and [transitions]. The states of our original VASS are q and r; the transitions are t_0 and t_1. Here we are testing reachability from $q(1, 0)$ to $r(0, 0)$.



(b) $\mathcal{G}_{\text{ex}}$ after SCC decomposition. We now have two components, the second of which is trivial. The adjoinment is marked by a dashed arrow. The shaded rectangles are separate components.

Fig. 1: A simple GVASS, $\mathcal{G}_{\text{ex}}$.

The procedure revolves around *Generalized VASS (GVASS)*, an extension of VASS. A GVASS $\mathcal{G}$ is a sequence $(C_1, .., C_n)$, of VASSs annotated with metadata, most notably constraints on their entry and exit configurations. The exit state of C_i is adjoined to the entry state of C_{i+1} by a transition. Reachability in our original VASS V is implied by reachability in an induced GVASS $\mathcal{G}(V)$, a singleton sequence where we constrain the entry and exit configurations as being equal to our initial $\mathbf{m_0}$ and target $\mathbf{m}$. Figure 1a gives an example.

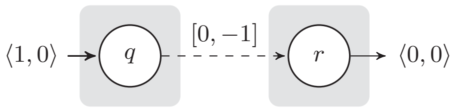
In outline, Kosaraju's algorithm operates as follows. At each step, given a strongly-connected GVASS $\mathcal{G}$:

– either $\mathcal{G}$ fulfils θ;
– or $\mathcal{G}$ violates θ, in which case we can *refine* $\mathcal{G}$ into a finite (possibly empty) set of modified GVASSs.

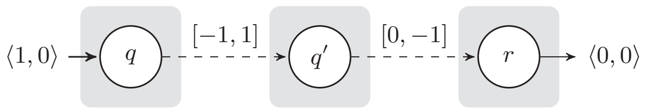
This produces a finitely-branching tree of GVASSs in which every branch forms a strictly descending chain with respect to a well-quasi-ordering [21]. The algorithm therefore always terminates, and $\mathbf{m}$ is reachable from $\mathbf{m_0}$ in the root GVASS $\mathcal{G}$ if and only if some GVASS in the tree satisfies θ.

The $\boldsymbol{\theta}$ Condition Kosaraju's main predicate comprises two parts. θ_1 is a global property of the system, while θ_2 must hold for each component.

θ_1 : There exist *pseudo-runs* through the GVASS which use every edge in every component unboundedly many times, and attain unboundedly large values for every unconstrained coordinate (here a *pseudo-run* is a run over $\mathbb{Z}^d$ rather than over $\mathbb{N}^d$). This condition can be formulated as an integer linear programming problem. From this we obtain a semilinear set of vectors of variables representing counts of transition occurrences and values of unconstrained coordinates. If there is no bounded variable then θ_1 holds. Otherwise, one such bounded variable is "refined" by either constraining the associated coordinate's value or by unfolding the associated transition. For example, we may deduce that the number of firings of t_0 never exceeds 1 in our example GVASS $\mathcal{G}_{\text{ex}}$; we generate the refinements as in Figure 2.



(a) Refinement when t_0 is activated 0 times.



(b) Refinement when t_0 is activated 1 time.

Fig. 2: Refinement of $\mathcal{G}_{\text{ex}}$ by removing bounded transition t_0.

θ_2 : Each non-trivial component of the GVASS contains some path from the initial to the final state, via which all unconstrained coordinates are increased. The same must also hold if the component's arcs are reversed. We can evaluate θ_2 using standard algorithms which compute the coverability set. If θ_2 fails, then some coordinate is bounded everywhere in the state space of the component; such a coordinate is refined by removing it entirely (making it *rigid*) and encoding its possible values into the component's states.

Solving Coverability We are able to reduce from the coverability problem to the reachability problem in the following way. Suppose we are intending to *cover* some vector $\mathbf{m}$—that is, we wish to reach any vector $\mathbf{m}'$ such that $\mathbf{m}' \geq \mathbf{m}$. We introduce a new state $\varDelta$, and add a transition δ from the final state of the original VASS to $\varDelta$, which subtracts $\mathbf{m}$ on activation. As $\varDelta$ can only be reached by subtracting $\mathbf{m}$ from our current vector, reaching a vector $\mathbf{m}' \geq \mathbf{m}$ (i.e. covering $\mathbf{m}$) is equivalent to reaching state $\varDelta$. For each vector coordinate we introduce a looping transition on $\varDelta$ which reduces the value of that coordinate by 1. This ensures that $(0, ..., 0)$ can be reached from any configuration in state $\varDelta$. As a result, covering $\mathbf{m}$ in the original net is equivalent to reaching $(\varDelta, (0, ..., 0))$ in the augmented version.

Implementation KReach is implemented in the Haskell programming language. This is a strongly-typed, functional language with lazy evaluation. The language was chosen for its high level of expressiveness, type-safety, and the ease of translation between algorithm and implementation.

We represent the algorithm as a function which takes a list of GVASSs, and returns a `KosrajuResult`. We perform a depth-first search of the refinement tree, either finding a refinement which permits reachability (`KosarajuHolds`) or exhausting all possibilities (`KosarajuDoesNotHold`). The algorithm is guaranteed to terminate [17], and so constitutes a full decision procedure for reachability.

The ILP subproblem (θ_1) is solved with the `SBV` (SMT Based Verification) package, an interface to a variety of SMT solvers. We formulate all the constraints as an integer linear program, and evaluate with the `ldn` function (Linear Diophantine equations over Naturals).

The coverability subproblem (θ_2) is solved by an implementation of the standard Karp-Miller algorithm for Vector Addition Systems [16]. This algorithm computes the *coverability set*—the upward closure of the set of all vectors that are reachable in a net from some starting vector. The extensible nature of the code allows the basic implementation to be swapped out for a more optimised one (e.g. based on [10]) at a later stage.

We ensure that the strongly connected property holds by decomposing the original GVASS via the SCC implementation found in the `Data.Graph` module.

Optimisations In spite of the ominous non-elementary complexity lower bound, some effort was still undertaken to improve the runtime of test cases. A number of minor improvements have been made over the standard algorithm which remove unneccessary computations.

For example, when constructing refinements for a GVASS $\mathcal{G}$, when a variable is bounded above by some constant c, Kosaraju suggests to generate refinements $R_i(\mathcal{G})$ for every i from $\{0, \ldots, c\}$. Instead, we refine only to $R_i(\mathcal{G})$ for values i that feature in the corresponding semilinear set.

The algorithm has also been multithreaded with Haskell's lightweight concurrency toolkit [1], so that it evaluates refinements in parallel rather than sequentially. Any return value of `KosarajuHolds` will terminate the program.

The program uses the `vass` library (released as part of this publication) to parse file formats. By default a parser for MIST's `.spec` format[1] is provided. This format is traditionally a representation of coverability problems; KReach translates these to reachability problems by replacing $p \geq n$ constraints by $p = n$ in target places.

3 Installation and Usage

Installation The KReach tool is available from a public GitHub repository. One can clone the repository in full with the following command:

```
git clone https://github.com/dixonary/kosaraju.git
```

The program is built against the Haskell `stack` toolchain[2]. In order to build the tool, a locally installed version of `stack` is required. The tool can be compiled and locally installed by running `stack install` in the cloned directory. One must also ensure that an SMT solver is installed and accessible on the user's binary path; `z3`[3] and `cvc4`[4] are supported. A compiled program binary, along with benchmarks, is provided on the "Releases" section of the GitHub page.

Usage The compiled `kosaraju` tool can be interacted with through the command line. Simple wrapper scripts are provided; the standard invocation is `kreach FILENAME` to check reachability, and `kcover FILENAME` for coverability. Intermediate output can be hidden by providing the `-q` (quiet) flag.

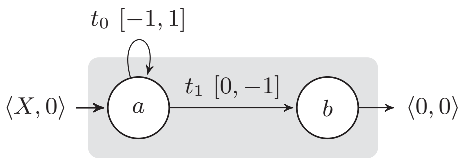
Figure 3b shows the relative performance of `z3` against `cvc4` for growing inputs. `cvc4` tends to far outperform `z3` on the constructed ILP problems.

4 Experimental Results

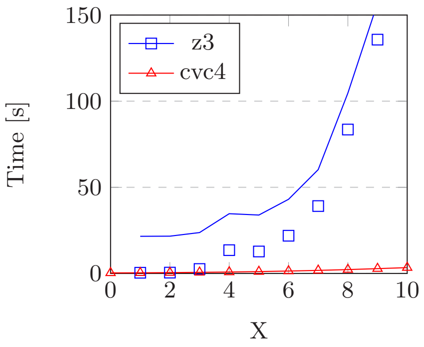
KCover allows us to use benchmarks for the coverability problem as a source of test cases for the reachability algorithm. The suite provided with the tool includes also a number of test cases for various aspects of the implementation, as well as examples from the non-elementary lower bound construction [7].

`KReach` was evaluated against many problems and solvers from the literature on coverability. `QCover` [4] implements coverability based on relaxation to continuous coverability; `ICover` [11] refines this further with inductive invariants.

[1] https://github.com/pierreganty/mist/wiki
[2] https://docs.haskellstack.org/en/stable/README/
[3] https://github.com/Z3Prover/z3
[4] https://cvc4.github.io



(a) The parameterized version of our original sample case $\mathcal{G}_{\text{ex}}$, notated $\mathcal{G}_{\text{ex}}(X)$.



(b) Time against Parameter X for $\mathcal{G}_{\text{ex}}(X)$ with the supported solvers.

Table 1 includes some specific instances which are representative of the broader trends in experimental results. On many safe cases, such as `Kanban` and `Bingham`, KReach is able to determine safety faster than state of the art coverability solvers by finding zero valid refinements (terminating the search immediately). On some safe nets such as `Manufacturing`, KReach cannot immediately rule out coverability in this way, and the refinement tree must be explored. The `Bug_Tracking` examples induced intractably large ILP problems. Unsafe cases such as `PNCSACover` induced large refinement trees, which were unable to be explored fully within the time limit.

Instance	Outcome	MIST (s)	Qcover (s)	Icover (s)	**KReach** (s)
Kanban	safe	404	TLE	TLE	1
Bingham_h150	safe	TLE	TLE	TLE	533
Manufacturing	safe	1	0	0	4
Bug_Tracking_x0	safe	MLE	13	33	TLE
PNCSACover	unsafe	3	27	59	TLE

Table 1: Sample of test cases. All results were computed on consumer hardware. MLE = Memory Limit Exceeded (4GB); TLE = Time Limit Exceeded (1 hour).

5 Concluding Remarks

The experimental results suggest that KReach may be a fruitful source of static invariants for ruling out coverability on some classes of Petri nets. One line of further work may be to attempt to formally classify those nets for which Kosaraju's algorithm is effective in practice.

Further work may also include optimisations based on the novel theoretical developments in the Ackermannian upper bound proof [21], and building parsers to enable experiments on instances of problems that are known to reduce to reachability in Petri nets (e.g., in logic [15,8], concurrent systems [9] or process calculi [25]).

Data Availability Statement The data analyzed here are available in the Figshare data repository: https://doi.org/10.6084/m9.figshare.11887956

References

1. Control.Concurrent - Haskell Package Database. http://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Concurrent.html
2. Mist - a safety checker for Petri nets (and monotonic extensions). https://github.com/pierreganty/mist
3. Angeli, D., De Leenheer, P., Sontag, E.D.: Persistence results for chemical reaction networks with time-dependent kinetics and no global conservation laws. SIAM Journal on Applied Mathematics **71**(1), 128–146 (2011). https://doi.org/10.1137/090779401
4. Blondin, M., Finkel, A., Haase, C., Haddad, S.: Approaching the coverability problem continuously. In: TACAS. LNCS, vol. 9636, pp. 480–496. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_28
5. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. ACM Trans. Program. Lang. Syst. **35**(3) (Nov 2013). https://doi.org/10.1145/2518188
6. Burns, F., Koelmans, A., Yakovlev, A.: WCET analysis of Superscalar Processors using Simulation with coloured Petri nets. Real-Time Systems **18**, 275–288 (2000). https://doi.org/10.1023/A:1008101416758
7. Czerwinski, W., Lasota, S., Lazic, R., Leroux, J., Mazowiecki, F.: The reachability problem for Petri nets is not elementary. In: STOC. pp. 24–33. ACM (2019). https://doi.org/10.1145/3313276.3316369
8. Demri, S., Figueira, D., Praveen, M.: Reasoning about data repetitions with counter systems. Logical Methods in Computer Science **12**(3) (2016). https://doi.org/10.2168/LMCS-12(3:1)2016
9. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Verification of population protocols. Acta Inf. **54**(2), 191–215 (2017). https://doi.org/10.1007/s00236-016-0272-3
10. Finkel, A., Haddad, S., Khmelnitsky, I.: Minimal coverability tree construction made complete and efficient. In: FoSSaCS (2020), https://hal.inria.fr/hal-02479879
11. 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
12. Greibach, S.A.: Remarks on blind and partially blind one-way multicounter machines. Theor. Comput. Sci. **7**, 311–324 (1978). https://doi.org/10.1016/0304-3975(78)90020-8
13. Hack, M.: The recursive equivalence of the reachability problem and the liveness problem for petri nets and vector addition systems. In: 15th Annual Symposium on Switching and Automata Theory (swat 1974). pp. 156–164 (Oct 1974). https://doi.org/10.1109/SWAT.1974.28
14. Hopcroft, J.E., Pansiot, J.: On the reachability problem for 5-dimensional vector addition systems. Theor. Comput. Sci. **8**, 135–159 (1979). https://doi.org/10.1016/0304-3975(79)90041-0
15. Kanovich, M.I.: Petri nets, Horn programs, linear logic and vector games. Ann. Pure Appl. Logic **75**(1–2), 107–135 (1995). https://doi.org/10.1016/0168-0072(94)00060-G
16. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. **3**(2), 147–195 (1969). https://doi.org/10.1016/S0022-0000(69)80011-5
17. Kosaraju, S.R.: Decidability of reachability in vector addition systems (preliminary version). In: STOC. pp. 267–281. ACM (1982). https://doi.org/10.1145/800070.802201

18. Lambert, J.: A structure to decide reachability in Petri nets. Theor. Comput. Sci. **99**(1), 79–104 (1992). https://doi.org/10.1016/0304-3975(92)90173-D
19. Lasota, S.: VASS reachability in three steps. CoRR **abs/1812.11966** (2018), http://arxiv.org/abs/1812.11966
20. Leroux, J.: The general vector addition system reachability problem by Presburger inductive invariants. Logical Methods in Computer Science **6**(3) (2010). https://doi.org/10.2168/LMCS-6(3:22)2010
21. Leroux, J., Schmitz, S.: Reachability in vector addition systems is primitive-recursive in fixed dimension. In: LICS. pp. 1–13. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785796
22. Li, Y., Deutsch, A., Vianu, V.: VERIFAS: A practical verifier for artifact systems. PVLDB **11**(3), 283–296 (2017). https://doi.org/10.14778/3157794.3157798
23. Lipton, R.J.: The reachability problem requires exponential space. Tech. Rep. 62, Yale University (1976), http://cpsc.yale.edu/sites/default/files/files/tr63.pdf
24. Mayr, E.W.: An algorithm for the general petri net reachability problem. SIAM J. Comput. **13**(3), 441–460 (1984). https://doi.org/10.1137/0213029
25. Meyer, R.: A theory of structural stationarity in the *pi*-calculus. Acta Inf. **46**(2), 87–137 (2009). https://doi.org/10.1007/s00236-009-0091-x
26. Petri, C.A.: Kommunikation mit Automaten. http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/ (1962), PhD thesis, Universität Hamburg
27. 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
28. Reutenauer, C.: The mathematics of Petri nets. Prentice Hall (1990), translated by Ian Craig
29. Reutenauer, C.: Aspects Mathématiques des Réseaux de Petri. Masson (1989)

AVR: Abstractly Verifying Reachability

Aman Goel(✉) and Karem Sakallah

University of Michigan, Ann Arbor MI 48105, USA
{amangoel,karem}@umich.edu

Abstract. We present AVR, a push-button model checker for verifying state transition systems directly at the source-code level. AVR uses information embedded in the word-level syntax of the design representation to automatically perform scalable model checking by combining a novel syntax-guided abstraction-refinement technique with a word-level implementation of the IC3 algorithm. AVR provides independently-verifiable certificates that offer provable assurance and are easy to relate to the word-level system. Moreover, proof certificates can be further used in innovative ways to extract key design information and are useful in a growing number of applications.

1 Introduction

Model checking [27,28] techniques based on incremental induction (like IC3 [19,31]) have gained significant success [21] due to their property-directed nature and clever use of incremental SAT solving. Bit-level implementations of IC3, however, struggle with scalability due to being overwhelmed by low-level propositional learning [33]. Rapid advances in SMT solving [54,12] offer a solution and allow for performing IC3 directly at the word level by combining the incremental induction algorithm with an abstraction-refinement procedure [18,41,23,34].

AVR [2] is a model checker designed, primarily, for verifying *safety* properties of hardware. It uses *syntax-guided abstraction* [34], a generalization of implicit predicate abstraction [22], to perform IC3-style reachability on a *first-order logic* encoding of the transition relation resulting in word-level clause learning. Upon termination, AVR will either produce a *proof certificate*, in the form of a state formula representing an *inductive invariant*, if the safety property holds or a counterexample execution trace if it fails. In both cases, confidence in the verification output is achieved by using an external proof checker to independently confirm the correctness of the proof certificate or a trace simulator depicting the sequence of transitions leading to the failure. Beyond hardware, these features allow AVR to be used in innovative ways including the verification of distributed protocols defined over unbounded domains [44,45]. AVR also provides a variety of complementary verification techniques, such as data abstraction and interpolation, to increase its scalability, as well as useful utilities, such as design statistics and graphical visualizations, to provide high-level insights on the input design. AVR was independently evaluated to be the best word-level verifier in the single bit-vector track of Hardware Model Checking Competition (HWMCC) 2019 [17].

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 413–422, 2020.
https://doi.org/10.1007/978-3-030-45190-5_23

2 Motivation

Consider a predicate $p := (a + b < 1)$ defined over two 32-bit variables a and b. An equivalent propositional-level representation of p will involve a bit-blasted expression involving 64 Boolean variables and several hundred clauses. As a consequence, bit-level model checking algorithms do not scale as variable bit widths increase and suffer from the so-called *state-space explosion* problem [26].

AVR derives its motivation from the fact that the word-level representation of a problem contains useful high-level information that can be exploited for better scalability. Building on our previous work [33,34], AVR uses this insight to infer an implicit syntax-guided abstraction using terms built from objects present in the word-level syntactic description of the problem (like a, b, 1, $+$, $<$). The approach can be further combined with data abstraction using uninterpreted functions [20,11] to simplify reasoning for the underlying query solver. This, coupled with efficient SMT solving, allows for an effective word-level model checking algorithm that can scale better than bit-level engines for a variety of verification problems. Moreover, the underlying induction-based verification procedure has the unique strength of producing word-level proof certificates that are useful in a variety of applications [32,37,45,44].

3 System Architecture

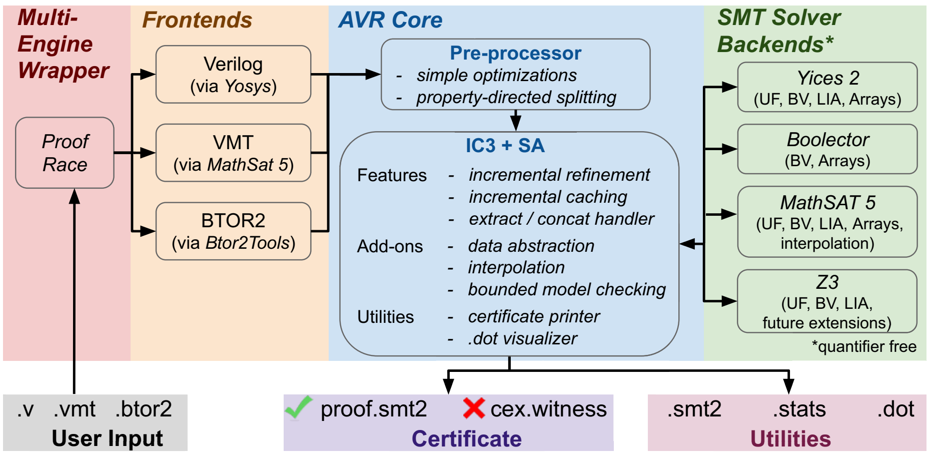


Fig. 1: Verification flow with AVR
UF: uninterpreted functions, BV: bit-vectors, LIA: linear integer arithmetic

Fig. 1 shows the architecture and verification flow of AVR.

Frontends in AVR extract the model checking problem from inputs in different formats using openly-available tools.

– Verilog + SystemVerilog Assertions [9]	(using Yosys [55])
– VMT [8]	(using MathSAT 5 [24])
– BTOR2 [51]	(using Btor2Tools [3])

AVR core performs IC3 with syntax-guided abstraction (IC3+SA) and implements several verification techniques and utilities (detailed in §3.1, §3.2).

SMT solver backends use the latest versions of state-of-the-art SMT solvers (Yices 2 [30], Boolector [50], MathSAT 5 [24] and Z3 [48]) to efficiently integrate incremental solver reasoning with *AVR core* using a C++ interface.

Multi-engine wrapper allows for process-level parallelism by running multiple instances of AVR in parallel using *proof race* (as elaborated later in §3.3).

3.1 Techniques

At its core, AVR implements a word-level IC3 procedure where terms in the implicit syntax of the problem are used as building blocks to perform IC3-style clause learning at the word level using SMT solving. The key differences between IC3+SA [34], as implemented in AVR, and bit-level IC3 [19,31] can be summarized as follows:

– IC3+SA uses relations defined over syntax terms (referred as *atoms*) instead of individual state bits to implicitly represent an abstract state space.
– SMT solving is used instead of propositional SAT solving for solver reasoning.
– Counterexample-guided abstraction refinement [25] is used to automatically eliminate the spurious behavior in the syntactically abstracted domain by identifying new terms from the proof of unsatisfiability [42].

Within the core IC3+SA framework, AVR implements several optimizations and important features that are helpful in improving model checking performance.

Core features

– *Pre-processor optimizations* perform simple transformations to standardize and optimize the input model extracted from different input formats.
– *Incremental refinement* performs abstract counterexample analysis in an incremental fashion by using single-step solver queries instead of conventional multi-step path queries.
– *Incremental caching* allows caching frequently-used data structures to speed up incremental SMT solving (at the cost of increasing memory usage).
– *Multiple SMT backends* allow configuring usage of different SMT solvers for different kinds of SMT queries based on the type of query.

Add-on techniques

– *Property-directed splitting* breaks wide words at bit-field extraction and concatenation boundaries [10] in a property-directed manner.
– *Data abstraction* focuses on the control structure of the problem by combining IC3+SA with data abstraction which converts data operations to uninterpreted functions [20,11,41],.
– *Interpolation* adds Craig interpolants [46] and incremental refinement to extract new terms from a spurious abstract counterexample.
– *Extract/Concat handler* adds a *novel* dedicated engine to deal with lightweight interpretation of bit-field extraction and concatenation operations.

– *Bounded model checking* (BMC) [15] allows for an alternative to the IC3+SA engine for quick bug hunting, especially for shallow bugs.
– *Other options* include adding global assumptions *lazily*, minimizing proof certificates, making syntax-guided abstraction closer to (resp. farther from) implicit predicate abstraction by decreasing (resp. increasing) abstraction *granularity*, exploiting *randomness* during solving, and a few others.

Utilities

AVR also provides a number of useful utilities to the user including:
– Printing the problem in SMT-LIB format [13].
– Graphical visualizations of the problem and the word-level clause learning.
– Detailed statistics report on the input design and the verification run.

3.2 Certificates

Once a model checking problem is solved, there can be two possible outcomes: either the property holds (safe), or it fails (unsafe).

If the property holds, IC3+SA produces an inductive invariant, i.e. an approximate fixpoint that establishes the property to be true in *all* executions of the system. Inductive invariants act as proof certificates that guarantee the correctness of the verification outcome. AVR prints such proof certificates directly in the SMT-LIB format, which allows for independent checking of their correctness using an external SMT solver like Yices 2 or Z3. Since proof certificates are in the word-level format, they are human-readable and much easier to relate to the word-level input directly at the source-code level (as against bit-level invariants which are usually too hard to understand). Proof certificates have many useful applications, including the derivation of inductive validity cores [32], gaining deeper insights on design behavior, deriving assume-guarantee verification conditions [37,53], deriving helper assertions during multi-property verification [36,29], and generalizing to quantified domains (as elaborated later in §4.3).

When the property fails, AVR produces a counterexample trace that establishes how to reach a bad state (a state where the property is false) starting from an initial state. AVR prints the counterexample witness in BTOR2 witness format [51], which allows for independent verification of the execution trace using a BTOR2 witness simulator [4]. This allows the designer to debug and pin-point the source of error by analyzing the execution leading to the buggy state.

3.3 Proof Race

AVR supports a variety of configurations and add-on features (as discussed in §3.1). Without detailed knowledge of the input, it is hard to tell upfront which technique will perform the best. Different configurations are useful to tackle different types of problems, though manually trying different configurations can become tedious for the user. To counter this, AVR offers a multi-engine wrapper called *proof race* that automatically runs multiple instances of AVR with different configurations in parallel and offers process-level parallelism. Given a set of specified resource limits, proof race initiates multiple AVR instances and terminates execution as soon as one of these instances successfully *races* to the result. Such a portfolio-based approach is crucial in practice for fast verification performance since no single technique performs best in all cases [21,16]. It is also further strengthened by complementing AVR's word-level techniques with state-of-the-art model checking engines like ABC *dprove* [14], IC3ia [23] etc.

4 Case Studies[1]

4.1 Apache Buffer Overflow

We consider patched versions of two buffer overflow vulnerabilities [40] from standard modules of the Apache web server [1].

apache-escape-absolute corrects a high severity vulnerability `CVE2006-3747` [7] that fixes the out-of-bounds buffer overflow exploitation which allows a remote attacker to cause a denial of service and execute arbitrary code via crafted URLs. The patched version corrects a check $(c < \texttt{TOKEN\_SZ})$ to $(c < \texttt{TOKEN\_SZ} - 1)$.

apache-get-tag fixes a medium severity vulnerability `CVE-2004-0940` [6] that exploits a buffer overflow when copying user-supplied tag strings into finite buffers. A local attacker may leverage this issue to execute arbitrary code on the affected computer with the privileges of the affected Apache server. The patched version corrects a check that validates the length of the tag strings.

In less than a minute, AVR successfully verifies that both of these buffer overflow exploits are unreachable in the patched versions for *any* buffer size. AVR also provides human-readable proof certificates that are externally verified using Z3, and provides provable assurance against these security vulnerabilities.

4.2 Public Key Authentication Protocol

The Needham-Schroeder public key authentication protocol [49] allows establishing mutual authentication between an initiator A and a responder B, after which some session involving the exchange of messages between them can take place. Unfortunately, this protocol is vulnerable to a man-in-the-middle attack [43]. If an intruder I can persuade A to initiate a session with him, he can relay the messages to B and convince B that he is communicating with A.

We consider an instance of the protocol from HWMCC'19 [17,52] with 3 initiators and responders each, and with an unsafe state defined as a responder being finished authentication with the intruder as a party. Within a minute, AVR finds an execution trace that establishes how to reach an unsafe state. The counterexample witness produced by AVR can be replayed using the BtorSIM simulator [4] to verify the execution trace and to debug the protocol.

4.3 Verifying Distributed Protocols

Beyond verifying model checking problems from finite domains, AVR has shown preliminary application in the verification of distributed protocols, which are

[1] All results presented in this paper can be replicated from [35,5].

generally expressed over unbounded domains (with an unbounded number of clients, servers, epochs, messages, etc.). The I4 system [45,44] demonstrates how AVR can be used to verify a simpler finite version of the protocol, followed by generalizing AVR's proof certificates to the unbounded domain. For example, a finite-domain invariant saying "clients C_1 and C_2 cannot both link to the server S" i.e. $\neg(link(C_1, S) \wedge link(C_2, S))$ can be generalized to the unbounded domain as "no two different clients can both link to a server" i.e.
$\forall_{C_1,C_2,S}\ (C_1 \neq C_2) \implies \neg(link(C_1, S) \wedge link(C_2, S))$.

5 Strengths

Control-centric properties, where much of the complexity lies in the control logic (such as sequential equivalence checking, microprocessor instruction control unit, key-value store) are much easier to verify using AVR. Syntax-guided abstraction hides the domain complexity outside of the problem syntax, and automatically separates important control-flow details from the irrelevant data component. This, combined with data abstraction, allows for scalable model checking with the capacity to scale independently of the variable bit widths [33,34].

Push-button verification using AVR eliminates the need for tedious human intervention in verification (such as manual identification of abstraction predicates, manually adding helper assertions) by automatic incremental construction of abstraction and word-level clauses using the IC3+SA algorithm.

Provable assurance on the verification outcome is guaranteed by AVR using independently-checkable proof certificates and counterexample traces.

Useful utilities that AVR provides, such as support for multiple input formats, efficient integration with state-of-the-art SMT solvers, proof race, high-level system statistics, graphical visualizations, etc. contribute to a user-friendly experience and ease of use.

6 Limitations

Heavy data dependency can make word-level techniques in AVR ineffective for certain problems, especially when a majority of bit-precise values in the data domain play an important role (for example, puzzle solving problems like Tower of Hanoi [39], Peg Solitaire [38], etc. formulated as reachability problems [52]). Logic synthesis and bit-level optimizations [14,47] can be very useful for such problems and help bit-level checkers perform better than word-level techniques by significantly decreasing the problem complexity at the bit level.

First-order logic fragments beyond quantifier-free bit-vectors, arrays and uninterpreted functions (such as non-linear arithmetic, floating-point numbers, quantifiers, etc.) and properties beyond safety (such as *liveness* and *fairness*) have limited support in the current tool implementation. AVR's primary focus has been on verification of safety properties defined on hardware systems.

7 Conclusions

AVR provides a variety of techniques to efficiently perform automatic word-level verification using SMT solvers with provable guarantees and security. AVR has been effective in hardware verification [17,33,34] and shows significant promise for the verification of distributed protocols [44,45]. In the future, we plan to address some of its current limitations and extend its application to practical verification problems beyond the hardware domain.

Data Availability Statement and Acknowledgments. The software and datasets generated and analyzed during the current study are available in the Zenodo repository: https://doi.org/10.5281/zenodo.3677545. The authors would like to thank Ranan Fraer, Ravi Prakash, Habeeb Farah and Ziyad Hanna from Cadence Design Systems for their help in shaping some of the concepts presented in this paper.

References

1. Apache HTTP server project. https://httpd.apache.org
2. AVR (github). https://github.com/aman-goel/avr
3. Btor2Tools. https://github.com/Boolector/btor2tools
4. BtorSIM. https://github.com/Boolector/btor2tools/tree/master/src/btorsim
5. Experiments. https://github.com/aman-goel/tacas20ae
6. National Vulnerability Database - CVE-2004-0940. https://nvd.nist.gov/vuln/detail/CVE-2004-0940
7. National Vulnerability Database - CVE-2006-3747. https://nvd.nist.gov/vuln/detail/CVE-2006-3747
8. Verification Modulo Theories. http://www.vmt-lib.org
9. Ieee standard for systemverilog–unified hardware design, specification, and verification language. IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) pp. 1–1315 (Feb 2018). https://doi.org/10.1109/IEEESTD.2018.8299595
10. Andraus, Z.S., Sakallah, K.A.: Automatic abstraction and verification of verilog models. In: Proceedings. 41st Design Automation Conference, 2004. pp. 218–223 (July 2004). https://doi.org/10.1145/996566.996629
11. Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. pp. 366–378. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
12. Barrett, C., Deters, M., de Moura, L., Oliveras, A., Stump, A.: 6 years of SMT-COMP. Journal of Automated Reasoning **50**(3), 243–277 (Apr 2012). https://doi.org/10.1007/s10817-012-9246-5
13. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
14. Berkeley Logic Synthesis and Verification Group: ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/ (2017)
15. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)

16. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 9–9. IEEE (2017)
17. Biere, A., Preiner, M.: Hardware model checking competition (HWMCC) 2019. http://fmv.jku.at/hwmcc19
18. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (ctigar). In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 831–848. Springer International Publishing, Cham (2014)
19. Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
20. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Dill, D.L. (ed.) Computer Aided Verification. pp. 68–80. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
21. Cabodi, G., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Vendraminetto, D., Biere, A., Heljanko, K.: Hardware model checking competition 2014: An analysis and comparison of model checkers and benchmarks. Journal on Satisfiability, Boolean Modeling and Computation **9**(1), 135–172 (Jan 2016). https://doi.org/10.3233/SAT190106
22. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Ic3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 46–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
23. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. Formal Methods in System Design **49**(3), 190–218 (Sep 2016). https://doi.org/10.1007/s10703-016-0257-4
24. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 93–107. Springer (2013)
25. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
26. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Progress on the state explosion problem in model checking. In: Informatics. pp. 176–194. Springer (2001)
27. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: algorithmic verification and debugging. Communications of the ACM **52**(11), 74–84 (2009)
28. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model checking. MIT press (2018)
29. Dureja, R., Rozier, K.Y.: Fuseic3: An algorithm for checking large design spaces. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 164–171 (Oct 2017). https://doi.org/10.23919/FMCAD.2017.8102255
30. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 737–744. Springer International Publishing, Cham (2014)
31. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: 2011 Formal Methods in Computer-Aided Design (FMCAD). pp. 125–134. IEEE (2011)
32. Ghassabani, E., Gacek, A., Whalen, M.W.: Efficient generation of inductive validity cores for safety properties. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 314–325 (2016)

33. Goel, A., Sakallah, K.: Empirical evaluation of ic3-based model checking techniques on verilog rtl designs. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 618–621 (March 2019). https://doi.org/10.23919/DATE.2019.8715289
34. Goel, A., Sakallah, K.: Model checking of verilog rtl using ic3 with syntax-guided abstraction. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods. pp. 166–185. Springer International Publishing, Cham (2019)
35. Goel, A., Sakallah, K.: AVR: Abstractly Verifying Reachability (Feb 2020). https://doi.org/10.5281/zenodo.3677545
36. Goldberg, E., Güdemann, M., Kroening, D., Mukherjee, R.: Efficient verification of multi-property designs (the benefit of wrong assumptions). In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 43–48 (March 2018). https://doi.org/10.23919/DATE.2018.8341977
37. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: Methodology and case studies. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification. pp. 440–451. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
38. Jefferson, C., Miguel, A., Miguel, I., Tarim, S.A.: Modelling and solving english peg solitaire. Computers & Operations Research **33**(10), 2935–2959 (Oct 2006). https://doi.org/10.1016/j.cor.2005.01.018
39. Kotovsky, K., Hayes, J., Simon, H.: Why are some problems hard? evidence from tower of hanoi. Cognitive Psychology **17**(2), 248–294 (Apr 1985). https://doi.org/10.1016/0010-0285(85)90009-x
40. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. p. 389–392. ASE '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1321631.1321691
41. Lee, S., Sakallah, K.A.: Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 849–865. Springer International Publishing, Cham (2014)
42. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning **40**(1), 1–33 (Sep 2007). https://doi.org/10.1007/s10817-007-9084-z
43. Lowe, G.: An attack on the needham-schroeder public-key authentication protocol. Information Processing Letters **56**(3), 131 – 133 (1995). https://doi.org/10.1016/0020-0190(95)00144-2
44. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: Incremental inference of inductive invariants for verification of distributed protocols. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 370–384. SOSP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359651
45. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: Towards automatic inference of inductive invariants. In: Proceedings of the Workshop on Hot Topics in Operating Systems. p. 30–36. HotOS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3317550.3321451
46. McMillan, K.L.: Applications of craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

47. Mishchenko, A., Case, M., Brayton, R., Jang, S.: Scalable and scalably-verifiable sequential synthesis. In: 2008 IEEE/ACM International Conference on Computer-Aided Design. pp. 234–241. IEEE (2008)
48. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
49. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (Dec 1978). https://doi.org/10.1145/359657.359659
50. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. Journal on Satisfiability, Boolean Modeling and Computation **9**(1), 53–58 (Jun 2015). https://doi.org/10.3233/SAT190101
51. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , btormc and boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 587–595. Springer International Publishing, Cham (2018)
52. Pelánek, R.: Beem: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) Model Checking Software. pp. 263–267. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
53. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., et al.: Dependent types and multi-monadic effects in f. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 256–270 (2016)
54. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The smt competition 2015 - 2018. Journal on Satisfiability, Boolean Modeling and Computation **11**(1), 221–259 (Sep 2019). https://doi.org/10.3233/SAT190123
55. Wolf, C.: Yosys open synthesis suite. http://www.clifford.at/yosys/

Timed and Probabilistic Systems

Verified Certification of Reachability Checking for Timed Automata

Simon Wimmer and Joshua von Mutius

Fakultät für Informatik,
Technische Universität München,
Munich, Germany
`wimmers@in.tum.de` `joshua.von-mutius@tum.de`

Abstract. Prior research has shown how to construct a mechanically verified model checker for timed automata, a popular formalism for modeling real-time systems.
In this paper, we shift the focus from verified model checking to certifying unreachability. This allows us to benefit from better approximation operations for symbolic states, and reduces execution time by exploring fewer states and by exploiting parallelism. Moreover, this gives us the ability to audit results of unverified model checkers that implement a range of further optimizations, including certificate compression.
The resulting tool is evaluated on a set of standard benchmarks to demonstrate its practicality, using a new unverified model checker implementation in Standard ML to construct the certificates.

Keywords: Timed automata · Certification · Model Checking · Interactive Theorem Proving · Isabelle/HOL

Timed automata [1] are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as Uppaal [4]. These tools can be understood as trust-multipliers: we trust their correctness to deduce trust in the safety of systems checked by these tools. As a consequence, one wants to ensure as rigorously as possible that the computation results of timed automata model checkers are correct.

Previous work [31] has addressed this problem by constructing a model checker for timed automata that is fully verified using Isabelle/HOL [25]. This tool is intended to be a reference implementation that can be used to scrutinize the correctness of other model checkers. As such, it is mainly able to check small and medium-sized benchmark examples, but the performance gap w.r.t. more practical model checkers prevents it from checking realistic benchmark models within reasonable time and space bounds.

We address this issue by shifting the focus from full verified model checking to only certifying that the result produced by an unverified model checker is correct. We only study reachability: it is the most important property that is checked with timed automata model checkers, and some model checkers only support reachability. It is crucial to ensure that a bad state is certainly not reachable if

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 425–443, 2020.
https://doi.org/10.1007/978-3-030-45190-5_24

the model checker claims so, thus we want to certify *unreachability*. Certifying that a state is indeed reachable would amount to extracting a timed trace and certifying that the trace is compatible with the model. While implementing this in a verified manner would be comparatively easy, we consider it less important because it corresponds to the bug finding functionality of model checkers, which carries less trust.

The recipe for certifying unreachability is simple: the model checker explores a number of states until it determines that there are no more states to be found. If none of the states fulfill the final state predicate (i.e. violates the safety property), then the model checker will answer "unreachable". We use the set of explored states as the unreachability certificate. In essence, we only need to check that the initial state is contained in this set, that there are no outgoing edges from this set, and that none of the states in the set fulfill the final state predicate.

The switch to certification holds many advantages. Timed automata model checking uses *over-approximations* of symbolic states to ensure termination. A large variety of these approximation operators has been studied [2,3,14]. Our previous work [29] has shown that, while formally proving the correctness of these approximation operations is feasible in principle with an interactive theorem prover, the effort is rather high. Instead, to *certify* unreachability, it is sufficient to only know that the approximation operator indeed yields a state that is at least as big as the precise symbolic state. Certifying this property is cheap.

Moreover, certification eases *parallelization*. Checking that a state is not final and that all its successors are covered by the state set are local properties. We show how to exploit this in a verified implementation, while only mildly increasing the verification effort and the size of the trusted code base.

Finally, the number of states explored by a model checker can vary immensely, depending on a range of factors such as the chosen approximation operator or the search order. Thus, an efficient unverified tool can exploit different heuristics and strategies to compute a state space that is as small as possible, and thereby speedup the certification effort. In this context, we also study a number of *compression techniques* to reduce the number of states in the certificate after the model checker has concluded its search.

We use a new unverified model checker called Mlunta, which is implemented in Standard ML (SML), to generate certificates for a set of standard benchmarks, and to evaluate our verified certifier's performance on these benchmarks [1].

Related Work This work is based on an existing Isabelle/HOL formalization of timed automata model checking [29,31]. Other proof-assistant formalizations of timed automata focus on proving elementary properties about the basic formalism [33,34], or proving properties about concrete automata [26,10,8], but none of them are concerned with model checking.

Earlier work formalizes a model checker for the modal μ-calculus [28], and constructs a verified finite state LTL model checker [9,24,6].

[1] Both tools are available online: https://doi.org/10.5281/zenodo.3679245.

The idea of extracting certificates from the model checking process has previously been studied in the context of the μ-calculus [23] and finite state LTL model checking [27]. However, these works are not accompanied by a verified certificate checker and do not attempt to scale the approach to practical examples. Only the recent work of Griggio et al. [11] provides a practical extraction mechanism and a certificate checker for LTL model checking, but the checker is not verified. To the best of our knowledge, we are the first to examine certification in the context of timed automata model checking.

Finally, in the context of software verification, the idea of producing certificates for the correctness of a program has been broadly studied [16,5].

Isabelle/HOL Isabelle/HOL [25] is an interactive theorem prover based on Higher-Order Logic (HOL). HOL can be thought of as a combination of a functional programming language and mathematical logic. Isabelle/HOL mostly resembles standard mathematical notation. Some conventions that are borrowed from functional programming need to be explained, however. Functions are mostly curried, i.e. of type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$ instead of $\tau_1 \times \tau_2 \Rightarrow \tau$. As a consequence, function application is usually denoted as $f\ a\ b$ instead of $f(a, b)$. Function abstraction with lambda terms uses the standard syntax $\lambda x.\ t$ (the function that maps x to t) and can also have paired arguments $\lambda(x, y).\ t$. Type variables are written $'a$, $'b$, etc. Compound types are written in postfix syntax: $\tau\ \mathit{set}$ is the type of sets of elements of type τ. We use the Isabelle/HOL convention that free variables are implicitly all-quantified throughout the paper. In parts of the paper, formulas or syntax have been simplified for readability, but we have stayed largely faithful to the Isabelle/HOL formalization.

Contributions In short, these are the main contributions of our work:

- To the best of our knowledge, we are the first to study certification of the model checking results of reachability checking for timed automata, including techniques to compress certificates.
- We construct a verified implementation of such a certificate checker, including a number of optimization techniques to make it practically usable.

Outline The remainder of the paper is organized as follows. The first section briefly recalls the theory of timed automata, and sketches the state-of-the-art model checking process. The second section details our approach to certification and explains how, starting from an abstract theory, a concrete verified implementation of the certificate checker can be obtained. Section three illustrates a number of techniques to improve the certificate checker's performance, while only mildly increasing the formalization effort. Section four discusses two methods for certificate compression. The paper is concluded by an experimental evaluation and remarks on potential future work.

1 Timed Automata and Model Checking

Transition Systems We take a very simple view of transition systems: they are simply a relation $\rightarrow$ of type $'a \Rightarrow 'a \Rightarrow bool$ for a type of states $'a$. We write $a \rightarrow^* b$ to denote that b can be reached from a via a sequence of $\rightarrow$-transitions.

Timed Automata To make the paper self-contained, this paragraph briefly describes timed automata and is mostly reproduced from Wimmer and Lammich [29]. For a thorough introduction see the tutorial paper of Bengtsson and Yi [4].

Compared to standard finite automata, timed automata introduce a notion of clocks. Figure 1 depicts an example of a timed automaton. We will assume that clocks are of type *nat*. A *clock valuation* u is a function of type $nat \Rightarrow real$. Locations and transitions are guarded by *clock constraints*, which have to be

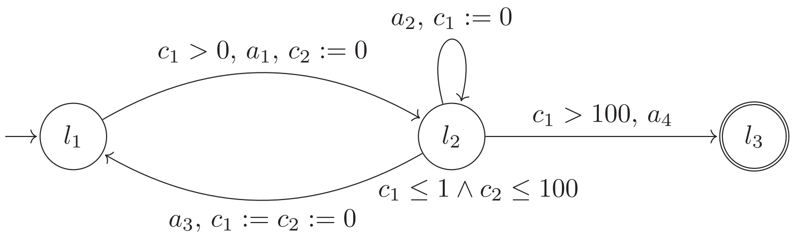


Fig. 1: Example of a timed automaton with two clocks.

fulfilled to stay in a location or to take a transition. Clock constraints are conjunctions of constraints of the form $c \sim d$ for a clock c, an integer d, and $\sim\ \in \{<, \le, =, \ge, >\}$. We write $u \models cc$ if the clock constraint cc holds for the clock valuation u. We define a timed automaton A as a pair $(\mathcal{T}, \mathcal{I})$ where $\mathcal{I}$ is a mapping from locations to clock constraints (also named invariants); and $\mathcal{T}$ is a set of transitions written as $A \vdash l \longrightarrow^{g,a,r} l'$ where l and l' are start and successor location, g is the guard of the transition, a is an action label, and r is a list of clocks that will be reset to zero when the transition is taken. States of timed automata are pairs of a location and a clock valuation. The operational semantics defines two kinds of steps (given as their HOL descriptions):

- Delay: $(l, u) \rightarrow^d (l, u \oplus d)$ if $d \ge 0$ and $u \oplus d \models \mathcal{I}\ l$;
- Action: $(l, u) \rightarrow_a (l', [r := 0]u)$
  if $A \vdash l \longrightarrow^{g,a,r} l'$, $u \models g$, and $[r := 0]u \models \mathcal{I}\ l'$;

where $u \oplus d = (\lambda c.\ u\ c + d)$ offsets all clocks by d in the valuation u, and $[r := 0]u = (\lambda c.\ \textsf{if}\ c \in r\ \textsf{then}\ 0\ \textsf{else}\ u\ c)$ resets all clocks in r to 0 in valuation u. For any (timed) automaton A, we consider the transition system

$$(l, u) \rightarrow_A (l', u') = (\exists d \ge 0.\ \exists a\ u''.\ (l, u) \rightarrow^d (l, u'') \wedge (l, u'') \rightarrow_a (l', u'))\,.$$

That is, each transition consists of a delay step that advances all clocks by some amount of time, followed by an action step that takes a transition and resets the clocks annotated to the transition. Given a final state predicate F and an initial state (l_0, u_0), we are interested in whether $(l_0, u_0) \rightarrow_A^* (l, u)$ for any l, u with $F\,l$. In Figure 1, the final state is l_3 (i.e. $F\,l \longleftrightarrow l = l_3$). As the guard for action a_4 is never enabled, l_3 is unreachable.

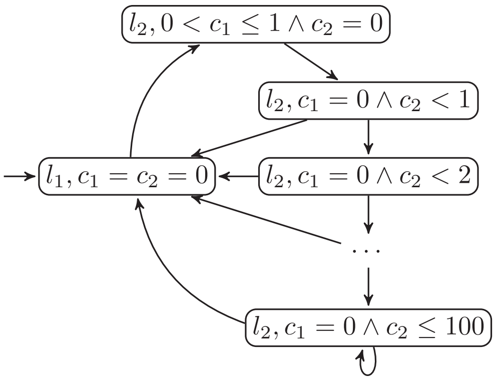
Model Checking Due to the use of clock valuations, the state space of timed automata is inherently infinite. Thus, model checking algorithms for timed automata are based on the idea of abstracting from concrete valuations to *sets* of clock valuations of type (*nat* ⇒ *real*) *set*, often called *zones*. The resulting transition system of reachable states from an initial zone is called the zone graph. It is explored in an *on-the-fly* manner, computing successors on zones, which are typically represented symbolically as *Difference Bound Matrices* (DBMs). Knowledge of this data structure is not necessary to understand the rest of the paper. Thus we refer the interested reader to Bengtsson and Yi [4] and to Wimmer and Lammich [29,31] for a verification of this data structure. In the remainder we will only use the term "zones" instead of referring to their implementation as DBMs.

The delicate part of this method is that the number of reachable zones could still be infinite. Therefore, over-approximations (or *abstractions*) of zones are computed to obtain a finite search space. For our purpose, it sufficient to assume an abstraction operator α indeed computes an over-approximation, i.e. $Z \subseteq \alpha(Z)$ for any zone Z. We call the version of the zone graph where abstractions are applied the *abstract zone graph* [13]. For a number of such abstraction operators, it can be shown that the abstract zone graph is sound and complete [2]. The proofs are rather intricate, however. Thus formalizing them would be a big effort. By focusing on certification of unreachability, this problem vanishes, as we only need to ensure that any state (l, Z) that we deem reachable in the zone graph is *subsumed* by some state (l, Z') with $Z \subseteq Z'$ that is part of the certificate and that was computed by the abstraction (i.e. $Z' = \alpha(Z_1)$ for some Z_1).

Certificates by Example Figure 2 depicts the zone graph of the automaton in Figure 1. Each zone Z is given as a clock constraint cc such that $Z = \{u \mid u \models cc\}$. A model checker like Munta would have to explore the full zone graph before being able to decide that l_3 is unreachable. Any model checker that uses the same abstraction technique as Munta [2] would not be able to benefit from abstractions for this example and thus the abstract zone graph is the same as the zone graph. However, such a model checker could apply subsumptions while exploring the zone graph. That is, when a symbolic state of the form $(l_2, \{u \mid u \models c_1 = 0 \wedge c_2 < k+1\})$ is explored, the state $(l_2, \{u \mid u \models c_1 = 0 \wedge c_2 < k\})$ can safely be discarded.

This means that at the end of the model checking process, only the three states in Figure 3a will be stored. The solid edges are part of the zone graph,

[2] Soundness: for every abstract run, there is a concrete instantiation. Completeness: every concrete run can be abstracted.



$l_2, 0 < c_1 \leq 1 \land c_2 = 0$
$l_2, c_1 = 0 \land c_2 < 1$
$l_1, c_1 = c_2 = 0$
$l_2, c_1 = 0 \land c_2 < 2$
$\dots$
$l_2, c_1 = 0 \land c_2 \leq 100$

Fig. 2: The zone graph of the automaton depicted in Figure 1.

while the dashed edge indicates that the zone at its tail has a successor in the zone graph ($(l_2, \{u \mid u \models c_1 = 0 \land c_2 < 1\})$) that is subsumed by the tip of the edge. The set of these three states can act as a *certificate* of unreachability. They essentially form an inductive invariant of the zone graph: for each state in the certificate, all its successors in the zone graph are either contained in the certificate themselves or subsumed by another state in the certificate. Thus we know that any symbolic state that is reachable from the initial state is subsumed by some state in the certificate, and as the final state is not contained in the certificate, we can conclude that it is unreachable.

Figure 3b shows a certificate with only two states that replaces the two states for l_2 by the state with a dashed border. Note that this state is not part of the original zone graph. The certificate fulfills the same invariant property and thus also proves unreachability. We will use this technique of adding larger states to the certificate that are not part of the zone graph for our compression techniques in section 4.

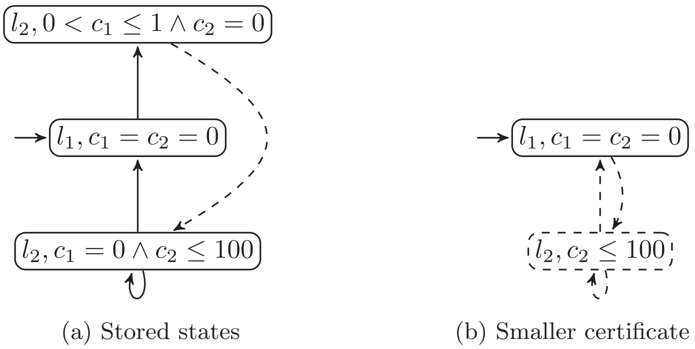


Fig. 3: Two certificates of unreachability for the automaton from Figure 1.

2 From Model Checking to Certifying Unreachability

This section first describes our approach to certification abstractly. Then, we detail how the existing formalization of a timed automata model checker was extended—with rather low effort—to a verified certifier. In practice, networks of timed automata with additional modeling features such as, e.g. shared state variables, are used. However, due to the existing verified product construction for such a formalism [31], it is sufficient to study the case of a single timed automaton here.

2.1 An Abstract Correctness Theorem

To work towards a rigorous justification of the certification process, we first study the problem on a more abstract level. Consider a transition system $\rightarrow$ on states of type $'l \times 's$ where $'l$ corresponds to the finite state part of timed automata and $'s$ corresponds to zones. We assume an invariant P on states, i.e.:

$$P\ (l_1,\ s_1)\ \wedge (l_1,\ s_1) \rightarrow (l_2,\ s_2) \Longrightarrow P\ (l_2,\ s_2)\,.$$

This invariant essentially represents a restriction of $\rightarrow$ to valid states. While this would usually be assumed implicitly, we explicate P here as it is technically more convenient to do so in the Isabelle/HOL formalization.

The interesting feature that sets timed automata model checking apart is subsumption. Recall that during the model checking process, it is possible to first discover some (symbolic) state (l, Z) (a pair of a discrete state l and a zone Z), and to find at some later point that another reachable state (l, Z') subsumes (l, Z) because Z' semantically contains Z', i.e. $Z \subseteq Z'$. At this point the state (l, Z) can be discarded as we know that anything that is reachable from (l, Z) is also reachable from (l, Z'). Abstractly, subsumption is modeled by some fixed preorder (i.e. a reflexive and transitive relation) $\preccurlyeq$ on $'s$ which is a simulation relation between $\rightarrow$ and itself:

$$\begin{array}{ll} & s_1 \preccurlyeq s_2 \wedge (l_1, s_1) \rightarrow (l_2, t_1) \wedge P\ (l_1, s_1) \wedge P\ (l_2, s_2) \\ \Longrightarrow & \exists t_2.\ t_1 \preccurlyeq t_2 \wedge (l_1, s_2) \rightarrow (l_2, t_2) \end{array}$$

In the abstract setting, a certificate consists of a set of discrete states L of type $'l\ set$, and a mapping M of type $'l \Rightarrow 's\ set$ that gives the set of reachable symbolic states that were computed for any discrete state $l \in L$. We say that (L, M) satisfies P if all states in the certificate (L, M) satisfy P:

$$l \in L \wedge s \in M\, l \Longrightarrow P\, (l,\, s)$$

Moreover, the certificate needs to be *closed*. Following Herbreteau et al. [13], we call a state *covered* if it is subsumed by another state in the certificate. A certificate is closed if for each state in the certificate all its successors are covered:

$$l_1 \in L \wedge s_1 \in M\, l_1 \wedge (l_1,\, s_1) \rightarrow (l_2,\, s_2) \Longrightarrow l_2 \in L \wedge (\exists s_3 \in M\, l_2.\, s_2 \preccurlyeq s_3) \quad (*)$$

The following key theorem states that all reachable states are covered if the initial state is covered:

Theorem 1. *Let (L, M) be closed and invariant under P. Assume $l_0 \in L$, $s_0' \in M\, l_0$, $s_0 \preccurlyeq s_0'$, and $(l_0,\, s_0) \rightarrow^* (l,\, s)$. Then $l \in L$ and there exists s' such that $s' \in M\, l$ and $s \preccurlyeq s'$.*

Proof. By induction on the number of steps in $(l_0,\, s_0) \rightarrow^* (l,\, s)$. The following sketches how the run of covering states is constructed. The first line represents $(l_0, s_0) \rightarrow^* (l, s)$ and the states in the third line are all part of the certificate.

$$\begin{tikzcd}
(l_0, s_0) \arrow[r] \arrow[dd, phantom, "\preceq"] & (l_1, s_1) \arrow[r] \arrow[d, phantom, "\preceq"] & \dots \arrow[r] & (l, s) \arrow[d, phantom, "\preceq"] \\
 & (l_1, t_1) \arrow[d, phantom, "\preceq"] & \dots & (l, t) \arrow[d, phantom, "\preceq"] \\
(l_0, s_0') \arrow[ur] & (l_1, s_1') \arrow[ur] & \dots \arrow[ur] & (l, s')
\end{tikzcd}$$

From the assumptions on l_0, s_0, and s_0', we can first apply the self-simulation property of $\rightarrow$ to $(l_0, s_0) \rightarrow (l_1, s_1)$ to obtain a t_1 such that $s_1 \preccurlyeq t_1$ and $(l_0, s_0') \rightarrow (l_1, t_1)$. As the certificate is closed we thus get $l_1 \in L$ and we can find an $s_1' \in M\, l_1$ such that $t_1 \preccurlyeq s_1'$ (and thus $s_1 \preccurlyeq s_1'$ by transitivity). The induction hypothesis can then be applied to l_1, s_1, and s_1'. □

We will now say that a certificate (L, M) is *admissible* iff

- it satisfies P,
- it is closed,
- it covers the initial state (i.e. there is an $s_0' \in M\, l_0$ such that $s_0 \preccurlyeq s_0'$),
- and there is no $l \in L$ with $F\, l$.

Corollary 1. *If F is monotone w.r.t. $\preccurlyeq$ and the certificate (L, M) is admissible, then $\nexists l\, s.\, (l_0,\, s_0) \rightarrow^* (l,\, s) \wedge F\, l$.*

2.2 An Abstract Certificate Checker

In practice, the certification process has to consider one additional complication. A model is typically described in terms of human-readable identifiers, while most model checkers and the verified model checker Munta [30] in particular represent these as natural numbers internally to allow for efficient indexing. In our certifier, this is accounted for by relabeling the human-readable identifiers in a given model to natural numbers in a first (verified) pre-processing step. To save additional transformations of the certificate after it was emitted, we let the unverified model checker additionally emit a textual description of such a renaming. The certifier then just needs to check that the given renaming is injective to ensure that it can safely be applied.

Together with the theoretical analysis laid out in the last section, we can thus derive the following strategy for certifying unreachability:

- An unverified model checker explores the reachable state space of a given model symbolically and checks that none of the discovered states (l, s) fulfills $F\, l$.

```
1  definition $check\ (L, M) \equiv$
2    $monadic\_list\_all\ L\ (\lambda l.$ do {
3      let $S = M\, l$ ;
4      let $next = succs\ l\ S$ ;
5      $monadic\_list\_all\ next\ (\lambda(l', S').$ do {
6        $xs \leftarrow$ SPEC $(\lambda xs.\ set\ xs = S')$ ;
7        if $xs = []$ then return True else do {
8          $b1 \leftarrow$ return $(l' \in L)$ ;
9          $ys \leftarrow$ SPEC $(\lambda xs.\ set\ xs = M\, l')$ ;
10         $b2 \leftarrow monadic\_list\_all\ xs\ (\lambda x.$
11           $monadic\_list\_ex\ ys\ (\lambda y.$ return $(x \preccurlyeq y))$
12         ) ;
13         return $(b1 \wedge b2)$
14       }
15     })
16   })
```

Listing 1.1: Monadic program to check whether a certificate is closed.

– The set of explored states is emitted as a certificate, possibly followed by compression (see section 4).
– The model, the final state predicate F, the certificate, and a description of the renaming that was used for the states are passed to the verified certifier.
– The certifier checks that the given renaming is injective, renames the model accordingly, applies the product construction and checks that the certificate is admissible.

If the process is successful, we can conclude by Corollary 1 that no "bad" state (l, s) (i.e. with $F\, l$) is reachable symbolically. We will argue that this really implies that the model is safe in the concrete case of timed automata in section 2.3.

We now lay out how a verified certificate checker that implements said strategy for an abstract transition system can be constructed in Isabelle/HOL. Listing 1.1 displays the definition of the core of the checker that checks whether the certificate is closed in the sense defined above. The program is defined in the non-determinism monad of the Imperative Refinement Framework (IRF) [20]. Some parts, such as checking set membership or converting a (finite) set to a list are still left abstract. A non-deterministic specification SPEC Q returns some value v with $Q\, v$.

The body of the program (lines 2-16) iterates over all discrete states in the certificate L and checks that all corresponding symbolic states are covered. Line 3 retrieves the symbolic states that correspond to discrete state l and in line 4 their symbolic successor states are computed. The result ($next$) is a list of pairs of a discrete state and the set of its corresponding symbolic states. The loop ranging from lines 5 to 15 iterates over this list to ensure that all the successor states are covered. Given a discrete state l' and a set of symbolic states S', line 6 first converts it into a list xs that can be iterated over. This turns into a vacuous operation when the algorithm is refined to an executable version where sets are implemented as lists. Line 8 checks that l' is also part of the certificate. Then, in line 9 the set of corresponding symbolic states is retrieved and converted to a list ys. Finally, lines 10-12 ensure that all states in xs are subsumed by some state in ys.

To prove soundness of *check*, we mainly need correctness theorems for the monadic combinators *monadic_list_all* and *monadic_list_ex*. Given a list xs and a monadic implementation Q_i of a predicate Q, they check whether all states (at least one state) in xs satisfy (satisfies) Q. This is the correctness theorem for *monadic_list_all*, for instance:

$$\begin{array}{ll} & (\forall x.\, Q_i\, x \le \mathsf{SPEC}\,(\lambda r.\, r \longleftrightarrow Q\, x)) \\ \Longrightarrow & \mathit{monadic\_list\_all}\, xs\, Q_i \le \mathsf{SPEC}\,(\lambda r.\, r \longleftrightarrow \mathit{list\_all}\, xs\, Q) \end{array}$$

where $\mathit{list\_all}\, xs\, Q$ holds if and only if Q holds for all elements in xs. After setting up the IRF's verification condition generator with this rule and the corresponding rule for *monadic_list_ex*, it is easy to prove that *check* is sound:

$$\mathit{check}\,(L, M) \le \mathsf{SPEC}\,(\lambda r.\, r \Longrightarrow \mathit{closed}\,(L, M))$$

where the property $\mathit{closed}\,(L, M)$ corresponds to condition $(*)$ from above.

We then use standard refinement techniques to obtain an algorithm check_i that refines *check*, replacing sets by lists. However, the algorithm is still specified in the non-determinism monad and therefore not executable. We use a simple technique to make it executable. Consider the following theorem for *monadic_list_all*:

$$\mathit{monadic\_list\_all}\, xs\,(\lambda x.\, \mathsf{return}\,(P\, x)) = \mathsf{return}\,(\mathit{list\_all}\, xs\, P)\,.$$

It allows us to replace the non-deterministic combinator *monadic_list_all* by the deterministic *list_all*, pushing return to the outside. By exhaustively applying a set of such rewrite rules we obtain an alternative definition of check_i where return appears only on the outermost level, and the inner term is deterministic and thus executable. Using these techniques, we obtain a simple certificate checker that is executable, provided that we can implement the elementary model checking primitives such as the subsumption check or computing the list of successors of a state.

2.3 Transferring the Correctness Theorem

For timed automata, the abstract transition system studied above is the zone graph $\rightarrow_{ZG(A)}$ of a given (single) automaton A. One can show that it simulates $\rightarrow_A$ (completeness of $\rightarrow_{ZG(A)}$):

$$(l, u) \rightarrow_A (l', u') \land u \in Z \Longrightarrow (\exists Z'.\, (l, Z) \rightarrow_{ZG(A)} (l', Z') \land u' \in Z')\,.$$

This simulation property is sufficient to establish that if there is no reachable state (l, Z) in $\rightarrow_{ZG(A)}$ with $F\, l$, then no final state (l, u) is reachable in $\rightarrow_A$:

$$\begin{array}{ll} & (\nexists l, Z.\, (l_0, Z_0) \rightarrow^*_{ZG(A)} (l, Z) \land F\, l) \land u_0 \in Z_0 \\ \Longrightarrow & (\nexists l, u.\, (l_0, u_0) \rightarrow^*_A (l, u) \land F\, l) \end{array}$$

In the formalization, these proofs rely on instantiating a general theory of simulations in transition systems that is derived from the theory of Wimmer and Lammich [31]. From Corollary 1 we get that there is no reachable final state in $\rightarrow_{ZG(A)}$ if the certificate check is passed. Finally, by correctness of the renaming process and the product construction, we can conclude that there is no final reachable state in the input model if there is no final reachable state in $\rightarrow_A$.

2.4 Implementing a Concrete Checker

All the elementary model checking primitives we need for certification have already been implemented [31]. The abstract implementation presented above assumes that the model checking primitives are implemented in a purely functional manner (as they are just regular HOL functions). The existing (verified) model checker [31], however, is an imperative implementation in the Imperative HOL framework. Imperative HOL [7] is a framework for specifying and reasoning about imperative programs in Isabelle/HOL. It provides a *heap monad* in which one can use—analogously to the ML family of programming languages—imperative references and arrays to express imperative programs. Usually, once we have used an imperative implementation anywhere, the whole program would need to be stated in the heap monad. However, we can employ a technique similar to the one that is used for Haskell's *ST* monad [21] to erase the heap monad in a safe way under certain circumstances.

More precisely, if it can be deduced from the type of an imperative computation that no information about references or arrays on the heap can be leaked to the outside of the computation in its result, then the heap monad can be erased for this computation, yielding a pure computation. In the certifier, this is primarily used for computing the symbolic successor of a zone Z for a certain transition. To that end, an immutable representation of the DBM M corresponding to Z is copied to the a newly allocated imperative array, then the imperative pipeline of computations to compute the successor M' is applied to M, and finally M' is copied back to an immutable array. Taken together, this whole computation does not contain the type of an array or reference in its result type, and thus can safely be turned into a pure computation. As a consequence, we are able to reuse the existing verified model checking primitives, while being able to state the certificate checking algorithm purely functionally.

In the concrete checker, the mapping M is implemented using a verified functional hash table implementation based on so-called *diff arrays* [19]. This data structure provides a purely functional interface to an underlying imperative array. When a diff array is updated, it performs the update on the imperative array, and stores a difference that can be used to re-compute the old state of the array. Reading from the most recent version of a diff array is fast as the value can directly be read from the underlying imperative array. If an old version is accessed, the whole array has to be copied to recompute the old version. This gives diff arrays good performance characteristics, as long as they are mostly used linearly. This is the case in our application as the hash table is filled in an initial phase, after which the hash table is used in a read-only manner.

2.5 Parallel Execution

The attentive reader may wonder why we care about a purely functional implementation of the certificate checker at all. Indeed, we could use existing techniques [31] to obtain an imperative implementation of the certificate checker in the heap monad. However, in this setting it would be hard to justify the soundness of executing parts of the checker in parallel. In the purely functional setting, this is much simpler. Our approach to parallel execution is minimalist: we only provide means to execute the *map* combinator on lists in parallel. This is achieved by another custom code translation that is part of the trusted code base. The parallel implementation of *map* uses a task queue that will contain the individual computations that need to be run for each element of xs, and uses a fixed number of threads to work through this list and assemble the final result.

We exploit this *map* implementation to work through the list of discrete states L in parallel, using the equivalence:

$$\mathit{list\_all}\, Q\, xs = \mathit{list\_all}\, \mathit{id}\, (\mathit{map}\, Q\, xs)\,.$$

In doing so, we lose the ability to stop execution early once a list element does not satisify Q. For the certificate checker, however, we assume that usually the certificate is correct, meaning that we have to go through the whole list anyway. We only parallelize the outermost loop of check_i because this should yield reasonably-sized work portions, given that the size of L will typically at least be in the hundreds.

3 Scaling Performance

In this section we discuss two techniques to improve the performance of the certificate checker without increasing the verification effort significantly.

3.1 Monomorphization

Isabelle/HOL supports polymorphism and type classes, which are valuable features for sizeable formalization efforts. Large parts of our formalization also make use of these features, e.g., most of the timed automata semantics are formalized for a general time domain, and operations on DBMs are applicable on DBMs whose entries are formed from more general algebraic structures than the ring of integers. While this yields an abstract and general formalized theory, it can get in our way when trying to obtain efficient code.

When generating SML code from HOL, Isabelle uses a so-called dictionary construction to compile out type classes, which are not supported by SML. This means that most functions carry a large number of additional parameters, which are used to look up elementary operations, such as addition of two numbers. These additional lookup operations degrade performance. One solution is to ensure that all relevant constants that are exported to SML are monomorphic (i.e. specialized to the integer type), eliminating the need for the dictionary construction in most places. Thus, we apply a semi-automated procedure to achieve this monomorphization.

3.2 Integer Representation

Types such as *int* or *nat* are unbounded in Isabelle/HOL meaning they are implemented with the help of big integers in the target languages. To improve performance, we want to use machine integers instead, and instruct Isabelle/HOL's code generator to do that. This is still sound: SML's standard integer operations throw an exception if an overflow occurs instead of silently wrapping around. The code generator can only achieve partial correctness anyway: if program execution does not fail, then its result is consistent with the evaluated HOL term.

3.3 Refined Code Equations

The last type of optimizations we use can be considered to belong to the category of micro-optimizations. These are improved code generator translations for elementary operations and combinators. We employ such improved translations to use native implementation language primitives to convert from mutable to immutable arrays and back. The other such optimizations we use, is to directly use integer values as counters in imperative loops instead of a natural number representation that would box the integers in a data constructor. In the same way, we use integers directly for array indexing.

4 Certificate Compression

In this section, we present two techniques to compress the unreachability certificate. By compression we mean reducing the number of zones that are present in the certificate for each discrete state, using the unverified model checker. The first technique relies on subsumption. As explained above, it is possible that the model checker adds a zone Z to the set of explored states and later another zone Z' with $Z \subseteq Z'$ (i.e. Z' subsumes Z). Thus the first technique simply filters the set w.r.t. $\subseteq$ in the end.

The second technique relies on the following idea: we replace one or more zones by their union, and check that the state space is still closed. This means that we have to check that all the successors of the larger zone are still covered by the current set of states. In that case, we can discard the old zones, and replace them by their union. As the union of two zones is not necessarily convex and thus cannot be represented as a DBM, we do not compute a precise union of zones but their convex hull. This operation is rather cheap as it amounts to taking the pointwise maximum of DBM entries. After computing the convex hull of a number of zones (in canonical form), we only need to apply the expensive operation to restore a canonical form once.

The latter technique yields a whole family of compression algorithms by iterating one of the following operations for each discrete state until a fixed-point is reached:

a) the convex hull of all zones is computed;
b) the convex hull of the first two zones is computed;

c) the convex hull of the first two zones that can successfully be joined is put to the front of the list;
d) same as c) but considering only discrete states for which compression was successful in the last round;
e) same as d) but iterating the operation until saturation.

The next section contains an experimental evaluation of these techniques.

Note that similar techniques for reducing the search-space could also be applied already during model checking. By doing so, the number of states explored and the runtime of model checking could be reduced. This, however, comes at the risk of producing spurious model checking results (i.e. a final state might be deemed reachable, although there is no corresponding reachable state in the timed automaton).

5 Experimental Evaluation

We evaluate the checker on a set of benchmarks that is derived from Uppaal's standard benchmark suite [22]. Additionally, to cover the advanced modeling features of committed locations and broadcast channels, we use a set of benchmarks that is derived from the pacemaker models of Jiang et al. [17] and a modified version of the FDDI benchmark with broadcast channels. A prototype SML implementation of a timed automata model checker (Mlunta) is used to compute the certificates. We use reachability properties of the form $\mathbf{E}\Diamond$ *false* to enforce that the model checker explores the complete state space. The results are given in Table 1. The problem size is specified as the number of automata in the network. We report the total runtime (wall time) of:

1. the tandem consisting of Mlunta (using the first compression technique) and the (verified) certificate checker, both compiled with MLton;
2. the individual runtime of the (verified) certificate checker for a varying number of threads for parallel computation, compiled with Poly/ML as it is the only SML compiler that supports multi-threading;
3. the runtime of Uppaal configured for depth-first search (like Mlunta);
4. the runtime of an unverified SML implementation of the certificate checker based on Mlunta (compiled with MLton);
5. and the runtime of the fully verified model checker Munta [31] extended with the improvements from sections 2 and 4 and compiled with MLton.

As can be seen from the results, the tandem is still one order of magnitude slower than Uppaal, but certificate checking in isolation is also up to one order of magnitude faster than the previous verified model checker [31]. Note that Mlunta explores significantly more states than Uppaal and Munta for "Pacemaker". Multi-core scale beyond two threads is relatively unsatisfactory, however. In micro-benchmarks, we have identified that the problem appears to be with memory allocation on the heap, even if no data is shared among threads (in our case, only the certificate is shared but successors are computed locally). There does not seem to be an obvious way to improve on this situation for SML implementations. Finally, one can see that the verified certifier is not drastically slower than the unverified implementation based on Mlunta, indicating that the verified certifier is not missing any obvious significant optimizations.

						Certifier for #threads				
Model	Size	Uppaal	Tandem	Munta	Unverif.	1-MLton	1	2	3	4
FDDI	8	0.33	0.79	1.01	0.14	0.21	0.99	0.64	0.57	0.53
	10	5.93	1.77	2.50	0.40	0.45	2.13	1.36	1.26	1.20
	12	92.66	3.93	5.42	0.90	1.20	3.41	2.33	2.25	2.18
	14	1874.28	7.28	10.73	1.86	2.22	5.36	3.94	3.90	3.88
	16	***	12.80	19.51	3.47	3.72	11.09	6.49	6.50	6.51
FDDI broad	8	0.34	0.28	1.07	0.10	0.08	0.26	0.19	0.17	0.16
Fischer	5	0.24	2.78	6.33	0.72	0.53	1.76	1.07	0.98	0.91
	6	34.74	143.72	377.70	40.99	26.58	40.60	25.47	24.16	23.67
CSMA	5	0.04	0.94	4.42	0.31	0.28	1.44	0.87	0.80	0.76
	6	1.53	13.48	65.16	5.24	4.18	12.16	8.04	7.87	7.76
	Mode									
Pacemaker	1	0.02	0.16	0.37	0.03	0.03	0.25	0.16	0.14	0.13
	2	0.02	0.75	3.20	0.17	0.26	1.23	0.75	0.68	0.65
	3	0.03	1.39	4.23	0.34	0.46	2.53	1.55	1.40	1.31
	4	0.02	11.80	0.70	3.24	3.71	12.13	8.38	6.69	6.66
	5	0.02	30.84	0.86	9.13	10.07	26.60	18.58	18.15	17.95

Table 1: Benchmarks results on a machine with 16 GB RAM and an Intel(R) Core(TM) i7-4610M CPU at 3.00GHz with two cores and two threads per core. The column labeled "Tandem" gives the runtime for a combination of the unverified SML tool and the verified certificate checker. The next column gives the runtime of the unverified SML certifier, followed by the runtimes of the verified checker for a varying number of threads. All times are given in seconds.

Table 2 gives the results of evaluating the different compression algorithms on the same set of benchmarks. The second variant is always applied to the compression result of the first variant to avoid trivial computations of the convex hull. Variant 2c) (the most expensive one) can produce drastically smaller certificates than any other variant, and its minimum compression factor is an order of a magnitude higher than for any other variant. Nevertheless, only variants 1 and 2a) appear to be useful in practice, as they are relatively cheap to compute. The other variants could prove useful if the certificates were produced by a significantly more efficient model checker, such as Uppaal or TChecker [15]. On a final note, we have constructed a more than 95% smaller but valid certificate for the Fischer benchmark, suggesting that there is room for improvement on the compression algorithms.

		Variant					
Model	Size	1	2a	2b	2c	2d	2e
FDDI	8	0.21	0.21	1.72	69.53	3.65	3.43
FDDI broadcast	8	0.00	48.94	48.94	48.94	1.06	1.06
Fischer	5	22.03	22.03	22.72	43.06	30.40	30.40
CSMA/CD	5	26.06	41.54	43.84	81.16	58.94	47.54
	6	24.86	41.91	44.02	88.35	63.24	47.02
	Mode						
Pacemaker	1	16.07	25.00	30.80	58.04	29.02	29.02
	2	24.00	26.38	30.37	58.68	35.87	35.22
	3	12.96	17.62	19.23	46.92	25.30	25.01
	4	13.82	20.02	23.60	41.48	26.16	24.71
	5	17.14	22.48	25.46	39.69	28.18	26.88
Average		15.71	26.61	29.07	57.58	30.18	27.03

Table 2: Certificate compression factors (given in %).

6 Conclusion and Future Work

We have presented a verified certifier of unreachability certificates for a timed automata. The certificates are ought to be produced by an unverified model checker. Experimentation shows that verified certificate checking in isolation is up to an order of magnitude faster than what was previously possible with a verified model checker [31]. The performance of a tandem of an unverified model checker and the verified certifier could be improved by replacing the certificate-producing part with a highly optimized tool, possibly opening room to use some of the more powerful certificate compression techniques we suggested above. As we pointed out above, there appears to be further room for improvement on the certificate compression algorithms as well.

Moreover, more sophisticated tools also employ more powerful abstraction techniques, for which our proposed certification technique is still suitable—to a large extent without requiring additional verification effort. An exception is the implicit abstraction technique studied by Herbreteau et al. [14] as it does not compute abstractions of zones explicitly but rather checks subsumptions of the form $Z \subseteq \alpha(Z')$ implicitly, meaning that one would have to prove correctness of the subsumption check to validate certificates produced by such a model checking process.

Finally, we intend to extend this work to certification of emptiness of timed Büchi automata in the future, using the idea of *subsumption graphs* [13] and relying on an unverified model checker implementation for timed Büchi automata to produce the certificates [13,18].

Data Availability Statement

The datasets generated and/or analyzed during the current study are available in the Zenodo repository [32]: https://doi.org/10.5281/zenodo.3679245. The artifact has been tested on the TACAS artifact evaluation VM [12].

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. pp. 254–270. Springer Berlin Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_18
3. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. pp. 312–326. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_25
4. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Lectures on Concurrency and Petri Nets: Advances in Petri Nets. LNCS, vol. 3908, pp. 87–124. Springer (2004). https://doi.org/10.1007/978-3-540-27755-2_3
5. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: FSE 2016. p. 326–337. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2950290.2950351
6. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. Journal of Automated Reasoning **60**(1), 3–21 (2018). https://doi.org/10.1007/s10817-017-9418-4
7. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Theorem Proving in Higher Order Logics (TPHOLs 2008). pp. 134–149 (2008). https://doi.org/10.1007/978-3-540-71067-7_14
8. Castéran, P., Rouillard, D.: Towards a generic tool for reasoning about labeled transition systems. In: TPHOLs 2001: Supplemental Proceedings (2001)
9. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer (2013)
10. Garnacho, M., Bodeveix, J., Filali-Amine, M.: A mechanized semantic framework for real-time systems. In: FORMATS 2013. pp. 106–120. LNCS 8053 (2013). https://doi.org/10.1007/978-3-642-40229-6_8
11. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. 2018 Formal Methods in Computer Aided Design (FMCAD) pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8603022
12. Hartmanns, A., Seidl, M.: tacas20ae.ova (Aug 2019). https://doi.org/10.6084/m9.figshare.9699839.v2, https://figshare.com/articles/tacas20ae_ova/9699839/2
13. Herbreteau, F., Srivathsan, B., Tran, T.T., Walukiewicz, I.: Why liveness for timed automata is hard, and what we can do about it. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) FSTTCS 2016. LIPIcs, vol. 65, pp. 48:1–48:14. Schloss Dagstuhl (2016). https://doi.org/10.4230/LIPIcs.FSTTCS.2016.48

14. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. Information and Computation **251**, 67–90 (2016). https://doi.org/10.1016/j.ic.2016.07.004
15. Herbreteau, F., Point, G.: TChecker (2019), https://github.com/fredher/tchecker
16. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: SPIN 2014. p. 30–39. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2632362.2632372
17. Jiang, Z., Pajic, M., Moarref, S., Alur, R., Mangharam, R.: Modeling and verification of a dual chamber implantable pacemaker. In: Flanagan, C., König, B. (eds.) TACAS 2012. pp. 188–203. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_14
18. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV 2013. pp. 968–983. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_69
19. Lammich, P.: Collections framework. Archive of Formal Proofs (Nov 2009), http://isa-afp.org/entries/Collections.html, Formal proof development
20. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015, Proceedings. LNCS, vol. 9236, pp. 253–269. Springer (2015). https://doi.org/10.1007/978-3-319-22102-1_17
21. Launchbury, J., Peyton Jones, S.: Lazy functional state threads. PLDI 1998 **29** (07 1998). https://doi.org/10.1145/178243.178246
22. Möller, M.O.: Uppaal benchmarks (2017), https://www.it.uu.se/research/group/darts/uppaal/benchmarks
23. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. pp. 2–13. Springer Berlin Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_2
24. Neumann, R.: Using promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) Verified Software: Theories, Tools and Experiments. pp. 105–114. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_7
25. Nipkow, T., Lawrence C. Paulson, Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9
26. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: STACS 2001. pp. 298–315. LNCS 2215 (2001). https://doi.org/10.1007/3-540-45500-0_15
27. Peled, D., Pnueli, A., Zuck, L.: From falsification to verification. In: Hariharan, R., Vinay, V., Mukund, M. (eds.) FSTTCS 2001. pp. 292–304. Springer Berlin Heidelberg (2001). https://doi.org/10.1007/3-540-45294-X_25
28. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: TACAS 1998. pp. 167–183. Springer, London, UK (1998). https://doi.org/10.1007/BFb0054171
29. Wimmer, S.: Formalized timed automata. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 425–440. Springer (2016). https://doi.org/10.1007/978-3-319-43144-4_26
30. Wimmer, S.: Munta: A fully verified model checker for realtime systems. https://github.com/wimmers/munta (2019)
31. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4

32. Wimmer, S., von Mutius, J.: Artifact for "Verified Certification of Reachability Checking for Timed Automata" (Feb 2020). https://doi.org/10.5281/zenodo.3679245, https://doi.org/10.5281/zenodo.3679245
33. Xu, Q., Miao, H.: Formal verification framework for safety of real-time system based on timed automata model in PVS. In: Proc. of the IASTED International Conference on Software Engineering, 2006. pp. 107–112 (2006)
34. Xu, Q., Miao, H.: Manipulating clocks in timed automata using PVS. In: SNPD 2009. pp. 555–560 (2009). https://doi.org/10.1109/SNPD.2009.69

Learning One-Clock Timed Automata*

Jie An[1(✉)], Mingshuai Chen[2,3,4], Bohua Zhan[3,4], Naijun Zhan[3,4(✉)], and Miaomiao Zhang[1(✉)]

[1] School of Software Engineering, Tongji University, Shanghai, China
{1510796,miaomiao}@tongji.edu.cn
[2] Lehrstuhl für Informatik 2, RWTH Aachen University, Aachen, Germany
chenms@cs.rwth-aachen.de
[3] State Key Lab. of Computer Science, Institute of Software, CAS, Beijing, China
{bzhan,znj}@ios.ac.cn
[4] University of Chinese Academy of Sciences, Beijing, China

Abstract. We present an algorithm for active learning of deterministic timed automata with a single clock. The algorithm is within the framework of Angluin's L^* algorithm and inspired by existing work on the active learning of symbolic automata. Due to the need of guessing for each transition whether it resets the clock, the algorithm is of exponential complexity in the size of the learned automata. Before presenting this algorithm, we propose a simpler version where the teacher is assumed to be *smart* in the sense of being able to provide the reset information. We show that this simpler setting yields a polynomial complexity of the learning process. Both of the algorithms are implemented and evaluated on a collection of randomly generated examples. We furthermore demonstrate the simpler algorithm on the functional specification of the TCP protocol.

Keywords: Automaton learning · Active learning · One-clock timed automata · Timed language · Reset-logical-timed language.

1 Introduction

In her seminal work [10], Angluin introduced the L^* algorithm for learning a regular language from queries and counterexamples within a query-answering framework. The Angluin-style learning therefore is also termed *active learning* or *query learning*, which is distinguished from *passive learning*, i.e., generating a model from a given data set. Following this line of research, an increasing number of efficient active learning methods (cf. [38]) have been proposed to learn, e.g., Mealy machines [34,30], I/O automata [2], register automata [25,1,15], nondeterministic finite automata [12], Büchi automata [19,28], symbolic automata [29,18,11] and Markov decision processes [36], to name just a few. Full-fledged libraries, tools and applications are also available for automata-learning tasks [13,27,20,21].

* This work has been partially funded by NSFC under grant No. 61625206, 61972284, 61732001 and 61872341, by the ERC Advanced Project FRAPPANT under grant No. 787914, and by the CAS Pioneer Hundred Talents Program under grant No. Y9RC585036.

For real-time systems where timing constraints play a key role, however, learning a formal model is much more complicated. As a classical model for real-time systems, timed automata [4] have an infinite set of timed actions. This yields a fundamental difference to finite automata featuring finite alphabets. Moreover, it is difficult to detect resets of clock variables from observable behaviors of the system. This makes learning formal models of timed systems a challenging yet interesting problem.

Various attempts have been carried out in the literature on learning timed models, which can be classified into two tracks. The first track pursues active learning methods, e.g. [22] for learning event-recording automata (ERA) [5] and [9] for learning real-time automata (RTA) [17]. ERA are time automata where, for every untimed action a, a clock is used to record the time of the last occurrence of a. The underlying learning algorithm [22], however, is prohibitively complex due to too many degrees of freedom and multiple clocks for recording events. RTA are a class of special timed automata with one clock to record the execution time of each action by resetting at the starting. The other track pursues passive learning. In [42,41], an algorithm was proposed to learn deterministic RTA. The basic idea is that the learner organizes a tree sketching traces of the data set while merging nodes of the tree following a certain heuristic function. A passive learning algorithm for timed automata with one clock was further proposed in [39,40]. A common weakness of passive learning methods is that the generated model merely accepts all positive traces while it rejects all negative ones for the given set of traces, without guaranteeing that it is a correct model of the target system. A theoretical result was established in [40] showing it is possible to obtain the target system by continuously enriching the data set, however the number of iterations is unknown. In addition, the passive learning methods cited above concern only discrete-time semantics of the underlying timed models, i.e., the clock takes values from non-negative integers. We furthermore refer the readers to [14,32] for learning specialized forms of practical timed systems in a passive manner, [37] for passively learning timed automata using genetic programming which scales to automata of large sizes, [33] for learning probabilistic real-time automata incorporating clustering techniques in machine learning, and [36] for L^*-based learning of Markov decision processes with testing and sampling.

In this paper, we present the first active learning method for deterministic one-clock timed automata (DOTAs) under continuous-time semantics[1]. Such timed automata provide simple models while preserving adequate expressiveness, and therefore have been widely used in practical real-time systems [35,3,16]. We present our approach in two steps. First, we describe a simpler algorithm, under the assumption that the teacher is *smart* in the sense of being able to provide information about clock resets in membership and equivalence queries. The basic idea is as follows. We define the *reset-logical-timed language* of a DOTA and show that the timed languages of two DOTAs are equivalent if their reset-logical-timed languages are equivalent, which reduces the learning problem to that of learning a reset-logical-timed language.Then we show how to learn the reset-logical-timed language following Maler and D'Antoni's learning algorithms for symbolic automata [29,18]. We claim the correctness, termination and polynomial complexity of this learning algorithm. Next, we extend this algorithm to the case of a normal teacher. The main difference is that the learner now needs to *guess* the reset

[1] The proposed learning method applies trivially to discrete-time semantics too.

information on transitions discovered in the observation table. Due to these guesses, the latter algorithm features exponential complexity in the size of the learned automata. The proposed learning methods are implemented and evaluated on randomly generated examples. We also demonstrate the simpler, polynomial algorithm on a practical case study concerning the functional specification of the TCP protocol. Detailed proofs for theorems and lemmas in this paper can be found in Appendix A of the full version [7].

In what follows, Sect. 2 provides preliminary definitions on one-clock timed automata. The learning algorithm with a smart teacher is presented and analyzed in Sect. 3. We then present the situation with a normal teacher in Sect. 4. The experimental results are reported in Sect. 5. Finally, Sect. 6 concludes this paper.

2 Preliminaries

Let $\mathbb{R}_{\geq 0}$ and $\mathbb{N}$ be the set of non-negative reals and natural numbers, respectively, and $\mathbb{B}$ the Boolean set. We use $\top$ to stand for true and $\bot$ for false. The projection of an n-tuple $\mathbf{x}$ onto its first two components is denoted by $\Pi_{\{1,2\}}\mathbf{x}$, which extends to a sequence of tuples as $\Pi_{\{1,2\}}(\mathbf{x}_1, \ldots, \mathbf{x}_k) = \left(\Pi_{\{1,2\}}\mathbf{x}_1, \ldots, \Pi_{\{1,2\}}\mathbf{x}_k\right)$.

Timed automata [4], a kind of finite automata extended with a finite set of real-valued clocks, are widely used to model real-time systems. In this paper, we consider a subclass of timed automata with a single clock, termed *one-clock timed automata* (OTAs). Let c be the clock variable, denote by Φ_c the set of clock constraints of the form $\phi ::= \top \mid c \bowtie m \mid \phi \wedge \phi$, where $m \in \mathbb{N}$ and $\bowtie \in \{=, <, >, \leq, \geq\}$.

Definition 1 (One-clock timed automata). *A one-clock timed automaton $\mathcal{A} = (\Sigma, Q, q_0, F, c, \Delta)$, where Σ is a finite set of actions, called the* alphabet*; Q is a finite set of locations; $q_0 \in Q$ is the initial location; $F \subseteq Q$ is a set of accepting locations; c is the unique clock; and $\Delta \subseteq Q \times \Sigma \times \Phi_c \times \mathbb{B} \times Q$ is a finite set of transitions.*

A transition $\delta = (q, \sigma, \phi, b, q')$ allows a jump from the *source location* q to the *target location* q' by performing the action $\sigma \in \Sigma$ if the constraint $\phi \in \Phi_c$ is satisfied. Meanwhile, clock c is reset to zero if $b = \top$, and remains unchanged otherwise.

A *clock valuation* is a function $\nu \colon c \mapsto \mathbb{R}_{\geq 0}$ that assigns a non-negative real number to the clock. For $t \in \mathbb{R}_{\geq 0}$, let $\nu + t$ be the clock valuation with $(\nu + t)(c) = \nu(c) + t$. According to the definitions of clock valuation and clock constraint, a transition *guard* can be represented as an interval whose endpoints are in $\mathbb{N} \cup \{\infty\}$. For example, $\phi_1 \colon c < 5 \wedge c \geq 3$ is represented as $[3, 5)$, $\phi_2 \colon c = 6$ as $[6, 6]$, and $\phi_3 \colon \top$ as $[0, \infty)$. We will use the inequality- and interval-representation interchangeably in this paper.

A *state* s of $\mathcal{A}$ is a pair (q, ν), where $q \in Q$ and ν is a clock valuation. A *run* ρ of $\mathcal{A}$ is a finite sequence $\rho = (q_0, \nu_0) \xrightarrow{t_1, \sigma_1} (q_1, \nu_1) \xrightarrow{t_2, \sigma_2} \cdots \xrightarrow{t_n, \sigma_n} (q_n, \nu_n)$, where $\nu_0(c) = 0$, $t_i \in \mathbb{R}_{\geq 0}$ stands for the time delay spending on q_{i-1} before $\delta_i = (q_{i-1}, \sigma_i, \phi_i, b_i, q_i) \in \Delta$ is taken, only if (1) $\nu_{i-1} + t_i$ satisfies ϕ_i, (2) $\nu_i(c) = \nu_{i-1}(c) + t_i$ if $b_i = \bot$, otherwise $\nu_i(c) = 0$, for all $1 \leq i \leq n$. A run ρ is *accepting* if $q_n \in F$.

The *trace* of a run ρ is a timed word, denoted by *trace*(ρ). *trace*$(\rho) = \epsilon$ if $\rho = (q_0, \nu_0)$, and *trace*$(\rho) = (\sigma_1, t_1)(\sigma_2, t_2) \cdots (\sigma_n, t_n)$ if $\rho = (q_0, \nu_0) \xrightarrow{t_1, \sigma_1} (q_1, \nu_1) \xrightarrow{t_2, \sigma_2} \cdots \xrightarrow{t_n, \sigma_n} (q_n, \nu_n)$. Since t_i is the time delay on q_{i-1}, for $1 \leq i \leq n$, such a timed word is also called *delay-timed word*. The corresponding *reset-delay-timed word* can be defined as $\mathit{trace}_r(\rho) = (\sigma_1, t_1, b_1)(\sigma_2, t_2, b_2) \cdots (\sigma_n, t_n, b_n)$, where b_i is the reset indicator for δ_i, for $1 \le i \le n$. If ρ is an accepting run of $\mathcal{A}$, $\mathit{trace}(\rho)$ is called an *accepting timed word*. The *recognized timed language* of $\mathcal{A}$ is the set of accepting delay-timed words, i.e., $\mathcal{L}(\mathcal{A}) = \{\mathit{trace}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{A}\}$. The *recognized reset-timed language* $\mathcal{L}_r(\mathcal{A})$ is defined as $\{\mathit{trace}_r(\rho) \mid \rho \text{ is an accepting run of } \mathcal{A}\}$.

The delay-timed word $\omega = (\sigma_1, t_1)(\sigma_2, t_2) \cdots (\sigma_n, t_n)$ is observed outside, from the view of the global clock. On the other hand, the behavior can also be observed inside, from the view of the local clock. This results in a *logical-timed word* of the form $\gamma = (\sigma_1, \mu_1)(\sigma_2, \mu_2) \cdots (\sigma_n, \mu_n)$ with $\mu_i = t_i$ if $i = 1 \vee b_{i-1} = \top$ and $\mu_i = \mu_{i-1} + t_i$ otherwise. We will denote the mapping from delay-timed words to logical-timed words above by Γ.

Similarly, we introduce *reset-logical-timed word* $\gamma_r = (\sigma_1, \mu_1, b_1)(\sigma_2, \mu_2, b_2) \cdots (\sigma_n, \mu_n, b_n)$ as the counterpart of $\omega_r = (\sigma_1, t_1, b_1)(\sigma_2, t_2, b_2) \cdots (\sigma_n, t_n, b_n)$ in terms of the local clock. Without any substantial change, we can extend the mapping Γ to map reset-delay-timed words to reset-logical-timed words. The *recognized logical-timed language* of $\mathcal{A}$ is given as $L(\mathcal{A}) = \{\Gamma(\mathit{trace}(\rho)) \mid \rho \text{ is an accepting run of } \mathcal{A}\}$, and the *recognized reset-logical-timed language* of $\mathcal{A}$ as $L_r(\mathcal{A}) = \{\Gamma(\mathit{trace}_r(\rho)) \mid \rho \text{ is an accepting run of } \mathcal{A}\}$.

An OTA is a *deterministic one-clock timed automaton* (DOTA) if there is at most one run for a given delay-timed word. In other words, for any location $q \in Q$ and action $\sigma \in \Sigma$, the guards of transitions outgoing from q labelled with σ are disjoint subsets of $\mathbb{R}_{\ge 0}$. We say a DOTA is *complete* if for any of its location $q \in Q$ and action $\sigma \in \Sigma$, the corresponding guards form a partition of $\mathbb{R}_{\ge 0}$. This means any given delay-timed word has exactly one run. Any DOTA $\mathcal{A}$ can be transformed into a complete DOTA (referred to as COTA) $\mathbb{A}$ accepting the same timed language as follows: (1) Augment Q with a “sink” location q_s which is not an accepting location; (2) For every $q \in Q$ and $\sigma \in \Sigma$, if there is no outgoing transition from q labelled with σ, introduce a (resetting) transition from q to q_s with label σ and guard $[0, \infty)$; (3) Otherwise, let S be the subset of $\mathbb{R}_{\ge 0}$ not covered by the guards of transitions from q with label σ. Write S as a union of intervals $I_1, \ldots, I_k$ in a minimal way, then introduce a (resetting) transition from q to q_s with label σ and guard I_j for each $1 \le j \le k$.

From now on, we therefore assume that we are working with COTAs.

Example 1. Fig. 1 depicts the transformation of a DOTA $\mathcal{A}$ (left part) into a COTA $\mathbb{A}$ (right part). First, a non-accepting “sink” location q_s is introduced. Second, we introduce three fresh transitions (marked in blue) from q_1 to q_s as well as transitions from q_s to itself. At last, for location q_0 and label a, the existing guards cover $(1, 3)$, with complement $[0, 1] \cup [3, \infty)$. Hence, we introduce transitions $(q_0, a, [0, 1], \top, q_s)$ and $(q_0, a, [3, \infty), \top, q_s)$. Two fresh transitions from q_1 to q_s are introduced similarly.

3 Learning from a Smart Teacher

In this section, we consider the case of learning a COTA $\mathbb{A}$ with a smart teacher. Our learning algorithm relies on the following reduction of the equivalence over timed languages to that of reset-logical timed languages.

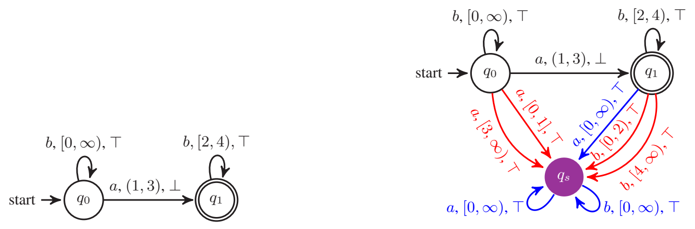


Fig. 1: A DOTA $\mathcal{A}$ on the left and the corresponding COTA $\mathbb{A}$ on the right. The initial location is indicated by 'start' and an accepting location is doubly circled.

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

Theorem 1 assures that $L_r(\mathcal{H}) = L_r(\mathbb{A})$ implies $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathbb{A})$, that is, to construct a COTA $\mathbb{A}$ that recognizes a target timed language $\mathcal{L} = \mathcal{L}(\mathbb{A})$, it suffices to learn a *hypothesis* $\mathcal{H}$ which recognizes the same reset-logical timed language. For equivalence queries, instead of checking directly whether $L_r(\mathcal{H}) = L_r(\mathbb{A})$, the contraposition of Theorem 1 guarantees that we can perform equivalence queries over their timed counterparts: if $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathbb{A})$, then $\mathcal{H}$ recognizes the target language already; otherwise, a counterexample making $\mathcal{L}(\mathcal{H}) \neq \mathcal{L}(\mathbb{A})$ yields an evidence also for $L_r(\mathcal{H}) \neq L_r(\mathbb{A})$.

We now describe the behavior of the teacher who keeps an automaton $\mathbb{A}$ to be learnt, while providing knowledge about the automaton by answering membership and equivalence queries through an oracle she maintains. For the membership query, the teacher receives a logical-timed word γ and returns whether γ is in $L(\mathbb{A})$. In addition, she is smart enough to return the reset-logical-timed word γ_r that corresponds to γ (the exact correspondence is described in Sect. 3.1). For the equivalence query, the teacher is given a hypothesis $\mathcal{H}$ and decides whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathbb{A})$. If not, she is smart enough to return a reset-delayed-timed word ω_r as a counterexample. The usual case where a teacher can deal with only standard delay-timed words will be discussed in Sect. 4.

Remark 1. The assumption that the teacher can respond with timed words coupled with reset information is reasonable, in the sense that the learner can always infer and detect the resets of the logical clock by referring to a global clock on the wall, as long as he can observe running states of $\mathbb{A}$, i.e., observing the clock valuation of the system whenever an event happens therein. This conforms with the idea of combining automata learning with white-box techniques, as exploited in [24], providing that in many application scenarios source code is available for the analysis.

In what follows, we elaborate the learning procedure including membership queries, hypotheses construction, equivalence queries and counterexample processing.

3.1 Membership query

In our setting, the oracle maintained by the smart teacher can be regarded as a COTA $\mathbb{A}$ that recognizes the target timed language $\mathcal{L}$, and thereby its logical-timed language $L(\mathbb{A})$ and reset-logical-timed counterpart $L_r(\mathbb{A})$. In order to collect enough information for constructing a hypothesis, the learner makes membership queries as "Is the logical-timed word γ in $L(\mathbb{A})$?". If there does not exist a run ρ such that $\Gamma(\mathit{trace}(\rho)) = \gamma$, meaning that there is some k such that the run is blocked after the k'th action (i.e. γ is *invalid*) and hence the teacher gives a negative answer, associated with a reset-logical-timed word γ_r where all b_i's with $i > k$ are set to $\top$; If there exists a run ρ (which is unique due to the determinacy of $\mathbb{A}$) that admits γ (i.e., γ is *valid*), the teacher answers "Yes", if ρ is accepting, or "No" otherwise, while in both cases providing the corresponding reset-logical-timed word γ_r, with $\Pi_{\{1,2\}}\gamma_r = \gamma$.

For the sake of simplicity, we define a function π that maps a logical-timed word to its unique reset-logical-timed counterpart in membership queries. Information gathered from the membership queries is stored in a timed observation table defined as follows.

Definition 2 (Timed observation table). *A timed observation table for a COTA $\mathbb{A}$ is a 7-tuple $\mathbf{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$ where Σ is the finite alphabet; $\boldsymbol{\Sigma} = \Sigma \times \mathbb{R}_{\geq 0}$ is the infinite set of logical-timed actions; $\boldsymbol{\Sigma_r} = \Sigma \times \mathbb{R}_{\geq 0} \times \mathbb{B}$ is the infinite set of reset-logical-timed actions; $\boldsymbol{S}, \boldsymbol{R} \subset \boldsymbol{\Sigma_r}^*$ and $\boldsymbol{E} \subset \boldsymbol{\Sigma}^*$ are finite sets of words, where $\boldsymbol{S}$ is called the set of prefixes, $\boldsymbol{R}$ the boundary, and $\boldsymbol{E}$ the set of suffixes. Specifically,*

- *$\boldsymbol{S}$ and $\boldsymbol{R}$ are disjoint, i.e., $\boldsymbol{S} \cup \boldsymbol{R} = \boldsymbol{S} \uplus \boldsymbol{R}$;*
- *The empty word is by default both a prefix and a suffix, i.e., $\epsilon \in \boldsymbol{E}$ and $\epsilon \in \boldsymbol{S}$;*
- *$f\colon (\boldsymbol{S} \cup \boldsymbol{R}) \cdot \boldsymbol{E} \mapsto \{-, +\}$ is a classification function such that for a reset-logical-timed word γ_r, $\gamma_r \cdot e \in (\boldsymbol{S} \cup \boldsymbol{R}) \cdot \boldsymbol{E}$, $f(\gamma_r \cdot e) = -$ if $\Pi_{\{1,2\}}\gamma_r \cdot e$ is invalid, otherwise if $\Pi_{\{1,2\}}\gamma_r \cdot e \notin L(\mathbb{A})$, $f(\gamma_r \cdot e) = -$, and $f(\gamma_r \cdot e) = +$ if $\Pi_{\{1,2\}}\gamma_r \cdot e \in L(\mathbb{A})$.*

Given a table $\mathbf{T}$, we define $\mathit{row}\colon \boldsymbol{S} \cup \boldsymbol{R} \mapsto (\boldsymbol{E} \mapsto \{+, -\})$ as a function mapping each $\gamma_r \in \boldsymbol{S} \cup \boldsymbol{R}$ to a vector indexed by $e \in \boldsymbol{E}$, each of whose components is defined as $f(\gamma_r \cdot e)$, denoting a potential location.

Before constructing a hypothesis $\mathcal{H}$ based on the timed observation table $\mathbf{T}$, the learner has to ensure that $\mathbf{T}$ satisfies the following conditions:

- *Reduced:* $\forall s, s' \in \boldsymbol{S}\colon s \neq s'$ implies $\mathit{row}(s) \neq \mathit{row}(s')$;
- *Closed:* $\forall r \in \boldsymbol{R}, \exists s \in \boldsymbol{S}\colon \mathit{row}(s) = \mathit{row}(r)$;
- *Consistent:* $\forall \gamma_r, \gamma_r' \in \boldsymbol{S} \cup \boldsymbol{R}$, $\mathit{row}(\gamma_r) = \mathit{row}(\gamma_r')$ implies $\mathit{row}(\gamma_r \cdot \boldsymbol{\sigma_r}) = \mathit{row}(\gamma_r' \cdot \boldsymbol{\sigma_r}')$, for all $\boldsymbol{\sigma_r}, \boldsymbol{\sigma_r}' \in \boldsymbol{\Sigma_r}$ satisfying $\gamma_r \cdot \boldsymbol{\sigma_r}, \gamma_r' \cdot \boldsymbol{\sigma_r}' \in \boldsymbol{S} \cup \boldsymbol{R}$ and $\Pi_{\{1,2\}}\boldsymbol{\sigma_r} = \Pi_{\{1,2\}}\boldsymbol{\sigma_r}'$;
- *Evidence-closed:* $\forall s \in \boldsymbol{S}$ and $\forall e \in \boldsymbol{E}$, the reset-logical-timed word $\pi(\Pi_{\{1,2\}} s \cdot e)$ belongs to $\boldsymbol{S} \cup \boldsymbol{R}$;
- *Prefix-closed:* $\boldsymbol{S} \cup \boldsymbol{R}$ is prefix-closed.

A timed observation table $\mathbf{T}$ is *prepared* if it satisfies the above five conditions. To get the table prepared, the learner can perform the following operations:

Making $\mathbf{T}$ *closed.* If $\mathbf{T}$ is not closed, there exists $r \in \boldsymbol{R}$ such that for all $s \in \boldsymbol{S}$ $\mathit{row}(r) \neq \mathit{row}(s)$. The learner thus can move such r from $\boldsymbol{R}$ to $\boldsymbol{S}$. Moreover, each reset-logical-timed word $\pi(\Pi_{\{1,2\}} r \cdot \boldsymbol{\sigma})$ needs to be added to $\boldsymbol{R}$, where $\boldsymbol{\sigma} = (\sigma, 0)$ for all $\sigma \in \Sigma$. Such an operation is important since it guarantees that at every location all actions in Σ are enabled, while specifying a clock valuation of these actions, despite that some invalid logical-timed words might be involved. Particularly, giving a bottom value 0 as the clock valuation satisfies the precondition of the partition functions that will be described in Sect. 3.2.

Making $\mathbf{T}$ *consistent.* If $\mathbf{T}$ is not consistent, one inconsistency is resolved by adding $\boldsymbol{\sigma} \cdot e$ to $\boldsymbol{E}$, where $\boldsymbol{\sigma}$ and e can be determined as follows. T being inconsistent implies that there exist two reset-logical-timed words $\gamma_r, \gamma_r{}' \in \boldsymbol{S} \cup \boldsymbol{R}$ at least, such that $\gamma_r \cdot \boldsymbol{\sigma_r}, \gamma_r{}' \cdot \boldsymbol{\sigma_r}{}' \in \boldsymbol{S} \cup \boldsymbol{R}$ and $\Pi_{\{1,2\}}\boldsymbol{\sigma_r} = \Pi_{\{1,2\}}\boldsymbol{\sigma_r}{}'$ for some $\boldsymbol{\sigma_r}, \boldsymbol{\sigma_r}{}' \in \boldsymbol{\Sigma_r}$, with $row(\gamma_r) = row(\gamma_r{}')$ but $row(\gamma_r \cdot \boldsymbol{\sigma_r}) \neq row(\gamma_r{}' \cdot \boldsymbol{\sigma_r}{}')$. So, let $\boldsymbol{\sigma} = \Pi_{\{1,2\}}\boldsymbol{\sigma_r} = \Pi_{\{1,2\}}\boldsymbol{\sigma_r}{}'$ and $e \in \boldsymbol{E}$ such that $f(\gamma_r\boldsymbol{\sigma_r} \cdot e) \neq f(\gamma_r{}'\boldsymbol{\sigma_r}{}' \cdot e)$. Thereafter, the learner fills the table by making membership queries. Note that this operation keeps the set $\boldsymbol{E}$ of suffixes being a set of logical-timed words.

Making $\mathbf{T}$ *evidence-closed.* If $\mathbf{T}$ is not evidence-closed, then the learner needs to add all prefixes of $\pi(\Pi_{\{1,2\}}s \cdot e)$ to $\boldsymbol{R}$ for every $s \in \boldsymbol{S}$ and $e \in \boldsymbol{E}$, except those already in $\boldsymbol{S} \cup \boldsymbol{R}$. Similarly, the learner needs to fill the table through membership queries.

The condition that a timed observation table $\mathbf{T}$ is reduced and prefix-closed is inherently preserved by the aforementioned operations, together with the counterexample processing described later in Sect. 3.3. Furthermore, a table may need several rounds of these operations before being prepared (cf. Algorithm 1), since certain conditions may be violated by different, interleaved operations.

3.2 Hypothesis construction

As soon as the timed observation table $\mathbf{T}$ is prepared, a hypothesis can be constructed in two steps, i.e., the learner first builds a DFA M based on the information in $\mathbf{T}$, and then transforms M to a hypothesis $\mathcal{H}$, which will later be shown as a COTA.

Given a prepared timed observation table $\mathbf{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$, a DFA $\mathrm{M} = (Q_M, \Sigma_M, \Delta_M, q_M^0, F_M)$ can be built as follows:

- the finite set of locations $Q_M = \{q_{row(s)} \mid s \in \boldsymbol{S}\}$;
- the initial location $q_M^0 = q_{row(\epsilon)}$ for $\epsilon \in \boldsymbol{S}$;
- the set of accepting locations $F_M = \{q_{row(s)} \mid f(s \cdot \epsilon) = + \text{ for } s \in \boldsymbol{S} \text{ and } \epsilon \in \boldsymbol{E}\}$;
- the finite alphabet $\Sigma_M = \{\boldsymbol{\sigma_r} \in \boldsymbol{\Sigma_r} \mid \gamma_r \cdot \boldsymbol{\sigma_r} \in \boldsymbol{S} \cup \boldsymbol{R} \text{ for } \gamma_r \in \boldsymbol{\Sigma_r^*}\}$;
- the finite set of transitions $\Delta_M = \{(q_{row(\gamma_r)}, \boldsymbol{\sigma_r}, q_{row(\gamma_r \cdot \boldsymbol{\sigma_r})}) \mid \gamma_r \cdot \boldsymbol{\sigma_r} \in \boldsymbol{S} \cup \boldsymbol{R} \text{ for } \gamma_r \in \boldsymbol{\Sigma_r^*} \text{ and } \boldsymbol{\sigma_r} \in \boldsymbol{\Sigma_r}\}$.

The constructed DFA M is compatible with the timed observation table $\mathbf{T}$ in the sense captured by the following lemma.

Lemma 1. *For a prepared timed observation table* $\mathbf{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$, *for every* $\gamma_r \cdot e \in (\boldsymbol{S} \cup \boldsymbol{R}) \cdot \boldsymbol{E}$, *the constructed DFA* $\mathrm{M} = (Q_M, \Sigma_M, \Delta_M, q_M^0, F_M)$ *accepts* $\pi(\Pi_{\{1,2\}}\gamma_r \cdot e)$ *if and only if* $f(\gamma_r \cdot e) = +$.

The learner then transforms the DFA M to a hypothesis $\mathcal{H} = (\Sigma, Q, q_0, F, c, \Delta)$, with $Q = Q_M$, $q_0 = q_M^0$, $F = F_M$, c being the clock and Σ the given alphabet as in $\mathbf{T}$. The set of transitions Δ in $\mathcal{H}$ can be constructed as follows: For any $q \in Q_M$ and $\sigma \in \Sigma$, let $\Psi_{q,\sigma} = \{\mu \mid (q, (\sigma, \mu, b), q') \in \Delta_M\}$, then applying the partition function $P^c(\cdot)$ (defined below) to $\Psi_{q,\sigma}$ returns k intervals, written as $I_1, \cdots, I_k$, satisfying $\mu_i \in I_i$ for any $1 \leq i \leq k$, where $k = |\Psi_{q,\sigma}|$; consequently, for every $(q, (\sigma, \mu_i, b_i), q') \in \Delta_M$, a fresh transition $\delta_i = (q, \sigma, I_i, b_i, q')$ is added to Δ.

$\mathbf{T_5}$	ϵ
ϵ	$-$
$(a, 1.1, \bot)$	$+$
$(a, 0, \top)$	$-$
$(b, 0, \top)$	$-$
$(a, 1.1, \bot)(a, 0, \top)$	$-$
$(a, 1.1, \bot)(b, 0, \top)$	$-$
$(a, 1.1, \bot)(b, 2, \top)$	$+$
$(a, 3, \top)$	$-$

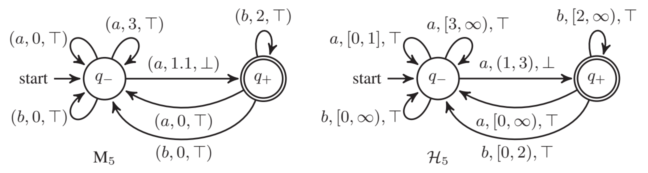


Fig. 2: The prepared timed observation table $\mathbf{T_5}$, the corresponding DFA M_5 and hypothesis $\mathcal{H}_5$.

Definition 3 (Partition function). *Given a list of clock valuations $\ell = \mu_0, \mu_1, \cdots, \mu_n$ with $0 = \mu_0 < \mu_1 \cdots < \mu_n$, and $\lfloor \mu_i \rfloor \neq \lfloor \mu_j \rfloor$ if $\mu_i, \mu_j \in \mathbb{R}_{\geq 0} \setminus \mathbb{N}$ and $i \neq j$ for all $1 \leq i, j \leq n$, let $\mu_{n+1} = \infty$, then a partition function $P^c(\cdot)$ mapping ℓ to a set of intervals $\{I_0, I_1, \ldots, I_n\}$, which is a partition of $\mathbb{R}_{\geq 0}$, is defined as*

$$I_i = \begin{cases} [\mu_i, \mu_{i+1}), & \textit{if } \mu_i \in \mathbb{N} \wedge \mu_{i+1} \in \mathbb{N}; \\ (\lfloor \mu_i \rfloor, \mu_{i+1}), & \textit{if } \mu_i \in \mathbb{R}_{\geq 0} \setminus \mathbb{N} \wedge \mu_{i+1} \in \mathbb{N}; \\ [\mu_i, \lfloor \mu_{i+1} \rfloor], & \textit{if } \mu_i \in \mathbb{N} \wedge \mu_{i+1} \in \mathbb{R}_{\geq 0} \setminus \mathbb{N}; \\ (\lfloor \mu_i \rfloor, \lfloor \mu_{i+1} \rfloor], & \textit{if } \mu_i \in \mathbb{R}_{\geq 0} \setminus \mathbb{N} \wedge \mu_{i+1} \in \mathbb{R}_{\geq 0} \setminus \mathbb{N}. \end{cases}$$

Remark 2. Definition 3 is adapted from that in [18] by imposing additional assumptions of the list of clock valuations in order to guarantee $\mu_i \in I_i$, for any $0 \leq i \leq n$, due to the underlying continuous-time semantics. Whereas, by $\mathbf{T}$ being prepared and the normalization function described in Sect. 3.3, the set of clock valuations $\Psi_{q,\sigma}$ can be arranged into a list $\ell_{q,\sigma} = \mu_0, \mu_1, \ldots, \mu_n$ satisfying such assumptions given in Definition 3 for any $q \in Q_M$ and $\sigma \in \Sigma$.

Example 2. Suppose $\mathbb{A}$ in Fig. 1 recognizes the target timed language. Then the prepared table $\mathbf{T_5}$, the corresponding DFA M_5 and hypothesis $\mathcal{H}_5$ are depicted in Fig. 2. Here, the subscript 5 indicates the fifth iteration of $\mathbf{T}$ (Details concerning the constructions and the entire learning process are enclosed in Appendix B of [7].).

Lemma 2. *Given a DFA* $\mathrm{M} = (Q_M, \Sigma_M, \delta_M, q_M^0, F_M)$*, which is generated from a prepared timed observation table* $\mathbf{T}$*, the hypothesis* $\mathcal{H} = (\Sigma, Q, q_0, F, c, \Delta)$ *is transformed from M. For all* $\gamma_r \cdot e \in (\boldsymbol{S} \cup \boldsymbol{R}) \cdot \boldsymbol{E}$*,* $\mathcal{H}$ *accepts the reset-logical-timed word* $\pi(\Pi_{\{1,2\}} \gamma_r \cdot e)$ *iff* $f(\gamma_r \cdot e) = +$.

Theorem 2. *The hypothesis $\mathcal{H}$ is a COTA.*

Given a clock valuation μ, we denote the *region* containing μ as $[\![\mu]\!]$, defined as $[\![\mu]\!] = [\mu, \mu]$ if $\mu \in \mathbb{N}$, and $[\![\mu]\!] = (\lfloor \mu \rfloor, \lfloor \mu \rfloor + 1)$ otherwise. The following theorem establishes the compatibility of the constructed hypothesis $\mathcal{H}$ with the timed observation table $\mathbf{T}$.

Theorem 3. *For* $\gamma_r \cdot e \in (\boldsymbol{S} \cup \boldsymbol{R}) \cdot \boldsymbol{E}$*, let* $\pi(\Pi_{\{1,2\}} \gamma_r \cdot e) = (\sigma_1, \mu_1, b_1) \cdots (\sigma_n, \mu_n, b_n)$*. Then for every* $\mu_i' \in [\![\mu_i]\!]$*, the hypothesis* $\mathcal{H}$ *accepts the reset-logical-timed word* $\gamma_r' = (\sigma_1, \mu_1', b_1) \cdots (\sigma_n, \mu_n', b_n)$ *if* $f(\gamma_r \cdot e) = +$*, and cannot accept it if* $f(\gamma_r \cdot e) = -$.

3.3 Equivalence query and counterexample processing

Suppose that the teacher knows a COTA $\mathbb{A}$ which recognizes the target timed language $\mathcal{L}$. Then to answer an equivalence query is to determine whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathbb{A})$, which can be divided into two timed language inclusion problems, i.e., whether $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathbb{A})$ and $\mathcal{L}(\mathbb{A}) \subseteq \mathcal{L}(\mathcal{H})$. Most decision procedures for language inclusion proceed by complementation and emptiness checking of the intersection [23]: $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ iff $\mathcal{L}(A) \cap \overline{\mathcal{L}(B)} = \emptyset$. The fact that deterministic timed automata can be complemented [6] enables solving the inclusion problem by checking the emptiness of the resulted product automata $\mathcal{H} \times \overline{\mathbb{A}}$ and $\overline{\mathcal{H}} \times \mathbb{A}$. The complementation technique, however, does not apply to nondeterministic timed automata even if with only one single clock [4], which we plan to incorporate in our learning framework in future work. We therefore opt for[2] the alternative method presented by Ouaknine and Worrell in [31] showing that the language inclusion problem of timed automata with one clock (regardless of their determinacy) is decidable by reduction to a reachability problem on an infinite graph. That is, there exists a delay-timed word ω that leads to a *bad configuration* if $\mathcal{L}(\mathcal{H}) \nsubseteq \mathcal{L}(\mathbb{A})$. In detail, the corresponding run ρ of ω ends in an accepting location in $\mathcal{H}$ but the counterpart ρ' of ω in $\mathbb{A}$ is not accepting. Consequently, the teacher can provide the reset-delay-timed word ω_r resulted from ω as a negative counterexample ctx_-. Similarly, a positive counterexample $ctx_+ = (\omega_r, +)$ can be generated if $\mathcal{L}(\mathbb{A}) \nsubseteq \mathcal{L}(\mathcal{H})$. An algorithm elaborating the equivalence query is provided in Appendix C of the full version [7].

When receiving a counterexample $ctx = (\omega_r, +/-)$, the learner first converts it to a reset-logical-timed word $\gamma_r = \Gamma(\omega_r) = (\sigma_1, \mu_1, b_1)(\sigma_2, \mu_2, b_2) \cdots (\sigma_n, \mu_n, b_n)$. By definition, γ_r and ω_r share the same sequence of transitions in $\mathbb{A}$. Furthermore, by the contraposition of Theorem 1, γ_r is an evidence for $L_r(\mathcal{H}) \neq L_r(\mathbb{A})$ if ω_r is an evidence for $\mathcal{L}(\mathcal{H}) \neq \mathcal{L}(\mathbb{A})$.

Additionally, by the definition of clock constraints Φ_c, at any location, if an action σ is enabled, i.e., its guard is satisfied, w.r.t. the clock value $\mu \in \mathbb{R}_{\geq 0} \setminus \mathbb{N}$, then σ should be enabled w.r.t. any clock value $\lfloor \mu \rfloor + \theta$ at the location, where $\theta \in (0, 1)$. Specifically, only one transition is available for σ at the location on the interval $[\![\mu]\!]$, because the target automaton is deterministic. Therefore, in order to avoid unnecessarily distinguishing timed words and violating the assumptions of the list ℓ for the partition function, the learner needs to apply a *normalization function* g to normalize γ_r.

Definition 4 (Normalization). *A normalization function g maps a reset-logical-timed word $\gamma_r = (\sigma_1, \mu_1, b_1)(\sigma_2, \mu_2, b_2) \cdots (\sigma_n, \mu_n, b_n)$ to another reset-logical-timed word by resetting any logical clock to its integer part plus a constant fractional part, i.e., $g(\gamma_r) = (\sigma_1, \mu'_1, b_1)(\sigma_2, \mu'_2, b_2) \cdots (\sigma_n, \mu'_n, b_n)$, where $\mu'_i = \mu_i$ if $\mu_i \in \mathbb{N}$, $\mu'_i = \lfloor \mu_i \rfloor + \theta$ for some fixed constant $\theta \in (0, 1)$ otherwise.*

We will instantiate $\theta = 0.1$ in what follows. Clearly our approach works for any other θ valued in $(0, 1)$. This *normalization* process guarantees the assumptions needed for Definition 3.

[2] Remark that the learning complexity (Sect. 3.5) is measured in terms of the number of queries rather than the time complexity of the specific method for checking the equivalence (nor membership). Additionally, the specific method of equivalence checking is not the main concern.

$\mathbf{T}_5$	ϵ
ϵ	$-$
$(a,1.1,\bot)$	$+$
$(a,0,\top)$	$-$
$(b,0,\top)$	$-$
$(a,1.1,\bot)(a,0,\top)$	$-$
$(a,1.1,\bot)(b,0,\top)$	$-$
$(a,1.1,\bot)(b,2,\top)$	$+$
$(a,3,\top)$	$-$

$$\xRightarrow[\gamma_r=(a,0,\top)(a,1.3,\top)]{\omega_r=(a,0,\top)(a,1.3,\top),-}$$

$\mathbf{T}_6$	ϵ
ϵ	$-$
$(a,1.1,\bot)$	$+$
$(a,0,\top)$	$-$
$(b,0,\top)$	$-$
$(a,1.1,\bot)(a,0,\top)$	$-$
$(a,1.1,\bot)(b,0,\top)$	$-$
$(a,1.1,\bot)(b,2,\top)$	$+$
$(a,3,\top)$	$-$
$(a,0,\top)(a,1.3,\top)$	$-$

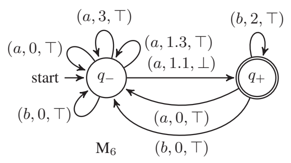


Fig. 3: An illustration of the necessity of normalization by the normalization function.

Algorithm 1: Learning one-clock timed automaton with a smart teacher

```
input : the timed observation table $\mathbf{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$.
output: the hypothesis $\mathcal{H}$ recognizing the target language $\mathcal{L}$.
$\boldsymbol{S} \leftarrow \{\epsilon\}$; $\boldsymbol{R} \leftarrow \{\Gamma(\omega) \mid \omega = (\sigma, 0), \forall \sigma \in \Sigma\}$; $\boldsymbol{E} \leftarrow \{\epsilon\}$ ;      // initialization
fill $\mathbf{T}$ by membership queries;
$equivalent \leftarrow \bot$;
while $equivalent = \bot$ do
    $prepared \leftarrow$ is_prepared($\mathbf{T}$) ;      // whether the table is prepared
    while $prepared = \bot$ do
        if $\mathbf{T}$ is not closed then make_closed($\mathbf{T}$) ;
        if $\mathbf{T}$ is not consistent then make_consistent($\mathbf{T}$) ;
        if $\mathbf{T}$ is not evidence-closed then make_evidence_closed($\mathbf{T}$) ;
        $prepared \leftarrow$ is_prepared($\mathbf{T}$);
    M $\leftarrow$ build_DFA($\mathbf{T}$) ;      // transforming $\mathbf{T}$ to a DFA M
    $\mathcal{H} \leftarrow$ build_hypothesis(M) ;      // constructing a hypothesis $\mathcal{H}$ from M
    $equivalent, ctx \leftarrow$ equivalence_query($\mathcal{H}$);
    if $equivalent = \bot$ then
        ctx_processing($\mathbf{T}$, $ctx$) ;      // counterexample processing
return $\mathcal{H}$;
```

Example 3. Consider the prepared table $\mathbf{T}_5$ in Fig. 3 (as in Fig. 2). When the leaner asks an equivalence query with hypothesis $\mathcal{H}_5$, the teacher answers that $\mathcal{L}(\mathcal{H}_5) \neq \mathcal{L}(\mathbb{A})$, where $\mathbb{A}$ in Fig. 1 is the target automaton, and provides a counterexample $(\omega_r, -)$ with $\omega_r = (a,0,\top)(a,1.3,\top)$, which can be transformed to a reset-logical-timed word $\gamma_r = (a,0,\top)(a,1.3,\top)$. If he adds prefixes of γ_r to the table directly, the learner will get a prepared table $\mathbf{T}_6$ and thus construct a DFA M_6. Unfortunately, the partition function defined in Definition 3 is not applicable to $(a,1.3,\top)$ and $(a,1.1,\bot)$ any more. On the other hand, if he adds the prefixes of the normalized reset-logical-timed word, i.e., $\gamma_r' = (a,0,\top)(a,1.1,\top)$, to $\mathbf{T}_5$, the learner will then get an inconsistent table whose consistency can be retrieved by the operation of "making $\mathbf{T}$ consistent" as expected.

The following theorem guarantees that the normalized reset-logical-timed word γ_r' is also an evidence for $L_r(\mathcal{H}) \neq L_r(\mathbb{A})$. Therefore, the learner can use it as a counterexample and thus adds all the prefixes of γ_r' to $\boldsymbol{R}$ except those already in $\boldsymbol{S} \cup \boldsymbol{R}$.

Theorem 4. *Given a valid reset-logical-timed word γ_r of $\mathbb{A}$, its normalization $\gamma_r' = g(\gamma_r)$ shares the same sequence of transitions in $\mathbb{A}$.*

3.4 Learning algorithm

We present in Algorithm 1 the learning procedure integrating all the previously stated ingredients, including preparing the table, membership and equivalence queries, hypothesis construction and counterexample processing. The learner first initializes the timed observation table $\mathbf{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$, where $\boldsymbol{S} = \{\epsilon\}$, $\boldsymbol{E} = \{\epsilon\}$, while for every $\sigma \in \Sigma$, he builds a logical-timed word $\gamma = (\sigma, 0)$ and then obtains its reset counterpart $\pi(\gamma) = (\sigma, 0, b)$ by triggering a membership query to the teacher, which is then added to $\boldsymbol{R}$. Thereafter, the learner can fill the table by additional membership queries. Before constructing a hypothesis, the learner performs several rounds of operations described in Sect. 3.1 until $\mathbf{T}$ is prepared. Then, a hypothesis $\mathcal{H}$ is constructed leveraging an intermediate DFA M and submitted to the teacher for an equivalence query. If the answer is positive, $\mathcal{H}$ recognizes the target language. Otherwise, the learner receives a counterexample ctx and then conducts the counterexample processing to update $\mathbf{T}$ as described in Sect. 3.3. The whole procedure repeats until the teacher gives a positive answer to an equivalence query.

To facilitate the analysis of correctness, termination and complexity of Algorithm 1, we introduce the notion of *symbolic state* that combines each location with its clock regions: a symbolic state of a COTA $\mathbb{A} = (\Sigma, Q, q_0, F, c, \Delta)$ is a pair $(q, [\![\mu]\!])$, where $q \in Q$ and $[\![\mu]\!]$ is a region containing μ. If κ is the maximal constant appearing in the clock constraints of $\mathbb{A}$, then there exist $2\kappa + 2$ such regions, including $[n, n]$ with $0 \leq n \leq \kappa$, $(n, n+1)$ with $0 \leq n < \kappa$, and (κ, ∞) for each location, so there are a total of $|Q| \times (2\kappa + 2)$ symbolic states. Then the correctness and termination of Algorithm 1 is stated in the following theorem, based on the fact that there is an injection from $\boldsymbol{S}$ (or equivalently, the locations of $\mathcal{H}$) to symbolic states of $\mathbb{A}$.

Theorem 5. *Algorithm 1 terminates and returns a COTA $\mathcal{H}$ which recognizes the target timed language $\mathcal{L}$.*

3.5 Complexity

Given a target timed language $\mathcal{L}$ which is recognized by a COTA $\mathbb{A}$, let $n = |Q|$ be the number of locations of $\mathbb{A}$, $m = |\Sigma|$ the size of the alphabet, and κ the maximal constant appearing in the clock constraints of $\mathbb{A}$. In what follows, we derive the complexity of Algorithm 1 in terms of the number of queries.

By the proof of Theorem 5, $\mathcal{H}$ has at most $n(2\kappa + 2)$ locations (the size of $\boldsymbol{S}$) distinguished by $\boldsymbol{E}$. Thus, $|\boldsymbol{E}|$ is at most $n(2\kappa + 2)$ in order to distinguish these locations. Therefore, the number of transitions of $\mathcal{H}$ is bounded by $mn^2(2\kappa + 2)^3$. Furthermore, as every counterexample adds at least one fresh transition to the hypothesis $\mathcal{H}$, where we consider each interval of the partition corresponds to a transition, this means that the number of counterexamples and equivalence queries is at most $mn^2(2\kappa + 2)^3$.

Now, we consider the number of membership queries, that is, to compute $(|\boldsymbol{S}| + |\boldsymbol{R}|) \times |\boldsymbol{E}|$. Let h be the maximal length of counterexamples returned by the teacher, which is polynomial in the size of $\mathbb{A}$ according to Theorem 5 in [40], bounded by n^2. There are three cases of extending $\boldsymbol{R}$ by adding fresh rows, namely during the processing of counterexamples, making $\mathbf{T}$ closed, and making $\mathbf{T}$ evidence-closed. The first case adds at most $hmn^2(2\kappa + 2)^3$ rows to $\boldsymbol{R}$, while the latter two add at most $n(2\kappa + 2) \times m$ and $n^2(2\kappa + 2)^2$, respectively, yielding that the size of $\boldsymbol{R}$ is bounded by $\mathcal{O}(hmn^2\kappa^3)$, where $\mathcal{O}(\cdot)$ is the big Omicron notation. As a consequence, the number of membership queries is bounded by $\mathcal{O}(mn^5\kappa^4)$. So, the total complexity is $\mathcal{O}(mn^5\kappa^4)$.

It is worth noting the above analysis is given in the worst case, where all partitions need to be fully refined. But, in practice we can learn the automaton without refining most partitions, and therefore the number of equivalence and membership queries, as well as the number of locations in the learned automaton are much fewer than the corresponding worst-case bounds. This will be demonstrated by examples in Sect. 5.

3.6 Accelerating Trick

In the timed observation table, the function f maps invalid reset-logical-timed words as well as certain valid ones to "$-$" when the teacher maintains a COTA $\mathbb{A}$ as the oracle. The learner thus needs multiple rounds of queries to distinguish the "sink" location from other unaccepting locations. If the function f is extended to map invalid reset-logical-timed words to a distinct symbol, say "$\times$", when we let a DOTA $\mathcal{A}$ be the oracle, then the learner will take much fewer queries. We will later show in the experiments that such a trick significantly accelerates the learning process.

4 Learning from a Normal Teacher

In this section, we consider the problem of learning timed automata with a normal teacher. As before, we assume the timed language to be learned comes from a complete DOTA. For the normal teacher, inputs to membership queries are delay-timed words, and the teacher returns whether the word is in the language (without giving any additional information). Inputs to equivalence queries are candidate DOTAs, and the teacher either answers they are equivalent or provides a delay-timed word as a counterexample.

The algorithm here is based on the procedure in the previous section. We still maintain observation tables where the elements in $\boldsymbol{S} \cup \boldsymbol{R}$ are reset-logical-timed words and the elements in $\boldsymbol{E}$ are logical-timed words. In order to obtain delay-timed words for the membership queries, we need to *guess* clock reset information for transitions in the table. More precisely, in order to convert a logical-timed word to a delay-timed word, it is necessary to know clock reset information for all but the last transition. Hence, it is necessary to guess reset information for each word in $\boldsymbol{S} \cup \boldsymbol{R}$ (since $\boldsymbol{S} \cup \boldsymbol{R}$ is prefix-closed, this is equivalent to guessing reset information for the last transition of each word). Also, for each entry in $(\boldsymbol{S} \cup \boldsymbol{R}) \times \boldsymbol{E}$, it is necessary to guess all but the last transition in $\boldsymbol{E}$. The algorithm can be thought of as exploring a search tree, where branching is caused by guesses, and successor nodes are constructed by the usual operations of preparing a table and dealing with a counterexample.

The detailed process is given in Algorithm 2. The learner maintains a set of table instances, named $\mathit{ToExplore}$, which contains all table instances that need to be explored.

The initial tables in $\mathit{ToExplore}$ are as follows. Each table has $\boldsymbol{S} = \boldsymbol{E} = \{\epsilon\}$. For each $\sigma \in \Sigma$, there is one row in $\boldsymbol{R}$ corresponding to the logical-timed word $\omega = (\sigma, 0)$. It is necessary to guess a reset b for each ω thereby transforming it to a reset-logical-timed word $\gamma_r = (\sigma, 0, b)$. There are $2^{|\Sigma|}$ possible combinations of guesses. These tables are filled by making membership queries (in this case, the membership queries for each table are the same). The resulting $2^{|\Sigma|}$ tables form the initial tables in $\mathit{ToExplore}$.

Algorithm 2: Learning one-clock timed automaton with a normal teacher

```
input  : the timed observation table $\mathbf{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$.
output: the hypothesis $\mathcal{H}$ recognizing the target language $\mathcal{L}$.
$ToExplore \leftarrow \emptyset$; $\boldsymbol{S} \leftarrow \{\epsilon\}$; $\boldsymbol{R} \leftarrow \{\pi(\Gamma(\omega)) \mid \omega = (\sigma, 0), \forall \sigma \in \Sigma\}$; $\boldsymbol{E} \leftarrow \{\epsilon\}$;
$currentTable \leftarrow (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$;
$tables \leftarrow$ guess_and_fill($currentTable$);   // guess resets and fill all table instances
$ToExplore$.insert($tables$);                    // insert table instances $tables$ into $ToExplore$
$currentTable \leftarrow ToExplore$.pop();      // pop out head instance as the current table
$equivalent \leftarrow \bot$;
while $equivalent = \bot$ do
    $prepared \leftarrow$ is_prepared($currentTable$);  // whether the current table is prepared
    while $prepared = \bot$ do
        if $currentTable$ is not closed then
            $tables \leftarrow$ guess_and_make_closed($currentTable$); $ToExplore$.insert($tables$);
            $currentTable \leftarrow ToExplore$.pop();
        if $currentTable$ is not consistent then
            $tables \leftarrow$ guess_and_make_consistent($currentTable$); $ToExplore$.insert($tables$);
            $currentTable \leftarrow ToExplore$.pop();
        if $currentTable$ is not evidence-closed then
            $tables \leftarrow$ guess_and_make_evidence_closed($currentTable$); $ToExplore$.insert($tables$);
            $currentTable \leftarrow ToExplore$.pop();
        $prepared \leftarrow$ is_prepared($currentTable$);
    M $\leftarrow$ build_DFA($currentTable$) ;          // transforming $currentTable$ to a DFA M
    $\mathcal{H} \leftarrow$ build_hypothesis(M) ;            // constructing a hypothesis $\mathcal{H}$ from M
    $equivalent$, $ctx \leftarrow$ equivalence_query($\mathcal{H}$);           // $ctx$ is a delay-timed word
    if $equivalent = \bot$ then
        $tables \leftarrow$ guess_and_ctx_processing($currentTable$, $ctx$) ;        // counterexample processing
        $ToExplore$.insert($tables$);
        $currentTable \leftarrow ToExplore$.pop();
return $\mathcal{H}$;
```

In each iteration of the algorithm, one table instance is taken out of $ToExplore$. The learner checks whether the table is closed, consistent, and evidence closed. If the table is not closed, i.e. there exists $r \in \boldsymbol{R}$ such that $row(r) \neq row(s)$ for all $s \in \boldsymbol{S}$, the learner moves r from $\boldsymbol{R}$ to $\boldsymbol{S}$. Then for each $\sigma \in \Sigma$, the element $r \cdot (\sigma, 0)$ is added to $\boldsymbol{R}$, and a guess has to be made for its reset information. Hence, $2^{|\Sigma|}$ unfilled table instances will be generated. Next, for each entry in the $|\Sigma|$ new rows of $\boldsymbol{R}$, it is necessary to guess reset information for all but the last transition in $e \in \boldsymbol{E}$. After this guess, it is now possible to fill the table instances by making membership queries with transformed delay-timed words. Hence, there are at most $2^{(\sum_{e_i \in \boldsymbol{E} \setminus \{\epsilon\}} (|e_i|-1)) \times |\Sigma|}$ filled table instances for one unfilled table instance. All filled table instances are inserted into $ToExplore$.

If the table is not consistent, i.e. there exist some $\gamma_r, \gamma_r' \in \boldsymbol{S} \cup \boldsymbol{R}$ and $\boldsymbol{\sigma_r} \in \boldsymbol{\Sigma_r}$ such that $\gamma_r \cdot \boldsymbol{\sigma_r}, \gamma_r' \cdot \boldsymbol{\sigma_r} \in \boldsymbol{S} \cup \boldsymbol{R}$ and $row(\gamma_r) = row(\gamma_r')$, but $row(\gamma_r \cdot \boldsymbol{\sigma_r}) \neq row(\gamma_r' \cdot \boldsymbol{\sigma_r})$. Let $e \in \boldsymbol{E}$ be one place where they are different. Then $\boldsymbol{\sigma_r} \cdot e$ needs to be added to $\boldsymbol{E}$. For each entry in $\boldsymbol{S} \cup \boldsymbol{R}$, all but the last transition in $\boldsymbol{\sigma_r} \cdot e$ need to be guessed, then the table can be filled. $2^{(|\boldsymbol{\sigma} \cdot e|-1) \times (|\boldsymbol{S}|+|\boldsymbol{R}|)}$ filled table instances will be generated and inserted into $ToExplore$. The operation for making tables evidence-closed is analogous.

Once the current table is prepared, the learner builds a hypothesis $\mathcal{H}$ and makes an equivalence query to the teacher. If the answer is positive, then $\mathcal{H}$ is a COTA which recognizes the target timed language $\mathcal{L}$; otherwise, the teacher gives a delay-timed word ω as a counterexample. The learner first finds the longest reset-logical-timed word in

$\boldsymbol{R}$ which, when converted to a delay-timed word, agrees with a prefix of ω. The remainder of ω, however, needs to be converted to a reset-logical-timed word by guessing reset information. The corresponding prefixes are then added to $\boldsymbol{R}$. Hence, at most $2^{|\omega|}$ unfilled table instances are generated. For each unfilled table instance, at most $2^{(\sum_{e_i \in \boldsymbol{E}\setminus\{\epsilon\}}(|e_i|-1))\times|\omega|}$ filled tables are produced and inserted into *ToExplore*.

Throughout the learning process, the learner adds a finite number of table instances to *ToExplore* at every iteration. Hence, the search tree is finite-branching. Moreover, if all guesses are correct, the resulting table instance will be identical to the observation table in the learning process with a smart teacher (apart from the guessing processes, the basic table operations are the same as those in Section 3.1). This means, with an appropriate search order, for example, taking the table instance that requires the least number of guesses in *ToExplore* at every iteration, the algorithm terminates and returns the same table as in the learning process with a smart teacher, which is a COTA that recognizes the target language $\mathcal{L}$. In conformity to Theorem 1, the algorithm may terminate even if the corresponding reset-logical-timed languages are not equivalent, while yielding correct COTAs recognizing the same delay-timed language.

Theorem 6. *Algorithm 2 terminates and returns a COTA $\mathcal{H}$ which recognizes the target timed language $\mathcal{L}$.*

Complexity analysis. If $\mathbf{T} = (\Sigma, \boldsymbol{\Sigma}, \boldsymbol{\Sigma_r}, \boldsymbol{S}, \boldsymbol{R}, \boldsymbol{E}, f)$ is the final observation table for the correct candidate COTA, the number of guessed resets in $\boldsymbol{S}\cup\boldsymbol{R}$ is $|\boldsymbol{S}|+|\boldsymbol{R}|$, and the number of guessed resets for entries in each row of the table is $\sum_{e_i\in\boldsymbol{E}\setminus\{\epsilon\}}(|e_i|-1)$. Hence, the total number of guessed resets is $(|\boldsymbol{S}|+|\boldsymbol{R}|)\times(1+\sum_{e_i\in\boldsymbol{E}\setminus\{\epsilon\}}(|e_i|-1))$. Assuming an appropriate search order (for example according to the number of guesses in each table), this yields the number of table instances considered before termination as $\mathcal{O}(2^{(|\boldsymbol{S}|+|\boldsymbol{R}|)\times(1+\sum_{e_i\in\boldsymbol{E}\setminus\{\epsilon\}}(|e_i|-1))})$.

5 Implementation and Experimental Results

To investigate the efficiency and scalability of the proposed methods, we implemented a prototype[3] in PYTHON for learning deterministic one-clock timed automata. The examples include a practical case concerning the functional specification of the TCP protocol [26] and a set of randomly generated DOTAs to be learnt. All of the evaluations have been carried out on a 3.6GHz Intel Core-i7 processor with 8GB RAM running 64-bit Ubuntu 16.04.

Functional specification of the TCP protocol. In [26], a state diagram on page 23 specifies state changes during a TCP connection triggered by causing events while leading to resulting actions. As observed by Ouaknine and Worrell in [31], such a functional specification of the protocol can be represented as a one-clock timed automaton. In our setting, the corresponding DOTA $\mathcal{A}$ to be learnt is configured to have $|Q| = 11$ states with the two `CLOSED` states collapsed, $|\Sigma| = 10$ after abstracting the causing events and the resulting actions, $|F| = 2$, and $|\Delta| = 19$ with appropriately specified timing constraints including guards and resets. Using the algorithm with the smart teacher, a

[3] Available at https://github.com/Leslieaj/OTALearning. The evaluated artifact is archived in [8].

Table 1: Experimental results on random examples for the smart teacher situation.

Case ID	$\|\Delta\|_{\text{mean}}$	#Membership			#Equivalence			n_{mean}	t_{mean}
		N_{min}	N_{mean}	N_{max}	N_{min}	N_{mean}	N_{max}		
4_4_20	16.3	118	245.0	650	20	30.1	42	4.5	24.7
7_2_10	16.9	568	920.8	1393	23	31.3	37	9.1	14.6
7_4_10	25.7	348	921.7	1296	34	50.9	64	9.3	38.0
7_6_10	26.0	351	634.5	1050	35	44.7	70	7.8	49.6
7_4_20	34.3	411	1183.4	1890	52	70.5	93	9.5	101.7
10_4_20	39.1	920	1580.9	2160	61	73.1	88	11.7	186.7
12_4_20	47.6	1090	2731.6	5733	66	97.4	125	16.0	521.8
14_4_20	58.4	1390	2238.6	4430	79	107.7	135	16.0	515.5

Case ID: n_m_κ, consisting of the number of locations, the size of the alphabet and the maximum constant appearing in the clock constraints, respectively, of the corresponding group of $\mathcal{A}$'s.
$|\Delta|_{\text{mean}}$: the average number of transitions in the corresponding group.
#Membership & #Equivalence: the number of conducted membership and equivalence queries, respectively. N_{min}: the minimal, N_{mean}: the mean, N_{max}: the maximum.
n_{mean}: the average number of locations of the learned automata in the corresponding group.
t_{mean}: the average wall-clock time in seconds, including that taken by the learner and the teacher.

correct DOTA $\mathcal{H}$ is learned in 155 seconds after 2600 membership queries and 28 equivalence queries. Specifically, $\mathcal{H}$ has 15 locations excluding a sink location connected by 28 transitions. The introduction of 4 new locations comes from splitting of guards along transitions, which however can be trivially merged back with other locations. The figures depicting $\mathcal{A}$ and $\mathcal{H}$ can be found in Appendix D of [7].

Random examples for a smart teacher. We randomly generated 80 DOTAs in eight groups, with each group having different numbers of locations, size of alphabet, and maximum constant appearing in clock constraints. As shown in Table 1, the proposed learning method succeeds in all cases in identifying a DOTA that recognizes the same timed language. In particular, the number of membership queries and that of equivalence queries appear to grow polynomially with the size of the problem[4], and are much smaller than the worst-case bounds estimated in Sect. 3.5. Moreover, the learned DOTAs do not have prominent increases in the number of locations (by comparing n_{mean} with the first component of Case IDs). The average wall-clock time including both time taken by the learner and by the teacher is recorded in the last column t_{mean}, of which, however, often over 90% is spent by the teacher for checking equivalences w.r.t. small $\mathbf{T}$'s while around 50% by the learner for checking the preparedness condition w.r.t. large $\mathbf{T}$'s.

It is worth noting that all of the results reported above are carried out on an implementation equipped with the *accelerating trick* discussed in Sect. 3.6. We remark that when *dropping* this trick, the average number of membership queries blow up with a factor of 0.83 (min) to 15.02 (max) with 2.16 in average for all the 8 groups, and 0.84 (min) to 1.71 (max) with 1.04 for the average number of equivalence queries, leading to dramatic increases also in the computation time (including that in operating tables).

[4] An exception w.r.t. the group 7_6_10 is due to relatively simple DOTAs generated occasionally.

Table 2: Experimental results on random examples for the normal teacher situation.

Case ID	$\lvert\Delta\rvert_{\text{mean}}$	#Membership			#Equivalence			n_{mean}	t_{mean}	$\#\mathbf{T}_{explored}$	#Learnt
		$N_{\min}$	N_{mean}	$N_{\max}$	$N_{\min}$	N_{mean}	$N_{\max}$				
3_2_10	4.8	43	83.7	167	5	8.8	14	3.0	0.9	149.1	10/10
4_2_10	6.8	67	134.0	345	6	13.3	24	4.0	7.4	563.0	10/10
5_2_10	8.8	75	223.9	375	9	15.2	24	5.0	35.5	2811.6	10/10
6_2_10	11.9	73	348.3	708	10	16.7	30	5.6	59.8	5077.6	7/10
4_4_20	16.3	231	371.0	564	27	30.9	40	4.0	137.5	8590.0	6/10

#Membership & #Equivalence: the number of conducted membership and equivalence queries with the cached methods, respectively. $N_{\min}$: the minimal, N_{mean}: the mean, $N_{\max}$: the maximum.
$\#\mathbf{T}_{explored}$: the average number of the explored table instances.
#Learnt: the number of the learnt DOTAs in the group (learnt/total).

The alternative implementation and experimental results without the accelerating trick can also be found in the tool page (under the `dev` branch).

Random examples for a normal teacher. Due to its high, exponential complexity, the algorithm with a normal teacher failed (out of memory) in identifying DOTAs for almost all the above examples, except 6 cases out of the 10 in group 4_4_20. We therefore randomly generated 40 extra DOTAs of smaller size classified into 4 groups. With the accelerating trick, the learner need not guess the resets in elements of $\boldsymbol{E}$ for an entry in $\boldsymbol{S} \cup \boldsymbol{R}$ if the querying result of the entry is the sink location. We also omitted the checking of the evidence-closed condition, since it may add redundant rows in $\boldsymbol{R}$, leading to more guesses and thereby a larger search space. The omission does not affect the correctness of the learnt DOTAs. Moreover, as different table instances may generate repeated queries, we cached the results of membership queries and counterexamples, such that the numbers of membership and equivalence queries to the teacher can be significantly reduced. Table 2 shows the performance of the algorithm in this setting. Results without caching are available in the tool page (under the `normal` branch).

6 Conclusion

We have presented a polynomial active learning method for deterministic one-clock timed automata from a smart teacher who can tell information about clock resets in membership and equivalence queries. Our technique is based on converting the problem to that of learning reset-logical-timed languages. We then extend the method to learning DOTAs from a normal teacher who receives delay-timed words for membership queries, while the learner guesses the reset information in the observation table. We evaluate both algorithms on randomly generated examples and, for the former case, the functional specification of the TCP protocol.

Moving forward, an extension of our active learning method to nondeterministic OTAs and timed automata involving multiple clocks is of particular interest.

Data Availability Statement The datasets generated and/or analyzed during the current study are available in the Figshare repository: https://doi.org/10.6084/m9.figshare.11545983.v3.

References

1. Aarts, F., Fiterau-Brostean, P., Kuppens, H., Vaandrager, F.W.: Learning register automata with fresh value generation. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) ICTAC 2015. LNCS, vol. 9399, pp. 165–183. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-319-25150-9_11
2. Aarts, F., Vaandrager, F.W.: Learning I/O automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_6
3. Abdullah, J., Dai, G., Mohaqeqi, M., Yi, W.: Schedulability analysis and software synthesis for graph-based task models with resource sharing. In: Proceedings of 24th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2018. pp. 261–270. IEEE Computer Society (2018)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
5. 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)
6. Alur, R., Madhusudan, P.: Decision problems for timed automata: A survey. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 1–24. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_1
7. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata (full version). arXiv:1910.10680 (2019), https://arxiv.org/abs/1910.10680
8. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. Figshare (2020), https://doi.org/10.6084/m9.figshare.11545983.v3
9. An, J., Wang, L., Zhan, B., Zhan, N., Zhang, M.: Learning real-time automata. SCIENCE CHINA Information Sciences (2020). https://doi.org/10.1007/s11432-019-2767-4, to appear.
10. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
11. 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, Heidelberg (2018). https://doi.org/10.1007/978-3-319-96145-3_23
12. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-style learning of NFA. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009. pp. 1004–1009. AAAI Press (2009)
13. Bollig, B., Katoen, J.P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: The automata learning framework. In: Touili, T., Cook, B., Jackson, P.B. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_32
14. Caldwell, B., Cardell-Oliver, R., French, T.: Learning time delay Mealy machines from programmable logic controllers. IEEE Trans. Automation Science and Engineering **13**(2), 1155–1164 (2016)
15. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–263 (2016)
16. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. Communications of the ACM **24**(8), 533–536 (1981)
17. Dima, C.: Real-time automata. Journal of Automata, Languages and Combinatorics **6**(1), 3–23 (2001)
18. 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

19. Farzan, A., Chen, Y., Clarke, E.M., Tsay, Y., Wang, B.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_2
20. Fiterau-Brostean, P., Janssen, R., Vaandrager, F.W.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 454–471. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-41540-6_25
21. Fiterau-Brostean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F.W., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, SPIN 2017. pp. 142–151. ACM (2017)
22. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. Theor. Comput. Sci. **411**(47), 4029–4054 (2010)
23. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company (1979)
24. Howar, F., Jonsson, B., Vaandrager, F.W.: Combining black-box and white-box techniques for learning register automata. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, LNCS, vol. 10000, pp. 563–588. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_26
25. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 251–266. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_17
26. Information Science Institute, University of Southern California: Transmission control protocol (DARPA internet program protocol specification). https://www.rfc-editor.org/rfc/rfc793.txt (1981)
27. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib - A framework for active automata learning. In: Kroening, D., Pasareanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-319-21690-4_32
28. Li, Y., Chen, Y., Zhang, L., Liu, D.: A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 208–226. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_12
29. Maler, O., Mens, I.: Learning regular languages over large alphabets. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 485–499. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_41
30. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: Proceedings of the 9th IEEE International High-Level Design Validation and Test Workshop, HLDVT 2004. pp. 95–100. IEEE Computer Society (2004)
31. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: Closing a decidability gap. In: Proceedings of the 19th IEEE Symposium on Logic in Computer Science, LICS 2004. pp. 54–63. IEEE Computer Society (2004)
32. Pastore, F., Micucci, D., Mariani, L.: Timed k-Tail: Automatic inference of timed automata. In: Proceedings of 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017. pp. 401–411. IEEE Computer Society (2017)
33. Schmidt, J., Ghorbani, A., Hapfelmeier, A., Kramer, S.: Learning probabilistic real-time automata from multi-attribute event logs. Intell. Data Anal. **17**(1), 93–123 (2013)
34. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009. LNCS, vol. 5850, pp. 207–222. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_14

35. Stigge, M., Ekberg, P., Guan, N., Yi, W.: The digraph real-time task model. In: Proceedings of 17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011. pp. 71–80. IEEE Computer Society (2011)
36. Tappler, M., Aichernig, B.K., Bacci, G., Eichlseder, M., Larsen, K.G.: L^*-based learning of Markov decision processes. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 651–669. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-30942-8_38
37. 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, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29662-9_13
38. Vaandrager, F.W.: Model learning. Communications of the ACM **60**(2), 86–95 (2017)
39. Verwer, S., de Weerdt, M., Witteveen, C.: One-clock deterministic timed automata are efficiently identifiable in the limit. In: Dediu, A., Ionescu, A., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 740–751. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00982-2_63
40. Verwer, S., de Weerdt, M., Witteveen, C.: The efficiency of identifying timed automata and the power of clocks. Information and Computation **209**(3), 606–625 (2011)
41. Verwer, S., de Weerdt, M., Witteveen, C.: Efficiently identifying deterministic real-time automata from labeled data. Machine Learning **86**(3), 295–333 (2012)
42. Verwer, S., Weerdt, M.D., Witteveen, C.: An algorithm for learning real-time automata. In: Proceedings of the 18th Annual Machine Learning Conference of Belgium and the Netherlands, Benelearn 2007. pp. 57–64 (2007)

Rare event simulation for non-Markovian repairable fault trees[☆]

Carlos E. Budde[1], Marco Biagi[2], Raúl E. Monti[1], Pedro R. D'Argenio[3,4,5], and Mariëlle Stoelinga[1,6]

[1] Formal Methods and Tools, University of Twente, Enschede, the Netherlands
[2] Department of Information Engineering, University of Florence, Florence, Italy
[3] FAMAF, Universidad Nacional de Córdoba, Córdoba, Argentina
[4] CONICET, Córdoba, Argentina
[5] Department of Computer Science, Saarland University, Saarbrücken, Germany
[6] Department of Software Science, Radboud University, Nijmegen, the Netherlands

Abstract. Dynamic fault trees (DFT) are widely adopted in industry to assess the dependability of safety-critical equipment. Since many systems are too large to be studied numerically, DFTs dependability is often analysed using Monte Carlo simulation. A bottleneck here is that many simulation samples are required in the case of rare events, e.g. in highly reliable systems where components fail seldomly. Rare event simulation (RES) provides techniques to reduce the number of samples in the case of rare events. We present a RES technique based on importance splitting, to study failures in highly reliable DFTs. Whereas RES usually requires meta-information from an expert, our method is fully automatic: By cleverly exploiting the fault tree structure we extract the so-called importance function. We handle DFTs with Markovian and non-Markovian failure and repair distributions—for which no numerical methods exist—and show the efficiency of our approach on several case studies.

1 Introduction

Reliability engineering is an important field that provides methods and tools to assess and mitigate the risks related to complex systems. Fault tree analysis (FTA) is a prominent technique here. Its application encompasses a large number of industrial domains that range from automotive and aerospace system engineering, to energy and telecommunication systems and protocols.

Fault trees. A fault tree (FT) describes how component failures occur and propagate through the system, eventually leading to system failures. Technically, an FT is a directed acyclic graph whose leaves model component failures, and whose other nodes (called gates) model failure propagation. Using fault trees one can compute dependability metrics to quantify how a system fares w.r.t.

[☆] This work was partially funded by NWO, NS, and ProRail project 15474 (*SEQUOIA*), ERC grant 695614 (*POWVER*), EU project 102112 (*SUCCESS*), ANPCyT PICT-2017-3894 (*RAFTSys*), and SeCyT project 33620180100354CB (*ARES*).

© The Author(s) 2020
A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 463–482, 2020.
https://doi.org/10.1007/978-3-030-45190-5_26

certain performance indicators. Two common metrics are system *reliability*—the probability that there are no system failures during a given mission time—and system *availability*—the average percentage of time that a system is operational.

In this paper we consider repairable dynamic fault trees. *Dynamic fault trees* (DFTs [17, 43]) are a common and widely applied variant of FTs, catering for common dependability patterns such as spare management and causal dependencies. *Repairs* [6] are not only crucial in fault-tolerant and resilient systems, they are also an important cost driver. Hence, repairable fault trees allow one to compare different repair strategies with respect to various dependability metrics.

Fault tree analysis. The reliability/availability of a fault tree can be computed via numerical methods, such as probabilistic model checking. This involves exhaustive explorations of state-based models such as interactive Markov chains [40]. Since the number of states (i.e. system configurations) is exponential in the number of tree elements, analysing large trees remains a challenge today [26, 1]. Moreover, numerical methods are usually restricted to exponential failure rates and combinations thereof, like Erlang and acyclic phase type distributions [40].

Alternatively, fault trees can be analysed using (standard) Monte Carlo simulation (SMC [22, 40, 38], aka statistical model checking). Here, a large number of simulated system runs (*samples*) is produced. Reliability and availability are then statistically estimated from the resulting sample set. Such sampling does not involve storing the full state space so, although the result provided can only be correct with a certain probability, SMC is much more memory efficient than numerical techniques. Furthermore, SMC is not restricted to exponential probability distributions. However, a known bottleneck of SMC are rare events: when the event of interest has a low probability (which is typically the case in highly reliable systems), millions of samples may be required to observe it. Producing these samples can take a unacceptably long simulation time.

Rare event simulation. To alleviate this problem, the field of rare event simulation (RES) provides techniques that reduce the number of samples [35]. The two leading techniques are importance sampling and importance splitting.

Importance sampling tweaks the probabilities in a model, then computes the metric of interest for the changed system, and finally adjusts the analysis results to the original model [23, 33]. Unfortunately it has specific requirements on the stochastic model: in particular, it is generally limited to Markov models.

Importance splitting, deployed in this paper, does not have this limitation. Importance splitting relies on rare events that arise as a sequence of less rare intermediate events [28, 2]. We exploit this fact by generating more (partial) samples on paths where such intermediate events are observed. As a simple example, consider a biased coin whose probability of heads is $p = 1/80$. Suppose we flip it eight times in a row, and say we are interested in observing at least three heads. If heads comes up at the first flip (H) then we are on a promising path. We can then clone (*split*) the current path H, generating e.g. 7 copies of it, each clone evolving independently from the second flip onwards. Say one clone observes three heads—the copied H plus two more. Then, this observation of the rare event (three heads) is counted as $1/7$ rather than as 1 observation, to account for the splitting where the clone was spawned. Now, if a clone observes a new head (HH), this is even more promising than H, so the splitting can be repeated. If we make 5 copies of the HH clone, then observing three heads in any of these copies counts as $\frac{1}{35} = \frac{1}{7} \cdot \frac{1}{5}$. Alternatively, observing tails as second flip (HT) is less promising than heads. One could then decide not to split such path.

This example highlights a key ingredient of importance splitting: the *importance function*, that indicates for each state how promising it is w.r.t. the event of interest. This function, together with other parameters such as thresholds [19], are used to choose e.g. the number of clones spawned when visiting a state. An importance function for our example could be the number of heads seen thus far. Another one could be such number, multiplied by the number of coin flips yet to come. The goal is to give *higher importance* to states from which observing the *rare event is more likely*. The efficiency of an importance splitting implementation increases as the importance function better reflects such property.

Rare event simulation has been successfully applied in several domains [34, 45, 49, 4, 5, 46]. However, a key bottleneck is that it critically relies on expert knowledge. In particular for importance splitting, finding a good importance function is a well-known highly non-trivial task [35, 25].

Our contribution: rare event simulation for fault trees. This paper presents an importance splitting method to analyse RFTs. In particular, we automatically derive an importance function by exploiting the description of a system as a fault tree. This is crucial, since the importance function is normally given manually in an ad hoc fashion by a domain or RES expert. We use a variety of RES algorithms based in our importance function, to estimate system unreliability and unavailability. Our approach can converge to precise estimations in increasingly reliable systems. This method has four advantages over earlier analysis methods for RFTs—which we overview in the related work section 6—namely: (1) we are able to estimate both the system reliability and availability; (2) we can handle arbitrary failure and repair distributions; (3) we can handle rare events; and (4) we can do it in a fully automatic fashion.

Technically, we build local importance functions for the (automata-semantics of the) nodes of the tree. We then aggregate these local functions into an importance function for the full tree. Aggregation uses structural induction in the layered description of the tree. Using our importance function, we implement importance splitting methods to run RES analyses. We implemented our theory in a full-stack tool chain. With it, we computed confidence intervals for the unreliability and unavailability of several case studies. Our case studies are RFTs whose failure and repair times are governed by arbitrary continuous probability density functions (PDFs). Each case study was analysed for a fixed runtime budget and in increasingly resilient configurations. In all cases our approach could estimate the narrowest intervals for the most resilient configurations.

Paper outline. Background on fault trees and RES is provided in Secs. 2 and 3. We detail our theory to implement RES for RFTs in Sec. 4. Using a tool chain, we performed an extensive experimental evaluation that we present in Sec. 5. We overview related work in Sec. 6 and conclude our contributions in Sec. 7.

2 Fault tree analysis

A fault tree '△' is a directed acyclic graph that models how component failures propagate and eventually cause the full system to fail. We consider repairable fault trees (RFTs), where failures and repairs are governed by arbitrary probability distributions.

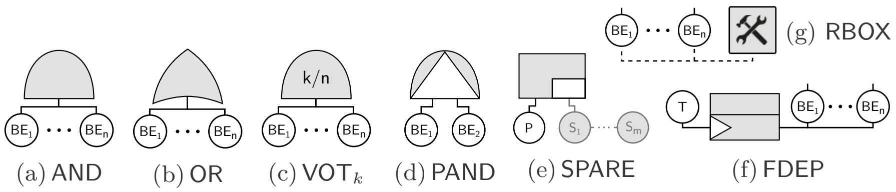


Fig. 1: Fault tree gates and the repair box

Basic elements. The leaves of the tree, called basic events or *basic elements* (BEs), model the failure of components. A BE b is equipped with a failure distribution F_b that governs the probability for b to fail before time t, and a repair distribution R_b governing its repair time. Some BEs are used as spare components: these (SBEs) replace a primary component when it fails. SBEs are equipped also with a dormancy distribution D_b, since spares fail less often when *dormant*, i.e. not in use. Only if an SBE becomes active, its failure distribution is given by F_b.

Gates. Non-leave nodes are called *intermediate events* and are labelled with *gates*, describing how combinations of lower failures propagate to upper levels. Fig. 1 shows their syntax. Their meaning is as follows: the AND, OR, and VOT$_k$ gates fail if respectively all, one, or k of their m children fail (with $1 \leqslant k \leqslant m$). The latter is called the *voting* or k out of m gate. Note that VOT$_1$ is equivalent to an OR gate, and VOT$_m$ is equivalent to an AND. The *priority-and gate* (PAND) is an AND gate that only fails if its children fail from left to right (or simultaneously). PANDs express failures that can only happen in a particular order, e.g. a short circuit in a pump can only occur after a leakage. SPARE gates have one *primary* child and one or more *spare* children: spares replace the primary when it fails. The FDEP gate has an input *trigger* and several *dependent events*: all dependent events become unavailable when the trigger fails. FDEPs can model for instance networks elements that become unavailable if their connecting bus fails.

Repair boxes. An RBOX determines which basic element is repaired next according to a given policy. Thus all its inputs are BEs or SBEs. Unlike gates, an RBOX has no output since it does not propagate failures.

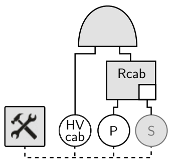


Fig. 2: Tiny RFT

Top level event. A full-system failure occurs if the *top event* (i.e. the root node) of the tree fails.

Example. The tree in Fig. 2 models a railway-signal system, which fails if its high voltage and relay cabinets fail [21, 39]. Thus, the top event is an AND gate with children HVcab (a BE) and Rcab. The latter is a SPARE gate with primary P and spare S. All BEs are managed by one RBOX with repair priority HVcab > P > S.

Notation. The nodes of a tree $\triangle$ are given by *nodes*$(\triangle) = \{0, 1, \ldots, n-1\}$. We let v, w range over *nodes*$(\triangle)$. A function $\mathit{type}^{\triangle} \colon \mathit{nodes}(\triangle) \to \{\mathsf{BE}, \mathsf{SBE}, \mathsf{AND}, \mathsf{OR}, \mathsf{VOT}_k, \mathsf{PAND}, \mathsf{SPARE}, \mathsf{FDEP}, \mathsf{RBOX}\}$ yields the type of each node in the tree. A function $\mathit{chil}^{\triangle} \colon \mathit{nodes}(\triangle) \to \mathit{nodes}(\triangle)^*$ returns the ordered list of children of a node. If clear from context, we omit the superscript $\triangle$ from function names.

Semantics. Following [32] we give semantics to RFT as Input/Output Stochastic Automata (IOSA), so that we can handle arbitrary probability distributions. Each state in the IOSA represents a system configuration, indicating which components are operational and which have failed. Transitions among states describe how the configuration changes when failures or repairs occur.

More precisely, a *state* in the IOSA is a tuple $\boldsymbol{x} = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}) \in \mathcal{S} \subseteq \mathbb{N}^n$, where $\mathcal{S}$ is the *state space* and $\boldsymbol{x}_v$ denotes the state of node v in $\triangle$. The possible values for $\boldsymbol{x}_v$ depend on the type of v. The *output* $\boldsymbol{z}_v \in \{0,1\}$ of node v indicates whether it is operational ($\boldsymbol{z}_v$=0) or failed ($\boldsymbol{z}_v$=1) and is calculated as follows:

- BEs (white circles in Fig. 1) have a binary state: $\boldsymbol{x}_v = 0$ if BE v is operational and $\boldsymbol{x}_v = 1$ if it is failed. The output of a BE is its state: $\boldsymbol{z}_v = \boldsymbol{x}_v$.
- SBEs (gray circles in Fig. 1e) have two additional states: $\boldsymbol{x}_v = 2, 3$ if a *dormant* SBE v is resp. operational, failed. Here $\boldsymbol{z}_v = \boldsymbol{x}_v \bmod 2$.
- ANDs have a binary state. Since the AND gate v fails iff all children fail: $\boldsymbol{x}_v = \min_{w \in \mathit{chil}(v)} \boldsymbol{z}_w$. An AND gate outputs its internal state: $\boldsymbol{z}_v = \boldsymbol{x}_v$.
- OR gates are analogous to AND gates, but fail iff any child fail, i.e. $\boldsymbol{z}_v = \boldsymbol{x}_v = \max_{w \in \mathit{chil}(v)} \boldsymbol{z}_w$ for OR gate v.
- VOT gates also have a binary state: a VOT_k gate fails iff $1 \leqslant k \leqslant m$ children fail, thus $\boldsymbol{z}_v = \boldsymbol{x}_v = 1$ if $k \leqslant \sum_{w \in \mathit{chil}(v)} \boldsymbol{z}_w$, and $\boldsymbol{z}_v = \boldsymbol{x}_v = 0$ otherwise.
- PAND gates admit multiple states to represent the failure order of the children. For PAND v with two children we let $\boldsymbol{x}_v$ equal: **0** if both children are operational; **1** if the left child failed, but the right one has not; **2** if the right child failed, but the left one has not; **3** if both children have failed, the right one first; **4** if both children have failed, otherwise. The output of PAND gate v is $\boldsymbol{z}_v = 1$ if $\boldsymbol{x}_v = 4$ and $\boldsymbol{z}_v = 0$ otherwise. PAND gates with more children are handled by exploiting $\mathsf{PAND}(w_1, w_2, w_3) = \mathsf{PAND}(\mathsf{PAND}(w_1, w_2), w_3)$.
- SPARE gate v leftmost input is its primary BE. All other (spare) inputs are SBEs. SBEs can be shared among SPARE gates. When the primary of v fails, it is replaced with an *available* SBE. An SBE is unavailable if it is failed, or if it is replacing the primary BE of another SPARE. The output of v is $\boldsymbol{z}_v = 1$ if its primary is failed and no spare is available. Else $\boldsymbol{z}_v = 0$.
- An FDEP gate has no output. All inputs are BEs and the leftmost is the trigger. We consider non-destructive FDEPs [7]: if the trigger fails, the output of all other BE is set to 1, without affecting the internal state. Since this can be modelled by a suitable combination of OR gates [32], we omit the details.

For example, the RFT from Fig. 2 starts with all operational elements, so the initial state is $\boldsymbol{x}^0 = (0,0,2,0,0)$. If then P fails, $\boldsymbol{x}_{\mathsf{P}}$ and $\boldsymbol{z}_{\mathsf{P}}$ are set to 1 (failed) and S becomes $\boldsymbol{x}_{\mathsf{S}} = 0$ (active and operational spare), so the state changes to $\boldsymbol{x}^1 = (0,1,0,0,0)$. The traces of the IOSA are given by $\boldsymbol{x}^0 \boldsymbol{x}^1 \cdots \boldsymbol{x}^n \in \mathcal{S}^*$, where a change from $\boldsymbol{x}^j$ to $\boldsymbol{x}^{j+1}$ corresponds to transitions triggered in the IOSA.

Nondeterminism. Dynamic fault trees may exhibit nondeterministic behaviour as a consequence of underspecified failure behaviour [15, 27]. This can happen e.g. when two SPAREs have a single shared SBE: if all elements are failed, and the SBE is repaired first, the failure behaviour depends on which SPARE gets the SBE. Monte Carlo simulation, however, requires fully stochastic models and cannot cope with nondeterminism. To overcome this problem we deploy the theory from [16, 32]. If a fault tree adheres to some mild syntactic conditions, then its IOSA semantics is *weakly deterministic*, meaning that all resolutions of the nondeterministic choices lead to the same probability value. In particular, we require that (1) each BE is connected to at most one SPARE gate, and (2) BEs and SBEs connected to SPAREs are not connected to FDEPs. In addition to this, some semantic decisions have been fixed, e.g. the semantics of PAND is fully specified, and policies should be provided for RBOX and spare assignments.

Dependability metrics. An important use of fault trees is to compute relevant dependability metrics. Let X_t denote the random variable that represents the state of the top event at time t [14]. Two popular metrics are:

– *system reliability*: the probability of observing no top event failure before some mission time $T > 0$, viz. $\mathrm{REL}_T = \mathit{Prob}\left(\forall_{t\in[0,T]} \,.\, X_t = 0\right)$;
– *system availability*: the proportion of time that the system remains operational in the long-run, viz. $\mathrm{AVA} = \lim_{t\to\infty} \mathit{Prob}\,(X_t = 0)$.

System *unreliability* and *unavailability* are the reverse of these metrics. That is: $\mathrm{UNREL}_T = 1 - \mathrm{REL}_T$ and $\mathrm{UNAVA} = 1 - \mathrm{AVA}$.

3 Stochastic simulation for Fault Trees

Standard Monte Carlo simulation (SMC). Monte Carlo simulation takes random samples from stochastic models to estimate a (dependability) metric of interest. For instance, to estimate the unreliability of a tree $\triangle$ we sample N independent traces from its IOSA semantics. An unbiased statistical estimator for $p = \mathrm{UNREL}_T$ is the proportion of traces observing a top level event, that is, $\hat{p}_N = \frac{1}{N}\sum_{j=1}^{N} X^j$ where $X^j = 1$ if the j-th trace exhibits a top level failure before time T and $X^j = 0$ otherwise. The statistical error of $\hat{p}$ is typically quantified with two numbers δ and ε s.t. $\hat{p} \in [p-\varepsilon, p+\varepsilon]$ with probability δ. The interval $\hat{p} \pm \varepsilon$ is called a *confidence interval* (CI) with coefficient δ and precision 2ε.

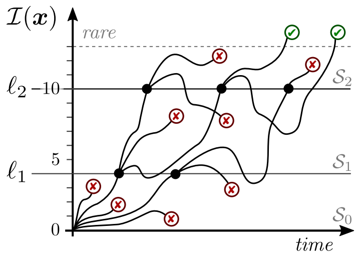
Such procedures scale linearly with the number of tree nodes and cater for a wide range of PDFs, even non-Markovian distributions. However, they encounter a bottleneck to estimate *rare events*: if $p \approx 0$, very few traces observe $X^j = 1$. Therefore, the variance of estimators like $\hat{p}$ becomes huge, and CIs become very broad, easily degenerating to the trivial interval $[0, 1]$. Increasing the number of traces alleviates this problem, but even standard CI settings—where ε is relative to p—require sampling an unacceptable number of traces [35]. Rare event simulation techniques solve this specific problem.

Rare Event Simulation (RES). RES techniques [35] increase the amount of traces that observe the rare event, e.g. a top level event in an RFT. Two prominent classes of RES techniques are *importance sampling*, which adjusts the PDF of failures and repairs, and *importance splitting* (ISPLIT [30]), which samples more (partial) traces from states that are closer to the rare event. We focus on ISPLIT due to its flexibility with respect to the probability distributions.

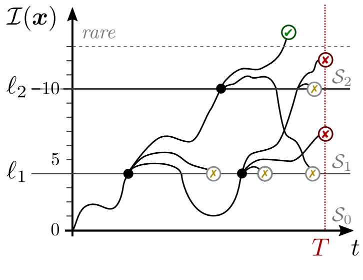
ISPLIT can be efficiently deployed as long as the rare event γ can be described as a nested sequence of less-rare events $\gamma = \gamma_M \subsetneq \gamma_{M-1} \subsetneq \cdots \subsetneq \gamma_0$. This decomposition allows ISPLIT to study the conditional probabilities $p_k = \mathit{Prob}(\gamma_{k+1} \mid \gamma_k)$ separately, to then compute $p = \mathit{Prob}(\gamma) = \prod_{k=0}^{M\text{-}1} \mathit{Prob}(\gamma_{k+1} \mid \gamma_k)$. Moreover, ISPLIT requires all conditional probabilities p_k to be much greater than p, so that estimating each p_k can be done efficiently with SMC.

The key idea behind ISPLIT is to define the events γ_k via a so called *importance function* $\mathcal{I}\colon \mathcal{S} \to \mathbb{N}$ that assigns an *importance* to each state $s \in \mathcal{S}$. The higher the importance of a state, the closer it is to the rare event γ_M. Event γ_k collects all states with importance at least ℓ_k, for certain sequence of *threshold levels* $0 = \ell_0 < \ell_1 < \cdots < \ell_M$. Formally: $\gamma_k = \{s \in \mathcal{S} \mid \mathcal{I}(s) \geqslant \ell_k\}$.

To exploit the importance function $\mathcal{I}$ in the simulation procedure, ISPLIT samples more (partial) traces from states with higher importance. Two well-known methods are deployed and compared in this paper: Fixed Effort and RESTART. *Fixed Effort* (FE [19]) samples a predefined amount of traces in each region $\mathcal{S}_k = \gamma_k \setminus \gamma_{k+1} = \{s \in \mathcal{S} \mid \ell_{k+1} > \mathcal{I}(s) \geqslant \ell_k\}$. Thus, starting at γ_0 it first estimates the proportion of traces that reach γ_1, i.e. $p_0 = \mathit{Prob}(\gamma_1 \mid \gamma_0) = \mathit{Prob}(\mathcal{S}_0)$. Next, from the states that reached γ_1 new traces are generated to estimate $p_1 = \mathit{Prob}(\mathcal{S}_1)$, and so on until p_M. Fixed Effort thus requires that (i) each trace has a clearly defined "end," so that estimations of each p_k finish with probability 1, and (ii) all rare events reside in the uppermost region.



(a) FE$_5$ for $\mathit{Prob}(\neg$ ✘ U ✔$)$



(b) RST$_{\text{ES}}$ for UNREL$_T$

Fig. 3: Importance Splitting algorithms Fixed Effort & RESTART

Example. Fig. 3a shows Fixed Effort estimating the probability to visit states labelled ✔ before others labelled ✘. States ✔ have importance >13, and thresholds $\ell_1, \ell_2 = 4, 10$ partition the state space in regions $\{\mathcal{S}_i\}_{i=0}^{2}$ s.t. all ✔ $\in \mathcal{S}_2$. The effort is 5 simulations per region, for all regions: we call this algorithm FE$_5$.

In region $\mathcal{S}_0$, 2 simulations made it from the initial state to threshold ℓ_1, i.e. they reached some state with importance 4 before visiting a state ✘. In $\mathcal{S}_1$, starting from these two states, 3 simulations reached ℓ_2. Finally, 2 out of 5 simulations visited states ✔ in $\mathcal{S}_2$. Thus, the estimated rare event probability of this run of FE 5 is $\hat{p} = \prod_{i=1}^{2} \hat{p}_i = \frac{2}{5}\frac{3}{5}\frac{2}{5} = 9.6 \times 10^{-2}$.

RESTART (RST [48, 47]) is another RES algorithm, which starts one trace in γ_0 and monitors the importance of the states visited. If the trace up-crosses threshold ℓ_1, the first state visited in $\mathcal{S}_1$ is saved and the trace is cloned, aka *split*—see Fig. 3b. This mechanism rewards traces that get closer to the rare event. Each clone then evolves independently, and if one up-crosses threshold ℓ_2 the splitting mechanism is repeated. Instead, if a state with importance below ℓ_1 is visited, the trace is *truncated* (⊗ in Fig. 3b). This penalises traces that move away from the rare event. To avoid truncating all traces, the one that spawned the clones in region $\mathcal{S}_k$ can go below importance ℓ_k. To deploy an unbiased estimator for p, RESTART measures how much split was required to visit a rare state [47]. In particular, RESTART does not need the rare event to be defined as γ_M [44], and it was devised for steady-state analysis [48] (e.g. to estimate UNAVA) although it can also been used for transient studies as depicted in Fig. 3b [45].

4 Importance Splitting for FTA

The effectiveness of ISPLIT crucially relies on the choice of the importance function $\mathcal{I}$ as well as the threshold levels ℓ_k [30]. Traditionally, these are given by domain and/or RES experts, requiring a lot of domain knowledge. This section presents a technique to obtain $\mathcal{I}$ and the ℓ_k automatically for an RFT.

4.1 Compositional importance functions for Fault Trees

By the core idea behind importance splitting, states that are more likely to lead to the rare event should have a higher importance. To achieve this, the key lies in defining an importance function $\mathcal{I}$ and thresholds ℓ_k that are sensitive to both the state space $\mathcal{S}$ and the transition probabilities of the system. For us, $\mathcal{S} \subseteq \mathbb{N}^n$ are all possible states of a repairable fault tree (RFT). Its top event fails when certain nodes fail in certain order, and remain failed before certain repairs occur. To exploit this for ISPLIT, the structure of the tree must be embedded into $\mathcal{I}$.

The strong dependence of the importance function $\mathcal{I}$ on the structure of the tree is easy to see in the following example. Take the RFT $\triangle$ from Fig. 2 and let its current state $\boldsymbol{x}$ be s.t. P is failed and HVcab and S are operational. If the next event is a repair of P, then the new state $\boldsymbol{x}'$ (where all basic elements are operational) is farther from a failure of the top event. Hence, a good importance function should satisfy $\mathcal{I}(\boldsymbol{x}) > \mathcal{I}(\boldsymbol{x}')$. Oppositely, if the next event had been a failure of S leading to state $\boldsymbol{x}''$, then one would want that $\mathcal{I}(\boldsymbol{x}) < \mathcal{I}(\boldsymbol{x}'')$. The key observation is that these inequalities depend on the structure of $\triangle$ as well as on the failures/repairs of basic elements.

In view of the above, any attempt to define an importance function for an arbitrary fault tree $\triangle$ must put its gate structure in the forefront. In Table 1 we introduce a compositional heuristic for this, which defines *local importance functions* distinguished per node type. The importance function associated to node v is $\mathcal{I}_v \colon \mathbb{N}^n \to \mathbb{N}$. We define the *global importance function* of the tree ($\mathcal{I}_\triangle$ or simply $\mathcal{I}$) as the local importance function of the top event node of $\triangle$.

Table 1: Compositional importance function for RFTs

$\mathit{type}(v)$	$\mathcal{I}_v(\boldsymbol{x})$
BE, SBE	$\boldsymbol{z}_v$
AND	$\mathrm{lcm}_v \cdot \sum_{w\in \mathit{chil}(v)} \frac{\mathcal{I}_w(\boldsymbol{x})}{\max^{\mathcal{I}}_w}$
OR	$\mathrm{lcm}_v \cdot \max\limits_{w\in \mathit{chil}(v)} \left\{\frac{\mathcal{I}_w(\boldsymbol{x})}{\max^{\mathcal{I}}_w}\right\}$
VOT$_k$	$\mathrm{lcm}_v \cdot \max\limits_{W\subseteq \mathit{chil}(v), \lvert W\rvert = k} \left\{\sum_{w\in W} \frac{\mathcal{I}_w(\boldsymbol{x})}{\max^{\mathcal{I}}_w}\right\}$
SPARE	$\mathrm{lcm}_v \cdot \max\left(\sum_{w\in \mathit{chil}(v)} \frac{\mathcal{I}_w(\boldsymbol{x})}{\max^{\mathcal{I}}_w}\ ,\ \boldsymbol{z}_v \cdot m\right)$
PAND	$\mathrm{lcm}_v \cdot \max\left(\frac{\mathcal{I}_l(\boldsymbol{x})}{\max^{\mathcal{I}}_l} + \mathit{ord}\,\frac{\mathcal{I}_r(\boldsymbol{x})}{\max^{\mathcal{I}}_r}\ ,\ \boldsymbol{z}_v \cdot 2\right)$ where $\mathit{ord} = 1$ if $\boldsymbol{x}_v \in \{1,4\}$ and $\mathit{ord} = -1$ otherwise
with $\max^{\mathcal{I}}_v = \max_{\boldsymbol{x}\in\mathcal{S}} \mathcal{I}_v(\boldsymbol{x})$ and $\mathrm{lcm}_v = \mathrm{lcm}\left\{\max^{\mathcal{I}}_w \,\middle|\, w \in \mathit{chil}(v)\right\}$	

Thus, $\mathcal{I}_v$ is defined in Table 1 via structural induction in the fault tree. It is defined so that it assigns to a *failed* node v its *highest importance value*. Functions with this property deploy the most efficient ISPLIT implementations [30], and some RES algorithms (e.g. Fixed Effort) require this property [19].

In the following we explain our definition of $\mathcal{I}_v$. If v is a failed BE or SBE, then its importance is 1; else it is 0. This matches the output of the node, thus $\mathcal{I}_v(\boldsymbol{x}) = \boldsymbol{z}_v$. Intuitively, this reflects how failures of basic elements are positively correlated to top event failures. The importance of AND, OR, and VOT$_k$ gates depends exclusively on their input. The importance of an AND is the sum of the importance of their children scaled by a normalisation factor. This reflects that AND gates fail when all their children fail, and each failure of a child brings an AND closer to its own failure, hence increasing its importance. Instead, since OR gates fail as soon as a single child fails, their importance is the maximum importance among its children. The importance of a VOT$_k$ gate is the sum of the k (out of m) children with highest importance value.

Omiting normalisation may yield an undesirable importance function. To understand why, suppose a binary AND gate v with children l and r, and define $\mathcal{I}^{\text{naive}}_v(\boldsymbol{x}) = \mathcal{I}_l(\boldsymbol{x}) + \mathcal{I}_r(\boldsymbol{x})$. Suppose that $\mathcal{I}_l$ takes it highest value in $\max^{\mathcal{I}}_l = 2$ while $\mathcal{I}_r$ in $\max^{\mathcal{I}}_r = 6$ and assume that states $\boldsymbol{x}$ and $\boldsymbol{x}'$ are s.t. $\mathcal{I}_l(\boldsymbol{x}) = 1$,

$\mathcal{I}_r(\boldsymbol{x}) = 0$, $\mathcal{I}_l(\boldsymbol{x}') = 0$, $\mathcal{I}_r(\boldsymbol{x}') = 3$. This means that in both states one child of v is "good-as-new" and the other is "half-failed" and hence the system is equally close to fail in both cases. Hence we expect $\mathcal{I}_v^{\text{naive}}(\boldsymbol{x}) = \mathcal{I}_v^{\text{naive}}(\boldsymbol{x}')$ when actually $\mathcal{I}_v^{\text{naive}}(\boldsymbol{x}) = 1 \neq 3 = \mathcal{I}_v^{\text{naive}}(\boldsymbol{x}')$. Instead, $\mathcal{I}_v$ operates with $\frac{\mathcal{I}_l(\boldsymbol{x})}{\max_l^{\mathcal{I}}}$ and $\frac{\mathcal{I}_r(\boldsymbol{x})}{\max_r^{\mathcal{I}}}$, which can be interpreted as the "percentage of failure" of the children of v. To make these numbers integers we scale them by lcm_v, the *least common multiple* of their max importance values. In our case $\text{lcm}_v = 6$ and hence $\mathcal{I}_v(\boldsymbol{x}) = \mathcal{I}_v(\boldsymbol{x}') = 3$. Similar problems arise whit all gates, hence normalization is applied in general.

SPARE gates with m children (including its primary) behave similarly to AND gates: every failed child brings the gate closer to failure, as reflected in the left operand of the max in Table 1. However, SPAREs fail when their primaries fail and no SBEs are *available*, e.g. possibly being used by another SPARE. This means that the gate could fail in spite of some children being operational. To account for this we exploit the gate output: multiplying $\boldsymbol{z}_v$ by m we give the gate its maximum value when it fails, even when this happens due to unavailable but operational SBEs. For a PAND gate v we have to carefully look at the states. If the left child l has failed, then the right child r contributes positively to the failure of the PAND and hence the importance function of the node v. If instead the right child has failed first, then the PAND gate will not fail and hence we let it contribute negatively to the importance function of v. Thus, we multiply $\frac{\mathcal{I}_r(\boldsymbol{x})}{\max_r^{\mathcal{I}}}$ (the normalized importance function of the right child) by -1 in the later case (i.e. when state $\boldsymbol{x}_v \notin \{1, 4\}$). Instead, the left child always contribute positively. Finally, the max operation is two-fold: on the one hand, $\boldsymbol{z}_v \cdot 2$ ensures that the importance value remains at its maximun while failing (PANDs remain failed even after the left child is repaired); on the other, it ensures that the smallest value posible is 0 while operational (since importance values can not be negative.)

4.2 Automatic importance splitting for FTA

Our compositional importance function is based on the distribution of operational/failed basic elements in the fault tree, and their failure order. This follows the core idea of importance splitting: the more failed BEs/SBEs (in the right order), the closer a tree is to its top event failure.

However, ISPLIT is about running more simulations from state with higher *probability* to lead to rare states. This is only partially reflected by whether basic element b is failed. Probabilities lie also in the distributions F_b, R_b, D_b. These distributions govern the transitions among states $\boldsymbol{x} \in \mathcal{S}$, and can be exploited for importance splitting. We do so using the two-phased approach of [11, 12], which in a first (static) phase computes an importance function, and in a second (dynamic) phase selects the thresholds from the resulting importance values.

In our current work, the first phase runs breadth-first search in the IOSA module of each tree node. This computes node-local importance functions, that are aggregated into a tree-global $\mathcal{I}$ using our compositional function in Table 1.

The second phase involves running "pilot simulations" on the importance-labelled states of the tree. Running simulations exercises the fail/repair distributions of BEs/SBEs, imprinting this information in the thresholds ℓ_k. Several algorithms can do such *selection of thresholds*. They operate sequentially, starting from the initial state—a fully operational tree—which has importance $i_0 = 0$. For instance, Expected Success [10] runs N finite-life simulations. If $K < \frac{N}{2}$ simulations reach the next smallest importance $i_1 > i_0$, then the first threshold will be $\ell_1 = i_1$. Next, N simulations start from states with importance i_1, to determine whether the next importance i_2 should be chosen as threshold ℓ_2, and so on.

Expected Success also computes the *effort* per splitting region $\mathcal{S}_k = \{\boldsymbol{x} \in \mathcal{S} \mid \ell_{k+1} > \mathcal{I}(\boldsymbol{x}) \geqslant \ell_k\}$. For Fixed Effort, "effort" is the base number of simulations to run in region $\mathcal{S}_k$. For RESTART, it is the number of clones spawned when threshold ℓ_{k+1} is up-crossed. In general, if K out of N pilot simulations make it from ℓ_{k-1} to ℓ_k, then the k-th effort is $\lceil \frac{N}{K} \rceil$. This is chosen so that, during RES estimations, one simulation makes it from threshold ℓ_{k-1} to ℓ_k on average.

Thus, using the method from [11, 12] based on our importance function $\mathcal{I}_\triangle$, we compute (automatically) the thresholds and their effort for tree $\triangle$. This is all the meta-information required to apply importance splitting RES [19, 18, 11].

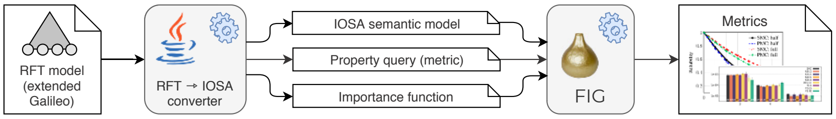


Fig. 4: Tool chain

Implementation. Fig. 4 outlines a tool chain implemented to deploy the theory described above. The input model is an RFT, described in the Galileo textual format [42, 41] extended with repairs and arbitrary PDFs. This RFT file is given as input to a Java converter that produces three outputs: the IOSA semantics of the tree, the property queries for its reliability or availability, and our compositional importance function in terms of variables of the IOSA semantic model. This information is dumped into a single text file and fed to FIG: a statistical model checker specialised in importance splitting RES. FIG interprets this importance function, deploying it into its internal model representation, which results in a global function for the whole tree. FIG can then use ISPLIT algorithms such as RESTART and Fixed Effort, via the automatic methods described above. The result are confidence intervals that estimate the reliability or availability of the RFT. In this way, we implemented automatic importance splitting for FTA. In [9] we provide more details about our tool chain and its capabilities.

5 Experimental evaluation

5.1 General setup

Using our tool chain, we computed the unreliability and unavailability of 26 highly-resilient repairable non-Markovian DFTs. These trees come from seven literature case studies, enriched with RBOX elements and non-Markovian PDFs. We estimated their UNREL_{10^3} or UNAVA in increasingly resilient configurations.

To estimate these values we used various simulation algorithms: Standard Monte Carlo (SMC); Fixed Effort [19] for different number of runs performed in each $\mathcal{S}_k$ region (FE_n for $n = 8, 12, 16, 24, 32$); RESTART [47] with thresholds selected via a Sequential Monte Carlo algorithm [12] for different global splitting values (RST_n for $n = 2, 3, 5, 8, 11$); and RESTART with thresholds selected via Expected Success [10], which computes splitting values independently for each threshold (RST_{ES}). FE_n, RST_n, and RST_{ES}, used the automatic ISPLIT framework based in our importance function, as described in Sec. 4.2.

An *instance* $\mathfrak{y}$ is a combination of an algorithm *algo*, an RFT, and a dependability metric. An RFT is identified by a case study ($\mathcal{cs}$) and a parameter ($\mathfrak{p}$), where larger parameters of the RFT $\mathcal{cs}_{\mathfrak{p}}$ indicate smaller dependability values $p_{\mathcal{cs}_{\mathfrak{p}}}$. Running *algo* for a fixed simulation time, instance $\mathfrak{y}$ estimates the value $p_{\mathfrak{y}} = p_{\mathcal{cs}_{\mathfrak{p}}}$. The resulting CI ($\hat{p}_{\mathfrak{y}}$) has a certain width $\|\hat{p}_{\mathfrak{y}}\| \in [0, 1]$ (we fix the confidence coefficient $\delta = 0.95$). The performance of *algo* can be measured by that width: the smaller $\|\hat{p}_{\mathfrak{y}}\|$, the more efficient the algorithm that achieved it.

The simulation time fixed for an RFT may not suffice to observe rare events, e.g. for SMC. In such cases the FIG tool reports a "null estimate" $\hat{p}_{\mathfrak{y}} = [0, 0]$. Moreover, the simulation of random events depends on the RNG—and its seed—used by FIG, so different runs may yield different results $\hat{p}_{\mathfrak{y}}$. Therefore, for each $\mathfrak{y}$ we repeated $n = 10$ times the computation of $\hat{p}_{\mathfrak{y}}$, to assess the performance of *algo* in $\mathfrak{y}$ by: (i) how many times it yielded not-null estimates, indicated with a bold number at the base of the bar corresponding to $\mathfrak{y}$ (e.g. **8** **10** in Fig. 5b); (ii) what was the average width $\|\hat{p}_{\mathfrak{y}}\|$, using not-null estimates only, indicated by the height of the bar; and (iii) what was the standard deviation of those widths, indicated by whiskers on top of the bar. We performed $n = 10$ repetitions to ensure statistical significance: a 95% CI for a plotted bar is narrower than the whiskers and, in the hardest configuration of every $\mathcal{cs}$, the whiskers of SMC bars never overlap with those of the best RES algorithm.

Case studies. Our seven parametric case studies are: the synthetic models DSPARE_n and VOT_m, with $n \in \{3, 4, 5\}$ SBEs the first, $m \in \{2, 3, 4\}$ shared BEs the second, and one RBOX each; FTPP_s [17], where we study one triad with $s \in \{4, 5, 6\}$ shared SBEs, using one RBOX for the processors and another for the network elements; HECS_o [43], with 2 memory interfaces, 4 RBOX (one per subsystem), $o \in \{1, \ldots, 5\}$ shared spare processors, and $2o$ parallel buses; and $\text{RWC}_{u \in \{4,\ldots,7\}}$ [22, 21, 39], which combines subsystems RC_v with one RBOX and $v \in \{3, \ldots, 6\}$ SPAREs, and HVC_w with another RBOX and $w \in \{2, \ldots, 4\}$ shared SBEs. In total these are 26 RFTs with PDFs that include exponential, Erlang, uniform, Rayleigh, Weibull, normal, and log-normal distributions. In an extended version of this work [9] we provide all details of our case studies.

Hardware. Experiments ran in two types of nodes in a SLURM cluster running Linux x64 (Ubuntu, kernel 3.13.0-168): *korenvliet* nodes have CPUs Intel® Xeon® E5-2630 v3 @ 2.40 GHz, and 64 GB of DDR4 RAM @ 1600 MHz; *caserta* has CPUs Intel® Xeon® E7-8890 v4 @ 2.20 GHz, and 2 TB of RAM DDR4 @ 1866 MHz.

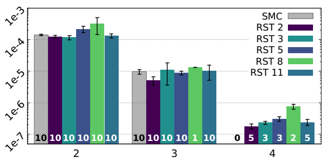
5.2 Experimental results and discussion

Using SMC and RESTART we computed UNAVA for $\mathrm{VOT}_{2,3,4}$, $\mathrm{HECS}_{1,\dots,5}$, $\mathrm{RC}_{3,\dots,6}$, and $\mathrm{RWC}_{1,\dots,4}$. FE was not used since it requires regeneration theory for steady-state analysis [19], which is not always feasible with non-Markovian models. The mean widths of the CIs achieved per instance are shown in Fig. 5.

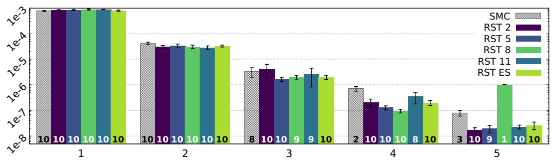
For example for VOT_2 (Fig. 5a), 10 independent computations with SMC ran in caserta for 5 min, and all converged to not-null CIs (**10**). The mean width of these CIs was $1.40{\times}10^{-4}$ and their standard deviation $7.96{\times}10^{-6}$. For VOT_3, all SMC computations yielded not-null CIs (after 30 min) with an average precision of $9.62{\times}10^{-6}$ and standard deviation $1.52{\times}10^{-6}$. For VOT_4 all SMC simulations yielded null CIs after 3 hours of simulation (**0**). Instead, RST_2 converged to 10, 10, and 5 not-null CIs resp. for $\mathrm{VOT}_{2,3,4}$, with mean widths (and standard deviation): $1.24{\times}10^{-4}$ ($1.19{\times}10^{-5}$), $5.09{\times}10^{-6}$ ($1.48{\times}10^{-6}$), and $1.79{\times}10^{-7}$ ($3.19{\times}10^{-8}$). Thus for the VOT case study, RST_2 was consistently more efficient than SMC, and the efficiency gap increased as UNAVA became rarer.

This trend repeats in all experiments: as expected, the rarer the metric, the wider the CIs computed in the time limit, until at some point it becomes very hard to converge to not-null CIs at all (specially for SMC). For the least resilient configuration of each case study, SMC can be competitive or even more efficient than some ISPLIT variants. For instance for VOT_1 and HECS_1 in Figs. 5a and 5b, all computations converged to not-null CIs for all algorithms, but SMC exhibits less variable CI widths, viz. smaller whiskers. This is reasonable: truncating and splitting traces in RESTART adds (*i*) simulation overhead that may not pay off to estimate not-so-rare events, and on top of it (*ii*) correlations of cloned traces that share a common history, increasing the variability among independent runs. On the other hand and as expected, SMC looses this competitiveness for all case studies as failures become rarer, here when UNAVA $\leqslant 1.0{\times}10^{-5}$. This

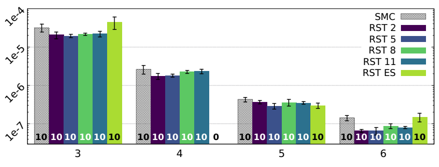
Fig. 5: CI precision for system unavailability



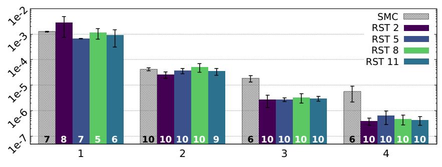
(a) VOT



(b) HECS



(c) RC

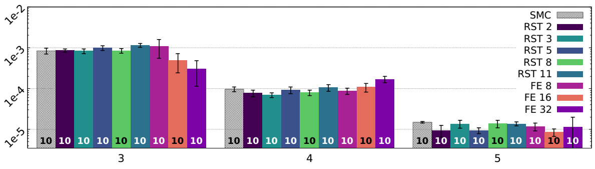


(d) RWC

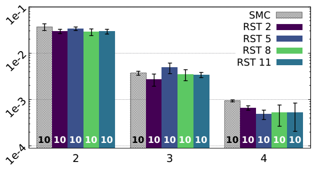
holds nicely for the biggest case studies: HECS$_5$[†] (a 42-nodes RFT whose IOSA has 126-not-clock variables $\approx 2.89{\times}10^{38}$ states, with 57 clocks of exponential, uniform, and log-normal PDFs) and RWC$_4$ (42 nodes, 181 variables $\approx 6.93{\times}10^{73}$ states, 62 clocks of exponential, Erlang, Rayleigh, uniform, and normal PDFs).

Using SMC, RESTART, and FE, we also estimated UNREL$_{1000}$ for RWC$_{2,3,4}$, DSPARE$_{3,4,5}$, FTPP$_{4,5,6}$, HVC$_{4,5,6,7}$, and HECS$_{2,3,4,5}$. For HVC (only) we ran 20 experiments per tree, 10 in each cluster node. Fig. 6 shows the results.

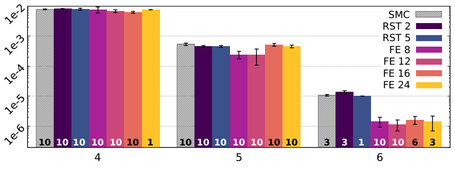
Fig. 6: CI precision for system unreliability



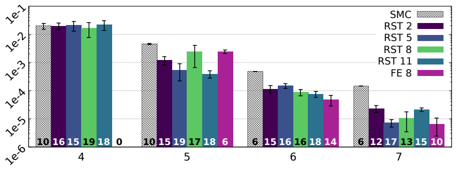
(a) DSPARE



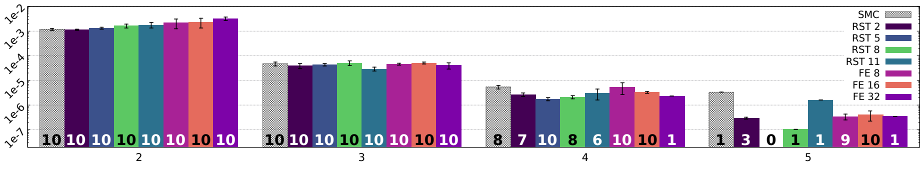
(b) RWC



(c) FTPP



(d) HVC



(e) HECS

The overall trend shown for unreliability estimations is similar to the previous unavailability cases. Here however it was possible to use Fixed Effort, since every simulation has a clearly defined end at time $T = 10^3$. It is interesting thus to compare the efficiency of RESTART vs. FE: we note for example that some variants of FE performed considerably better than any other approach in the most resilient configurations of FTPP and HECS. It is nevertheless difficult to draw general conclusions from Figs. 6a to 6e, since some variants that performed best in a case study—e.g. FE$_{16}$ in HECS—did worse in others—e.g. FTPP, where the best algorithms were FE$_{8,12}$. Furthermore, FE$_8$, which is always better than

[†]RST$_8$ for HECS$_5$ escapes this trend: analysing the execution logs it was found that FIG crashed during the second computation.

SMC when $\mathrm{UNREL}_{1000} < 10^{-3}$, did not perform very well in HVC, where the algorithms that achieved the narrowest and most not-null CIs were $\mathrm{RST}_{5,11}$. Such cases notwithstanding, FE is a solid competitor of RESTART in our benchmark.

Another relevant point of study is the optimal effort e for $\mathrm{RST}_{\mathrm{e}}$ or FE_{e}, which shows no clear trend in our experiments. Here, e is a "global effort" used by these algorithms, equal for all $\mathcal{S}_k$ regions. e also alters the way in which the thresholds selection algorithm Sequential Monte Carlo (SEQ [12]) selects the ℓ_k. The lack of guidelines to select a value for e that works well across different systems was raised in [8]. This motivated the development of Expected Success (ES [10]), which selects efforts individually per $\mathcal{S}_k$ (or ℓ_k). Thus, in $\mathrm{RST}_{\mathrm{ES}}$, a trace up-crossing threshold ℓ_k is split according to the individual effort e_k selected by ES. In the benchmark of [10], which consists mostly of queueing systems, ES was shown superior to SEQ. However, experimental outcomes on DFTs in this work are different: for UNAVA, $\mathrm{RST}_{\mathrm{ES}}$ yielded mildly good results for HECS and RC; for the other case studies and for all UNREL_{1000} experiments, $\mathrm{RST}_{\mathrm{ES}}$ always yielded null CIs. It was found that the effort selected for most thresholds ℓ_k was either too small—so splitting in e_k was not enough for the $\mathrm{RST}_{\mathrm{ES}}$ trace to reach ℓ_{k+1}—or too large—so there was a splitting/truncation overhead. This point is further addressed in the conclusions.

Beyond comparisons among the specific algorithms, be these for RES or for selecting thresholds, it seems clear that our approach to FTA via ISPLIT deploys the expected results. For each parameterised case study $\mathcal{CS}_{\mathfrak{p}}$, we could find a value of the parameter $\mathfrak{p}$ where the level of resilience is such, that SMC is less efficient than our automatically-constructed ISPLIT framework. This is particularly significant for big DFTs like HECS and RWC, whose complex structure could be exploited by our importance function.

6 Related work

Most work on DFT analysis assumes discrete [43, 3] or exponentially distributed [15, 29] components failure. Furthermore, components repair is seldom studied in conjunction with dynamic gates [6, 3, 40, 29, 31]. In this work we address repairable DFTs, whose failure and repair times can follow arbitrary PDFs. More in detail, RFTs were first formally introduced as stochastic Petri nets in [6, 13]. Our work stands on [32], which reviews [13] in the context of stochastic automata with arbitrary PDFs. In particular we also address non-Markovian continuous distributions: in Sec. 5 we experimented with exponential, Erlang, uniform, Rayleigh, Weibull, normal, and log-normal PDFs. Furthermore and for the first time, we consider the application of [13, 32] to study rare events.

Much effort in RES has been dedicated to study highly reliable systems, deploying either importance splitting or sampling. Typically, importance sampling can be used when the system takes a particular shape. For instance, a common assumption is that all failure (and repair) times are exponentially distributed with parameters λ^i, for some $\lambda \in \mathbb{R}$ and $i \in \mathbb{N}_{>0}$. In these cases, a favourable change of measure can be computed analytically [20, 23, 33, 34, 49, 39].

In contrast, when the fail/repair times follow less-structured distributions, importance splitting is more easily applicable. As long as a full system failure can be broken down into several smaller components failures, an importance splitting method can be devised. Of course, its efficiency relies heavily on the choice of importance function. This choice is typically done ad hoc for the model under study [44, 30, 46]. In that sense [24, 25, 11, 12] are among the first to attempt a heuristic derivation of all parameters required to implement splitting. This is based on formal specifications of the model and property query (the dependability metric). Here we extended [11, 12, 8], using the structure of the fault tree to define composition operands. With these operands we aggregate the automatically-computed local importance functions of the tree nodes. This aggregation results in an importance function for the whole model.

7 Conclusions

We have presented a theory to deploy *automatic importance splitting* (ISPLIT) for fault tree analysis of repairable dynamic fault trees (RFTs). This Rare Event Simulation approach supports arbitrary probability distributions of components failure and repair. The core of our theory is an importance function $\mathcal{I}_\triangle$ defined structurally on the tree. From such function we implemented ISPLIT algorithms, and used them to estimate the *unreliability* and *unavailability* of highly-resilient RFTs. Departing from classical approaches, that define importance functions ad hoc using expert knowledge, our theory computes all metadata required for RES from the model and metric specifications. Nonetheless, we have shown that for a fixed simulation time budget and in the most resilient RFTs, diverse ISPLIT algorithms can be automatically implemented from $\mathcal{I}_\triangle$, and always converge to narrower confidence intervals than standard Monte Carlo simulation.

There are several paths open for future development. First and foremost, we are looking into new ways to define the importance function, e.g. to cover more general categories of FTs such as fault maintenance trees [37]. It would also be interesting to look into possible correlations among specific RES algorithms and tree structures, that yield the most efficient estimations for a particular metric. Moreover, we have defined $\mathcal{I}_\triangle$ based on the tree structure alone. It would be interesting to further include stochastic information in this phase, and not only afterwards during the thresholds-selection phase.

Regarding thresholds, the relatively bad performance of the Expected Success algorithm shows a spot for improvement. In general, we believe that enhancing its statistical properties should alleviate the behaviour mentioned in Sec. 5.2. Moreover, techniques to increase trace independence during splitting (e.g. resampling) could further improve the performance of the ISPLIT algorithms. Finally, we are investigating enhancements in IOSA and our tool chain, to exploit the ratio between fail and dormancy PDFs of SBEs in warm SPARE gates.

Acknowledgments

The authors thank José and Manuel Villén-Altamirano, for fruitful discussions that helped to better understand the application scope of our approach.

References

1. Abate, A., Budde, C.E., Cauchi, N., Hoque, K.A., Stoelinga, M.: Assessment of maintenance policies for smart buildings: Application of formal methods to fault maintenance trees. PHM Society European Conference **4**(1) (2018), https://www.phmpapers.org/index.php/phme/article/view/385
2. Bayes, A.J.: Statistical techniques for simulation models. Australian computer journal **2**(4), 180–184 (1970)
3. Beccuti, M., Codetta-Raiteri, D., Franceschinis, G., Haddad, S.: Non deterministic repairable fault trees for computing optimal repair strategy. In: VALUETOOLS 2008 (2010). https://doi.org/10.4108/ICST.VALUETOOLS2008.4411
4. Blanchet, J., Mandjes, M.: Rare event simulation for queues. In: Rubino and Tuffin [36], pp. 87–124. https://doi.org/10.1002/9780470745403.ch5
5. Blom, H.A.P., Bakker, G.J.B., Krystul, J.: Rare event estimation for a large-scale stochastic hybrid system with air traffic application. In: Rubino and Tuffin [36], pp. 193–214. https://doi.org/10.1002/9780470745403.ch9
6. Bobbio, A., Codetta-Raiteri, D.: Parametric fault trees with dynamic gates and repair boxes. In: RAMS 2004. pp. 459–465. IEEE (2004). https://doi.org/10.1109/RAMS.2004.1285491
7. Boudali, H., Crouzen, P., Haverkort, B.R., Kuntz, M., Stoelinga, M.: Architectural dependability evaluation with arcade. In: DSN'08. pp. 512–521. IEEE Computer Society (2008). https://doi.org/10.1109/DSN.2008.4630122
8. Budde, C.E.: Automation of Importance Splitting Techniques for Rare Event Simulation. Ph.D. thesis, FAMAF, Universidad Nacional de Córdoba, Córdoba, Argentina (2017), https://famaf.biblio.unc.edu.ar/cgi-bin/koha/opac-detail.pl?biblionumber=18143
9. Budde, C.E., Biagi, M., Monti, R.E., D'Argenio, P.R., Stoelinga, M.: Rare event simulation for non-Markovian repairable fault trees. arXiv e-prints arXiv:1910.11672 (2019), https://arxiv.org/abs/1910.11672
10. Budde, C.E., D'Argenio, P.R., Hartmanns, A.: Better automated importance splitting for transient rare events. In: SETTA. LNCS, vol. 10606, pp. 42–58. Springer (2017). https://doi.org/10.1007/978-3-319-69483-2_3
11. Budde, C.E., D'Argenio, P.R., Hermanns, H.: Rare event simulation with fully automated importance splitting. In: EPEW 2015. LNCS, vol. 9272, pp. 275–290. Springer (2015). https://doi.org/10.1007/978-3-319-23267-6_18
12. Budde, C.E., D'Argenio, P.R., Monti, R.E.: Compositional construction of importance functions in fully automated importance splitting. In: VALUETOOLS 2016. pp. 30–37 (2017). https://doi.org/10.4108/eai.25-10-2016.2266501
13. Codetta-Raiteri, D., Iacono, M., Franceschinis, G., Vittorini, V.: Repairable fault tree for the automatic evaluation of repair policies. In: DSN 2004. pp. 659–668. IEEE Computer Society (2004). https://doi.org/10.1109/DSN.2004.1311936
14. Coppit, D., Sullivan, K.J., Dugan, J.B.: Formal semantics of models for computational engineering: a case study on dynamic fault trees. In: ISSRE 2000. pp. 270–282 (2000). https://doi.org/10.1109/ISSRE.2000.885878
15. Crouzen, P., Boudali, H., Stoelinga, M.: Dynamic fault tree analysis using input/output interactive Markov chains. In: DSN 2007. pp. 708–717. IEEE Computer Society (2007). https://doi.org/10.1109/DSN.2007.37
16. D'Argenio, P.R., Monti, R.E.: Input/Output Stochastic Automata with Urgency: Confluence and weak determinism. In: ICTAC. LNCS, vol. 11187, pp. 132–152. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_8

17. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Fault trees and sequence dependencies. In: ARMS 1990. pp. 286–293. IEEE (1990). https://doi.org/10.1109/ARMS.1990.67971
18. Garvels, M.J.J., van Ommeren, J.K.C.W., Kroese, D.P.: On the importance function in splitting simulation. European Transactions on Telecommunications **13**(4), 363–371 (2002). https://doi.org/10.1002/ett.4460130408
19. Garvels, M.J.J.: The splitting method in rare event simulation. Ph.D. thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands (2000), http://eprints.eemcs.utwente.nl/14291/
20. Goyal, A., Shahabuddin, P., Heidelberger, P., Nicola, V.F., Glynn, P.W.: A unified framework for simulating Markovian models of highly dependable systems. IEEE Transactions on Computers **41**(1), 36–51 (1992). https://doi.org/10.1109/12.123381
21. Guck, D., Spel, J., Stoelinga, M.: DFTCalc: Reliability centered maintenance via fault tree analysis (tool paper). In: ICFEM 2015. LNCS, vol. 9407, pp. 304–311. Springer (2015). https://doi.org/10.1007/978-3-319-25423-4_19
22. Guck, D., Katoen, J.P., Stoelinga, M., Luiten, T., Romijn, J.: Smart railroad maintenance engineering with stochastic model checking. In: Railways 2014. Civil-Comp Proceedings, Civil-Comp Press (2014). https://doi.org/10.4203/ccp.104.299
23. Heidelberger, P.: Fast simulation of rare events in queueing and reliability models. ACM Trans. Model. Comput. Simul. **5**(1), 43–85 (1995). https://doi.org/10.1145/203091.203094
24. Jegourel, C., Legay, A., Sedwards, S.: Importance splitting for statistical model checking rare properties. In: CAV 2013. LNCS, vol. 8044, pp. 576–591. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_38
25. Jégourel, C., Legay, A., Sedwards, S., Traonouez, L.M.: Distributed verification of rare properties using importance splitting observers. In: AVoCS 2015. ECEASST, vol. 72 (2015). https://doi.org/10.14279/tuj.eceasst.72.1024
26. Junges, S., Guck, D., Katoen, J.P., Rensink, A., Stoelinga, M.: Fault trees on a diet. In: SETTA 2015. LNCS, vol. 9409, pp. 3–18. Springer (2015). https://doi.org/10.1007/978-3-319-25942-0_1
27. Junges, S., Guck, D., Katoen, J., Stoelinga, M.: Uncovering dynamic fault trees. In: DSN 2016. pp. 299–310. IEEE Computer Society (2016). https://doi.org/10.1109/DSN.2016.35
28. Kahn, H., Harris, T.E.: Estimation of particle transmission by random sampling. National Bureau of Standards applied mathematics series **12**, 27–30 (1951)
29. Katoen, J.P., Stoelinga, M.: Boosting Fault Tree Analysis by Formal Methods, LNCS, vol. 10500, pp. 368–389. Springer (2017). https://doi.org/10.1007/978-3-319-68270-9_19
30. L'Ecuyer, P., Le Gland, F., Lezaud, P., Tuffin, B.: Splitting techniques. In: Rubino and Tuffin [36], pp. 39–61. https://doi.org/10.1002/9780470745403.ch3
31. Liu, Y., Wu, Y., Kalbarczyk, Z.: Smart maintenance via dynamic fault tree analysis: A case study on Singapore MRT system. In: DSN 2017. pp. 511–518. IEEE Computer Society (2017). https://doi.org/10.1109/DSN.2017.50
32. Monti, R.E.: Stochastic Automata for Fault Tolerant Concurrent Systems. Ph.D. thesis, FAMAF, Universidad Nacional de Córdoba, Córdoba, Argentina (2018)
33. Nicola, V.F., Shahabuddin, P., Nakayama, M.K.: Techniques for fast simulation of models of highly dependable systems. IEEE Transactions on Reliability **50**(3), 246–264 (2001). https://doi.org/10.1109/24.974122

34. Ridder, A.: Importance sampling simulations of Markovian reliability systems using cross-entropy. Annals of Operations Research **134**(1), 119–136 (2005). https://doi.org/10.1007/s10479-005-5727-9
35. Rubino, G., Tuffin, B.: Introduction to rare event simulation. In: Rare Event Simulation Using Monte Carlo Methods [36], pp. 1–13. https://doi.org/10.1002/9780470745403.ch1
36. Rubino, G., Tuffin, B. (eds.): Rare Event Simulation Using Monte Carlo Methods. John Wiley & Sons, Ltd (2009)
37. Ruijters, E., Guck, D., Drolenga, P., Peters, M., Stoelinga, M.: Maintenance analysis and optimization via statistical model checking. In: QEST 2016. LNCS, vol. 9826, pp. 331–347. Springer (2016). https://doi.org/10.1007/978-3-319-43425-4_22
38. Ruijters, E., Guck, D., van Noort, M., Stoelinga, M.: Reliability-centered maintenance of the electrically insulated railway joint via fault tree analysis: A practical experience report. In: DSN 2016. pp. 662–669. IEEE Computer Society (2016). https://doi.org/10.1109/DSN.2016.67
39. Ruijters, E., Reijsbergen, D., de Boer, P.T., Stoelinga, M.: Rare event simulation for dynamic fault trees. Reliability Engineering & System Safety **186**, 220–231 (2019). https://doi.org/10.1016/j.ress.2019.02.004
40. Ruijters, E., Stoelinga, M.: Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. Computer Science Review **15-16**, 29–62 (2015). https://doi.org/10.1016/j.cosrev.2015.03.001
41. Sullivan, K.J., Dugan, J.B.: Galileo user's manual & design overview. https://www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm (1998), v2.1-alpha
42. Sullivan, K., Dugan, J., Coppit, D.: The Galileo fault tree analysis tool. In: 29th Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352). pp. 232–235. IEEE (1999). https://doi.org/10.1109/FTCS.1999.781056
43. Vesely, W., Stamatelatos, M., Dugan, J., Fragola, J., Minarick, J., Railsback, J.: Fault tree handbook with aerospace applications. NASA Office of Safety and Mission Assurance (2002), version 1.1
44. Villén-Altamirano, J.: RESTART method for the case where rare events can occur in retrials from any threshold. Int. J. Electron. Commun. **52**(3), 183–189 (1998)
45. Villén-Altamirano, J.: Importance functions for RESTART simulation of highly-dependable systems. Simulation **83**(12), 821–828 (2007). https://doi.org/10.1177/0037549707081257
46. Villén-Altamirano, J.: RESTART vs splitting: A comparative study. Performance Evaluation **121-122**, 38–47 (2018). https://doi.org/10.1016/j.peva.2018.02.002
47. Villén-Altamirano, M., Martínez-Marrón, A., Gamo, J., Fernández-Cuesta, F.: Enhancement of the accelerated simulation method RESTART by considering multiple thresholds. In: Proc. 14th Int. Teletraffic Congress, Teletraffic Science and Engineering, vol. 1, pp. 797–810. Elsevier (1994). https://doi.org/10.1016/B978-0-444-82031-0.50084-6
48. Villén-Altamirano, M., Villén-Altamirano, J.: RESTART: a method for accelerating rare event simulations. In: Queueing, Performance and Control in ATM (ITC-13). pp. 71–76. Elsevier (1991)
49. Xiao, G., Li, Z., Li, T.: Dependability estimation for non-Markov consecutive-k-out-of-n: F repairable systems by fast simulation. Reliability Engineering & System Safety **92**(3), 293–299 (2007). https://doi.org/10.1016/j.ress.2006.04.004

FIG: the Finite Improbability Generator[✠]

Carlos E. Budde[1]

Formal Methods and Tools, University of Twente,
Enschede, the Netherlands c.e.budde@utwente.nl

Abstract. This paper introduces the statistical model checker FIG 1.2, that estimates transient and steady-state reachability properties in stochastic automata. This software tool specialises in Rare Event Simulation via importance splitting, and implements the algorithms RESTART and Fixed Effort. FIG is push-button automatic since the user need not define an importance function: this function is derived from the model specification plus the property query. The tool operates with Input/Output Stochastic Automata with Urgency, aka IOSA models, described either in the native syntax or in the JANI exchange format. The theory backing FIG has demonstrated good efficiency, comparable to optimal importance splitting implemented ad hoc for specific models. Written in C++, FIG can outperform other state-of-the-art tools for Rare Event Simulation.

1 Introduction

In formal analysis of stochastic systems, statistical model checking (SMC [33]) emerges as an alternative to numerical techniques such as (exhaustive) probabilistic model checking. Its partial, on-demand state exploration offers a memory-lightweight option to exhaustive explorations. At its core, SMC integrates Monte Carlo simulation with formal models, where traces of states are generated dynamically e.g. via discrete event simulation. Such traces are samples of the states where a (possibly non-Markovian) stochastic model usually ferrets. Given a temporal logic property φ that characterises certain states, an SMC analysis yields an estimate $\hat{\gamma}$ of the actual probability γ with which the model satisfies φ. The estimate $\hat{\gamma}$ typically comes together with a quantification of the statistical error: two numbers $\delta \in (0,1)$ and $\varepsilon > 0$ such that $\hat{\gamma} \in [\gamma - \varepsilon, \gamma + \varepsilon]$ with probability δ. Thus, if n traces are sampled, the full SMC outcome is the tuple $(n, \hat{\gamma}, \delta, \varepsilon)$.

With this statistical quantification—usually presented as a confidence interval (CI) around $\hat{\gamma}$—an idea of the quality of an estimation is conveyed. To increase the quality one must increase the *precision* (smaller ε) or the *confidence* (bigger δ). For fixed confidence, this means a narrower CI around $\hat{\gamma}$. The number of traces n is inversely proportional to ε and to the CI width, so SMC trades memory for runtime or precision when compared to exhaustive methods [5].

This trade-off of SMC comes with one up and one down. The **up** is the capability to analyse systems whose stochastic transitions can have non-Markovian

[✠] This work was partially funded by NWO, NS, and ProRail project 15474 (*SEQUOIA*) and EU project 102112 (*SUCCESS*).

distributions. In spite of gallant efforts, this is still out of reach for most other model checking approaches, making SMC unique. The **down** are rare events. If there is a very low probability to visit the states characterised by the property φ, then most traces will not visit them. Thus the estimate $\hat{\gamma}$ is either (an incorrect) 0 or, if a few traces do visit these states, statistical error quantification make ε skyrocket. To counter such phenomenon, n must increase as γ decreases. Unfortunately, for typical estimates such as the sample mean, it takes $n \geqslant {}^{384}\!/\gamma$ to build a (rather lax!) CI where $\delta = 0.95$ and $\varepsilon = \frac{\gamma}{10}$. If e.g. $\gamma \approx 10^{-8}$ then $n \geqslant 38400000000$ traces are needed, causing trace-sampling times to grow unacceptably long. Rare Event Simulation (RES [24]) methods tackle this issue.

The two main RES methods are importance sampling (IS) and importance splitting (ISPLIT). IS compromises the aforementioned **up**, since it must tamper the stochastic transitions of the model. Given that the study of non-Markovian systems is a chief reason to use SMC, FIG, a statistical model checker specialised in RES, implements ISPLIT. To deploy an efficient implementation, however, both importance sampling and splitting require expert knowledge. The novelty of FIG lies on its automatic derivation of the importance function (and thresholds and splitting values) required by ISPLIT. This derivation exploits the model and property under study, resulting in a push-button application of RES for SMC.

Outline. The way in which FIG approaches RES is explained in Sec. 2. Its model and properties input syntax are presented in Sec. 3. Finally, Sec. 4 mentions some features of FIG 1.2, before ending the paper with the briefest experimental display.

Related work. Other statistical model checkers offer RES methods to some degree of automation. Plasma Lab implements automatic IS [18] and semiautomatic ISPLIT [21] for Markov chains. Its ISPLIT engine offers a wizard that guides the user to choose an importance function. The wizard exploits a layered decomposition of the property query—not the system model. Via APIs, the ISPLIT engine of Plasma Lab could be extended beyond DTMC models. $\mathcal{S}$BIP 2.0 [22] implements the same (semiautomatic, property-based) engine for DTMCs. $\mathcal{S}$BIP offers a richer set of temporal logics to define the property query in. Cosmos [1] and FTRES [26] implement importance sampling on Markov chains, the latter specialising in systems described as repairable Dynamic Fault Trees (DFTs). All these tools can operate directly on Markovian models, and none offers fully automated ISPLIT. Instead, the SMC tool modes [5] supports non-Markovian probability distributions and is much closer to the capabilities of FIG, offering a similar degree of automation. As a matter of fact, all core RES algorithms in modes were inspired in or motivated by the theory behind FIG. On the one hand, FIG is restricted to fully-stochastic IOSA models, whereas modes can also cope with nondeterminism (e.g. in Markov automata) using the LSS algorithm [10, 5]. On the other hand, using the batch means method, FIG can estimate steady-state properties, which modes cannot currently do. Moreover, FIG 1.2 implements basic functionality to tailor importance functions for DFTs.

Previous versions of FIG have been used for scientific experimentation and research: the theory of [6] was first implemented and exercised with FIG 1.0; and FIG 1.1 was presented in [2], and last used in an extended journal version of [5].

2 Rare Event Simulation

RES methods make more traces visit the rare states that satisfy a property φ (the set $\mathcal{S}_\varphi$), to reduce the variance of SMC estimators. For a fixed budget of traces n, this yields more precise CIs than classical Monte Carlo simulation (CMC).

FIG implements *importance splitting*: a main RES method that can work on non-Markovian systems without special considerations. ISPLIT splits the states of the model into layers that wrap $\mathcal{S}_\varphi$ like an onion. Reaching a state in $\mathcal{S}_\varphi$ from the surface is then broken down into many steps. The i-th step estimates the conditional probability to reach (the inner) layer $i+1$ from (the outer) layer i. This stepwise estimation of conditional probabilities can be much more efficient than trying to go in one leap from the surface of the onion to its core [20].

Formally, let $\mathcal{S}$ be the states of a model with initial states $\mathcal{S}_0$ and rare states $\mathcal{S}_\varphi$. ISPLIT works on a partition $\biguplus_{i=0}^{M} \mathcal{S}_i = \mathcal{S}$, where $\mathcal{S}_\varphi = \mathcal{S}_M$. To estimate the probability $\gamma = \mathit{Prob}(\mathcal{S}_\varphi \,|\, \mathcal{S}_0)$, each conditional probability $\gamma_i = \mathit{Prob}(\mathcal{S}_i \,|\, \mathcal{S}_{i-1})$ is estimated separately via CMC. Then simply $\hat{\gamma} = \prod_{i=1}^{M} \hat{\gamma}_i \approx \prod_{i=1}^{M} \gamma_i = \gamma$.

This approach is correct, i.e. it yields an unbiased estimator $\hat{\gamma} \xrightarrow{n\to\infty} \gamma$. However, it is efficient iff $\forall_{i=1}^{M} . \gamma_i \gg \gamma$, which depends on how the $\mathcal{S}_i$ layers where chosen. For this, an *importance function* $f\colon \mathcal{S} \to \mathbb{R}_{\geqslant 0}$ and *thresholds* $\ell_i \in \mathbb{R}_{\geqslant 0}$ are defined: then $\mathcal{S}_i = \{s \in \mathcal{S} \mid \ell_i \leqslant f(s) < \ell_{i+1}\}$, where $\ell_0 = 0$, and $\mathcal{S}_\varphi$ are the states with highest *importance*, i.e. $f(s) \geqslant \ell_M$. The efficiency of ISPLIT is thus delegated to the choice of $\{\ell_i\}_{i=1}^{M}$ and the importance function f.

These choices are *the* key challenge in ISPLIT [20]. Theoretical developments assume f is given [12, 8], and applications define it ad hoc via (RES and domain) expert knowledge [30, 27]. Yet there is one general rule: importance must be proportional to the probability of reaching $\mathcal{S}_\varphi$. Thus for $s, s' \in \mathcal{S}$, if a trace that visits s' is more likely to observe a rare state, one wants $f(s) \leqslant f(s')$. This means that f depends both on the model M and the property φ that define $\mathcal{S}_\varphi$.

FIG, an SMC tool, exploits the formal definitions of M and φ to derive f and $\{\ell_i\}_{i=1}^{M}$ so as to reflect this rule. For this, FIG runs BFS from $\mathcal{S}_\varphi$ on the (inverted) transitions of M. This computes the number-of-transitions distance from each state to $\mathcal{S}_\varphi$. The heuristic importance function of FIG, $f^\star$, is the inverse of this distance, stored as an array the size of $\mathcal{S}$. To avoid the state explosion FIG works on modular formalisms, deriving local $f_i^\star$ for the M_i whose parallel composition forms M. $f^\star$ is an aggregation of these functions, e.g. adding the $f_i^\star$ of every M_i with variables in φ. Details are in [2] and also in [5], where the difference with the (later) implementation in modes is that FIG uses the DNF of φ.

$f^\star$ is solely based on the number-of-transitions distance. Stochastic behaviour of M omitted by $f^\star$, such as probabilistic labels in the transitions, is captured in the thresholds ℓ_i. For this, FIG runs short simulations that start from $\mathcal{S}_0$. Say K_1 out of N simulations visit states with importance $i_1 > i_0 = f^\star(\mathcal{S}_0)$. Then, 1 out of $e_1 = \left\lceil \frac{N}{K_1} \right\rceil$ simulations are expected to reach threshold $\ell_1 = i_1$. Next, repeat this procedure starting from states with importance i_1 to choose ℓ_2 and e_2. Etc. Such threshold-selection algorithms (see Sec. 4) are fully described in [4].

Thus, just from M and φ, FIG enables ISPLIT by computing $f^\star$ and $\{\ell_i, e_i\}_{i=1}^{M}$.

3 Modelling formalism and input languages

IOSA. FIG models are Input/Output Stochastic Automata with urgency [11]. In IOSA, continuous variables called *clocks* sample random values from arbitrary distributions (PDFs). As time evolves, all clocks count down at the same rate. The first to reach zero can trigger events and synchronise with other modules, broadcasting an *output* action that synchronises with homonymous *input* actions (IOSA are input-enabled). Actions can be urgent, where urgent outputs have maximal progress. IOSA can thus be nondeterministic: to allow simulation, [23] gives conditions to ensure determinism modulo weak bisimulation. IOSA *variables* are clocks, integers, or Booleans. *Constants* can also be floats and have global scope (variables are module-local). FIG offers array variables and can get e.g. "a-random/the-smallest value." Code 1 shows the guarded command language of FIG models. Decorators `?`/`!` tell an action is input/output, e.g. `fl!`. Double decorators (`r??`) are for urgency. Non-urgent outputs can be sent only on clock expiration (`[fl!]`···`@ fc ->`). A clock can sample random values (`fc'=`μ).

```
module M1
  fc,rc    : clock;
  inf,brk  : [0..2] init 0;
  [fl!] brk==0 @ fc -> (inf'=1)
                     & (brk'=1);
  [r??] brk==1 ->(brk'=2) & (rc'=γ);
  [up!] brk==2 @ rc -> (inf'=2)
                     & (brk'=0)
                     & (fc'=μ);
  [f!!] inf==1 -> (inf'=0);
  [u!!] inf==2 -> (inf'=0);
endmodule
```

Code 1: IOSA module in FIG 1.2

JANI. Besides its native input syntax, FIG 1.2 reads models written in the JANI exchange format [7]. Model types supported are CTMC and a subset of STA that matches IOSA, e.g. with a single PDF per clock and broadcast synchronisation. FIG also translates IOSA to JANI as STA, to share models with tools such as the Modest Toolset [16] and Storm [13]. This is used in Sec. 4 for comparisons.

Properties. FIG estimates the probability with which input models satisfy temporal logic formulæ. A formula is specified as a (transient or steady-state) property query in the model file. Transient properties in FIG correspond to the PCTL-like query `P=?` in PRISM [19]: e.g. the first property in Code 2 asks the probability of assigning value 8 to variable `q2` before it takes a value $\leqslant 0$. Steady-state properties in FIG correspond to the unbounded CSL-like query `S=?` in PRISM: e.g. `S(q2>=8)`. For steady-state estimations FIG implements batch means [9]. The initial (discarded) transient simulation time, and the batch time, can be heuristically computed by the tool. These values can also be given by the user—in Code 2, the last property specifies 9 and 999 resp.

```
properties
  P( q2>0 U q2==8 )
  S( q2>=8 )
  S[9:999]( q2>=8 )
endproperties
```

Code 2: Property queries in FIG

4 FIG 1.2 showcase

The *Finite Improbability Generator* is written in C++14 and is available at https://git.snt.utwente.nl/buddece/fig under the GNU GPLv3. FIG is built in modules across three categories: simulation engines, importance functions, and thresholds builders. Engines are `nosplit`, `restart`, and `sfe`, which resp. run CMC, RESTART (RST [31]), and Fixed Effort (FE [14]) simulations. The latter two are ISPLIT algorithms: FE was described in Sec. 2, and works for transient properties; RST also works for steady-state analysis (steady-state via FE requires regeneration theory [15], seldom applicable to non-Markovian models and unsupported by FIG 1.2). RST and FE work with an *effort* e. FE_e means e simulations are ran in a layer $\mathcal{S}_i$. RST_e means $e-1$ clones are spawned when a simulation up-crosses a threshold ℓ_i. Omitting e makes FIG 1.2 use respectively FE_8 or RST_3.

A RES run yields a random value $r \in [0,1]$ of unknown distribution, so FIG computes standard CLT confidence intervals with Student's t-distribution quantiles. r has a Bernoulli distribution only for transient properties estimated with CMC: FIG can then use Wilson score intervals [32]. Floating-point precision loss is reduced by using the logarithm of r and of the number of runs.

FIG reads or computes importance functions. Option `--adhoc` takes as mandatory argument a function on the variables of the IOSA modules. Instead, `--amono` automatically builds $f^\star$ on the parallel composition of all modules, and `--acomp` builds a local $f_i^\star$ per IOSA module—see Sec. 2. For `--acomp`, FIG takes an optional argument to aggregate all local $f_i^\star$ into one global $f^\star$. This can be an associative binary arithmetic operator, or a custom function on the names of the IOSA modules. By default, $f^\star$ is computed as the sum of all local functions. Option `--dft 0` indicates that the model is a fault tree: FIG then builds specialised local importance functions for certain modules, e.g. basic events and PAND gates.

Two algorithms in FIG 1.2 can compute the thresholds and efforts $\{\ell_i, e_i\}_{i=1}^{M}$. Sequential Monte Carlo [8, 6] (SEQ, option `-t hyb`) is characterised by one effort for all regions $\mathcal{S}_i$, set with `-g` e. Instead, Expected Success [4] (ES, `-t es`) determines each effort e_i per $\mathcal{S}_i$ region. By default FIG 1.2 uses `-e restart -g 3 -t hyb`. Other customisable options are the RNG, its seed, the floating point precision, and a timeout. Mandatory arguments for FIG invocation are the model and properties file, the simulation type (`--flat` for CMC, or `--adhoc/amono/acomp` for RES), and a stop criterion (either time, or confidence and precision of the CI).

Experimental demonstration. We display the capabilities of FIG via three experiments. First, we show how ISPLIT implemented in FIG 1.2 is as automatic but more efficient than CMC to estimate rare properties. Second, we test the degree to which $f^\star$ in FIG can approximate optimal importance functions chosen ad hoc for some models. Third, we compare FIG and its closest competitor: modes. All these experiments can be reproduced via the artifact freely available in [3].

We test different configurations of engines, efforts, and thresholds. For each configuration we run simulations until some timeout. This yields a CI with precision 2ε for confidence coefficient $\delta = 0.95$. The smaller the ε, the narrower the CI, and the better the performance of the configuration (and tool) that produced it.

First, we analyse repairable DFTs with warm spares and exponential (fail), normal (repair), and lognormal (dormancy) PDFs. Using CMC, $\mathrm{FE}_{8,16,32}$ and $\mathrm{RST}_{3,4,6}$ we estimate the probability of a top level event after the first failure, before all components are repaired, in trees with 6, 7, and 8 spares (the smallest IOSA has 116 variables and $> 2.5\,\mathrm{e}\,37$ states). For ISPLIT we used SEQ thresholds with `--dft 0 --acomp` and no arguments, i.e. as automatic as CMC.

With a 20 min timeout, each configuration was repeated 13 times in a Xeon E5-2683v4 CPU running Linux x64 4.4.0. The height of the bars in the top plot of Fig. 1 is the average CI precision (lower is better), using $\text{Z-score}_{m=2}$ to remove

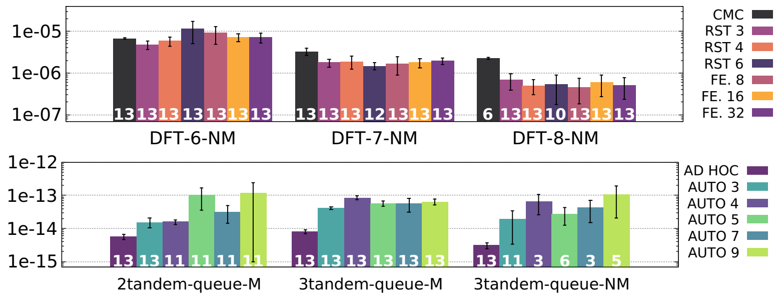


Fig. 1: CI precision. Top: DFTs (transient). Bottom: queues (steady-state).

outliers [17]. Whiskers are standard deviation, and white numbers indicate how many runs yielded not-null estimates. Clearly, RES algorithms outperform CMC in the hardest cases: less than half of CMC runs in DFT-8 could build (wide) CIs.

Second, we estimate the steady-state overflow probability in the last node of tandem queues, on a Markovian case with 2 buffers [29], 3 buffers [28], and a non-Markovian 3-buffers case [30]. We study how FIG—using --amono, SEQ, and $\mathrm{RST}_{3,4,5,7,9}$—approximates each optimal ad hoc function and thresholds of [29, 28, 30]. Experiments ran as before: the bottom plot of Fig. 1 shows that FIG's default (RST_3 with SEQ, legend "AUTO 3") is always closest to the optimal.

Third, we compare FIG and modes in the original benchmark of the latter [5]. We do so for FE-SEQ, RST-SEQ, RST-ES, using each tool's default options. We ran each benchmark instance 15 min, thrice per tool, in an Intel i7-6700 CPU with Linux x64 5.3.1. The scatter plots of Fig. 2 show the median of the CI precisions. Sub-plots on the bottom-right are a zoom-ins in the range $[10^{-10},10^{-5}]$.

An (x,y) point is an instance whose median CI width was x for FIG 1.2 and y for modes netcore-3.0.150, single threaded. A point over the solid diagonal line means FIG built a narrower CI. A point on the upper boundary means that modes built no CIs in all runs. Dotted diagonal lines indicate CIs twice as wide. Fig. 2 shows that both tools perform similarly, with a slight trend in favour of FIG. This could be caused by modes operating on JANI STA (translated from IOSA by FIG): modes must assign values to variables and then compare them to clocks.

Albeit modes is multi-threaded, these experiments ran on a single thread to compare both tools on equal conditions. On the other hand, FIG also estimates the probability of steady-state properties, for which there is no support in modes.

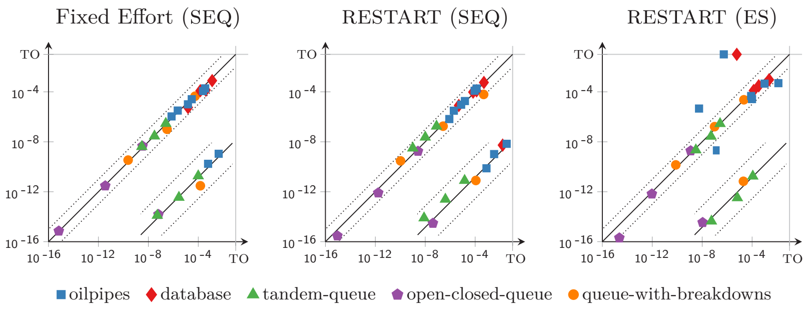


Fig. 2: CI precision of FIG (x-axis) vs. modes (y-axis): medians of 3 runs × 15 min

References

1. Barbot, B., Haddad, S., Picaronny, C.: Coupling and importance sampling for statistical model checking. In: TACAS. LNCS, vol. 7214, pp. 331–346. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_23
2. Budde, C.E.: Automation of Importance Splitting Techniques for Rare Event Simulation. Ph.D. thesis, FAMAF, Universidad Nacional de Córdoba, Córdoba, Argentina (2017), https://famaf.biblio.unc.edu.ar/cgi-bin/koha/opac-detail.pl?biblionumber=18143
3. Budde, C.E.: FIG: the Finite Improbability Generator. 4TU.Centre for Research Data (2020). https://doi.org/10.4121/uuid:1d5ddcd6-b3a9-4425-92b3-c46db98b7d8e
4. Budde, C.E., D'Argenio, P.R., Hartmanns, A.: Better automated importance splitting for transient rare events. In: SETTA. LNCS, vol. 10606, pp. 42–58. Springer (2017). https://doi.org/10.1007/978-3-319-69483-2_3
5. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: A statistical model checker for nondeterminism and rare events. In: TACAS. LNCS, vol. 10806, pp. 340–358. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_20
6. Budde, C.E., D'Argenio, P.R., Monti, R.E.: Compositional construction of importance functions in fully automated importance splitting. In: VALUETOOLS. ICST (2016). https://doi.org/10.4108/eai.25-10-2016.2266501
7. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: TACAS. LNCS, vol. 10206, pp. 151–168. Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_9
8. Cérou, F., Del Moral, P., Furon, T., Guyader, A.: Sequential Monte Carlo for rare event estimation. Statistics and Computing **22**(3), 795–808 (2012). https://doi.org/10.1007/s11222-011-9231-6
9. Conway, R.: Some tactical problems in digital simulation. Management Science **10**(1), 47–61 (1963). https://doi.org/10.1287/mnsc.10.1.47
10. D'Argenio, P.R., Legay, A., Sedwards, S., Traonouez, L.M.: Smart sampling for lightweight verification of Markov decision processes. STTT **17**(4), 469–484 (2015). https://doi.org/10.1007/s10009-015-0383-0
11. D'Argenio, P.R., Monti, R.E.: Input/Output Stochastic Automata with Urgency: Confluence and weak determinism. In: ICTAC. LNCS, vol. 11187, pp. 132–152. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_8
12. Dean, T., Dupuis, P.: Splitting for rare event simulation: A large deviation approach to design and analysis. Stochastic Processes and their Applications **119**(2), 562–587 (2009). https://doi.org/10.1016/j.spa.2008.02.017
13. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: CAV. LNCS, vol. 10427, pp. 592–600. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_31
14. Garvels, M.J.J., van Ommeren, J.C.W., Kroese, D.P.: On the importance function in splitting simulation. Eur. Trans. Telecommun. **13**(4), 363–371 (2002). https://doi.org/10.1002/ett.4460130408
15. Garvels, M.J.J.: The splitting method in rare event simulation. Ph.D. thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands (2000), http://eprints.eemcs.utwente.nl/14291/
16. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS. LNCS, vol. 8413, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_51

17. Iglewicz, B., Hoaglin, D.: How to Detect and Handle Outliers. ASQC basic references in quality control, ASQC Quality Press (1993)
18. Jégourel, C., Legay, A., Sedwards, S.: Command-based importance sampling for statistical model checking. Theor. Comput. Sci. **649**, 1–24 (2016). https://doi.org/10.1016/j.tcs.2016.08.009
19. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. LNCS, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47
20. L'Ecuyer, P., Le Gland, F., Lezaud, P., Tuffin, B.: Splitting techniques. In: Rubino and Tuffin [25], pp. 39–61. https://doi.org/10.1002/9780470745403.ch3
21. Legay, A., Sedwards, S., Traonouez, L.M.: Plasma Lab: A modular statistical model checking platform. In: ISoLA. LNCS, vol. 9952, pp. 77–93 (2016). https://doi.org/10.1007/978-3-319-47166-2_6
22. Mediouni, B.L., Nouri, A., Bozga, M., Dellabani, M., Legay, A., Bensalem, S.: $\mathcal{S}$BIP 2.0: Statistical model checking stochastic real-time systems. In: ATVA. LNCS, vol. 11138, pp. 536–542. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_33
23. Monti, R.E.: Stochastic Automata for Fault Tolerant Concurrent Systems. Ph.D. thesis, FAMAF, Universidad Nacional de Córdoba, Córdoba, Argentina (2018)
24. Rubino, G., Tuffin, B.: Introduction to rare event simulation. In: Rubino and Tuffin [25], pp. 1–13. https://doi.org/10.1002/9780470745403.ch1
25. Rubino, G., Tuffin, B. (eds.): Rare Event Simulation Using Monte Carlo Methods. Wiley (2009). https://doi.org/10.1002/9780470745403
26. Ruijters, E., Reijsbergen, D., de Boer, P.T., Stoelinga, M.: Rare event simulation for dynamic fault trees. Reliability Engineering & System Safety **186**, 220–231 (2019). https://doi.org/10.1016/j.ress.2019.02.004
27. Turati, P., Pedroni, N., Zio, E.: Advanced RESTART method for the estimation of the probability of failure of highly reliable hybrid dynamic systems. Reliability Engineering & System Safety **154**(C), 117–126 (2016). https://doi.org/10.1016/j.ress.2016.04.020
28. Villén-Altamirano, J.: Importance functions for restart simulation of general Jackson networks. European Journal of Operational Research **203**(1), 156–165 (2010). https://doi.org/10.1016/j.ejor.2009.07.013
29. Villén-Altamirano, J.: RESTART vs Splitting: A comparative study. Performance Evaluation **121–122**, 38–47 (2018). https://doi.org/10.1016/j.peva.2018.02.002
30. Villén-Altamirano, J., Villén-Altamirano, M.: Rare event simulation of non-Markovian queueing networks using RESTART method. Simulation Modelling Practice and Theory **37**, 70–78 (2013). https://doi.org/10.1016/j.simpat.2013.05.012
31. Villén-Altamirano, M., Villén-Altamirano, J.: RESTART: a method for accelerating rare event simulations. In: Queueing, Performance and Control in ATM (ITC-13). pp. 71–76. Elsevier (1991)
32. Wilson, E.B.: Probable inference, the law of succession, and statistical inference. Journal of the American Statistical Association **22**(158), 209–212 (1927). https://doi.org/10.1080/01621459.1927.10502953
33. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV. LNCS, vol. 2404, pp. 223–235. Springer (2002). https://doi.org/10.1007/3-540-45657-0_17

Acknowledgments. The author thanks Arnd Hartmanns for excellent discussions that originally motivated and subsequently helped to shape this work.

Mora - Automatic Generation of Moment-Based Invariants

Ezio Bartocci[1], Laura Kovács[1,2], and Miroslav Stankovič[1]

[1] TU Wien, Vienna, Austria
[2] Chalmers, Gothenburg, Sweden

Abstract. We introduce Mora, an automated tool for generating invariants of probabilistic programs. Inputs to Mora are so-called Prob-solvable loops, that is probabilistic programs with polynomial assignments over random variables and parametrized distributions. Combining methods from symbolic computation and statistics, Mora computes invariant properties over higher-order moments of loop variables, expressing, for example, statistical properties, such as expected values and variances, over the value distribution of loop variables.

1 Introduction

Probabilistic programs (PPs) are becoming more and more commonplace. Originally employed in randomized algorithms and cryptographic/privacy protocols, now gaining momentum due to the several emerging applications in the areas of machine learning and AI [5]. By introducing randomness into the program, program variables can no longer be treated as having single values; we must think about them as distributions. Dealing with distributions is much more challenging and some simplifications are required. Existing approaches, see e.g. [1,3,7,9,10], usually take into consideration only expected values or upper and lower bounds over program variables, or rely on user guidance for providing templates and hints.

One of the main challenges in analyzing PPs and computing their higher-order moments comes with the presence of loops and the burden of computing so-called *quantitative invariants* [7]. Quantitative invariants are properties that are true before and after each loop iteration and are crucial for analyzing the behavior of PP loops.

In this paper, we introduce the Mora tool for computing quantitative invariants of a class of PPs, called *Prob-solvable loops* [2], with random assignments, parametrized distributions, and polynomial probabilistic updates. Our implementation is available at:

https://github.com/miroslav21/mora,

and successfully evaluated on a number of challenging examples. Unlike other existing approaches, e.g. [1,3,7,9], Mora computes non-linear invariants in a fully automatic way, without relying on user-provided templates/hints. The proposed automatic approach can handle an arbitrary number of loop iterations and also infinite loops. On the contrary, tools like PSI [4] support only the automatic analysis of probabilistic programs with a specified number of loop iterations.

This research was supported by the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014 TheProSE, and the Austrian FWF project W1255-N23.

```
x=0
while true:
  u = RV(uniform, 0, b)
  g = RV(gauss, 0, 1)
  x = x - u @ 1/2; x + u @ 1/2
  y = y + x + g
```

Loop conditions are ignored, yielding non-deterministic PPs. The value of the random variable `u` is sampled by a uniform distribution with support in the real interval `[0, b]`, whereas the value of `g` is a random number from a normal distribution with mean (first moment) 0 and variance (second moment) 1. Updates to variable `x` are probabilistic: with probability `1/2`, the variable `x` is updated by `x-u`. Similarly, with probability `1/2`, `x` is updated by `x+u`. Further, updates to `u` and `g` do not depend on other variables; the update to `x` depends only on itself and `u`.

Fig. 1. An illustrative example of a Prob-solvable loop.

Moreover, the invariants inferred by MORA are not restricted to expected values but are quantitative invariants over the higher-order moments of program variables. We refer to such invariants as *moment-based invariants* [2]. To the best of our knowledge, no other approach can so far automatically compute higher-order moments of PPs, not even for the restricted yet expressive enough class of Prob-solvable loop.

The purpose of this paper is to describe what MORA can do and how it can be used. The paper is intended as a tool demonstration and guide for potential users of MORA. We focus on the usage and implementation aspects of MORA. For details on theoretical foundations and algorithmic aspects of MORA for computing moment-based invariants, we refer to [2]. We note however that, when compared to the experimental setup of [2], MORA comes with a completely new design, fully implemented in `python` and supporting an easy installation and use by even non-experts in PPs.

2 MORA– Programming Model

Input programs to MORA are PP loops that are Prob-solvable [2]. In Figure 1, we give an example of a Prob-solvable loop and use this example as a running example to guide the potential users of MORA in the rest of this paper.

In a nutshell, the probabilistic assignments of Prob-solvable loops involve (i) variable values drawn from random distributions, such as uniform or normal distributions, and (ii) random variable updates. In the sequel, we write `RV` to refer to a random variable. Input programs to MORA thus satisfy the following two properties:

(1) Input programs to MORA are PPs generated from the grammar in Figure 2.

(2) In addition to the grammar of Figure 2, MORA requires its PP input to be Prob-solvable, imposing further restrictions as follows:

- PP loop variables are different from each other and from parameters;
- probabilities used within a variable update sum up to 1;
- updated variables depend on themselves linearly and may depend polynomially only on other variables that have been previously updated.

Note that Figure 1 satisfies all constraints above, and thus is Prob-solvable.

Grammar defining PP inputs to Mora

```
PROGRAM → INIT_ASSIGNS "while true:" RV_ASSIGNS UPD_ASSIGNS

INIT_ASSIGNS → INIT_ASSIGN | INIT_ASSIGN INIT_ASSIGNS
RV_ASSIGNS → RV_ASSIGN | RV_ASSIGN RV_ASSIGNS
UPD_ASSIGNS → UPD_ASSIGN | UPD_ASSIGN UPD_ASSIGNS

INIT_ASSIGN → VAR " = " INIT_EXPR
RV_ASSIGN → VAR " = " RV_EXPR
UPD_ASSIGN → VAR " = " UPD_BRANCHES

UPD_BRANCHES → UPD_BRANCH | UPD_BRANCH UPD_BRANCHES
UPD_BRANCH → UPD_EXPR "@" UPD_PROB
UPD_PROB → SIMP_EXPR

INIT_EXPR → RV_EXPR |  SIMP_EXPR
RV_EXPR →  "RV(uniform, " SIMP_EXPR ", " SIMP_EXPR ")"
                | "RV(gauss, " SIMP_EXPR ", " SIMP_EXPR ")"
UPD_EXPR → UPD_EXPR OP UPD_EXPR | VAR | ATOM
SIMP_EXPR → SIMP_EXPR OP SIMP_EXPR | ATOM

ATOM → NUM | PARAMETER
OP → [∗+−]
VAR → [a–zA–Z][a–zA–Z0−9]∗
PARAMETER → [a–zA–Z][a–zA–Z0−9]∗
NUM → [−]?[0−9]+[.]?[0−9]∗([\/][1−9][0−9]∗)?
```

Fig. 2.

3 Mora– Usage

We describe the easiest way Mora can be used to generate moment-based invariants:

– Save a Prob-solvable loop to a file, for example save Figure 1 in the file `running`
– In the main Mora folder invoke `python` with `python3.7` and execute:

```
from mora.mora import mora
```

– Run Mora using the command:

```
mora("running", goal=GOAL),
```

where `GOAL` can be (i) a specific natural number $k \geq 1$, in which case Mora computes the kth moments of all variables from `running`; (ii) a specific moment of one loop variable of `running` (e.g. `"x^2"` specifying the second moment of a variable `x` of Figure 1); or (iii) a list containing the goals as just specified. One can specify finitely many goals as inputs to Mora; yet, at least one goal is required. For example, by running `mora("running", [1, "x^2", "x^3"])`, Mora computes the expected values (first moments, i.e. 1) of all variables from Figure 1, as well as the second and third moments of variable `x` of Figure 1 (specified by `x^2` and `x^3`, respectively).

Mora is completely automatic. That is, once an execution of Mora is started on a given Prob-solvable loop and input goals, Mora outputs the higher-order moments, and thus moment-based invariants, of its loop w.r.t. the specified input goals. To this end, Mora computes the expected values of all monomials over loop variables, on which one of the goals from `Goal` depends. In general, computing the `k`th moment requires computing the expected values of all monomial expressions over loop variables, such that the total degree of the monomials is less or equal than k – see [2] for more details.

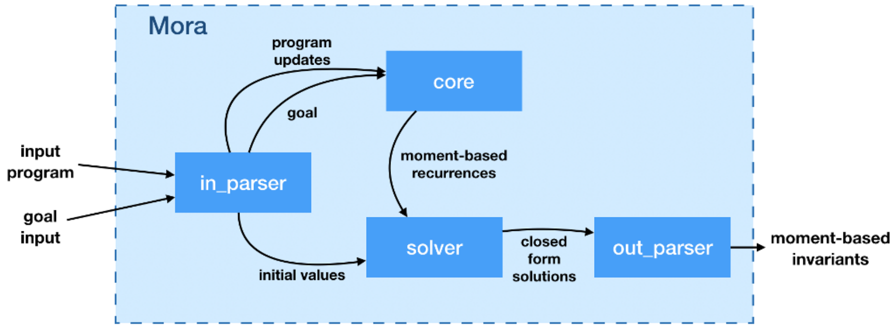


Fig. 3. Mora workflow diagram.

In the rest of the paper, we will illustrate the main steps of Mora, by considering Figure 1 as its input loop and `[1, 2]` as its list of input goals. With such an input goal, Mora is set to compute the first and second moments of each variable of Figure 1. Note, that even if `1` was omitted from the aforementioned input goal, Mora would still need to compute some of the first moments of the variables, as they are required for computing the second-order moments. In the sequel, we show-case the Mora behaviour for:

```
mora("running", [1, 2]).                                (1)
```

4 Mora– Tool Overview

We first give details on our implementation. We then present the overall workflow of Mora in Figure 3, based on which we overview the main components of our tool.

Overall Implementation. Mora is implemented in `python3`, requiring `python` version of at least 3.7. Mora relies on the `diofant` and `scipy` libraries: (i) the `python` library `diofant` is used in Mora for symbolic mathematical computations and recurrence solving; (ii) the `scipy` library, and in particular its statistics module `scipy.stats`, is used in Mora to handle probability distributions and statistical functions, as well as to simplify and compute expressions involving probability distributions and initial values of variables. Altogether, our implementation comprises of around 350 lines of code.

Mora – *Parser.* Mora first checks whether a given input program is Prob-solvable, by checking the requirements of Section 2. If the input program is not Prob-solvable, an error is reported, and the execution of Mora stops. Otherwise, within its parser module, Mora extracts initial values from its input loop, rewrites loop updates into equations over expected values of monomial expressions over loop variables, and processes the list of its input goals to identify which higher-order moments need to be computed.

For our demo execution (1), Mora extracts the initial value `x(0)=0`, where `x(0)` denotes the initial value of `x` before the loop. Using the input goals specified in (1), Mora is set to compute the expected values of {`u, g, x, y, u^2, g^2, x^2,` `y^2` characterizing the first and second moments of all loop variables of Figure 1. Further, the loop updates of Figure 1 are rewritten by MORA into equations over expected values, as follows:

$$\begin{cases} E[x^k(n+1)] = E[1/2 \cdot (x(n) - u(n+1))^k + 1/2 \cdot (x(n) + u(n+1))^k] \\ E[y^k(n+1)] = E[(y(n) + x(n+1) + g(n+1))^k] \end{cases}, \tag{2}$$

where $n \geq 0$ is the loop counter of Figure 1, $x(n)$ denotes the value of `x` at the nth loop iteration, and $E[expr]$ is the expected value of an expression $expr$.

MORA – *Core*. After rewriting probabilistic loop updates into equations over expected values, MORA rewrites these equations into non-probabilistic recurrences over so-called E-variables, with the loop counter n being the recurrence index. E-variables are simply variables created from monomials over original variables. Thanks to the restrictions defining PPs to be Prob-solvable, the resulting recurrences are linear recurrences with constant coefficients, that is C-finite recurrences, whose closed forms can always be computed [8]. MORA solves these recurrences by calling its *Solver* module.

Using the equations (2) over expected values, the non-probabilistic recurrences of Figure 1 generated by MORA are as follows, using the MORA synthax:

```
y = x + y
g ∗ ∗2 = 1
x = x
u = b/2
x ∗ ∗2 = b ∗ ∗2/3 + x ∗ ∗2                                   (3)
u ∗ ∗2 = b ∗ ∗2/3
y ∗ ∗2 = b ∗ ∗2/3 + x ∗ ∗2 + 2 ∗ x ∗ y + y ∗ ∗2 + 1
g = 0
x ∗ y = b ∗ ∗2/3 + x ∗ ∗2 + x ∗ y
```

The left-hand sides of these equations represent values of E-variables at iteration $n+1$, while monomials over original variables on the right-hand side represent E-variables at iteration n. For example, the first equation of (3) stands for $E[y(n+1)] = E[x(n)] + E[y(n)]$. On the other hand, the fourth equation of (3) represents $E[x(n+1)^2] = \frac{b^2}{3} + E[x(n)^2]$, as b is a constant parameter and `x**k` in `python` denotes the kth power of x.

Solver. In this module, MORA extracts and solves recurrences from the non-probabilistic equations over E-variables computed by its *Core* module. By exploiting the structure of Prob-solvable programs, MORA also optimizes the order in which recurrences are solved, e.g. independent recurrences are solved first. Partial solutions can be used to reduce the complexity of the latter recurrences. MORA then uses the `diofant` library to handle and solve single recurrences.

For Figure 1, using the E-variable equations of (3), the following closed form solutions are computed by MORA:

$$\begin{aligned}
&E[u^2] = \tfrac{b^2}{3}\\
&E[x^1] = 0\\
&E[y^1] = y(0)\\
&E[x^2] = \tfrac{b^2 n}{3}\\
&E[u^1] = \tfrac{b}{2}\\
&E[y^1x^1] = \tfrac{b^2 n}{6}\,(n+1)\\
&E[y^2] = \tfrac{n}{18}\,\left(2b^2n^2 + 3b^2n + b^2 + 18\right) + y(0)^2\\
&E[g^1] = 0\\
&E[g^2] = 1
\end{aligned} \tag{4}$$

with $y(0)$ standing for the initial value of y (treated as a parameter, since not specified).

Mora – *Out_Parser.* Mora's output consists of basic information about the program and the goal, moment-based invariants computed, and computation time. By default, the Mora output is shown only on the screen. However, an optional argument can specify if an output file should be created. Two possible values for `output_format` are (i) `"txt"`, producing a simple human-readable file, and (ii) `"tex"`, producing a file with invariants in LaTeX format (as given in (4) above).

5 Evaluation

A proof-of-concept implementation, together with initial experiments, were already given in our work on generating moment-based invariants [2]. Mora comes however with a new design and re-implementation of [2], significantly improving the experimental setting and evaluations of [2]. Table 1 compares Mora against the experiments of [2], on a subset of Prob-solvable loops from [2], evidencing that Mora is faster than our initial proof-of-concept implementation. This is due to the following reasons:

- Mora now optimizes the order in which recurrences are sent to the `diofant` recurrence solver. This reduces the amount of necessary symbolic computation and speeds up the process.
- While Mora is implemented entirely in `python`, with limited usage of external libraries, the previous implementation was done in `Julia` and relied on calls to the `sympy` library of `python`.
- Mora does not rely on `Aligator` [6] for handling systems of recurrences, allowing us to eliminate some intermediate and redundant steps.

Program	Moment	Runtime PoC (s)	Runtime Mora (s)
SUM_RND_SERIES	1	0.31	0.22
	2	2.89	0.93
	3	17.7	2.47
STUTTERINGA	1	0.44	0.25
	2	2.20	1.07
	3	8.48	3.35
STUTTERINGC	1	1.80	0.66
	2	72.5	12.2
	3	2144	73.9
SQUARE	1	0.38	0.22
	2	2.46	0.73
	3	8.70	1.67

Table 1. Comparison of Mora vs. proof-of-concept (PoC) implementation of [2].

6 Conclusion

We described Mora, a fully automated tool for generating invariants of probabilistic programs. Mora combines recurrence solving, symbolic summation and statistical reasoning, and derives higher-order moments of loop variables in probabilistic programs.

References

1. Barthe, G., Espitau, T., Fioriti, L.M.F., Hsu, J.: Synthesizing Probabilistic Invariants via Doob's Decomposition. In: CAV. LNCS, vol. 9779, pp. 43–61. Springer (2016)
2. Bartocci, E., Kovács, L., Stankovic, M.: Automatic generation of moment-based invariants for prob-solvable loops. In: Proc. of ATVA 2019: the 17th International Symposium on Automated Technology for Verification and Analysis. LNCS, vol. 11781, pp. 255–276 (2019)
3. Chakarov, A., Sankaranarayanan, S.: Expectation Invariants for Probabilistic Program Loops as Fixed Points. In: SAS. LNCS, vol. 8723, pp. 85–100 (2014)
4. Gehr, T., Misailovic, S., Vechev, M.T.: PSI: Exact Symbolic Inference for Probabilistic Programs. In: CAV. LNCS, vol. 9779, pp. 62–83 (2016)
5. Ghahramani, Z.: Probabilistic Machine Learning and Artificial Intelligence. Nature **521**(7553), 452–459 (2015)
6. Humenberger, A., Jaroschek, M., Kovács, L.: Aligator.jl - A Julia Package for Loop Invariant Generation. In: CICM. LNCS, vol. 11006, pp. 111–117 (2018)
7. Katoen, J.P., McIver, A.K., Meinicke, L.A., Morgan, C.C.: Linear-Invariant Generation for Probabilistic Programs: Automated Support for Proof-Based Methods. In: SAS. LNCS, vol. 6337, pp. 390–406 (2010)
8. Kauers, M., Paule, P.: The Concrete Tetrahedron - Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates. Texts & Monographs in Symbolic Computation, Springer (2011)
9. Kura, S., Urabe, N., Hasuo, I.: Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments. In: TACAS. LNCS, vol. 11428, pp. 135–153 (2019)
10. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer (2005)

Author Index