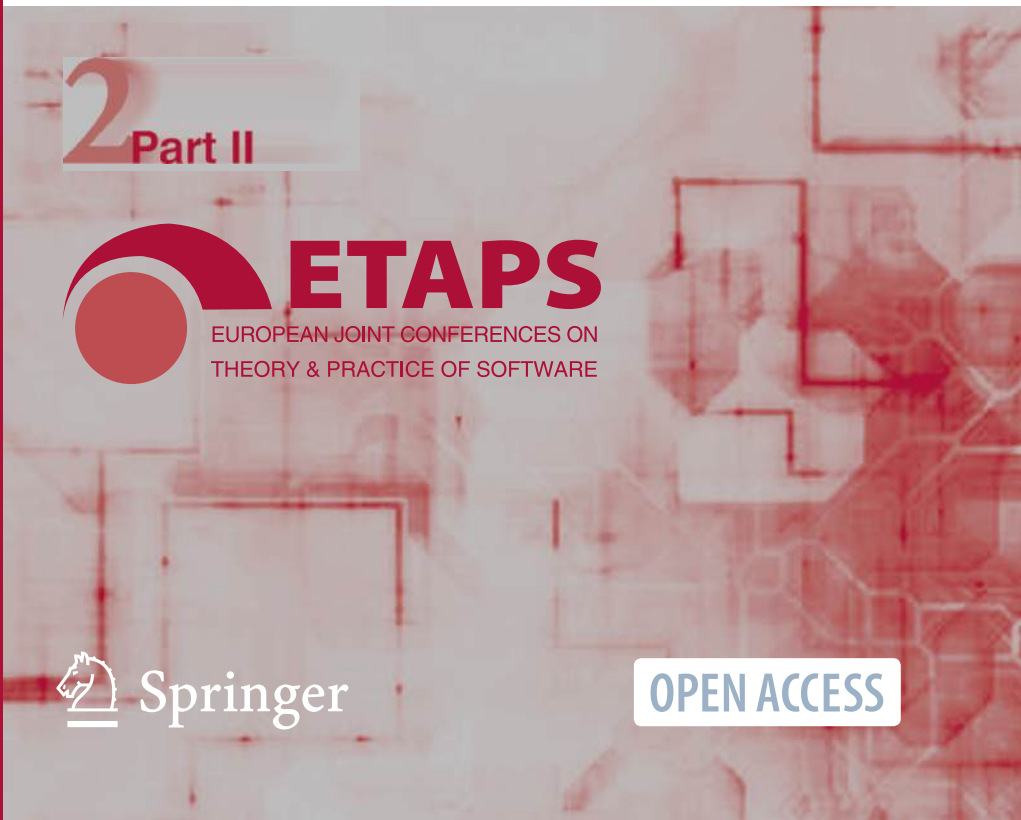
Dana Fisman
Grigore Rosu (Eds.)

LNCS 13244 ARCoSS

Tools and Algorithms for the Construction and Analysis of Systems

**28th International Conference, TACAS 2022
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2022
Munich, Germany, April 2–7, 2022
Proceedings, Part II**



More information about this series at https://link.springer.com/bookseries/558

Dana Fisman · Grigore Rosu (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

28th International Conference, TACAS 2022
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2022
Munich, Germany, April 2–7, 2022
Proceedings, Part II

Springer

Editors
Dana Fisman
Ben-Gurion University of the Negev
Be'er Sheva, Israel

Grigore Rosu
University of Illinois Urbana-Champaign
Urbana, IL, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-99526-3 ISBN 978-3-030-99527-0 (eBook)
https://doi.org/10.1007/978-3-030-99527-0

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

ETAPS Foreword

Welcome to the 25th ETAPS! ETAPS 2022 took place in Munich, the beautiful capital of Bavaria, in Germany.

ETAPS 2022 is the 25th 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 languages, 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 attract many researchers from all over the globe.

ETAPS 2022 received 362 submissions in total, 111 of which were accepted, yielding an overall acceptance rate of 30.7%. 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 2022 featured the unifying invited speakers Alexandra Silva (University College London, UK, and Cornell University, USA) and Tomáš Vojnar (Brno University of Technology, Czech Republic) and the conference-specific invited speakers Nathalie Bertrand (Inria Rennes, France) for FoSSaCS and Lenore Zuck (University of Illinois at Chicago, USA) for TACAS. Invited tutorials were provided by Stacey Jeffery (CWI and QuSoft, The Netherlands) on quantum computing and Nicholas Lane (University of Cambridge and Samsung AI Lab, UK) on federated learning.

As this event was the 25th edition of ETAPS, part of the program was a special celebration where we looked back on the achievements of ETAPS and its constituting conferences in the past, but we also looked into the future, and discussed the challenges ahead for research in software science. This edition also reinstated the ETAPS mentoring workshop for PhD students.

ETAPS 2022 took place in Munich, Germany, and was organized jointly by the Technical University of Munich (TUM) and the LMU Munich. The former was founded in 1868, and the latter in 1472 as the 6th oldest German university still running today. Together, they have 100,000 enrolled students, regularly rank among the top 100 universities worldwide (with TUM's computer-science department ranked #1 in the European Union), and their researchers and alumni include 60 Nobel laureates.

The local organization team consisted of Jan Křetínský (general chair), Dirk Beyer (general, financial, and workshop chair), Julia Eisentraut (organization chair), and Alexandros Evangelidis (local proceedings chair).

ETAPS 2022 was 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 ETAPS Steering Committee 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 (Twente, chair), Jan Kofroň (Prague), Barbara König (Duisburg), Thomas Noll (Aachen), Caterina Urban (Paris), Tarmo Uustalu (Reykjavik and Tallinn), and Lenore Zuck (Chicago).

Other members of the Steering Committee are Patricia Bouyer (Paris), Einar Broch Johnsen (Oslo), Dana Fisman (Be'er Sheva), Reiko Heckel (Leicester), Joost-Pieter Katoen (Aachen and Twente), Fabrice Kordon (Paris), Jan Křetínský (Munich), Orna Kupferman (Jerusalem), Leen Lambers (Cottbus), Tiziana Margaria (Limerick), Andrew M. Pitts (Cambridge), Elizabeth Polgreen (Edinburgh), Grigore Roşu (Illinois), Peter Ryan (Luxembourg), Sriram Sankaranarayanan (Boulder), Don Sannella (Edinburgh), Lutz Schröder (Erlangen), Ilya Sergey (Singapore), Natasha Sharygina (Lugano), Pawel Sobocinski (Tallinn), Peter Thiemann (Freiburg), Sebastián Uchitel (London and Buenos Aires), Jan Vitek (Prague), Andrzej Wasowski (Copenhagen), Thomas Wies (New York), Anton Wijs (Eindhoven), and Manuel Wimmer (Linz).

I'd like to take this opportunity to thank all authors, attendees, organizers of the satellite workshops, and Springer-Verlag GmbH for their support. I hope you all enjoyed ETAPS 2022.

Finally, a big thanks to Jan, Julia, Dirk, and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

February 2022

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

TACAS 2022 was the 28th edition of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022 was part of the 25th European Joint Conferences on Theory and Practice of Software (ETAPS 2022), which was held from April 2 to April 7 in Munich, Germany, as well as online due to the COVID-19 pandemic. TACAS is a forum for researchers, developers, and users interested in rigorous 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 computer-controlled systems.

There were four submission categories for TACAS 2022:

1. Research papers advancing the theoretical foundations for the construction and analysis of systems.
2. Case study papers with an emphasis on a real-world setting.
3. Regular tool papers presenting a new tool, a new tool component, or novel extensions to an existing tool.
4. Tool demonstration papers focusing on the usage aspects of tools.

Papers of categories 1–3 were restricted to 16 pages, and papers of category 4 to six pages.

This year 159 papers were submitted to TACAS, consisting of 112 research papers, five case study papers, 33 regular tool papers, and nine tool demo papers. Authors were allowed to submit up to four papers. Each paper was reviewed by three Program Committee (PC) members, who made use of subreviewers. Similarly to previous years, it was possible to submit an artifact alongside a paper, which was mandatory for regular tool and tool demo papers.

An artifact might consist of a tool, models, proofs, or other data required for validation of the results of the paper. The Artifact Evaluation Committee (AEC) was tasked with reviewing the artifacts based on their documentation, ease of use, and, most importantly, whether the results presented in the corresponding paper could be accurately reproduced. Most of the evaluation was carried out using a standardized virtual machine to ensure consistency of the results, except for those artifacts that had special hardware or software requirements. The evaluation consisted of two rounds. The first round was carried out in parallel with the work of the PC. The judgment of the AEC was communicated to the PC and weighed in their discussion. The second round took place after paper acceptance notifications were sent out; authors of accepted research papers who did not submit an artifact in the first round could submit their artifact at this time. In total, 86 artifacts were submitted (79 in the first round and seven in the second) and evaluated by the AEC regarding their availability, functionality, and/or reusability. Papers with an artifact that was successfully evaluated include one or more badges on the first page, certifying the respective properties.

Selected authors were requested to provide a rebuttal for both papers and artifacts in case a review gave rise to questions. Using the review reports and rebuttals, the Program and the Artifact Evaluation Committees extensively discussed the papers and artifacts and ultimately decided to accept 33 research papers, one case study, 12 tool papers, and four tool demos.

This corresponds to an acceptance rate of 29.46% for research papers and an overall acceptance rate of 31.44%.

Besides the regular conference papers, this two-volume proceedings also contains 16 short papers that describe the participating verification systems and a competition report presenting the results of the 11th SV-COMP, the competition on automatic software verifiers for C and Java programs. These papers were reviewed by a separate Program Committee (PC); each of the papers was assessed by at least three reviewers. A total of 47 verification systems with developers from 11 countries entered the systematic comparative evaluation, including four submissions from industry. Two sessions in the TACAS program were reserved for the presentation of the results: (1) a summary by the competition chair and of the participating tools by the developer teams in the first session, and (2) an open community meeting in the second session.

We would like to thank all the people who helped to make TACAS 2022 successful. First, we would like to thank the authors for submitting their papers to TACAS 2022. The PC members and additional reviewers did a great job in reviewing papers: they contributed informed and detailed reports and engaged in the PC discussions. We also thank the steering committee, and especially its chair, Joost-Pieter Katoen, for his valuable advice. Lastly, we would like to thank the overall organization team of ETAPS 2022.

April 2022

Dana Fisman
Grigore Rosu
PC Chairs

Swen Jacobs
Andrew Reynolds
AEC Chairs, Tools, and Case-study Chairs

Dirk Beyer
Competition Chair

Organization

Program Committee

Parosh Aziz Abdulla	Uppsala University, Sweden
Luca Aceto	Reykjavik University, Iceland
Timos Antonopoulos	Yale University, USA
Saddek Bensalem	Verimag, France
Dirk Beyer	LMU Munich, Germany
Nikolaj Bjorner	Microsoft, USA
Jasmin Blanchette	Vrije Universiteit Amsterdam, The Netherlands
Udi Boker	Interdisciplinary Center Herzliya, Israel
Hana Chockler	King's College London, UK
Rance Cleaveland	University of Maryland, USA
Alessandro Coglio	Kestrel Institute, USA
Pedro R. D'Argenio	Universidad Nacional de Córdoba, Argentina
Javier Esparza	Technical University of Munich, Germany
Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Dana Fisman (Chair)	Ben-Gurion University, Israel
Martin Fränzle	University of Oldenburg, Germany
Felipe Gorostiaga	IMDEA Software Institute, Spain
Susanne Graf	Université Joseph Fourier, France
Radu Grosu	Stony Brook University, USA
Arie Gurfinkel	University of Waterloo, Canada
Klaus Havelund	Jet Propulsion Laboratory, USA
Holger Hermanns	Saarland University, Germany
Falk Howar	TU Clausthal / IPSSE, Germany
Swen Jacobs	CISPA Helmholtz Center for Information Security, Germany
Ranjit Jhala	University of California, San Diego, USA
Jan Kretinsky	Technical University of Munich, Germany
Viktor Kuncak	Ecole Polytechnique Fédérale de Lausanne, Switzerland
Kim Larsen	Aalborg University, Denmark
Konstantinos Mamouras	Rice University, USA
Daniel Neider	Max Planck Institute for Software Systems, Germany
Dejan Nickovic	AIT Austrian Institute of Technology, Austria
Corina Pasareanu	Carnegie Mellon University, NASA, and KBR, USA
Doron Peled	Bar Ilan University, Israel
Anna Philippou	University of Cyprus, Cyprus
Andrew Reynolds	University of Iowa, USA

Grigore Rosu (Chair)	University of Illinois at Urbana-Champaign, USA
Kristin Yvonne Rozier	Iowa State University, USA
Cesar Sanchez	IMDEA Software Institute, Spain
Sven Schewe	University of Liverpool, UK
Natasha Sharygina	Università della Svizzera italiana, Italy
Jan Strejček	Masaryk University, Czech Republic
Cesare Tinelli	University of Iowa, USA
Stavros Tripakis	Northeastern University, USA
Frits Vaandrager	Radboud University, The Netherlands
Tomas Vojnar	Brno University of Technology, Czech Republic
Christoph M. Wintersteiger	Microsoft, USA
Lijun Zhang	Institute of Software, Chinese Academy of Sciences, China
Lingming Zhang	University of Illinois at Urbana-Champaign, USA
Lenore Zuck	University of Illinois at Chicago, USA

Artifact Evaluation Committee

Pavel Andrianov	Ivannikov Institute for System Programming of the RAS, Russia
Michael Backenköhler	Saarland University, Germany
Sebastian Biewer	Saarland University, Germany
Benjamin Bisping	TU Berlin, Germany
Olav Bunte	Eindhoven University of Technology, The Netherlands
Damien Busatto-Gaston	Université Libre de Bruxelles, Belgium
Marek Chalupa	IST Austria, Austria, and Masaryk University, Czech Republic
Priyanka Darke	Tata Consultancy Services, India
Alexandre Duret-Lutz	LRDE, France
Shenghua Feng	Institute of Software, Chinese Academy of Sciences, Beijing, China
Mathias Fleury	University of Freiburg, Germany
Kush Grover	Technical University of Munich, Germany
Dominik Harmim	Brno University of Technology, Czech Republic
Swen Jacobs (Chair)	CISPA Helmholtz Center for Information Security, Germany
Xiangyu Jin	Institute of Software, Chinese Academy of Sciences
Juraj Sič	Masaryk University, Czech Republic
Daniela Kaufmann	Johannes Kepler University Linz, Austria
Maximilian Alexander Köhl	Saarland University, Germany
Mitja Kulczynski	Kiel University, Germany
Maurice Laveaux	Eindhoven University of Technology, The Netherlands
Yong Li	Institute of Software, Chinese Academy of Sciences, China
Debasmita Lohar	Max Planck Institute for Software Systems, Germany
Makai Mann	Stanford University, USA

Fabian Meyer	RWTH Aachen University, Germany
Stefanie Mohr	Technical University of Munich, Germany
Malte Mues	TU Dortmund, Germany
Yuki Nishida	Kyoto University, Japan
Philip Offtermatt	Université de Sherbrooke, Canada
Muhammad Osama	Eindhoven University of Technology, The Netherlands
Jiří Pavela	Brno University of Technology, Czech Republic
Adrien Pommellet	LRDE, France
Mathias Preiner	Stanford University, USA
José Proença	CISTER-ISEP and HASLab-INESC TEC, Portugal
Tim Quatmann	RWTH Aachen University, Germany
Etienne Renault	LRDE, France
Andrew Reynolds (Chair)	University of Iowa, USA
Mouhammad Sakr	University of Luxembourg, Luxembourg
Morten Konggaard Schou	Aalborg University, Denmark
Philipp Schlehuber-Caissier	LRDE, France
Hans-Jörg Schurr	Inria Nancy - Grand Est, France
Michael Schwarz	Technische Universität München, Germany
Joseph Scott	University of Waterloo, Canada
Ali Shamakhi	Tehran Institute for Advanced Studies, Iran
Lei Shi	University of Pennsylvania, USA
Matthew Sotoudeh	University of California, Davis, USA
Jip Spel	RWTH Aachen University, Germany
Veronika Šoková	Brno University of Technology, Czech Republic

Program Committee and Jury — SV-COMP

Fatimah Aljaafari	University of Manchester, UK
Lei Bu	Nanjing University, China
Thomas Bunk	LMU Munich, Germany
Marek Chalupa	Masaryk University, Czech Republic
Priyanka Darke	Tata Consultancy Services, India
Daniel Dietsch	University of Freiburg, Germany
Gidon Ernst	LMU Munich, Germany
Fei He	Tsinghua University, China
Matthias Heizmann	University of Freiburg, Germany
Jera Hensel	RWTH Aachen University, Germany
Falk Howar	TU Dortmund, Germany
Soha Hussein	University of Minnesota, USA
Dominik Klumpp	University of Freiburg, Germany
Henrich Lauko	Masaryk University, Czech Republic
Will Leeson	University of Virginia, USA
Xie Li	Chinese Academy of Sciences, China
Viktor Malík	Brno University of Technology, Czech Republic
Raveendra Kumar Medicherla	Tata Consultancy Services, India

Rafael Sá Menezes	University of Manchester, UK
Vince Molnár	Budapest University of Technology and Economics, Hungary
Hernán Ponce de León	Bundeswehr University Munich, Germany
Cedric Richter	University of Oldenburg, Germany
Simmo Saan	University of Tartu, Estonia
Emerson Sales	Gran Sasso Science Institute, Italy
Peter Schrammel	University of Sussex and Diffblue, UK
Frank Schüssele	University of Freiburg, Germany
Ryan Scott	Galois, USA
Ali Shamakhi	Tehran Institute for Advanced Studies, Iran
Martin Spiessl	LMU Munich, Germany
Michael Tautschnig	Queen Mary University of London, UK
Anton Vasilyev	ISP RAS, Russia
Vesal Vojdani	University of Tartu, Estonia

Steering Committee

Dirk Beyer	Ludwig-Maximilians-Universität München, Germany
Rance Cleaveland	University of Maryland, USA
Holger Hermanns	Universität des Saarlandes, Germany
Joost-Pieter Katoen (Chair)	RWTH Aachen University, Germany, and Universiteit Twente, The Netherlands
Kim G. Larsen	Aalborg University, Denmark
Bernhard Steffen	Technische Universität Dortmund, Germany

Additional Reviewers

Abraham, Erika
Aguilar, Edgar
Akshay, S.
Asadi, Sepideh
Attard, Duncan
Avni, Guy
Azeem, Muqsit
Bacci, Giorgio
Balasubramanian, A. R.
Barbanera, Franco
Bard, Joachim
Basset, Nicolas
Bendík, Jaroslav
Berani Abdelwahab, Erzana
Beutner, Raven
Bhandary, Shrajan
Biewer, Sebastian
Blicha, Martin
Brandstätter, Andreas
Bright, Curtis
Britikov, Konstantin
Brunnbauer, Axel
Capretto, Margarita
Castiglioni, Valentina
Castro, Pablo
Ceska, Milan
Chadha, Rohit
Chalupa, Marek
Changshun, Wu
Chen, Xiaohong
Cruciani, Emilio
Dahmen, Sander
Dang, Thao
Danielsson, Luis Miguel

Degiovanni, Renzo
Dell'Erba, Daniele
Demasi, Ramiro
Desharnais, Martin
Dierl, Simon
Dubslaff, Clemens
Egolf, Derek
Evangelidis, Alexandros
Fedyukovich, Grigory
Fiedor, Jan
Fitzpatrick, Stephen
Fleury, Mathias
Frenkel, Hadar
Gamboa Guzman, Laura P.
Garcia-Contreras, Isabel
Gianola, Alessandro
Goorden, Martijn
Gorostiaga, Felipe
Gorrieri, Roberto
Grahn, Samuel
Grastien, Alban
Grover, Kush
Grünbacher, Sophie
Guha, Shibashis
Gutiérrez Brida, Simón Emmanuel
Havlena, Vojtěch
He, Jie
Helfrich, Martin
Henkel, Elisabeth
Hicks, Michael
Hirschkoff, Daniel
Hofmann, Jana
Hojjat, Hossein
Holík, Lukáš
Hospodár, Michal
Huang, Chao
Hyvärinen, Antti
Inverso, Omar
Itzhaky, Shachar
Jaksic, Stefan
Jansen, David N.
Jin, Xiangyu
Jonas, Martin
Kanav, Sudeep
Karra, Shyam Lal
Katsaros, Panagiotis
Kempa, Brian
Klauck, Michaela
Kreitz, Christoph
Kröger, Paul
Köhl, Maximilian Alexander
König, Barbara
Lahijanian, Morteza
Larraz, Daniel
Le, Nham
Lemberger, Thomas
Lengal, Ondrej
Li, Chunxiao
Li, Jianlin
Lorber, Florian
Lung, David
Luppen, Zachary
Lybech, Stian
Major, Juraj
Manganini, Giorgio
McCarthy, Eric
Mediouni, Braham Lotfi
Meggendorfer, Tobias
Meira-Goes, Romulo
Melcer, Daniel
Metzger, Niklas
Milovancevic, Dragana
Mohr, Stefanie
Najib, Muhammad
Noetzli, Andres
Nouri, Ayoub
Offtermatt, Philip
Otoni, Rodrigo
Paoletti, Nicola
Parizek, Pavel
Parker, Dave
Parys, Paweł
Passing, Noemi
Perez Dominguez, Ivan
Perez, Guillermo
Pinna, G. Michele
Pous, Damien
Priya, Siddharth
Putruele, Luciano
Pérez, Jorge A.
Qu, Meixun
Raskin, Mikhail

Rauh, Andreas
Reger, Giles
Reynouard, Raphaël
Riener, Heinz
Rogalewicz, Adam
Roy, Rajarshi
Ruemmer, Philipp
Ruijters, Enno
Schilling, Christian
Schmitt, Frederik
Schneider, Tibor
Scholl, Christoph
Schultz, William
Schupp, Stefan
Schurr, Hans-Jörg
Schwammberger, Maike
Shafiei, Nastaran
Siber, Julian
Sickert, Salomon
Singh, Gagandeep
Smith, Douglas
Somenzi, Fabio
Stewing, Richard
Stock, Gregory
Su, Yusen
Tang, Qiyi
Tibo, Alessandro
Trefler, Richard
Trtík, Marek
Turrini, Andrea
Vaezipoor, Pashootan
van Dijk, Tom
Vašíček, Ondřej
Vediramana Krishnan, Hari Govind
Wang, Wenxi
Wendler, Philipp
Westfold, Stephen
Winter, Stefan
Wolovick, Nicolás
Yakusheva, Sophia
Yang, Pengfei
Zeljić, Aleksandar
Zhou, Yuhao
Zimmermann, Martin

Contents – Part II

Contents – Part I

Probabilistic Systems

A Probabilistic Logic for Verifying Continuous-time Markov Chains

Ji Guan[1] and Nengkun Yu[2](✉)

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China.
guanji1992@gmail.com
[2] Centre for Quantum Software and Information, University of Technology Sydney, Sydney, Australia.
nengkunyu@gmail.com

Abstract. A continuous-time Markov chain (CTMC) execution is a continuous class of probability distributions over states. This paper proposes a probabilistic linear-time temporal logic, namely continuous-time linear logic (CLL), to reason about the probability distribution execution of CTMCs. We define the syntax of CLL on the space of probability distributions. The syntax of CLL includes multiphase timed until formulas, and the semantics of CLL allows time reset to study relatively temporal properties. We derive a corresponding model-checking algorithm for CLL formulas. The correctness of the model-checking algorithm depends on Schanuel's conjecture, a central open problem in transcendental number theory. Furthermore, we provide a running example of CTMCs to illustrate our method.

1 Introduction

As a popular model of probabilistic continuous-time systems, *continuous-time Markov chains (CTMCs)* have been extensively studied since Kolmogorov [25]. In the recent 20 years, probabilistic continuous-time model checking receives much attention. Adopting *probabilistic computational tree logic (PCTL)* [22] to this context with extra *multiphase timed until formulas* $\Phi_1 U^{\mathcal{T}_1} \Phi_2 \cdots U^{\mathcal{T}_K} \Phi_{K+1}$, for state formula Φ and time interval $\mathcal{T}$, Aziz et al. proposed *continuous stochastic logic (CSL)* to specify the branching-time properties of CTMCs and the model-checking problem for CSL is decidable [8]. After that, efficient model-checking algorithms were developed by transient analysis of CTMCs using uniformization [9] and stratification [41] for a restricted version (path formulas are restricted to single until formulas $\Phi_1 U^{\mathcal{I}} \Phi_2$) and a full version of CSL, respectively. These algorithms have been practically implemented in model checkers PRISM [26], MRMC [24] and STORM [18]. Further details can be found in an excellent survey [23].

There are also different ways to specify the linear-time properties of CTMCs. Timed automata were first used to achieve this task [11,13,14,15,19], and then

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 3–21, 2022.
https://doi.org/10.1007/978-3-030-99527-0_1

metric temporal logic (MTL) [12] was also considered in this context. Subsequently, the probability of "the system being in *state* s_0 within five-time units after having continuously remained in *state* s_1" can be computed. However, some statements cannot be specified and verified because of the lack of a probabilistic linear-time temporal logic, for instance "the system being in *state* s_0 with high probability (≥ 0.9) within five-time units after having continuously remained in *state* s_1 with low probability (≤ 0.1)". Furthermore, this probabilistic property cannot be expressed by CSL because CSL cannot express properties that are defined across several state transitions of the same time length in the execution of a CTMC.

In this paper, targeting to express the mentioned probabilistic linear-time properties, we introduce *continuous-time linear logic (CLL)*. In particular, we adopt the viewpoint used in [2] by regarding CTMCs as transformers of probability distributions over states. CLL studies the properties of the probability distribution execution generated by a given initial probability distribution over time. By the fundamental difference between the views of state executions and probability distribution executions of CTMCs, CLL and CSL are incomparable and complementary, as the relation between *probabilistic linear-time temporal logic (PLTL)* and PCTL in model checking discrete-time Markov chains [2, Section 3.3].

The atomic propositions of CLL are explained on the space of probability distributions over states of CTMCs. We apply the method of symbolic dynamics to the probability distributions of CTMCs. To be specific, we symbolize the probability value space $[0, 1]$ into a finite set of intervals $\mathscr{I} = \{\mathcal{I}_k \subseteq [0, 1]\}_{k=1}^{m}$. A probability distribution μ over its set of states $S = \{s_0, s_2, \ldots, s_{d-1}\}$ is then represented symbolically as a set of symbols

$$\mathbb{S}(\mu) = \{\langle s, \mathcal{I}\rangle \in S \times \mathscr{I} : \mu(s) \in \mathcal{I}\}$$

where each symbol $\langle s, \mathcal{I}\rangle$ asserts $\mu(s) \in \mathcal{I}$, i.e., the probability of state s in distribution μ falls in interval $\mathcal{I}$. For example, $\langle s_0, [0.9, 1]\rangle$ means the system is in *state* s_0 with a probability in 0.9 to 1. The symbolization idea of distributions has been considered in [2]: choosing a disjoint cover of $[0, 1]$:

$$\mathcal{I} = \{[0, p_1), [p_1, p_2), ..., [p_n, 1]\}.$$

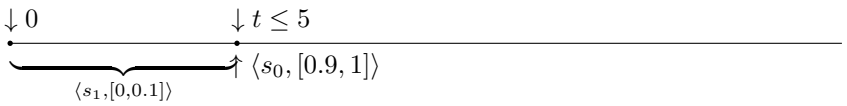
Here, we remove this restriction and enrich the expressiveness of $\mathscr{I}$. A crucial fact about this symbolization is that the set $S \times \mathscr{I}$ is finite. Consequently, the (probability distribution) execution path generated by an initial probability distribution μ induces a sequence of symbols in $S \times \mathscr{I}$ over time. Therefore, the dynamics of CTMCs can be studied in terms of a (real-time) language over the alphabet $S \times \mathscr{I}$, which is the set of atomic propositions of CLL.

Different from non-probabilistic linear-time temporal logics — *linear-time temporal logic (LTL)* and MTL, CLL has two types of formulas: state formulas and path formulas. The state formulas are constructed using propositional connectives. The path formulas are obtained by propositional connectives and a temporal modal operator *timed until* $U^{\mathcal{T}}$ for a bounded time interval $\mathcal{T}$, as in

MTL and CSL. The standard next-step temporal operator in LTL is meaningless in continuous-time systems since the time domain (real numbers) is uncountable. As a result, CLL can express the above mentioned probabilistic property "the system is at *state* s_0 with high probability (≥ 0.9) within 5 time units after having continuously remained at *state* s_1 with low probability (≤ 0.1)" in a path formula:

$$\varphi = \langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle.$$

In this single until formula, there is a time instant $0 \leq t \leq 5$ at which *state* s_1 with low probability transits to *state* s_0 with high probability. Then we illustrate this on the following timeline.

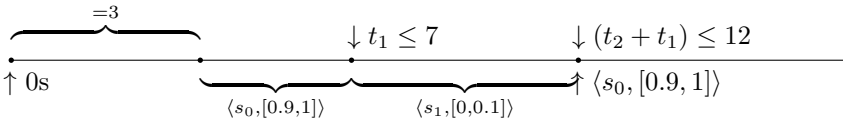


Furthermore, CLL allows *multiphase timed until formulas*. The semantics of the formulas focuses on relative time intervals, i.e., time can be reset as in timed automata [5,6], while those of CSL [8] are for absolute time intervals. Subsequently, CLL can express not only *relatively* but also *absolutely* temporal properties of CTMCs.

We illustrate the significant difference between *relatively* temporal properties and *absolutely* temporal properties of CTMCs. For instance, "before probability distributions transition φ happening in 3 to 7 time units, the system always stays at *state* s_0 with a high probability (≥ 0.9)" can be formalized in path formulae

$$\varphi' = \langle s_0, [0.9, 1]\rangle U^{[3,7]} (\langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle).$$

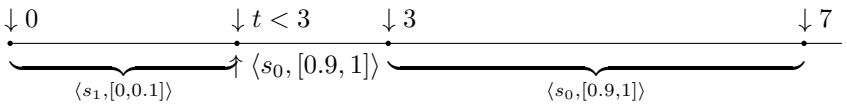
As we can see, there are two time instants, namely t_1 and t_2, happening distribution transitions. Time is reset to 0 after the first distribution transition happens and thus t_2 is relative to t_1. More clearly, we depict this on the following timeline.



An absolute version is "probability distribution transition φ happens and the system always stays at *state* s_0 with a high probability (≥ 0.9) in 3 to 7 time units"

$$\varphi'' = \Box^{[3,7]} \langle s_0, [0.9, 1]\rangle \wedge \langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle).$$

We can get a clear timeline representation by simply adding $\Box^{[3,7]} \langle s_0, [0.9, 1]\rangle$ to that of φ. Assume that $t < 3$,



Time reset enriches the expressiveness of CLL but introduces more difficulties to model checking CLL than CSL. We cross this by translating relative time to the absolute one. As a result, we develop an algorithm to model check CTMCs against CLL formulas. More precisely, we reduce the model-checking problem to a reachability problem of absolute time intervals. The reachability problem corresponds to the real root isolation problem of *real polynomial-exponential functions (PEFs)* over the field of algebraic numbers, an extensively studied question in recent symbolic and algebraic computation community (e.g. [1,20,28]). By developing a state-of-the-art real root isolation algorithm, we resolve the latter problem under the assumption of the validity of Schanuel's conjecture, a central open question in transcendental number theory [27]. This conjecture has also been the footstone of the correctness of many recent model-checking algorithms, including the decidability of continuous-time Markov decision processes [30], the synthesizing inductive invariants for continuous linear dynamical systems [4], the termination analysis for probabilistic programs with delays [39], and reachability analysis for dynamical systems [20].

In summary, the main contributions of this paper are as follows.

- *Introducing a probabilistic logic*, namely continuous-time linear logic (CLL), for reasoning about CTMCs;
- *Developing* a state-of-the-art real root isolation algorithm for PEFs over the field of algebraic numbers for checking atomic propositions of CLL;
- *Proving* that model checking CTMCs against CLL formulas is decidable subject to Schanuel's conjecture.

Organization of this paper. In the next section, we give the mathematical preliminaries used in this paper. In Section 3, we recall the view of CTMCs as distribution transformers. After that, the symbolic dynamics of CTMCs are introduced by symbolizing distributions over states of CTMCs in Section 4. In the subsequent section, we present our continuous-time probabilistic temporal logic CLL. In Section 6, we develop an algorithm to solve the CLL model checking problem. A case study and related works are shown in Sections 7 and 8, respectively. We summarize our results and point out future research directions in the final section.

2 Preliminaries

For the convenience of the readers, we review basic definitions and notations of number theory, particularly Schanuel's conjecture.

Throughout this paper, we write $\mathbb{C}, \mathbb{R}, \mathbb{Q}$ and $\mathbb{A}$ for the fields of all complex, real, rational and algebraic numbers, respectively. In addition, $\mathbb{Z}$ denotes the set of all integer numbers. For $\mathbb{F} \in \{\mathbb{C}, \mathbb{R}, \mathbb{Q}, \mathbb{Z}, \mathbb{A}\}$, we use $\mathbb{F}[t]$ and $\mathbb{F}^{n\times m}$ to denote the set of polynomials in t with coefficients in $\mathbb{F}$ and n-by-m matrices with every entry in $\mathbb{F}$, respectively. Furthermore, for $\mathbb{F} \in \{\mathbb{R}, \mathbb{Q}, \mathbb{Z}\}$, we use $\mathbb{F}^+$ to denote the set of positive elements (including 0) of $\mathbb{F}$.

A bounded (time) *interval* $\mathcal{T}$ is a subset of $\mathbb{R}^+$, which may be open, half-open or closed with one of the following forms:

$$[t_1, t_2], [t_1, t_2), (t_1, t_2], (t_1, t_2),$$

where $t_1, t_2 \in \mathbb{R}^+$ and $t_2 \geq t_1$ ($t_1 = t_2$ is only allowed in the case of $[t_1, t_2]$). Here, t_1 and t_2 are called the *left* and *right endpoints* of $\mathcal{T}$, respectively. Conveniently, we use $\inf \mathcal{T}$ and $\sup \mathcal{T}$ to denote t_1 and t_2, respectively. In this paper, we only consider **bounded intervals**.

For reasoning about the temporal properties, we further define the *addition* and *subtraction* of (time) intervals. The expression $\mathcal{T} + t$ or $t + \mathcal{T}$, for $t \in \mathbb{R}^+$, denotes the interval $\{t + t' : t' \in \mathcal{T}\}$. Similarly, $\mathcal{T} - t$ stands for the interval $\{-t + t' : t' \in \mathcal{T}\}$ if $t \leq \inf \mathcal{T}$. Furthermore, for two intervals $\mathcal{T}_1$ and $\mathcal{T}_2$,

$$\mathcal{T}_1 + \mathcal{T}_2 = \{t \in (t' + \mathcal{T}_2) : t' \in \mathcal{T}_1\} = \{t_1 + t_2 : t_1 \in \mathcal{T}_1 \text{ and } t_2 \in \mathcal{T}_2\}.$$

Two intervals $\mathcal{T}_1$ and $\mathcal{T}_2$ are *disjoint* if their intersection is an empty set, i.e., $\mathcal{T}_1 \cap \mathcal{T}_2 = \emptyset$. Let us see some concrete examples: $1 + (2, 3) = (3, 4)$, $(2, 3) - 1 = (1, 2)$, $(2, 3) + [3, 4] = (5, 7)$ and $(2, 3), [3, 4]$ are disjoint. It is obvious that all calculations of time intervals in the above are easy to be computed.

An *algebraic number* is a complex number that is a root of a non-zero polynomial in one variable with rational coefficients (or equivalent to integer coefficients, by eliminating denominators). An algebraic number α is represented by $(P, (a, b), \varepsilon)$ where P is the minimal polynomial of α, $a, b \in \mathbb{Q}$ and $a + bi$ is an approximation of α such that $|\alpha - (a + bi)| < \varepsilon$ and α is the only root of P in the open ball $B(a + bi, \varepsilon)$. The minimal polynomial of α is the polynomial with the smallest degree in $\mathbb{Q}[t]$ such that α is a root of the polynomial and the coefficient of the highest-degree term is 1. Any root of $f(t) \in \mathbb{A}[t]$ is algebraic. Moreover, given the representations of $a, b \in \mathbb{A}$, the representations of $a \pm b$, $\frac{a}{b}$ and $a \cdot b$ can be computed in polynomial time, so does the equality checking [17].

Furthermore, a complex number is called *transcendental* if it is not an algebraic number. In general, it is challenging to verify relationships between transcendental numbers [33]. On the other hand, one can use the Lindemann-Weierstrass theorem to compare some transcendental numbers. The transcendence of e and π are direct corollaries of this theorem.

Theorem 1 (Lindemann-Weierstrass theorem). *Let $\eta_1, \cdots, \eta_n$ be pairwise distinct algebraic complex numbers. Then $\sum_k \lambda_k e^{\eta_k} \neq 0$ for non-zero algebraic numbers $\lambda_1, \cdots, \lambda_n$.*

The following concepts are introduced to study the general relation between transcendental numbers.

Definition 1 (Algebraic independence). *A set of complex numbers $S = \{a_1, \cdots, a_n\}$ is* algebraically independent *over $\mathbb{Q}$ if the elements of S do not satisfy any nontrivial (non-constant) polynomial equation with coefficients in $\mathbb{Q}$.*

By the above definition, for any transcendental number u, $\{u\}$ is algebraically independent over $\mathbb{Q}$, while $\{a\}$ for any algebraic number $a \in \mathbb{A}$ is not. Thus, a

set of complex numbers that is algebraically independent over $\mathbb{Q}$ must consist of transcendental numbers. $\{\pi, e^{\pi\sqrt{n}}\}$ is also algebraically independent over $\mathbb{Q}$ for any positive integer n [31]. Checking the algebraic independence is challenging. For example, it is still widely open whether $\{e, \pi\}$ is algebraically independent over $\mathbb{Q}$.

Definition 2 (Extension field). *Given two fields $E \subseteq F$, F is an extension field of E, denoted by F/E, if the operations of E are those of F restricted to E.*

For example, under the usual notions of addition and multiplication, the field of complex numbers is an extension field of real numbers.

Definition 3 (Transcendence degree). *Let L be an extension field of $\mathbb{Q}$, the transcendence degree of L over $\mathbb{Q}$ is defined as the largest cardinality of an algebraically independent subset of L over $\mathbb{Q}$.*

For instance, let $\mathbb{Q}(e)/\mathbb{Q} = \{a + be \mid a, b \in \mathbb{Q}\}$ and $\mathbb{Q}(\sqrt{2})/\mathbb{Q} = \{a + b\sqrt{2} \mid a, b \in \mathbb{Q}\}$ be two extension fields of $\mathbb{Q}$. Then the transcendence degree of them are 1 and 0, respectively, by noting that e is a transcendental number and $\sqrt{2}$ is an algebraic number.

Now, Schanuel's conjecture is ready to be presented.

Conjecture 1 (Schanuel's conjecture). Given any complex numbers $z_1, \cdots, z_n$ that are linearly independent over $\mathbb{Q}$, the extension field $\mathbb{Q}(z_1, ..., z_n, e^{z_1}, ..., e^{z_n})$ has transcendence degree of at least n over $\mathbb{Q}$.

Stephen Schanuel proposed this conjecture during a course given by Serge Lang at Columbia in the 1960s [27]. Schanuel's conjecture concerns the transcendence degree of certain field extensions of the rational numbers. The conjecture, if proven, would generalize the most well-known results in transcendental number theory significantly [29,37]. For example, the algebraical independence of $\{e, \pi\}$ would simply follow by setting $z_1 = 1$ and $z_2 = \pi i$, and using *Euler's identity* $e^{\pi i} + 1 = 0$.

3 Continuous-time Markov Chains as Distributions Transformers

We begin with the definition of *continuous-time Markov chains (CTMCs)*. A CTMC is a Markovian (memoryless) *stochastic process* that takes values on a finite state set S ($|S| = d < \infty$) and evolves in continuous-time $t \in \mathbb{R}^+$. Formally,

Definition 4. *A CTMC is a pair $\mathcal{M} = (S, Q)$, where S ($|S| = d$) is a finite state set and $Q \in \mathbb{Q}^{d \times d}$ is a* transition rate matrix.

A transition rate matrix Q is a matrix whose off-diagonal entries $\{Q_{i,j}\}_{i \neq j}$ are nonnegative rational numbers, representing the transition rate from state i to

state j, while the diagonal entries $\{Q_{j,j}\}$ are constrained to be $-\sum_{j\neq i} Q_{i,j}$ for all $1 \leq j \leq d$. Consequently, the column summations of Q are all zero.

The evolution of a CTMC can be regarded as a *distribution transformer*. Given initial distribution $\mu \in \mathbb{Q}^{d\times 1} \in \mathcal{D}(\mathcal{S})$, the distribution at time $t \in \mathbb{R}^+$ is:

$$\mu_t = e^{Qt}\mu,$$

where $\mathcal{D}(\mathcal{S})$ is denoted as the set of all probability distributions over S. We call $\mathcal{D}(\mathcal{S})$ the *probability distribution space* of CTMCs. An *execution path* of CTMCs is a continuous function indexed by initial distribution $\mu \in \mathcal{D}(\mathcal{S})$:

$$\sigma_\mu \colon \mathbb{R}^+ \to \mathcal{D}(\mathcal{S}), \qquad \sigma_\mu(t) = e^{Qt}\mu. \tag{1}$$

Example 1. We recall the illustrating example of CTMC $\mathcal{M} = (S, Q)$ in [8, Figure 1] as the running example in our work. In particular, $\mathcal{M}$ is a 5-dimensional CTMC with initial distribution μ, where $S = \{s_0, s_1, s_2, s_3, s_4\}$ and

$$Q = \begin{pmatrix} -3 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & -7 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \end{pmatrix} \qquad \mu = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \\ 0 \end{pmatrix}.$$

4 Symbolic Dynamics of CTMCs

In this section, we introduce symbolic dynamics to characterize the properties of the probability distribution space of CTMCs.

First, we fix a finite set of intervals $\mathscr{I} = \{\mathcal{I}_k \subseteq [0,1]\}_{k\in K}$, where the endpoints of each $\mathcal{I}_k$ are rational numbers. With the states $S = \{s_0, s_1, \cdots, s_{d-1}\}$, we define the *symbolization* of distributions as a function:

$$\mathbb{S} : \mathcal{D}(\mathcal{S}) \to 2^{S\times\mathscr{I}} \qquad \mathbb{S}(\mu) = \{\langle s, \mathcal{I}\rangle \in S \times \mathscr{I} : \mu(s) \in \mathcal{I}\}, \tag{2}$$

where $\times$ denotes the Cartesian product, and $2^{S\times\mathscr{I}}$ is the power set of $S \times \mathscr{I}$. $\langle s, \mathcal{I}\rangle \in \mathbb{S}(\mu)$ asserts that the probability of *state* s in distribution μ is in the interval $\mathcal{I}$. The *symbolization of distributions* is a generalization of the discretization of distributions with $\mathcal{I}_k \cap \mathcal{I}_m = \emptyset$ for all $k \neq m$ which was studied in [2]. This generalization increases the expressiveness of our *continuous linear-time logic* introduced in the next section. Now, we can represent any given probability distribution by finite symbols from $S \times \mathscr{I}$. For example, suppose

$$\mathscr{I} = \{[0, 0.1], (0.1, 0.9), [0.9, 1], [1, 1], [0.4, 0.4]\}, \tag{3}$$

and then the initial distribution μ in Example 1 is symbolized as

$$\begin{aligned} \mathbb{S}(\mu) = \{&\langle s_0, [0, 0.1]\rangle, \langle s_1, (0.1, 0.9)\rangle, \langle s_2, (0.1, 0.9)\rangle, \\ &\langle s_3, (0.1, 0.9)\rangle, \langle s_3, [0.4, 0.4]\rangle, \langle s_4, [0, 0.1]\rangle\}. \end{aligned} \tag{4}$$

As we can see from the above example, the symbolization of distributions on states considers the exact probabilities (singleton intervals) of the states and the range of their possibilities.

Next, we introduce the symbolization to CTMCs,

Definition 5. *A* symbolized *CTMC is a tuple* $\mathcal{SM} = (S, Q, \mathscr{I})$*, where* $\mathcal{M} = (S, Q)$ *is a CTMC and* $\mathscr{I}$ *is a finite set of intervals in* $[0, 1]$.

As we can see, the set of intervals is picked depending on CTMCs. Then, we extend this symbolization to the path σ_μ:

$$\mathbb{S} \circ \sigma_\mu : \mathbb{R}^+ \to 2^{S \times \mathscr{I}}. \tag{5}$$

Definition 6. *Given a symbolized CTMC* $\mathcal{SM} = (S, Q, \mathscr{I})$*,* $\mathbb{S} \circ \sigma_\mu$ *is a* symbolic execution path *of* $\mathcal{M} = (S, Q)$.

Given a symbolized CTMC $\mathcal{SM} = (S, Q, \mathscr{I})$, the path σ_μ of CTMC $\mathcal{M} = (S, Q)$ over real numbers $\mathbb{R}^+$ generated by probability distribution μ induces a symbolic execution path $\mathbb{S} \circ \sigma_\mu$ over finite symbols $S \times \mathscr{I}$. Subsequently, the dynamics of CTMCs can be studied in terms of a language over $S \times \mathscr{I}$. In other words, we can study the temporal properties of CTMCs in the context of symbolized CTMCs.

5 Continuous Linear-time Logic

In this section, we introduce *continuous linear-time logic (CLL)*, a probabilistic linear-time temporal logic, to specify the temporal properties of a symbolized CTMC $\mathcal{SM} = (S, Q, \mathscr{I})$.

CLL has two types of formulas: state formulas and path formulas. The state formulas are constructed using propositional connectives. The path formulas are obtained by propositional connectives and a temporal modal operator *timed until* $U^{\mathcal{T}}$ for a bounded time interval $\mathcal{T}$, as in MTL and CSL. Furthermore, *multiphase timed until formulas* $\Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2 \dots U^{\mathcal{T}_n} \Phi_n$ are allowed to enrich the expressiveness of CLL. More importantly, *time reset* is involved in these multiphase formulas. Thus absolutely and relatively temporal properties of CTMCs can be studied.

Definition 7. *The* state formulas *of CLL are described according to the following syntax:*

$$\Phi := \mathbf{true} \mid a \in AP \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2$$

where AP *denotes* $S \times \mathscr{I}$ *as the set of atomic propositions.*

The path formulas *of CLL are constructed by the following syntax:*

$$\varphi := \mathbf{true} \mid \Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2 \dots U^{\mathcal{T}_n} \Phi_n \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$$

where $n \in \mathbb{Z}^+$ *is a positive integer, for all* $0 \leq k \leq n$*,* Φ_k *is a state formula, and* $\mathcal{T}_k$*'s are* time *intervals with the endpoints in* $\mathbb{Q}^+$*, i.e., each* $\mathcal{T}_k$ *is one of the following forms:*

$$(a, b), [a, b], (a, b], [a, b) \qquad \forall a, b \in \mathbb{Q}^+.$$

The semantics of CLL state formulas is defined on the set $\mathcal{D}(\mathcal{S})$ of probability distributions over S with the symbolized function $\mathbb{S}$ in Eq.(2) of Section 4.

(1) $\mu \models \mathbf{true}$ for all probability distributions $\mu \in \mathcal{D}(\mathcal{S})$;
(2) $\mu \models a$ iff $a \in \mathbb{S}(\mu)$;
(3) $\mu \models \neg\Phi$ iff it is not the case that $\mu \models \Phi$ (or written $\mu \not\models \Phi$);
(4) $\mu \models \Phi_1 \wedge \Phi_2$ iff $\mu \models \Phi_1$ and $\mu \models \Phi_2$.

The semantics of CLL path formulas is defined on execution paths $\{\sigma_\mu\}_{\mu\in\mathcal{D}(\mathcal{S})}$ of CTMC $\mathcal{M} = (S, Q)$.

(1) $\sigma_\mu \models \mathbf{true}$ for all probability distributions $\mu \in \mathcal{D}(\mathcal{S})$;
(2) $\sigma_\mu \models \Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2 \ldots U^{\mathcal{T}_n} \Phi_n$ iff there is a time instant $t \in \mathcal{T}_1$ such that $\sigma_{\mu_t} \models \Phi_1 U^{\mathcal{T}_2} \Phi_2 \ldots U^{\mathcal{T}_n} \Phi_n$, and for any $t' \in \mathcal{T}_1 \cap [0, t)$, $\mu_{t'} \models \Phi_0$, where $\sigma_{\mu_t} \models \Phi$ iff $\mu_t \models \Phi$, and μ_t is the distribution of the chain at time instant t, i.e., $\mu_t = e^{Qt}\mu \ \forall t \in \mathbb{R}^+$;
(3) $\sigma_\mu \models \neg\varphi$ iff it is not the case that $\sigma_\mu \models \varphi$ (written $\sigma_\mu \not\models \varphi$);
(4) $\sigma_\mu \models \varphi_1 \wedge \varphi_2$ iff $\sigma_\mu \models \varphi_1$ and $\sigma_\mu \models \varphi_2$.

Not surprisingly, other Boolean connectives are derived in the standard way, i.e., $\mathbf{false} = \neg\mathbf{true}$, $\Phi_1 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$ and $\Phi_1 \rightarrow \Phi_2 = \neg\Phi_1 \vee \Phi_2$, and the path formula φ follows the same way. Furthermore, we generalize temporal operators $\Diamond$ ("eventually") and $\Box$ ("always") of discrete-time systems into their timed variant $\Diamond^{\mathcal{T}}$ and $\Box^{\mathcal{T}}$, respectively, in the following:

$$\Diamond^{\mathcal{T}}\Phi = \mathbf{true} U^{\mathcal{T}} \Phi \qquad \Box^{\mathcal{T}}\Phi = \neg\Diamond^{\mathcal{T}}\neg\Phi.$$

For $n = 1$ in multiphase timed until formulas, the until operator $U^{\mathcal{T}_1}$ is a timed variant of the until operator of LTL; the path formula $\Phi_0 U^{\mathcal{T}_1} \Phi_1$ asserts that Φ_1 is satisfied at some time instant in the interval $\mathcal{T}_1$ and that at all preceding time instants in $\mathcal{T}_1$, Φ_0 holds. For example,

$$\varphi = \langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle,$$

as mentioned in introduction section.

For general n, the CLL path formula $\Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2 \ldots U^{\mathcal{T}_n} \Phi_n$ is explained over the induction on n. We first mention that $U^{\mathcal{T}}$ is right-associative, e.g., $\Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2$ stands for $\Phi_0 U^{\mathcal{T}_1} (\Phi_1 U^{\mathcal{T}_2} \Phi_2)$. This makes time reset, i.e., $\mathcal{T}_1$ and $\mathcal{T}_2$ do not have to be disjoint, and the starting time point of $\mathcal{T}_2$ is based on some time instant in $\mathcal{T}_1$. Recall the multiphase timed until formula in introduction section and this formula expresses a relative time property:

$$\varphi' = \langle s_0, [0.9, 1]\rangle U^{[3,7]} (\langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle),$$

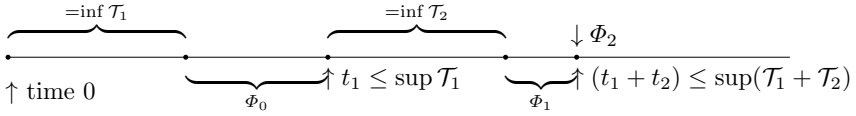
which is different to the following CLL path formula representing an absolutely temporal property of CTMCs:

$$\varphi'' = \Box^{[3,7]} \langle s_0, [0.9, 1]\rangle \wedge \langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle).$$

As an example, we clarify the semantics of CLL by comparing the above two path formulas in general forms:

$$\Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2 \quad \text{and} \quad \Phi_0 U^{\mathcal{T}_1} \Phi_1 \wedge \Phi_1 U^{\mathcal{T}_2} \Phi_2.$$

(1) $\sigma_\mu \models \Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2$ asserts that there are time instants $t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2$ such that $\mu_{t_1+t_2} \models \Phi_2$ and for any $t_1' \in \mathcal{T}_1 \cap [0, t_1)$ and $t_2' \in \mathcal{T}_2 \cap [0, t_2)$, $\mu_{t_1'} \models \Phi_0$ and $\mu_{t_1+t_2'} \models \Phi_1$, where $\mu_t = e^{Qt}\mu \ \forall t \in \mathbb{R}^+$. This is more clear in the following timeline.



(2) $\sigma_\mu \models \Phi_0 U^{\mathcal{T}_1} \Phi_1 \wedge \Phi_1 U^{\mathcal{T}_2} \Phi_2$ asserts that there are time instants $t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2$ such that $\mu_{t_1} \models \Phi_1$ and $\mu_{t_2} \models \Phi_2$, and for any $t_1' \in \mathcal{T}_1 \cap [0, t_1)$ and $t_2' \in \mathcal{T}_2 \cap [0, t_2)$, $\mu_{t_1'} \models \Phi_0$ and $\mu_{t_2'} \models \Phi_1$, where $\mu_t = e^{Qt}\mu \ \forall t \in \mathbb{R}^+$.

Before solving the model-checking problem of CTMCs against CLL formulas in the next section, we shall first discuss what can be specified in our logic CLL.

Given a CTMC (S, Q), CLL path formula $\Diamond^{[0,1000]}\langle s, [1,1]\rangle$ expresses a liveness property that *state* $s \in S$ is eventually reached with probability one before time instant 1000. In terms of safety properties, formula $\Box^{[100,1000]}\langle s, [0,0]\rangle$ represents that state $s \in S$ is never reached (reached with probability zero) between time instants 100 and 1000. Furthermore, setting the intervals nontrivial (neither $[0,0]$ or $[1,1]$), liveness and safety properties can be asserted with probabilities, such as $\Diamond^{[0,1000]}\langle s, [0.5,1]\rangle$ and $\Box^{[100,1000]}\langle s, [0,0.5]\rangle$. For multiphase timed until formula $\langle s, [0.7,1]\rangle U^{[2,3]}\langle s, [0.7,1]\rangle \ldots U^{[2,3]}\langle s, [0.7,1]\rangle$, where the number of $U^{[2,3]}$ is 100, asserts that the probability of *state* s is beyond 0.7 in every time instant 2 to 3, and this happens at least 100 times.

Next, we can classify members of $\mathscr{I}$ as representing "low" and "high" probabilities. For example, if $\mathscr{I}$ contains 3 intervals $\{[0, 0.1], (0.1, 0.9), [0.9, 1]\}$, we can declare the first interval as "low" and the last interval as "high". In this case $\Box^{[10,1000)}(\langle s_0, [0, 0.1]\rangle \rightarrow \langle s_1, [0.9, 1]\rangle)$ says that, in time interval $[10, 1000)$, whenever the probability of state s_0 is low, the probability of state s_1 will be high.

6 CLL Model Checking

In this section, we provide an algorithm to model check CTMCs against CLL formulas, i.e., the following CLL model-checking problem — Problem 1 is decidable.

Problem 1 (CLL Model-checking Problem). Given a symbolized CTMC $\mathcal{SM} = (S, Q, \mathscr{I})$ with an initial distribution μ and a CLL path formula φ on $AP = S \times \mathscr{I}$, the goal is to decide whether $\sigma_\mu \models \varphi$, where $\sigma_\mu(t) = e^{Qt}\mu$ is an execution path defined in Eq.(1).

In particular, we show that

Theorem 2. *Under the condition that Schanuel's conjecture holds, the CLL model-checking problem in Problem 1 is decidable.*

In the following, we prove the above theorem from checking basic formulas — atomic propositions to the most complex one — nontrivial multiphase timed until formulas. For readability, we put the proofs of all results in Appendix A of the extended version [21] of this paper.

We start with the simplest case of atomic proposition $\langle s, \mathcal{I}\rangle$. By the semantics of CLL, $\mu_t \models \langle s, \mathcal{I}\rangle$ if and only if $\mu_t = e^{Qt}\mu(s) \in \mathcal{I}$. To check this, we first observe that the execution path $e^{Qt}\mu$ of CTMCs is a system of *polynomial exponential functions (PEFs).*

Definition 8. *A function* $f : \mathbb{R} \to \mathbb{R}$ *is a* polynomial-exponential function (PEF) *if* f *has the following form:*

$$f(t) = \sum_{k=0}^{K} f_k(t) e^{\lambda_k t} \tag{6}$$

where for all $0 \leq k \leq K < \infty$, $f_k(t) \in \mathbb{F}_1[t], f_k(t) \neq 0$, $\lambda_k \in \mathbb{F}_2$ *and* $\mathbb{F}_1, \mathbb{F}_2$ *are fields. Without loss of generality, we assume that* λ_k*'s are distinct.*

Generally, for a PEF $f(t)$ with the range in complex numbers $\mathbb{C}$, $g(t) = f(t) + f^*(t)$ is a PEF with the range in real numbers $\mathbb{R}$, where $f^*(t)$ is the complex conjugate of $f(t)$. The factor t is omitted whenever convenient, i.e., $f = f(t)$. t is called a *root* of a function f if $f(t) = 0$. PEFs often appear in transcendental number theory as auxiliary functions in the proofs involving the exponential function [10].

Lemma 1. *Given a CTMC* $\mathcal{M} = (S, Q)$ *with* $S = \{s_0, \ldots, s_{d-1}\}$, $Q \in \mathbb{Q}^{d\times d}$, *and an initial distribution* $\mu \in \mathbb{Q}^{d\times 1}$, *for any* $0 \leq i \leq d-1$, $e^{Qt}\mu(s_i)$, *the* i*-th entry of* $e^{Qt}\mu$, *can be expressed as a PEF* $f : \mathbb{R}^+ \to [0, 1]$ *as in Eq.(6) with* $\mathbb{F}_1 = \mathbb{F}_2 = \mathbb{A}$.

By the above lemma, for a given t in some bounded time interval $\mathcal{T}$ (to be specific in the latter discussion), $e^{Qt}\mu(s) \in \mathcal{I}$ is determined by the algebraic structure of PEF $g(t) = e^{Qt}\mu(s)$ in $\mathcal{T}$. That is all *maximum intervals* $\mathcal{T}_{\max} \subseteq \mathcal{T}$ such that $g(t) \in \mathcal{I}$ for all $t \in \mathcal{T}_{\max}$, where interval $\mathcal{T}_{\max} \neq \emptyset$ is called maximum for $g(t) \in \mathcal{I}$ if no sub-intervals $\mathcal{T}' \subsetneq \mathcal{T}_{\max}$ such that the property holds, i.e., $g(t) \in \mathcal{I}$ for all $t \in \mathcal{T}'$. Then $e^{Qt}\mu(s) \in \mathcal{I}$ if and only if $t \in \mathcal{T}_{\max}$ for some maximum interval $\mathcal{T}_{\max}$. So, we aim to compute the set $\mathscr{T}$ of all maximum intervals. By the continuity of PEF $g(t)$, this can be done by identifying a *real root isolation* of the following PEF $f(t)$ in $\mathcal{T}$: $f(t) = (g(t) - \inf \mathcal{I})(g(t) - \sup \mathcal{I})$.

A (real) *root isolation* of function $f(t)$ in interval $\mathcal{T}$ is a set of mutually disjoint intervals, denoted by $\mathrm{Iso}(f)_{\mathcal{T}} = \{(a_j, b_j) \subseteq \mathcal{T}\}$ for $a_j, b_j \in \mathbb{Q}$ such that

- for any j, there is one and only one root of $f(t)$ in (a_j, b_j);

– for any root t^* of $f(t)$, $t^* \in (a_j, b_j)$ for some j.

Furthermore, if f has no any root in $\mathcal{T}$, then $\text{Iso}(f)_{\mathcal{T}} = \emptyset$.

Although there are infinite kinds of real root isolations of $f(t)$ in $\mathcal{T}$, the number of isolation intervals equals to the number of distinct roots of $f(t)$ in $\mathcal{T}$.

Finding real root isolations of PEFs is a long-standing problem and can be at least backtracked to Ritt's paper [34] in 1929. Some following results were obtained since the last century (e.g. [7,38]). This problem is essential in the reachability analysis of dynamical systems, one active field of symbolic and algebraic computation. In the case of $\mathbb{F}_1 = \mathbb{Q}$ and $\mathbb{F}_2 = \mathbb{N}^+$ in [1], an algorithm named ISOL was proposed to isolate all real roots of $f(t)$. Later, this algorithm has been extended to the case of $\mathbb{F}_1 = \mathbb{Q}$ and $\mathbb{F}_2 = \mathbb{R}$ [20]. A variant of the problem has also been studied in [28]. The correctness of these algorithms is based on Schanuel's conjecture. Other works are using Schanuel's conjecture to do the root isolation of other functions, such as exp-log functions [35] and tame elementary functions [36].

By Lemma 1, we pursue this problem in the context of CTMCs. The distinct feature of solving real root isolations of PEFs in our paper is to deal with complex numbers $\mathbb{C}$, more specifically algebraic numbers $\mathbb{A}$, i.e., $\mathbb{F}_1 = \mathbb{F}_2 = \mathbb{A}$. At the same time, to the best of our knowledge, all the previous works can only handle the case over $\mathbb{R}$. Here, we develop a state-of-the-art real root isolation algorithm for PEFs over algebraic numbers. Thus from now on, we always assume that PEFs are over $\mathbb{A}$, i.e., $\mathbb{F}_1 = \mathbb{F}_2 = \mathbb{A}$ in Eq.(6). In this case, it is worth noting that whether a PEF has a root in a given interval, $\mathcal{T} \subseteq \mathbb{R}^+$ is decidable subject to Schanuel's Conjecture if $\mathcal{T}$ is bounded [16], which falls in the situation we consider in this paper.

Theorem 3 ([16]). *Under the condition that Schanuel's conjecture holds, there is an algorithm to check whether a PEF $f(t)$ has a root in interval $\mathcal{T}$, i.e., whether $Iso(f)_{\mathcal{T}} = \emptyset$.*

In this paper, we extend the above checking $\text{Iso}(f)_{\mathcal{T}} = \emptyset$ to computing $\text{Iso}(f)_{\mathcal{T}}$ of PEF $f(t)$.

Theorem 4. *Under the condition that Schanuel's conjecture holds, there is an algorithm to find real root isolation $Iso(f)_{\mathcal{T}}$ for any PEF $f(t)$ and interval $\mathcal{T}$. Furthermore, the number of real roots is finite, i.e., $|Iso(f)_{\mathcal{T}}| < \infty$.*

We can compute the set $\mathscr{T}$ of all maximum intervals with the above theorem to check atomic propositions. Furthermore, we can compare the values of any real roots of PEFs, which is important in model checking general multiphase timed until formulas at the end of this section.

Lemma 2. *Let $f_1(t)$ and $f_2(t)$ be two PEFs with the domains in $\mathcal{T}_1$ and $\mathcal{T}_2$, and $t_1 \in \mathcal{T}_1$ and $t_2 \in \mathcal{T}_2$ are roots of them, respectively. Under the condition that Schanuel's conjecture holds, there is an efficient way to check whether or not $t_1 - t_2 < g$ for any given rational number $g \in \mathbb{Q}$.*

For model checking general state formula Φ, we can also use real root isolation of some PEF to obtain the set of all maximum intervals $\mathcal{T}_{\max}$ such that $\mu_t \models \Phi$ for all $t \in \mathcal{T}_{\max}$. The reason is that Φ admits *conjunctive normal form* consisting of atomic propositions. See the proof of the following lemma in Appendix A of the extended version [21] of this paper for the details.

Lemma 3. *Under the condition that Schanuel's conjecture holds, given a time interval $\mathcal{T}$, the set $\mathscr{T}$ of all maximum intervals in $\mathcal{T}$ satisfying $\mu_t \models \Phi$ can be computed, where Φ is a state formula of CLL. Furthermore, the number of all intervals in $\mathscr{T}$ is finite; the left and right endpoints of each interval in $\mathscr{T}$ are roots of PEFs.*

At last, we characterize the multiphase timed until formulas by the reachability analysis of time intervals (instants).

Lemma 4. *$\sigma_\mu \models \Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2 \cdots U^{\mathcal{T}_n} \Phi_n$ if and only if there exist time intervals $\{\mathcal{I}_k \subseteq \mathbb{R}^+\}_{k=0}^n$ with $\mathcal{I}_0 = [0,0]$ such that*

- The satisfaction of intervals*: for all $1 \le k \le n$, $\mu_t \models \Phi_{k-1}$ for all $t \in \mathcal{I}_k$, and $\mu_{t^*} \models \Phi_n$, where $t^* = \sup \mathcal{I}_n$ and $\mu_t = e^{Qt}\mu$ $\forall t \in \mathbb{R}^+$;*
- The order of intervals*: for all $1 \le k \le n$, $\mathcal{I}_k \subseteq \mathcal{I}_{k-1} + \mathcal{T}_k$ and $\inf \mathcal{I}_k = \sup \mathcal{I}_{k-1} + \inf \mathcal{T}_k$.*

By the above lemma, the problem of checking multiphase timed until formulas is reduced to verify the existence of a sequence of time intervals.

Now we can show the proof of Theorem 2.

Proof. Recall that the nontrivial step is to model check multiphase timed until formula $\Phi_0 U^{\mathcal{T}_1} \Phi_1 U^{\mathcal{T}_2} \Phi_2 \cdots U^{\mathcal{T}_n} \Phi_n$, where $\{\mathcal{T}_j\}_{j=1}^n$ is a set of bounded rational intervals in $\mathbb{R}^+$, and for $0 \le k \le n+1$, Φ_k is a state formula.

By Lemma 4, for model checking the above formula, we only need to check the existence of time intervals $\{\mathcal{I}_k\}_{k=0}^n$ illustrated in the lemma. The following procedure can construct such a set of intervals if it exists:

- (1) Let $\mathscr{I}_0 = \{\mathcal{I}_0 = [0,0]\}$;
- (2) For each $1 \le k \le n$, obtaining the set $\mathscr{I}_k$ in $[0, \sum_{j=1}^k \sup \mathcal{T}_j]$ of all maximum intervals such that $\mu_t \models \Phi_{k-1}$ for all $t \in \mathcal{I}$ of $\mathcal{I} \in \mathscr{I}$, where $\mu_t = e^{Qt}\mu$; this can be done by Lemma 3. Noting that $\mathscr{I}_k$ can be the empty set, i.e., $\mathscr{I}_k = \emptyset$;
- (3) Let k from 1 to n. First, updating $\mathscr{I}_k$:

$$\mathscr{I}_k = \{\mathcal{I} \cap (\mathcal{I}' + \mathcal{T}_k) : \mathcal{I} \in \mathscr{I}_k \text{ and } \mathcal{I}' \in \mathscr{I}_{k-1}\}. \tag{7}$$

  The above updates can be finished by Lemma 2. If $\mathscr{I}_k = \emptyset$, then the formula is not satisfied;
- (4) Updating $\mathscr{I}_n$: for each $\mathcal{I} \in \mathscr{I}_n$, we replace $\mathcal{I}$ with $[s-\varepsilon, s)$ for some constant $\varepsilon > 0$ if there is an $s \in \mathcal{I}$ with $s - \varepsilon \in \mathcal{I}$ such that $\mu_s \models \Phi_n$ where $\mu_s = e^{Qs}\mu$; Otherwise, remove this element from $\mathscr{I}_n$. Again, this can be done by Lemma 3. If $\mathscr{I}_n = \emptyset$, then the formula is not satisfied;

– (5) Finally, let k from $n-1$ to 1, updating $\mathscr{I}_k$:

$$\mathscr{I}_k = \{[s - \inf \mathcal{T}_k, s - \inf \mathcal{T}_k] : [s-\varepsilon, s) \in \mathscr{I}_{k+1}]\}.$$

Thus after the above procedure, we have non-empty sets $\{\mathscr{I}_k\}_{k=0}^n$ with the following properties.

– for each $1 \leq k \leq n$, $\mu_t \models \Phi_{k-1}$ for all $t \in \mathcal{I}_k$ and $\mathcal{I}_k \in \mathscr{I}_k$, and $\mu_{t^*} \models \Phi_n$, where $t^* = \sup \mathcal{I}_n$;
– for each $1 \leq k \leq n$, $\mathcal{I} \in \mathscr{I}_k$, there exists at least one $\mathcal{I}' \in \mathscr{I}_{k-1}$ such that $\mathcal{I} \subseteq \sup \mathcal{I}' + \mathcal{T}_k$ and $\inf \mathcal{I} = \sup \mathcal{I}' + \inf \mathcal{T}_k$.

Therefore, we can get a set of intervals $\{\mathcal{I}_k\}_{k=0}^n$ satisfying the two conditions in Lemma 4 if it exists. On the other hand, it is easy to check that all such $\{\mathcal{I}_k\}_{k=0}^n$ must be in $\{\mathscr{I}_k\}_{k=0}^n$, i.e., for each k, $\mathcal{I}_k \subseteq \mathcal{I}$ for some $\mathcal{I} \in \mathscr{I}_k$. This ensures the correctness of the above procedure.

By the above constructive analysis, we give an algorithm for model checking CTMCs against CLL formulas. Focusing on the decidability problem, we do not provide the pseudocode of the algorithm. Alternatively, we implement a numerical experiment to illustrate the checking procedure in the next section.

7 Numerical Implementation

In this section, we implement a case study of checking CTMCs against CLL formulas. Here, we consider a symbolized CTMC $\mathcal{SM} = (S, Q, \mathscr{I})$, where $\mathcal{M} = (S, Q)$ is the CTMC in Example 1 and finite set $\mathscr{I}$ is the one considered in Eq.(3). We check the properties of $\mathcal{M}$ given by the following two CLL path formulas mentioned in the introduction for different initial distributions.

$$\varphi = \langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle.$$
$$\varphi' = \langle s_0, [0.9, 1]\rangle U^{[3,7]} \langle s_1, [0, 0.1]\rangle U^{[0,5]} \langle s_0, [0.9, 1]\rangle.$$

By Jordan decomposition, we have $Q = SJS^{-1}$ where

$$S = \begin{pmatrix} 0 & -6 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 1 \\ -7 & -3 & 0 & 0 & 0 \\ 3 & 3 & 0 & 1 & 0 \\ 4 & 4 & 1 & 0 & 0 \end{pmatrix} \quad J = \begin{pmatrix} -7 & 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad S^{-1} = \begin{pmatrix} \frac{1}{14} & 0 & -\frac{1}{7} & 0 & 0 \\ -\frac{1}{6} & 0 & 0 & 0 & 0 \\ \frac{8}{21} & 0 & \frac{4}{7} & 0 & 1 \\ \frac{2}{7} & 0 & \frac{3}{7} & 1 & 0 \\ \frac{1}{3} & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Then, we consider an initial distribution μ as the same as the one in Example 1. Then we have that the value of $e^{Qt}\mu$ is as follows:

$$\begin{pmatrix} e^{-3t} & 0 & 0 & 0 & 0 \\ -\frac{1}{3}(e^{-3t}-1) & 1 & 0 & 0 & 0 \\ \frac{1}{2}(e^{-3t}-e^{-7t}) & 0 & e^{-7t} & 0 & 0 \\ \frac{3}{14}e^{-7t} - \frac{1}{2}e^{-3t} + \frac{2}{7} & 0 & -\frac{3}{7}e^{-7t} + \frac{3}{7} & 1 & 0 \\ \frac{2}{7}e^{-7t} - \frac{2}{3}e^{-3t} + \frac{8}{21} & 0 & -\frac{4}{7}e^{-7t} + \frac{4}{7} & 0 & 1 \end{pmatrix} \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{10}e^{-3t} \\ -\frac{1}{30}e^{-3t} + \frac{7}{30} \\ \frac{1}{20}e^{-3t} + \frac{1}{4}e^{-7t} \\ -\frac{1}{20}e^{-3t} - \frac{3}{28}e^{-7t} + \frac{39}{70} \\ -\frac{1}{15}e^{-3t} - \frac{1}{7}e^{-7t} + \frac{22}{105} \end{pmatrix}.$$

As we only consider states s_0 and s_1 in formulas φ and φ', we focus on the following PEFs: $f_0(t) = \frac{1}{10}e^{-3t}$ and $f_1(t) = -\frac{1}{30}e^{-3t} + \frac{7}{30}$.

Next, we initialize the model checking procedures introduced in the proof of Theorem 2. First, we compute the set $\mathscr{T}$ of all maximum intervals $\mathcal{T} \subseteq [0,5]$ such that $e^{Qt}\mu \models \langle s_0, [0.9,1]\rangle$ for $t \in \mathcal{T}$, i.e., $f_0(t) \in [0.9,1]$ for $t \in \mathcal{T}$. We obtain $\mathscr{T} = \emptyset$ by the real root isolation algorithm mentioned in Theorem 4, and this indicates that $\sigma_\mu \not\models \varphi$ where $\sigma_\mu(t) = e^{Qt}\mu$ is the path induced by μ and defined in Eq.(1).

To check whether $\sigma_\mu \models \varphi'$, we compute the set $\mathscr{T}$ of all maximum intervals $\mathcal{T} \subseteq [0,12]$ such that $e^{Qt}\mu \models \langle s_0, [0.9,1]\rangle$ for $t \in \mathcal{T}$, i.e., $f_0(t) \in [0.9,1]$ for $t \in \mathcal{T}$. Again, we obtain $\mathscr{T} = \emptyset$ by the real root isolation algorithm in Theorem 4. Therefore, $\sigma_\mu \not\models \varphi'$.

In the following, we consider a different initial distribution μ_1 as follows:

$$e^{Qt}\mu_1 = e^{Qt}\begin{pmatrix}0.9\\0\\0.1\\0\\0\end{pmatrix} = \begin{pmatrix}\frac{9}{10}e^{-3t}\\ -\frac{3}{10}(e^{-3t}-1)\\ \frac{9}{20}e^{-3t} - \frac{7}{20}e^{-7t}\\ -\frac{9}{20}e^{-3t} + \frac{3}{20}e^{-7t} + \frac{3}{10}\\ -\frac{3}{5}e^{-3t} + \frac{1}{5}e^{-7t} + \frac{2}{5}\end{pmatrix}.$$

The key PEFs are: $g_0(t) = \frac{9}{10}e^{-3t}$ and $g_1(t) = -\frac{3}{10}(e^{-3t}-1)$.

Again, we initialize the model checking procedures introduced in the proof of Theorem 2. We first compute the set $\mathscr{T}$ of all maximum intervals $\mathcal{T} \subseteq [0,5]$ such that $e^{Qt}\mu_1 \models \langle s_1, [0,0.1]\rangle$ for $t \in \mathcal{T}$, i.e., $g_1(t) \in [0,0.1]$ for $t \in \mathcal{T}$. This can be done by finding a real root isolation of the following PEF: $g_1^0(t) = -\frac{3}{10}(e^{-3t} - 1) - \frac{1}{10}$.

By implementing the real root isolation algorithm in Theorem 4, we have

$$\mathrm{Iso}(g_1^0)_{[0,5]} = \{(0.13, 0.14)\} \text{ and then } \mathscr{T} = \{[0, t^*]\} \text{ for } t^* \in (0.13, 0.14).$$

Following the same way, we compute $\mathscr{T}$ for $e^{Qt}\mu_1 \models \langle s_0, [0.9,1]\rangle$. Then we complete the model checking procedures in the proof of Theorem 2, and we conclude: $\sigma_{\mu_1} \models \varphi$. By repeating these, the result of the second formula φ' is $\sigma_{\mu_1} \not\models \varphi'$.

8 Related Works

Agrawal *et al.* [2] introduced probabilistic linear-time temporal logic (PLTL) to reason about discrete-time Markov chains in the context of distribution transformers as we did for CTMCs in this paper. Interestingly, the Skolem Problem can be reduced to the model checking problem for the logic PLTL [3]. The Skolem Problem asks whether a given linear recurrence sequence has a zero term and plays a vital role in the reachability analysis of linear dynamical systems. Unfortunately, the decidability of the problem remains open [32]. Recently, the Continuous Skolem Problem has been proposed with good behavior (the problem is decidable) and forms a fundamental decision problem concerning reachability

in continuous-time linear dynamical systems [16]. Not surprisingly, the Continuous Skolem Problem can be reduced to model-checking CLL. The primary step of verifying CLL formulas is to find a real root isolation of a PEF in a given interval. Chonev, Ouaknine and Worrell reformulated the Continuous Skolem Problem in terms of whether a PEF has a root in a given interval, which is decidable subject to Schanuel's conjecture [16]. An algorithm for finding root isolation can also answer the problem of checking the existence of the roots of a PEF. However, the reverse does not work in general. Therefore, the decidability of the Continuous Skolem Problem cannot be applied to establish that of our CLL model checking.

Remark 1. By adopting the method in this paper, we established the decidability of model checking quantum CTMCs against signal temporal logic [40]. Again, we need Schanuel's conjecture to guarantee the correctness. A Lindblad's master equation governs a quantum CTMC and a more general real-time probabilistic Markov model than a CTMC, i.e., a CTMC is an instance of quantum CTMCs. We converted the evolution of Lindblad's master equation into a distribution transformer that preserves the laws of quantum mechanics. We reduced the model-checking problem of quantum CTMCs to the real root isolation problem, which we considered in this paper, and thus our method could be applied to it.

9 Conclusion

This paper revisited the study of temporal properties of finite-state CTMCs by symbolizing the probability value space $[0, 1]$ into a finite set of intervals. To specify relatively and absolutely temporal properties, we propose a probabilistic logic for CTMCs, namely continuous linear-time logic (CLL). We have considered the model checking problem in this setting. Our main result is that a state-of-the-art real root isolation algorithm over the field of algebraic numbers was proposed to establish the decidability of the model checking problem under the condition that Schanuel's conjecture holds.

This paper aims to show decidability in as simple a fashion as possible without paying much attention to complexity issues. Faster algorithms on our current constructions would significantly improve from a practical standpoint.

Acknowledgments

We want to thank Professor Joost-Pieter Katoen for his invaluable feedback and for pointing out the references [14,15,30]. This work is supported by the National Key R&D Program of China (Grant No: 2018YFA0306701), the National Natural Science Foundation of China (Grant No: 61832015), ARC Discovery Program (#DP210102449) and ARC DECRA (#DE180100156).

References

1. Achatz, M., McCallum, S., Weispfenning, V.: Deciding polynomial-exponential problems. In: Proceedings of the Twenty-first International Symposium on Symbolic and Algebraic Computation. pp. 215–222. ACM (2008)
2. Agrawal, M., Akshay, S., Genest, B., Thiagarajan, P.: Approximate verification of the symbolic dynamics of Markov chains. Journal of the ACM (JACM) **62**(1), 2 (2015)
3. Akshay, S., Antonopoulos, T., Ouaknine, J., Worrell, J.: Reachability problems for Markov chains. Information Processing Letters **115**(2), 155–158 (2015)
4. Almagor, S., Kelmendi, E., Ouaknine, J., Worrell, J.: Invariants for continuous linear dynamical systems. arXiv preprint arXiv:2004.11661 (2020)
5. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**, 183–235 (1994)
6. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing. pp. 592–601 (1993)
7. Avellar, C.E., Hale, J.K.: On the zeros of exponential polynomials. Journal of Mathematical Analysis and Applications **73**(2), 434–452 (1980)
8. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. ACM Transactions on Computational Logic **1**(1), 162–170 (2000)
9. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Transactions on Software Engineering **29**(6), 524–541 (2003)
10. Baker, A.: Transcendental number theory. Cambridge university press (1990)
11. Barbot, B., Chen, T., Han, T., Katoen, J.P., Mereacre, A.: Efficient CTMC model checking of linear real-time objectives. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 128–142. Springer (2011)
12. Chen, T., Diciolla, M., Kwiatkowska, M., Mereacre, A.: Time-bounded verification of CTMCs against real-time specifications. In: International Conference on Formal Modeling and Analysis of Timed Systems. pp. 26–42. Springer (2011)
13. Chen, T., Han, T., Katoen, J.P., Mereacre, A.: Quantitative model checking of continuous-time Markov chains against timed automata specifications. In: 2009 24th Annual IEEE Symposium on Logic In Computer Science. pp. 309–318. IEEE (2009)
14. Chen, T., Han, T., Katoen, J.P., Mereacre, A.: Model checking of continuous-time Markov chains against timed automata specifications. Logical Methods in Computer Science **7**(1) (Mar 2011)
15. Chen, T., Han, T., Katoen, J.P., Mereacre, A.: Observing continuous-time MDPs by 1-clock timed automata. In: International Workshop on Reachability Problems. pp. 2–25. Springer (2011)
16. Chonev, V., Ouaknine, J., Worrell, J.: On the skolem problem for continuous linear dynamical systems. In: Chatzigiannakis, I., Mitzenmacher, M., Rabani, Y., Sangiorgi, D. (eds.) 43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 55, pp. 100:1–100:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016)
17. Cohen, H.: A course in computational algebraic number theory, vol. 138. Springer Science & Business Media (2013)

18. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A STORM is coming: A modern probabilistic model checker. In: International Conference on Computer Aided Verification. pp. 592–600. Springer (2017)
19. Feng, Y., Katoen, J.P., Li, H., Xia, B., Zhan, N.: Monitoring CTMCs by multi-clock timed automata. In: International Conference on Computer Aided Verification. pp. 507–526. Springer (2018)
20. Gan, T., Chen, M., Li, Y., Xia, B., Zhan, N.: Reachability analysis for solvable dynamical systems. IEEE Transactions on Automatic Control **63**(7), 2003–2018 (2017)
21. Guan, J., Yu, N.: A probabilistic logic for verifying continuous-time markov chains. arXiv preprint arXiv:2004.08059 (2020)
22. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6**(5), 512–535 (1994)
23. Katoen, J.P.: The probabilistic model checking landscape. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 31–45. ACM (2016)
24. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Performance Evaluation **68**(2), 90–104 (2011)
25. Kolmogoroff, A.: Über die analytischen methoden in der wahrscheinlichkeitsrechnung. Mathematische Annalen **104**(1), 415–458 (1931)
26. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. pp. 200–204. Springer (2002)
27. Lang, S.: Introduction to transcendental numbers. Addison-Wesley Pub. Co. (1966)
28. Li, J.C., Huang, C.C., Xu, M., Li, Z.B.: Positive root isolation for poly-powers. In: Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation. pp. 325–332. ACM (2016)
29. Macintyre, A., Wilkie, A.J.: On the decidability of the real exponential field (1996)
30. Majumdar, R., Salamati, M., Soudjani, S.: On decidability of time-bounded reachability in CTMDPs. In: Czumaj, A., Dawar, A., Merelli, E. (eds.) 47th International Colloquium on Automata, Languages, and Programming (ICALP 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 168, pp. 133:1–133:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020)
31. Nesterenko, Y.: Modular functions and transcendence problems. Comptes rendus de l'Académie des sciences. Série 1, Mathématique **322**(10), 909–914 (1996)
32. Ouaknine, J., Worrell, J.: Decision problems for linear recurrence sequences. In: International Workshop on Reachability Problems. pp. 21–28. Springer (2012)
33. Richardson, D.: How to recognize zero. Journal of Symbolic Computation **24**(6), 627–645 (1997)
34. Ritt, J.F.: On the zeros of exponential polynomials. Transactions of the American Mathematical Society **31**(4), 680–686 (1929)
35. Strzebonski, A.: Real root isolation for exp-log functions. In: Proceedings of the Twenty-first International Symposium on Symbolic and Algebraic Computation. pp. 303–314 (2008)
36. Strzebonski, A.: Real root isolation for tame elementary functions. In: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation. pp. 341–350 (2009)
37. Terzo, G.: Some consequences of Schanuel's conjecture in exponential rings. Communications in Algebra® **36**(3), 1171–1189 (2008)

38. Tijdeman, R.: On the number of zeros of general exponential polynomials. In: Indagationes Mathematicae (Proceedings). vol. 74, pp. 1–7. North-Holland (1971)
39. Xu, M., Deng, Y.: Time-bounded termination analysis for probabilistic programs with delays. Information and Computation **275**, 104634 (2020)
40. Xu, M., Mei, J., Guan, J., Yu, N.: Model checking quantum continuous-time Markov chains. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory (CONCUR 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 203, pp. 13:1–13:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021)
41. Zhang, L., Jansen, D.N., Nielson, F., Hermanns, H.: Automata-based CSL model checking. In: International Colloquium on Automata, Languages, and Programming. pp. 271–282. Springer (2011)

Under-Approximating Expected Total Rewards in POMDPs*

Alexander Bork[1](✉), Joost-Pieter Katoen[1], and Tim Quatmann[1]

RWTH Aachen University, Aachen, Germany
alexander.bork@cs.rwth-aachen.de

Abstract We consider the problem: is the optimal expected total reward to reach a goal state in a partially observable Markov decision process (POMDP) below a given threshold? We tackle this—generally undecidable—problem by computing under-approximations on these total expected rewards. This is done by abstracting finite unfoldings of the infinite belief MDP of the POMDP. The key issue is to find a suitable under-approximation of the value function. We provide two techniques: a simple (cut-off) technique that uses a good policy on the POMDP, and a more advanced technique (belief clipping) that uses minimal shifts of probabilities between beliefs. We use mixed-integer linear programming (MILP) to find such minimal probability shifts and experimentally show that our techniques scale quite well while providing tight lower bounds on the expected total reward.

1 Introduction

The relevance of POMDPs. Partially observable Markov decision processes (POMDPs) originated in operations research and nowadays are a pivotal model for planning in AI [40]. They inherit all features of classical MDPs: each state has a set of discrete probability distributions over the states and rewards are earned when taking transitions. However, states are *not* fully observable. Intuitively, certain aspects of the states can be identified, such as a state's colour, but states themselves cannot be observed. This partial observability reflects, for example, a robot's view of its environment while only having the limited perspective of its sensors at its disposal. The main goal is to obtain a policy—a plan how to resolve the non-determinism in the model—for a given objective. The key problem here is that POMDP policies must base their decisions *only* on the observable aspects (e.g. colours) of states. This stands in contrast to policies for MDPs which can make decisions dependent on the entire history of *full* state information.

Analysing POMDPs. Typical POMDP planning problems consider either finite-horizon objectives or infinite-horizon objectives under discounting. Finite-horizon objectives focus on reaching a certain goal state (such as *"the robot has collected all items"*) within a given number of steps. For infinite horizons, no step bound

* This work is funded by the DFG RTG 2236 "UnRAVeL".

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 22–40, 2022.
https://doi.org/10.1007/978-3-030-99527-0_2

is provided and typically rewards along a run are weighted by a discounting factor that indicates how much immediate rewards are favoured over more distant ones. Existing techniques to treat these objectives include variations of value iteration [46,36,20,18,52,53] and policy trees [29]. Point-based techniques [38,42] approximate a POMDP's value function using a finite subset of beliefs which is iteratively updated. Algorithms include *PBVI* [38], *Perseus* [48], *SARSOP* [30] and *HSVI* [45]. Point-based methods can treat large POMDPs for both finite- and discounted infinite-horizon objectives [42].

Problem statement. In this paper we consider the problem: *is the maximal expected total reward to reach a given goal state in a POMDP below a given threshold?* We thus consider an infinite-horizon objective *without* discounting—also called an *indefinite-horizon* objective. A specific instance of the considered problem is the reachability probability to eventually reach a given goal state in a POMDP. This problem is undecidable [33,34] in general. Intuitively, this is due to the fact that POMDP policies need to consider the entire (infinite) observation history to make optimal decisions. For a POMDP, this notion is captured by an infinite, fully observable MDP, its *belief MDP*. This MDP is obtained from observation sequences inducing probabilities of being in certain states of the POMDP.

Previously proposed methods to solve the problem are e.g. to use approximate value iteration [22], optimisation and search techniques [1,12], dynamic programming [6], Monte Carlo simulation [43], game-based abstraction [51], and machine learning [13,14,19]. Other approaches restrict the memory size of the policies [35]. The synthesis of (possibly randomised) finite-memory policies is ETR-complete[1] [28]. Techniques to obtain finite-memory policies use e.g. parameter synthesis [28] or satisfiability checking and SMT solving [15,50].

Our approach. We tackle the aforementioned problem by computing under-approximations on maximal total expected rewards. This is done by considering finite unfoldings of the infinite belief MDP of the POMDP, and then applying abstraction. The key issue here is to find a suitable under-approximation of the POMDP's value function. We provide two techniques: a simple (cut-off) technique that uses a good policy on the POMDP, and a more advanced technique (belief clipping) that uses minimal shifts of probabilities between beliefs and can be applied on top of the simple approach. We use mixed-integer linear programming (MILP) to find such minimal probability shifts. Cut-off techniques for indefinite-horizon objectives have been used on computation trees—rather than on the belief MDP as used here—in *Goal-HSVI* [24]. Belief clipping amends the probabilities in a belief to be in a state of the POMDP yielding discretised values, i.e. an abstraction of the probability range $[0, 1]$ is applied. Such grid-based approximations are inspired by Lovejoy's grid-based belief MDP discretisation method [32]. They have also been used in [7] in the context of dynamic programming for POMDPs, and to over-approximate the value function in model checking of POMDPs [8]. In fact, this paper on determining lower bounds for

[1] A decision problem is ETR-complete if it can be reduced to a polynomial-length sentence in the Existential Theory of the Reals (for which the satisfiability problem is decidable) in polynomial time, and there is such a reduction in the reverse direction.

indefinite-horizon objectives can be seen as the dual counterpart of [8]. Our key challenge—compared to the approach of [8]—is that the value at a certain belief cannot easily be under-approximated with a convex combination of values of nearby beliefs. On the other hand, an under-approximation can benefit from a "good" guess of some initial POMDP policy. In the context of [8], such a guessed policy is of limited use for over-approximating values in the POMDP induced by an *optimal* policy. Although our approach is applicable to all thresholds, the focus of our work is on determining under-approximations for *quantitative* objectives. Dedicated verification techniques for the qualitative setting—almost-sure reachability—are presented in [17,16,27].

Experimental results. We have implemented our cut-off and belief clipping approaches on top of the probabilistic model checker Storm [23] and applied it to a range of various benchmarks. We provide a comparison with the model checking approach in [37], and determine the tightness of our under-approximations by comparing them to over-approximations obtained using the algorithm from [8]. Our main findings from the experimental validation are:

- Cut-offs often generate tight bounds while being computationally inexpensive.
- The clipping approach may further improve the accuracy of the approximation.
- Our implementation can deal with POMDPs with tens of thousands of states.
- Mostly, the obtained under-approximations are less than 10% off.

2 Preliminaries and Problem Statement

Let $Dist(A) := \{\mu : A \to [0,1] \mid \sum_{a \in A} \mu(a) = 1\}$ denote the set of probability distributions over a finite set A. The set $supp(\mu) := \{a \in A \mid \mu(a) > 0\}$ is the *support* of $\mu \in Dist(A)$. Let $\mathbb{R}^\infty := \mathbb{R} \cup \{\infty, -\infty\}$. We use Iverson bracket notation, where $[x] = 1$ if the Boolean expression x is *true* and $[x] = 0$ otherwise.

2.1 Partially Observable MDPs

Definition 1 (MDP). *A* Markov decision process (MDP) *is a tuple* $M = \langle S, Act, \mathbf{P}, s_{init} \rangle$ *with a (finite or infinite) set of states* S*, a finite set of actions* Act*, a transition function* $\mathbf{P} \colon S \times Act \times S \to [0,1]$ *with* $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{0, 1\}$ *for all* $s \in S$ *and* $\alpha \in Act$*, and an initial state* s_{init}*.*

We fix an MDP $M := \langle S, Act, \mathbf{P}, s_{init} \rangle$. For $s \in S$ and $\alpha \in Act$, let $post^M(s, \alpha) := \{s' \in S \mid \mathbf{P}(s, \alpha, s') > 0\}$ denote the set of α-successors of s in M. The set of *enabled actions* in $s \in S$ is given by $Act(s) := \{\alpha \in Act \mid post^M(s, \alpha) \neq \emptyset\}$.

Definition 2 (POMDP). *A* partially observable MDP (POMDP) *is a tuple* $\mathcal{M} = \langle M, Z, O \rangle$*, where* M *is the underlying MDP with* $|S| \in \mathbb{N}$*, i.e.* S *is finite,* Z *is a finite set of observations, and* $O \colon S \to Z$ *is an observation function such that* $O(s) = O(s') \implies Act(s) = Act(s')$ *for all* $s, s' \in S$*.*

We fix a POMDP $\mathcal{M} := \langle M, Z, O \rangle$ with underlying MDP M. We lift the notion of enabled actions to observations $z \in Z$ by setting $Act(z) := Act(s)$ for some

$s \in S$ with $O(s) = z$ which is valid since states with the same observations are required to have the same enabled actions. The notions defined for MDPs below also straightforwardly apply to POMDPs.

Remark 1. More general observation functions of the form $O : S \times Act \to Dist(Z)$ can be encoded in this formalism by using a polynomially larger state space [16].

An *infinite path* through an MDP (and a POMDP) is a sequence $\tilde{\pi} = s_0\alpha_1 s_1\alpha_2 \ldots$ such that $\alpha_{i+1} \in Act(s_i)$ and $s_{i+1} \in post^M(s_i, \alpha_{i+1})$ for all $i \in \mathbb{N}$. A *finite path* is a finite prefix $\hat{\pi} = s_0\alpha_1 \ldots \alpha_n s_n$ of an infinite path $\tilde{\pi}$. For finite $\hat{\pi}$ let $last(\hat{\pi}) := s_n$ and $|\hat{\pi}| := n$. For infinite $\tilde{\pi}$ set $|\tilde{\pi}| := \infty$ and let $\tilde{\pi}[i]$ denote the finite prefix of length $i \in \mathbb{N}$. We denote the set of finite and infinite paths in M by $Paths^M_{\text{fin}}$ and $Paths^M_{\text{inf}}$, respectively. Let $Paths^M := Paths^M_{\text{fin}} \cup Paths^M_{\text{inf}}$. Paths are lifted to the observation level by *observation traces*. The observation trace of a (finite or infinite) path $\pi = s_0\alpha_1 s_1\alpha_2 \ldots \in Paths^M$ is $O(\pi) := O(s_0)\alpha_1 O(s_1)\alpha_2 \ldots$. Two paths $\pi, \pi' \in Paths^M$ are *observation-equivalent* if $O(\pi) = O(\pi')$.

Policies resolve the non-determinism present in MDPs (and POMDPs). Given a finite path $\hat{\pi}$, a policy determines the action to take at $last(\hat{\pi})$.

Definition 3 (Policy). *A* policy *for M is a function $\sigma : Paths^M_{\text{fin}} \to Dist(Act)$ such that for each path $\hat{\pi} \in Paths^M_{\text{fin}}$, $supp(\sigma(\hat{\pi})) \subseteq Act(last(\hat{\pi}))$.*

A policy σ is *deterministic* if $|supp(\sigma(\hat{\pi}))| = 1$ for all $\hat{\pi} \in Paths^M_{\text{fin}}$. Otherwise it is *randomised*. σ is *memoryless* if for all $\hat{\pi}, \hat{\pi}' \in Paths^M_{\text{fin}}$ we have $last(\hat{\pi}) = last(\hat{\pi}') \implies \sigma(\hat{\pi}) = \sigma(\hat{\pi}')$. σ is *observation-based* if for all $\hat{\pi}, \hat{\pi}' \in Paths^M_{\text{fin}}$ it holds that $O(\hat{\pi}) = O(\hat{\pi}') \implies \sigma(\hat{\pi}) = \sigma(\hat{\pi}')$. We denote the set of policies for M by Σ^M and the set of observation-based policies for $\mathcal{M}$ by $\Sigma^{\mathcal{M}}_{\text{obs}}$. A *finite-memory* policy (fm-policy) can be represented by a finite automaton where the current memory state and the state of the MDP determine the actions to take [4].

The *probability measure* $\mu^{\sigma,s}_M$ for paths in M under policy σ and initial state s is the probability measure of the Markov chain induced by M, σ, and s [4].

We use *reward structures* to model quantities like time, or energy consumption.

Definition 4 (Reward Structure). *A* reward structure *for M is a function $\mathbf{R}\colon S \times Act \times S \to \mathbb{R}$ such that either for all $s, s' \in S$, $\alpha \in Act$, $\mathbf{R}(s, \alpha, s') \geq 0$ or for all $s, s' \in S$, $\alpha \in Act$, $\mathbf{R}(s, \alpha, s') \leq 0$ holds. In the former case, we call $\mathbf{R}$* positive*, otherwise* negative*.*

We fix a reward structure $\mathbf{R}$ for M. The *total reward* along a path π is defined as $\mathsf{rew}_{M,\mathbf{R}}(\pi) := \sum_{i=1}^{|\pi|} \mathbf{R}(s_{i-1}, \alpha_i, s_i)$. The total reward is always well-defined—even if π is infinite—since all rewards are assumed to be either non-negative or non-positive. For an infinite path $\tilde{\pi}$ we define the *total reward* until reaching a set of goal states $G \subseteq S$ by

$$\mathsf{rew}_{M,\mathbf{R},G}(\tilde{\pi}) := \begin{cases} \mathsf{rew}_{M,\mathbf{R}}(\hat{\pi}) & \text{if } \exists i \in \mathbb{N} : \hat{\pi} = \tilde{\pi}[i] \ \wedge \ last(\hat{\pi}) \in G \ \wedge \\ & \quad \forall j < i : last(\tilde{\pi}[j]) \notin G, \\ \mathsf{rew}_{M,\mathbf{R}}(\tilde{\pi}) & \text{otherwise.} \end{cases}$$

Intuitively, $\mathsf{rew}_{M,\mathbf{R},G}(\tilde{\pi})$ accumulates reward along $\tilde{\pi}$ until the first visit of a goal state $s \in G$. If no goal state is reached, reward is accumulated along the infinite path. The *expected* total reward until reaching G for policy σ and state s is

$$\mathsf{ER}^{\sigma}_{M,\mathbf{R}}(s \models \Diamond G) := \int_{\tilde{\pi} \in Paths^{M}_{\text{inf}}} \mathsf{rew}_{M,\mathbf{R},G}(\tilde{\pi}) \cdot \mu^{\sigma,s}_{M}(d\tilde{\pi}).$$

Observation-based policies capture the notion that a decision procedure for a POMDP only accesses the observations and their history and not the entire state of the system. We are interested in reasoning about *minimal* and *maximal* values over *all* observation-based policies. For our explanations we focus on maximising (non-negative or non-positive) expected rewards. Minimisation can be achieved by negating all rewards.

Definition 5 (Maximal Expected Total Reward). *The* maximal *expected total reward until reaching G from s in POMDP $\mathcal{M}$ is*

$$\mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(s \models \Diamond G) := \sup_{\sigma \in \Sigma^{\mathcal{M}}_{\text{obs}}} \mathsf{ER}^{\sigma}_{\mathcal{M},\mathbf{R}}(s \models \Diamond G).$$

We define $\mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(\Diamond G) := \mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(s_{init} \models \Diamond G)$.

The central problem of our work, the *indefinite-horizon total reward problem*, asks the question whether the maximal expected total reward until reaching a goal exceeds a given threshold.

Problem 1. Given a POMDP $\mathcal{M}$, reward structure $\mathbf{R}$, set of goal states $G \subseteq S$, and threshold $\lambda \in \mathbb{R}$, decide whether $\mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(\Diamond G) \leq \lambda$.

Example 1. Fig. 1 shows a POMDP $\mathcal{M}$ with three states and two observations: $O(s_0) = O(s_1) =$ □ and $O(s_2) =$ ●. A reward of 1 is collected when transitioning from s_1 to s_2 via the β-action. All other rewards are zero.

The policy that always selects α at s_0 and β at s_1 maximizes the expected total reward to reach $G = \{s_2\}$ but is not observation-based. The observation-based policy that for the first $n \in \mathbb{N}$ transition steps selects α and then selects β afterwards yields an expected total reward of $1 - (1/2)^n$. With $n \to \infty$ we obtain $\mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(\Diamond\{s_2\}) = 1$.

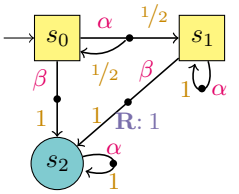


Figure 1. POMDP $\mathcal{M}$

As computing maximal expected rewards exactly in POMDPs is undecidable [34], we aim at under-approximating the actual value $\mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(\Diamond G)$. This allows us to answer our problem negatively if the computed lower bound exceeds λ.

Remark 2. Expected rewards can be used to describe *reachability probabilities* by assigning reward 1 to all transitions entering G and assigning reward 0 to all other transitions. Our approach can thus be used to obtain lower bounds on reachability probabilities in POMDPs. This also holds for almost-sure reachability (i.e. *"is the reachability probabilty one?"*), though dedicated methods like those presented in [17,16,27] are better suited for that setting.

2.2 Beliefs

The semantics of a POMDP $\mathcal{M}$ are captured by its (fully observable) *belief MDP*. The infinite state space of this MDP consists of *beliefs* [3,44]. A belief is a distribution over the states of the POMDP where each component describes the likelihood to be in a POMDP state given a history of observations. We denote the set of all beliefs for $\mathcal{M}$ by $\mathcal{B}_\mathcal{M} := \{b \in \mathit{Dist}(S) \mid \forall s, s' \in \mathit{supp}(b) : O(s) = O(s')\}$ and write $O(b) \in Z$ for the unique observation $O(s)$ of all $s \in \mathit{supp}(b)$.

The belief MDP of $\mathcal{M}$ is constructed by starting in the belief corresponding to the initial state and computing successor beliefs to unfold the MDP. Let $\mathbf{P}(s, \alpha, z) := \sum_{s' \in S} [O(s') = z] \cdot \mathbf{P}(s, \alpha, s')$ be the probability to observe $z \in Z$ after taking action α in POMDP state s. Then, the probability to observe z after taking action α in belief b is $\mathbf{P}(b, \alpha, z) := \sum_{s \in S} b(s) \cdot \mathbf{P}(s, \alpha, z)$. We refer to $[\![b|\alpha, z]\!] \in \mathcal{B}_\mathcal{M}$—the belief after taking α in b, conditioned on observing z—as the *α-z-successor of b*. If $\mathbf{P}(b, \alpha, z) > 0$, it is defined component-wise as

$$[\![b|\alpha, z]\!](s) := \frac{[O(s) = z] \cdot \sum_{s' \in S} b(s') \cdot \mathbf{P}(s', \alpha, s)}{\mathbf{P}(b, \alpha, z)}$$

for all $s \in S$. Otherwise $[\![b|\alpha, z]\!]$ is *undefined*.

Definition 6 (Belief MDP). *The* belief MDP *of $\mathcal{M}$ is the MDP $\mathit{bel}(\mathcal{M}) = \langle \mathcal{B}_\mathcal{M}, \mathit{Act}, \mathbf{P}^B, b_{\mathit{init}} \rangle$, where $\mathcal{B}_\mathcal{M}$ is the set of all beliefs in $\mathcal{M}$, Act is as for $\mathcal{M}$, $b_{\mathit{init}} := \{s_{\mathit{init}} \mapsto 1\}$ is the initial belief, and $\mathbf{P}^B \colon \mathcal{B}_\mathcal{M} \times \mathit{Act} \times \mathcal{B}_\mathcal{M} \to [0, 1]$ is the belief transition function with*

$$\mathbf{P}^B(b, \alpha, b') := \begin{cases} \mathbf{P}(b, \alpha, z) & \text{if } b' = [\![b|\alpha, z]\!], \\ 0 & \text{otherwise.} \end{cases}$$

We lift a POMDP reward structure $\mathbf{R}$ to the belief MDP [25].

Definition 7 (Belief Reward Structure). *For beliefs $b, b' \in \mathcal{B}_\mathcal{M}$ and action $\alpha \in \mathit{Act}$, the* belief reward structure $\mathbf{R}^B$ *based on $\mathbf{R}$ associated with $\mathit{bel}(\mathcal{M})$ is given by*

$$\mathbf{R}^B(b, \alpha, b') := \frac{\sum_{s \in S} b(s) \cdot \sum_{s' \in S} [O(s') = O(b')] \cdot \mathbf{R}(s, \alpha, s') \cdot \mathbf{P}(s, \alpha, s')}{\mathbf{P}(b, \alpha, O(b'))}.$$

Given a set of goal states $G \subseteq S$, we assume—for simplicity—that there is a set of observations $Z' \subseteq Z$ such that $s \in G$ iff $O(s) \in Z'$. This assumption can always be ensured by transforming the POMDP $\mathcal{M}$. See the full technical report [10] for details. The set of *goal beliefs* for G is given by $G_\mathcal{B} := \{b \in \mathcal{B}_\mathcal{M} \mid \mathit{supp}(b) \subseteq G\}$.

We now lift the computation of expected rewards to the belief level. Based on the well-known Bellman equations [5], the belief MDP induces a function that maps every belief to the expected total reward accumulated from that belief.

Definition 8 (POMDP Value Function). *For $b \in \mathcal{B}_\mathcal{M}$, the* n-step value function $V_n : \mathcal{B}_\mathcal{M} \to \mathbb{R}$ *of $\mathcal{M}$ is defined recursively as $V_0(b) := 0$ and*

$$V_n(b) := [b \notin G_\mathcal{B}] \cdot \max_{\alpha \in \mathit{Act}} \sum_{b' \in \mathit{post}^{\mathit{bel}(\mathcal{M})}(b, \alpha)} \mathbf{P}^B(b, \alpha, b') \cdot \left(\mathbf{R}^B(b, \alpha, b') + V_{n-1}(b')\right).$$

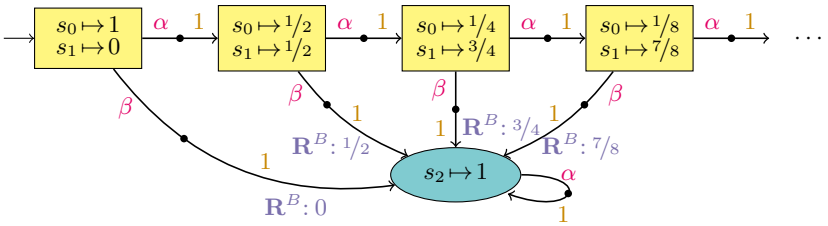


Figure 2. Belief MDP $bel(\mathcal{M})$ of POMDP $\mathcal{M}$ from Fig. 1

The (optimal) value function $V^* : \mathcal{B}_\mathcal{M} \to \mathbb{R}^\infty$ *is given by* $V^*(b) := \lim_{n\to\infty} V_n(b)$.

The n-step value function is piecewise linear and convex [44]. Thus, the optimal value function can be approximated arbitrarily close by a piecewise linear convex function [47]. The value function yields expected total rewards in $\mathcal{M}$ and $bel(\mathcal{M})$:

$$\mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(s \models \Diamond G) \;=\; \mathsf{ER}^{\max}_{bel(\mathcal{M}),\mathbf{R}^B}(\{s \mapsto 1\} \models \Diamond G_\mathcal{B}) \;=\; V^*(\{s \mapsto 1\}).$$

Example 2. Fig. 2 shows a fragment of the belief MDP of the POMDP from Fig. 1. Observe $\mathsf{ER}^{\max}_{bel(\mathcal{M}),\mathbf{R}^B}(\Diamond\, \{s_2 \mapsto 1\}) = 1$.

We reformulate our problem statement to focus on the belief MDP.

> *Problem 2 (equivalent to Problem 1).* For a POMDP $\mathcal{M}$, reward structure $\mathbf{R}$, goal states $G \subseteq S$, and threshold $\lambda \in \mathbb{R}$, decide whether $V^*(\{s_{init} \mapsto 1\}) \leq \lambda$.

As the belief MDP is fully observable, standard results for MDPs apply. However, an exhaustive analysis of $bel(\mathcal{M})$ is intractable since the belief MDP is—in general—infinitely large[2].

3 Finite Exploration Under-Approximation

Instead of approximating values directly on the POMDP, we consider approximations of the corresponding belief MDP. The basic idea is to construct a finite abstraction of the belief MDP by unfolding parts of it and approximate values at beliefs where we decide not to explore. In the resulting finite MDP, under-approximative expected reward values can be computed by standard model checking techniques. We present two approaches for abstraction: *belief cut-offs* and *belief clipping*. We incorporate those techniques into an algorithmic framework that yields arbitrarily tight under-approximations.

The technical report [10] contains formal proofs of our claims.

[2] The set of all beliefs—i.e. the state space of $bel(\mathcal{M})$—is uncountable. The reachable fragment is countable, though, since each belief has at most $|Z|$ many successors.

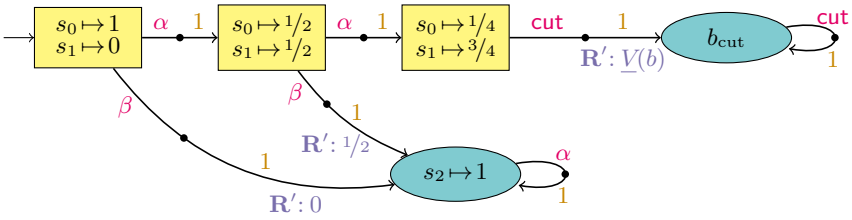


Figure 3. Applying belief cut-offs to the belief MDP from Fig. 2

3.1 Belief Cut-Offs

The general idea of *belief cut-offs* is to stop exploring the belief MDP at certain beliefs—the *cut-off beliefs*—and assume that a goal state is immediately reached while sub-optimal reward is collected. Similar techniques have been discussed in the context of fully observable MDPs and other model types [11,26,49,2]. Our work adapts the idea of cut-offs for POMDP *over-approximations* described in [8] to under-approximations. The main idea of belief cut-offs shares similarities with the *SARSOP* [30] and *Goal-HSVI* [24] approaches. While they apply cut-offs on the level of the computation tree, our approach directly manipulates the belief MDP to yield a finite model.

Let $\underline{V} : \mathcal{B}_{\mathcal{M}} \to \mathbb{R}^{\infty}$ with $\underline{V}(b) \leq V^*(b)$ for all $b \in \mathcal{B}_{\mathcal{M}}$. We call $\underline{V}$ an *under-approximative value function* and $\underline{V}(b)$ the *cut-off value* of b. In each of the cut-off beliefs b, instead of adding the regular transitions to its successors, we add a transition with probability 1 to a dedicated goal state b_{cut}. In the modified reward structure $\mathbf{R}'$, this *cut-off transition* is assigned a reward[3] of $\underline{V}(b)$, causing the value for a cut-off belief b in the modified MDP to coincide with $\underline{V}(b)$. Hence, the exact value of the cut-off belief—and thus the value of all other explored beliefs—is under-approximated.

Example 3. Fig. 3 shows the resulting *finite* MDP obtained when considering the belief MDP from Fig. 2 with single cut-off belief $b = \{s_0 \mapsto 1/4,\ s_1 \mapsto 3/4\}$.

Computing cut-off values. The question of finding a suitable under-approximative value function $\underline{V}$ is central to the cut-off approach. For an effective approximation, such a function should be easy to compute while still providing values close to the optimum. If we assume a positive reward structure, the constant value 0 is always a valid under-approximation. A more sophisticated approach is to compute suboptimal expected reward values for the states of the POMDP using *some* arbitrary, fixed observation-based policy $\sigma \in \Sigma^{\mathcal{M}}_{\text{obs}}$. Let $U^{\sigma} : S \to \mathbb{R}^{\infty}$ such that for all $s \in S$, $U^{\sigma}(s) = \mathsf{ER}^{\sigma}_{\mathcal{M},\mathbf{R}}(s \models \Diamond G)$. Then, we define the function $\mathfrak{U}^{\sigma} : \mathcal{B}_{\mathcal{M}} \to \mathbb{R}^{\infty}$ as $\mathfrak{U}^{\sigma}(b) := \sum_{s \in supp(b)} b(s) \cdot U^{\sigma}(s)$.

[3] We slightly deviate from Def. 4 by allowing transition rewards to be $-\infty$ or $+\infty$. Alternatively, we could introduce new sink states with a non-zero self-loop reward.

Lemma 1. *$\mathfrak{U}^\sigma$ is an under-approximative value function, i.e. for all $b \in \mathcal{B}_\mathcal{M}$:*

$$\mathfrak{U}^\sigma(b) := \sum_{s \in supp(b)} b(s) \cdot U^\sigma(s) \leq V^*(b).$$

Thus, finding a suitable under-approximative value function reduces to finding "good" policies for $\mathcal{M}$, e.g. by using randomly guessed fm-policies, machine learning methods [13], or a transformation to a parametric model [28].

3.2 Belief Clipping

The cut-off approach provides a universal way to construct an MDP which under-approximates the expected total reward value for a given POMDP. The quality of the approximation, however, is highly dependent on the under-approximative value function used. Furthermore, regions where the belief MDP slowly converges towards a belief may pose problems in practice.

As a potential remedy for these problems, we propose a different concept called *belief clipping*. Intuitively, the procedure shifts some of the probability mass of a belief b in order to transform b to another belief $\tilde{b}$. We then connect b to $\tilde{b}$ in a way that the accuracy of our approximation of the value $V^*(b)$ depends only on the approximation of $V^*(\tilde{b})$ and the so-called *clipping value*—some notion of distance between b and $\tilde{b}$ that we discuss below. We can thus focus on exploring the successors of $\tilde{b}$ to obtain good approximations for both beliefs b and $\tilde{b}$.

Definition 9 (Belief Clip). *For $b \in \mathcal{B}_\mathcal{M}$, we call $\mu\colon supp(b) \to [0,1]$ a* belief clip *if $\forall s \in supp(b)\colon \mu(s) \leq b(s)$ and $\sum(\mu) := \sum_{s \in supp(b)} \mu(s) < 1$. The belief $(b \ominus \mu) \in \mathcal{B}_M$ induced by μ is defined by*

$$\forall s \in supp(b)\colon \ (b \ominus \mu)(s) \ := \ \frac{b(s) - \mu(s)}{1 - \sum(\mu)}.$$

Intuitively, a belief clip μ for b describes for each $s \in supp(b)$ the probability mass that is removed ("clipped away") from $b(s)$. The induced belief is obtained when normalising the resulting values so that they sum up to one.

Example 4. For belief $b = \{s_0 \mapsto 1/4, s_1 \mapsto 3/4\}$, consider the two belief clips $\mu_1 = \{s_0 \mapsto 1/4, s_1 \mapsto 1/4\}$ and $\mu_2 = \{s_0 \mapsto 1/4, s_1 \mapsto 0\}$. Both induce the same belief: $(b \ominus \mu_1) = (b \ominus \mu_2) = \{s_0 \mapsto 0, s_1 \mapsto 1\}$.

We have $supp((b \ominus \mu)) \subseteq supp(b)$, which also implies $O((b \ominus \mu)) = O(b)$. Given some candidate belief $\tilde{b}$, consider the set of inducing belief clips:

$$\mathcal{C}(b, \tilde{b}) := \left\{ \mu\colon supp(b) \to [0,1] \mid \mu \text{ is a belief clip for } b \text{ with } \tilde{b} = (b \ominus \mu) \right\}.$$

Belief $\tilde{b}$ is called an adequate clipping candidate for b iff $\mathcal{C}(b, \tilde{b}) \neq \emptyset$.

Definition 10 (Clipping Value). *For $b \in \mathcal{B}_\mathcal{M}$ and adequate clipping candidate $\tilde{b}$, the* clipping value *is $\Delta_{b \to \tilde{b}} := \sum(\delta_{b \to \tilde{b}})$, where $\delta_{b \to \tilde{b}} := \arg\min_{\mu \in \mathcal{C}(b,\tilde{b})} \sum(\mu)$. The values $\delta_{b \to \tilde{b}}(s)$ for $s \in supp(b)$ are the* state clipping values.

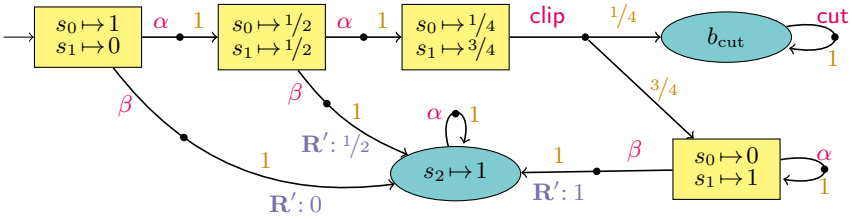


Figure 4. Applying belief clipping to the belief MDP from Fig. 2

Given a belief b and an adequate clipping candidate $\tilde{b}$, we outline how the notion of belief clipping is used to obtain valid under-approximations. We assume $b \neq \tilde{b}$, implying $0 < \Delta_{b \to \tilde{b}} < 1$. Instead of exploring all successors of b in $bel(\mathcal{M})$, the approach is to add a transition from b to $\tilde{b}$. The newly added transition has probability $1 - \Delta_{b \to \tilde{b}}$ and gets assigned a reward of 0. The remaining probability mass (i.e. $\Delta_{b \to \tilde{b}}$) leads to a designated goal state b_{cut}. To guarantee that—in general—the clipping procedure yields a valid under-approximation, we need to add a corrective reward value to the transition from b to b_{cut}. Let $\mathfrak{L} : S \to \mathbb{R}^{\infty}$ which maps each POMDP state to its *minimum* expected reward in the underlying, fully observable MDP M of $\mathcal{M}$[4], i.e. $\mathfrak{L}(s) = \mathsf{ER}^{\min}_{M,\mathbf{R}}(s \models \Diamond G)$. This function soundly under-approximates the state values which can be achieved by *any* observation-based policy. It can be generated using standard MDP analysis. Given state clipping values $\delta_{b \to \tilde{b}}(s)$ for $s \in supp(b)$, the reward for the transition from b to b_{cut} is $\sum_{s \in supp(b)} (\delta_{b \to \tilde{b}}(s) / \Delta_{b \to \tilde{b}}) \cdot \mathfrak{L}(s)$.

Example 5. For the belief MDP from Fig. 2, belief $b = \{s_0 \mapsto 1/4,\ s_1 \mapsto 3/4\}$, and clipping candidate $\tilde{b} = \{s_0 \mapsto 0,\ s_1 \mapsto 1\}$ we get $\Delta_{b \to \tilde{b}} = 1/4$, as $\delta_{b \to \tilde{b}} = \mu_2 = \{s_0 \mapsto 1/4,\ s_1 \mapsto 0\}$ with the belief clip μ_2 as in Example 4. Furthermore, $\mathfrak{L}(s_0) = 0$. The resulting MDP following our construction above is given in Fig. 4.

The following lemma shows that the construction yields an under-approximation.

Lemma 2. $(1 - \Delta_{b \to \tilde{b}}) \cdot V^*(\tilde{b}) \ + \ \Delta_{b \to \tilde{b}} \cdot \sum_{s \in supp(b)} \frac{\delta_{b \to \tilde{b}}(s)}{\Delta_{b \to \tilde{b}}} \cdot \mathfrak{L}(s) \ \leq \ V^*(b).$

Proof (sketch). To gain some intuition, consider the special case, where $\Delta_{b \to \tilde{b}} = \delta_{b \to \tilde{b}}(s) = b(s)$ for some $s \in supp(b)$. The clipping candidate $\tilde{b}$ can be interpreted as the conditional probability distribution arising from distribution b given that s is *not* the current state. The value $V^*(b)$ can be split into the sum of (i) the probability that s is *not* the current state times the reward accumulated from belief $\tilde{b}$ and (ii) the probability that s *is* the current state times the reward accumulated from s, i.e. from the belief $\{s \mapsto 1\}$. However, for the two summands

[4] When rewards are negative, we might have $\mathfrak{L}(s) = -\infty$ for many $s \in S \setminus G$ in which case the applicability of the clipping approach is very limited.

we must consider a policy that does not distinguish between the beliefs b, $\tilde{b}$, and $\{s \mapsto 1\}$ as well as their observation-equivalent successors. In other words, the same sequence of actions must be executed when the same observations are made.

We consider such a policy that in addition is optimal at $\tilde{b}$, i.e. the reward accumulated from $\tilde{b}$ is equal to $V^*(\tilde{b})$. For the reward accumulated from $\{s \mapsto 1\}$, $\mathfrak{L}(s)$ provides a lower bound. Hence, $(1 - b(s)) \cdot V^*(b) + b(s) \cdot \mathfrak{L}(s)$ is a lower bound for the reward accumulated from b. A formal proof is given in [10]. □

To find a suitable clipping candidate for a given belief b, we consider a finite *candidate set* $\mathfrak{B} \subseteq \mathcal{B}_{\mathcal{M}}$ consisting of beliefs with observation $O(b)$. These beliefs do not need to be reachable in the belief MDP. The set can be constructed, e.g. by taking already explored beliefs or by using a fixed, discretised set of beliefs.

We are interested in minimising the clipping value $\Delta_{b \to b'}$ over all candidate beliefs $b' \in \mathfrak{B}$. A naive approach is to explicitly compute all clipping values for all candidates. We are using *mixed-integer linear programming (MILP)* [41] instead. An MILP is a system of linear inequalities (*constraints*) and a linear *objective function* considering real-valued and integer variables. A *feasible solution* of the MILP is a variable assignment that satisfies all constraints. An *optimal solution* is a feasible solution that minimises the objective function.

Definition 11 (Belief Clipping MILP). *The belief clipping MILP for belief $b \in \mathcal{B}_{\mathcal{M}}$ and finite set of candidates $\mathfrak{B} \subseteq \{b' \in \mathcal{B}_{\mathcal{M}} \mid O(b') = O(b)\}$ is given by:*

minimise Δ such that:

$$\sum_{b' \in \mathfrak{B}} a_{b'} = 1 \qquad \triangleright \textit{Select exactly one candidate } b' \tag{1}$$

$$\forall b' \in \mathfrak{B}\colon \quad a_{b'} \in \{0, 1\} \tag{2}$$

$$\sum_{s \in \mathit{supp}(b)} \delta_s = \Delta \qquad \triangleright \textit{Compute clipping value for selected } b' \tag{3}$$

$$\forall s \in \mathit{supp}(b)\colon \quad \delta_s \in [0,\, b(s)] \tag{4}$$

$$\forall b' \in \mathfrak{B}\colon \quad \delta_s \geq b(s) - (1 - \Delta) \cdot b'(s) - (1 - a_{b'}) \tag{5}$$

The MILP consists of $\mathcal{O}(|\mathit{supp}(b)| + |\mathfrak{B}|)$ variables and $\mathcal{O}(|\mathit{supp}(b)| \cdot |\mathfrak{B}|)$ constraints. For $b' \in \mathfrak{B}$, the binary variable $a_{b'}$ indicates whether b' has been chosen as the clipping candidate. Moreover, we have variables δ_s for $s \in \mathit{supp}(b)$ and a variable Δ to represent the (state) clipping values for b and the chosen candidate b'. Constraints 1 and 2 enforce that exactly one of the $a_{b'}$ variables is one, i.e. exactly one belief is chosen. Constraint 3 forces Δ to be the sum of all state clipping values. δ_s variables get a value between zero and $b(s)$ (Constraint 4). Constraint 5 only affects δ_s if the corresponding belief is chosen. Otherwise, $a_{b'}$ is set to 0 and the value on the right-hand side becomes negative. If a belief b' is chosen, the minimisation forces Constraint 5 to hold with equality as the right-hand side is greater or equal to 0. Assuming Δ is set to a value below 1, we obtain a valid clipping values as

$$\forall s \in \mathit{supp}(b)\colon \quad \delta_s = b(s) - (1 - \Delta) \cdot b'(s) \quad \iff \quad b'(s) = \frac{b(s) - \delta_s}{1 - \Delta}.$$

Input : POMDP $\mathcal{M} = \langle M, Z, O\rangle$ with $M = \langle S, Act, \mathbf{P}, s_{init}\rangle$, reward structure $\mathbf{R}$, goal states $G \subseteq S$, under-approx. value function $\underline{V}$, function $\mathfrak{L} : S \to \mathbb{R}^{\infty}$ with $\mathfrak{L}(s) = \mathsf{ER}^{\min}_{M,\mathbf{R}}(s \models \Diamond G)$

Output : Clipping belief MDP $\mathcal{K}_{\mathcal{M}}$ and reward structure $\mathbf{R}^{\mathcal{K}}$

1 $S^{\mathcal{K}} \leftarrow \{b_{init}, b_{\text{cut}}\}$ with $b_{init} = \{s_{init} \mapsto 1\}$ and a new belief state b_{cut}
2 $\mathbf{P}^{\mathcal{K}}(b_{\text{cut}}, \mathsf{cut}, b_{\text{cut}}) \leftarrow 1$, $\mathbf{R}^{\mathcal{K}}(b_{\text{cut}}, \mathsf{cut}, b_{\text{cut}}) \leftarrow 0$ *// add self-loop*
3 $Q \leftarrow \{b_{init}\}$ *// initialize exploration set*
4 **while** $Q \neq \emptyset$ **do**
5 $b \leftarrow$ `chooseBelief`(Q), $Q \leftarrow Q \setminus \{b\}$ *// pop next belief to explore from Q*
6 **if** $supp(b) \subseteq G$ **then** $\mathbf{P}^{\mathcal{K}}(b, \mathsf{goal}, b) \leftarrow 1$, $\mathbf{R}^{\mathcal{K}}(b, \mathsf{goal}, b) \leftarrow 0$ *// add self-loop*
7 **else if** `exploreBelief`(b) **then** *// expand b*
8 **foreach** $\alpha \in Act(b)$ **do** *// Using bel($\mathcal{M}$) and $\mathbf{R}^B$ as in Defs. 6 and 7*
9 **foreach** $b' \in post^{bel(\mathcal{M})}(b, \alpha)$ **do**
10 $\mathbf{P}^{\mathcal{K}}(b, \alpha, b') \leftarrow \mathbf{P}^{B}(b, \alpha, b')$, $\mathbf{R}^{\mathcal{K}}(b, \alpha, b') \leftarrow \mathbf{R}^{B}(b, \alpha, b')$
11 **if** $b' \notin S^{\mathcal{K}}$ **then** $S^{\mathcal{K}} \leftarrow S^{\mathcal{K}} \cup \{b'\}$, $Q \leftarrow Q \cup \{b'\}$
12 **else** *// apply cut-off and clipping to b*
13 $\mathbf{P}^{\mathcal{K}}(b, \mathsf{cut}, b_{\text{cut}}) \leftarrow 1$, $\mathbf{R}^{\mathcal{K}}(b, \mathsf{cut}, b_{\text{cut}}) \leftarrow \underline{V}(b)$ *// add cut-off transition*
14 choose a finite set $\mathfrak{B} \subseteq \mathcal{B}_{\mathcal{M}}$ of clipping candidates for b
15 $\tilde{b}, \Delta_{b\to\tilde{b}}, \delta_{b\to\tilde{b}} \leftarrow$ `solveClippingMILP`$(b, \mathfrak{B})$
16 **if** $\tilde{b} \neq b$ and $\tilde{b}$ is adequate **then** *// Clip b using $\tilde{b}$*
17 $\mathbf{P}^{\mathcal{K}}(b, \mathsf{clip}, \tilde{b}) \leftarrow (1 - \Delta_{b\to\tilde{b}})$, $\mathbf{P}^{\mathcal{K}}(b, \mathsf{clip}, b_{\text{cut}}) \leftarrow \Delta_{b\to\tilde{b}}$
18 $\mathbf{R}^{\mathcal{K}}(b, \mathsf{clip}, \tilde{b}) \leftarrow 0$, $\mathbf{R}^{\mathcal{K}}(b, \mathsf{clip}, b_{\text{cut}}) \leftarrow \sum_{s \in supp(b)} \frac{\delta_{b\to\tilde{b}}(s)}{\Delta_{b\to\tilde{b}}} \cdot \mathfrak{L}(s)$
19 **if** $\tilde{b} \notin S^{\mathcal{K}}$ **then** $S^{\mathcal{K}} \leftarrow S^{\mathcal{K}} \cup \{\tilde{b}\}$, $Q \leftarrow Q \cup \{\tilde{b}\}$
20 **return** $\mathcal{K}_{\mathcal{M}} = \left\langle S^{\mathcal{K}}, Act \uplus \{\mathsf{goal}, \mathsf{cut}, \mathsf{clip}\}, \mathbf{P}^{\mathcal{K}}, b_{init}\right\rangle$ and $\mathbf{R}^{\mathcal{K}}$

Algorithm 1: Belief exploration algorithm with cut-offs and clipping

A trivial solution of the MILP is always obtained by setting $a_{b'}$ and Δ to 1 and δ_s to $b(s)$ for all s and an arbitrary $b' \in \mathfrak{B}$. This corresponds to an invalid belief clip. However, as we minimise the value for Δ, we can conclude that *no* belief in the candidate set is adequate for clipping if Δ is 1 in an optimal solution.

Theorem 1. *An optimal solution to the belief clipping MILP for belief b and candidate set $\mathfrak{B}$ sets $a_{\tilde{b}}$ to 1 and Δ to a value below 1 iff $\tilde{b} \in \mathfrak{B}$ is an adequate clipping candidate for b with minimal clipping value.*

3.3 Algorithm

We incorporate belief cut-offs and belief clipping into an algorithmic framework outlined in Algorithm 1. As input, the algorithm takes an instance of Problems 1 and 2, i.e. a POMDP $\mathcal{M}$ with reward structure $\mathbf{R}$ and goal states G. In addition, the algorithm considers an under-approximative value function $\underline{V}$ (Sect. 3.1) and a function $\mathfrak{L}$ for the computation of corrective reward values (Sect. 3.2).

Lines 1 and 2 initialise the state set $S^{\mathcal{K}}$ of the under-approximative MDP $\mathcal{K}_{\mathcal{M}}$ with the initial belief b_{init} and the designated goal state b_{cut} which has only one

transition to itself with reward 0. Furthermore, we initialise the *exploration set* Q by adding b_{init} (Line 3). During the computation, Q is used to keep track of all beliefs we still need to process. We then execute the exploration loop (Lines 4 to 19) until Q becomes empty. In each exploration step, a belief b is selected[5] and removed from Q. There are three cases for the currently processed belief b.

If $supp(b) \subseteq G$, i.e. b is a goal belief, we add a self-loop with reward 0 to b and continue with the next belief (Line 6). b is not expanded as successors of goal beliefs will not influence the result of the computation.

If b is not a goal belief, we use a heuristic function[6] `exploreBelief` to decide if b is expanded in Line 7. Lines 8 to 11 outline the expansion step. The transitions from b to its successor beliefs and the corresponding rewards as in the original belief MDP (see Sect. 2.2) are added. Furthermore, the successor beliefs that have not been encountered before are added to the set of states $S^{\mathcal{K}}$ and the exploration set Q.

If b is *not* expanded, we apply the cut-off approach *and* the clipping approach to b in Lines 12 to 19. In Line 13 we add a cut-off transition from b to b_{cut} with a new action cut. We use the given under-approximative value function $\underline{V}$ to compute the cut-off reward. Towards the clipping approach, a set of candidate beliefs is chosen and the belief clipping MILP for b and the candidate set is constructed as described in Def. 11 (Lines 14 and 15). If an adequate candidate $\tilde{b}$ with clipping values $\Delta_{b\to\tilde{b}}$ and $\delta_{b\to\tilde{b}}(s)$ for $s \in supp(b)$ has been found, we add the transitions from b to b_{cut} and to $\tilde{b}$ using a new action clip and probabilities $\Delta_{b\to\tilde{b}}$ and $1 - \Delta_{b\to\tilde{b}}$, respectively. Furthermore, we equip the transitions with reward values as described in Sect. 3.2 using the given function $\mathfrak{L}$ (Lines 16 to 18). If the clipping candidate $\tilde{b}$ has not been encountered before, we add it to the state space of the MDP and to the exploration set in Line 19.

The result of the algorithm is an MDP $\mathcal{K}_{\mathcal{M}}$ with reward structure $\mathbf{R}^{\mathcal{K}}$. The set of states $S^{\mathcal{K}}$ of $\mathcal{K}_{\mathcal{M}}$ contains all encountered beliefs. To guarantee termination of the algorithm, the decision heuristic `exploreBelief` has to stop exploring further beliefs at some point. Moreover, the handling of clipping candidates in Line 19 should not add new beliefs to Q infinitely often. We therefore fix a finite set of candidate beliefs $\mathcal{B}^{\#} \subseteq \mathcal{B}_{\mathcal{M}}$ and make sure that the candidate sets $\mathfrak{B}$ in Line 14 satisfy $(\mathfrak{B} \setminus S^{\mathcal{K}}) \subseteq \mathcal{B}^{\#}$. To ensure a certain progress in the exploration "clip-cycles"—i.e. paths of the form $b_1\, \mathsf{clip} \ldots \mathsf{clip}\, b_n\, \mathsf{clip}\, b_1$—are avoided in $\mathcal{K}_{\mathcal{M}}$. This can be done, e.g. by always expanding the candidate beliefs $b \in \mathcal{B}^{\#}$.

Expected total rewards until reaching the extended set of goal beliefs $G_{\text{cut}} := G_{\mathcal{B}} \cup \{b_{\text{cut}}\}$ in $\mathcal{K}_{\mathcal{M}}$ under-approximate the values in the belief MDP:

Theorem 2. *For all beliefs $b \in S^{\mathcal{K}} \setminus \{b_{cut}\}$ it holds that*

$$\mathsf{ER}^{\max}_{\mathcal{K}_{\mathcal{M}},\mathbf{R}^{\mathcal{K}}}(b \models \Diamond G_{cut}) \leq V^*(b) = \mathsf{ER}^{\max}_{bel(\mathcal{M}),\mathbf{R}^{B}}(b \models \Diamond G_{\mathcal{B}}).$$

Corollary 1. $\mathsf{ER}^{\max}_{\mathcal{K}_{\mathcal{M}},\mathbf{R}^{\mathcal{K}}}(\Diamond G_{cut}) \leq \mathsf{ER}^{\max}_{\mathcal{M},\mathbf{R}}(\Diamond G)$.

[5] For example, Q can be implemented as a FIFO queue.

[6] The decision can be made for example by considering the size of the already explored state space such that the expansion is stopped if a size threshold has been reached. More involved decision heuristics are subject to further research.

Table 1. Results for benchmark POMDPs with maximisation objective

Benchmark Model	ϕ	Data $S/Act/Z$	Prism	Storm					
				Cut-Off Only	Cut-Off + Clipping η=2	η=3	η=4	η=6	Over-Approx.
Drone 4-1	$P_{\max}$	1226 2954 384	TO / MO	≥ 0.79 < 1s $3{\cdot}10^4$	≥ 0.79 1360s $3{\cdot}10^4$	TO	TO	TO	≤ 0.94
Drone 4-2	$P_{\max}$	1226 2954 761	TO / MO	≥ 0.86 < 1s $2{\cdot}10^4$	≥ 0.91 249s $2{\cdot}10^4$	≥ 0.92 1902s $2{\cdot}10^4$	TO	TO	≤ 0.97
Grid-av 4-0	$P_{\max}$	17 59 4	[**0.21**, 1.0] 5.14s η =6	≥ 0.86 < 1s 238	≥ 0.93 < 1s 312	≥ 0.93 1.77s 472	≥ 0.93 3.63s 663	≥ 0.93 13.9s 1300	≤ 0.98
Grid-av 4-0.1	$P_{\max}$	17 59 4	[**0.21**, 1.0] 1.47s η =3	≥ 0.82 < 1s 238	≥ 0.85 26.1s 317	≥ 0.82 198s 461	≥ 0.85 1913s 759	TO	≤ 0.99
Netw-p 2-8-20	$R_{\max}$	$2{\cdot}10^4$ $3{\cdot}10^4$ 4909	[**557**, 557] 2355s η =10	≥ 537 2.3s $8{\cdot}10^4$	≥ 537 98.5s $1{\cdot}10^5$	≥ 537 320s $1{\cdot}10^5$	≥ 537 651s $1{\cdot}10^5$	≥ 537 2368s $1{\cdot}10^5$	≤ 558
Netw-p 3-8-20	$R_{\max}$	$2{\cdot}10^5$ $3{\cdot}10^5$ $2{\cdot}10^4$	TO / MO	≥ 769 290s $1{\cdot}10^6$	≥ 769 6640s $1{\cdot}10^6$	TO	TO	TO	≤ 819
Refuel 06	$P_{\max}$	208 565 50	[**0.67**, 0.72] 4625s η =3	≥ 0.67 < 1s 4576	≥ 0.67 5.89s 4834	≥ 0.67 24.3s 5204	≥ 0.67 92s 5603	≥ 0.67 2076s 6135	≤ 0.69
Refuel 08	$P_{\max}$	470 1431 66	TO / MO	≥ 0.45 < 1s $2{\cdot}10^4$	≥ 0.45 839s $2{\cdot}10^4$	TO	TO	TO	≤ 0.51

4 Experimental Evaluation

Implementation details. We integrated Algorithm 1 in the probabilistic model checker Storm [23] as an extension of the POMDP verification framework described in [8]. Inputs are a POMDP—encoded either explicitly or using an extension of the Prism language [37]—and a property specification. Internally, POMDPs and MDPs are represented using sparse matrices. The implementation supports minimisation[7] and maximisation of reachability probabilities, reach-avoid probabilities (i.e. the probability to avoid a set of bad state until a set of goal states is reached), and expected total rewards. In a preprocessing step, functions $\underline{V}$ and $\mathfrak{L}$ as considered in Algorithm 1 are generated. For $\underline{V}$, we consider the function $\mathfrak{U}^\sigma$ as in Lemma 1, where σ is a memoryless observation-based policy given by a heuristic[8]. For the function $\mathfrak{L}$, we apply standard MDP analysis on the underlying MDP. When exploring the abstraction MDP $\mathcal{K}_{\mathcal{M}}$, our heuristic expands a belief iff $|S^{\mathcal{K}}| \leq |S| \cdot \max_{z \in z} |O^{-1}(z)|$, where $|S^{\mathcal{K}}|$ is the number of already explored beliefs and $|O^{-1}(z)|$ is the number of POMDP states with observation z. Belief clipping can either be disabled entirely, or we consider candidate sets $\mathfrak{B} \subseteq \mathcal{B}^{\#}_{\eta}$, where $\mathcal{B}^{\#}_{\eta} := \{b \in \mathcal{B} \mid \forall s \in S : b(s) \in \{i/\eta \mid i \in \mathbb{N}, 0 \leq i \leq \eta\}\}$ forms a finite, regular *grid* of beliefs with resolution $\eta \in \mathbb{N} \setminus \{0\}$. Grid beliefs $b \in \mathcal{B}^{\#}_{\eta}$ are always expanded.

[7] For minimisation, the under-approximation yields *upper bounds*.

[8] The heuristic uses optimal values obtained on the fully observable underlying MDP.

Table 2. Results for benchmark POMDPs with minimisation objective

Benchmark Model	ϕ	Data $S/Act/Z$	Prism	Storm Cut-Off Only	Storm Cut-Off + Clipping η=2	η=3	η=4	η=6	Storm Over-Approx.
Grid 4-0.1	$R_{\min}$	17	$[4.52, \mathbf{4.7}]$	≤ 4.78	≤ 4.78	≤ 4.78	≤ 4.78		≥ 4.52
		62	649s	< 1s	15.6s	148s	1940s	TO	
		3	$\eta = 10$	258	255	255	255		
Grid 4-0.3	$R_{\min}$	17	$[6.12, \mathbf{6.31}]$	≤ 6.56	≤ 6.56	≤ 6.56	≤ 6.56		≥ 6.08
		62	1077s	< 1s	15.8s	148s	1983s	TO	
		3	$\eta = 10$	255	256	256	256		
Maze2 0.1	$R_{\min}$	15	$[6.32, \mathbf{6.32}]$	≤ 6.34	≤ 6.34	≤ 6.34	≤ 6.34	≤ 6.34	≥ 6.32
		54	1.79s	< 1s	< 1s	< 1s	< 1s	2.02s	
		8	$\eta = 10$	91	90	90	90	90	
Netw 2-8-20	$R_{\min}$	4589	$[3.17, \mathbf{3.2}]$	≤ 6.56	≤ 6.56	≤ 6.56	≤ 6.56	≤ 6.56	≥ 3.14
		6973	211s	< 1s	5.31s	17.2s	42.3s	167s	
		1173	$\eta = 10$	$2{\cdot}10^4$	$2{\cdot}10^4$	$2{\cdot}10^4$	$3{\cdot}10^4$	$3{\cdot}10^4$	
Netw 3-8-20	$R_{\min}$	$2{\cdot}10^4$	$[5.61, \mathbf{6.79}]$	≤ 11.9	≤ 11.9	≤ 11.9	≤ 11.9		≥ 6.13
		$3{\cdot}10^4$	7133s	3.51s	214s	1372s	4910s	TO	
		2205	$\eta = 6$	$1{\cdot}10^5$	$2{\cdot}10^5$	$2{\cdot}10^5$	$2{\cdot}10^5$		
Rocks 12	$R_{\min}$	6553		≤ 38	≤ 38	≤ 38	≤ 20	≤ 21	≥ 20
		$3{\cdot}10^4$	TO / MO	1.39s	61.1s	138s	230s	532s	
		1645		$3{\cdot}10^4$	$3{\cdot}10^4$	$3{\cdot}10^4$	$5{\cdot}10^4$	$6{\cdot}10^4$	
Rocks 16	$R_{\min}$	$1{\cdot}10^4$		≤ 44	≤ 44	≤ 44	≤ 26	≤ 27	≥ 26
		$5{\cdot}10^4$	TO / MO	3.85s	114s	230s	399s	1062s	
		2761		$4{\cdot}10^4$	$4{\cdot}10^4$	$4{\cdot}10^4$	$6{\cdot}10^4$	$1{\cdot}10^5$	

Furthermore, we exclude clipping candidates $\tilde{b}$ with $\delta_{b \to \tilde{b}}(s) > 0$ for s with $\mathfrak{L}(s) = -\infty$; clipping with such candidates is not useful as it induces a value of $-\infty$. Expected total rewards on fully observable MDPs are computed using *Sound Value Iteration* [39] with relative precision 10^{-6}. MILPs are solved using Gurobi [21].

Set-up. We evaluate our under-approximation approach with cut-offs only and with enabled belief clipping procedure using grid resolutions $\eta = 2, 3, 4, 6$. We consider the same POMDP benchmarks[9] as in [37,8]. The POMDPs are scalable versions of case studies stemming from various application domains. To establish an external baseline, we compare with the approach of [37] implemented in Prism [31]. Prism generates an under-approximation based on an optimal policy for an over-approximative MDP which—in contrast to Storm—means that always both, under- and over-approximations, have to be computed. We ran Prism with resolutions $\eta = 2, 3, 4, 6, 8, 10$ and report on the *best* approximation obtained. To provide a further reference for the tightness of our under-approximation, we compute over-approximative bounds as in [8] using the implementation in Storm with a resolution of $\eta = 8$. All experiments were run on an Intel® Xeon® Platinum 8160 CPU using 4 threads[10], 64GB RAM and a time limit of 2 hours.

Results. Tables 1 and 2 show our results for maximising and minimising properties, respectively. The first columns contain for each POMDP the benchmark name,

[9] Instances with a finite belief MDP that would be fully explored by our algorithm are omitted since the exact value can be obtained without approximation techniques.

[10] For our implementation, only Gurobi runs multi-threaded. Prism uses multiple threads for garbage collection.

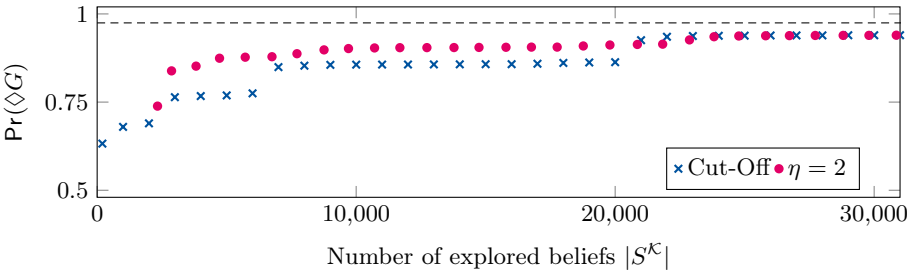


Figure 5. Accuracy for Drone 4-2 with different sizes of approximation MDP $\mathcal{K}_{\mathcal{M}}$

model parameters, property type (probabilities (P) or rewards (R)), and the numbers of states, state-action pairs, and observations. Column Prism gives the result with the smallest gap between over- and under-approximation computed with the approach of [37]. For maximising (minimising) properties, our approach competes with the lower (upper) bound of the provided interval. The relevant value is marked in bold. We also provide the computation time and the considered resolution η. For our implementation, we give results for the configuration with disabled clipping and for clipping with different resolutions η. In each cell, we give the obtained value, the computation time and the number of states in the abstraction MDP $\mathcal{K}_{\mathcal{M}}$. Time- and memory-outs are indicated by TO and MO. The right-most column indicates the over-approximation value computed via [8].

Discussion. The pure cut-off approach yields valid under-approximations in *all* benchmark instances—often exceeding the accuracy of the approach of [37] while being consistently faster. In some cases, the resulting values improve when clipping is enabled. However, larger candidate sets significantly increase the computation time which stems from the fact that many clipping MILPs have to be solved.

For Drone 4-2, Fig. 5 plots the resulting under-approximation values (y-axis) for varying sizes of the explored MDP $\mathcal{K}_{\mathcal{M}}$ (x-axis). The horizontal, dashed line indicates the computed over-approximation value. The quality of the approximation further improves with an increased number of explored beliefs.

5 Conclusion

We presented techniques to safely under-approximate expected total rewards in POMDPs. The approach scales to large POMDPs and often produces tight lower bounds. Belief clipping generally does not improve on the simpler cut-off approach in terms of results and performance. However, considering—and optimising—the approach for particular classes of POMDPs might prove beneficial. Future work includes integrating the algorithm into a refinement loop that also considers over-approximation techniques from [8]. Furthermore, lifting our approach to partially observable stochastic games is promising.

Data Availability. The artifact [9] accompanying this paper contains source code, benchmark files, and replication scripts for our experiments.

References

1. Amato, C., Bernstein, D.S., Zilberstein, S.: Optimizing fixed-size stochastic controllers for POMDPs and decentralized POMDPs. Auton. Agents Multi Agent Syst. **21**(3), 293–320 (2010)
2. Ashok, P., Butkova, Y., Hermanns, H., Kretínský, J.: Continuous-time Markov decisions based on partial exploration. In: ATVA. Lecture Notes in Computer Science, vol. 11138, pp. 317–334. Springer (2018)
3. Aström, K.J.: Optimal control of Markov processes with incomplete state information. J. of Mathematical Analysis and Applications **10**(1), 174–205 (1965)
4. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
5. Bellman, R.: A Markovian decision process. Journal of Mathematics and Mechanics **6**, 679–684 (1957)
6. Bonet, B.: Solving large POMDPs using real time dynamic programming. In: AAAI Fall Symp. on POMDPs (1998)
7. Bonet, B., Geffner, H.: Solving POMDPs: RTDP-Bel vs. Point-based Algorithms. In: IJCAI. pp. 1641–1646 (2009)
8. Bork, A., Junges, S., Katoen, J., Quatmann, T.: Verification of indefinite-horizon POMDPs. In: ATVA. Lecture Notes in Computer Science, vol. 12302, pp. 288–304. Springer (2020)
9. Bork, A., Katoen, J.P., Quatmann, T.: Artifact for Paper: Under-Approximating Expected Total Rewards in POMDPs. Zenodo (2022). https://doi.org/10.5281/zenodo.5643643
10. Bork, A., Katoen, J.P., Quatmann, T.: Under-Approximating Expected Total Rewards in POMDPs. arXiv e-print (2022), https://arxiv.org/abs/2201.08772
11. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Křetínskỳ, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: ATVA. Lecture Notes in Computer Science, vol. 8837, pp. 98–114. Springer (2014)
12. Braziunas, D., Boutilier, C.: Stochastic local search for POMDP controllers. In: AAAI. pp. 690–696. AAAI Press / The MIT Press (2004)
13. Carr, S., Jansen, N., Topcu, U.: Verifiable rnn-based policies for POMDPs under temporal logic constraints. In: IJCAI. pp. 4121–4127. ijcai.org (2020)
14. Carr, S., Jansen, N., Wimmer, R., Serban, A.C., Becker, B., Topcu, U.: Counterexample-guided strategy improvement for POMDPs using recurrent neural networks. In: IJCAI. pp. 5532–5539. ijcai.org (2019)
15. Chatterjee, K., Chmelík, M., Davies, J.: A symbolic SAT-based algorithm for almost-sure reachability with small strategies in POMDPs. In: AAAI. pp. 3225–3232 (2016)
16. Chatterjee, K., Chmelík, M., Gupta, R., Kanodia, A.: Optimal cost almost-sure reachability in POMDPs. Artificial Intelligence **234**, 26–48 (2016)
17. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative analysis of partially-observable Markov decision processes. In: MFCS. Lecture Notes in Computer Science, vol. 6281, pp. 258–269. Springer (2010)
18. Cheng, H.T.: Algorithms for partially observable Markov decision processes. Ph.D. thesis, University of British Columbia (1988)
19. Doshi, F., Pineau, J., Roy, N.: Reinforcement learning with limited reinforcement: Using Bayes risk for active learning in POMDPs. In: ICML. pp. 256–263 (2008)
20. Eagle, J.N.: The optimal search for a moving target when the search path is constrained. Operations Research **32**(5), 1107–1115 (1984)

21. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021), https://www.gurobi.com
22. Hauskrecht, M.: Value-function approximations for partially observable Markov decision processes. J. Artif. Intell. Res. **13**, 33–94 (2000)
23. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. on Software Tools for Technology Transfer (2021). https://doi.org/10.1007/s10009-021-00633-z
24. Horák, K., Bošanský, B., Chatterjee, K.: Goal-HSVI: Heuristic Search Value Iteration for Goal POMDPs. In: IJCAI. pp. 4764–4770. ijcai.org (7 2018)
25. Itoh, H., Nakamura, K.: Partially observable Markov decision processes with imprecise parameters. Artificial Intelligence **171**(8-9), 453–490 (2007)
26. Jansen, N., Dehnert, C., Kaminski, B.L., Katoen, J., Westhofen, L.: Bounded model checking for probabilistic programs. In: ATVA. Lecture Notes in Computer Science, vol. 9938, pp. 68–85 (2016)
27. Junges, S., Jansen, N., Seshia, S.A.: Enforcing almost-sure reachability in POMDPs. In: CAV (2). Lecture Notes in Computer Science, vol. 12760, pp. 602–625. Springer (2021)
28. Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L., Katoen, J.P., Becker, B.: Finite-state Controllers of POMDPs via Parameter Synthesis. In: UAI. pp. 519–529. AUAI Press (2018)
29. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial Intelligence **101**(1-2), 99–134 (1998)
30. Kurniawati, H., Hsu, D., Lee, W.S.: SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In: Robotics: Science and Systems. vol. 2008 (2008)
31. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011)
32. Lovejoy, W.S.: Computationally feasible bounds for partially observed Markov decision processes. Operations Research **39**(1), 162–175 (1991)
33. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In: AAAI/IAAI. pp. 541–548 (1999)
34. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and related stochastic optimization problems. Artificial Intelligence **147**(1-2), 5–34 (2003)
35. Meuleau, N., Kim, K.E., Kaelbling, L.P., Cassandra, A.R.: Solving POMDPs by searching the space of finite policies. In: UAI. pp. 417–426 (1999)
36. Monahan, G.E.: State of the art — a survey of partially observable Markov decision processes: theory, models, and algorithms. Management Science **28**(1), 1–16 (1982)
37. Norman, G., Parker, D., Zou, X.: Verification and Control of Partially Observable Probabilistic Systems. Real-Time Systems **53**(3), 354–402 (2017)
38. Pineau, J., Gordon, G., Thrun, S.: Point-based value iteration: An anytime algorithm for POMDPs. In: IJCAI. vol. 3, pp. 1025–1032 (2003)
39. Quatmann, T., Katoen, J.: Sound value iteration. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 643–661. Springer (2018)
40. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach (4th Edition). Pearson (2020)
41. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons (1986)

42. Shani, G., Pineau, J., Kaplow, R.: A survey of point-based POMDP solvers. Autonomous Agents and Multi-Agent Systems **27**(1), 1–51 (2013)
43. Silver, D., Veness, J.: Monte-Carlo planning in large POMDPs. In: NIPS. pp. 2164–2172 (2010)
44. Smallwood, R.D., Sondik, E.J.: The optimal control of partially observable Markov processes over a finite horizon. Operations Research **21**(5), 1071–1088 (1973)
45. Smith, T., Simmons, R.: Heuristic search value iteration for POMDPs. In: UAI. pp. 520–527 (2004)
46. Sondik, E.J.: The Optimal Control of Partially Observable Markov Processes. Ph.D. thesis, Stanford Univ Calif Stanford Electronics Labs (1971)
47. Sondik, E.J.: The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. Operations research **26**(2), 282–304 (1978)
48. Spaan, M.T., Vlassis, N.: Perseus: Randomized point-based value iteration for POMDPs. J. of Artificial Intelligence Research **24**, 195–220 (2005)
49. Volk, M., Junges, S., Katoen, J.P.: Fast dynamic fault tree analysis by model checking techniques. IEEE Transactions on Industrial Informatics **14**(1), 370–379 (2017)
50. Wang, Y., Chaudhuri, S., Kavraki, L.E.: Bounded Policy Synthesis for POMDPs with Safe-Reachability Objectives. In: AAMAS. pp. 238–246 (2018)
51. Winterer, L., Junges, S., Wimmer, R., Jansen, N., Topcu, U., Katoen, J.P., Becker, B.: Motion planning under partial observability using game-based abstraction. In: CDC. pp. 2201–2208. IEEE (2017)
52. Zhang, N.L., Lee, S.S.: Planning with partially observable Markov decision processes: advances in exact solution method. In: UAI. pp. 523–530 (1998)
53. Zhang, N.L., Zhang, W.: Speeding up the convergence of value iteration in partially observable Markov decision processes. Journal of Artificial Intelligence Research **14**, 29–51 (2001)

Correct Probabilistic Model Checking with Floating-Point Arithmetic*

Arnd Hartmanns✉

University of Twente, Enschede, The Netherlands
a.hartmanns@utwente.nl

Abstract. Probabilistic model checking computes probabilities and expected values related to designated behaviours of interest in Markov models. As a formal verification approach, it is applied to critical systems; thus we trust that probabilistic model checkers deliver correct results. To achieve scalability and performance, however, these tools use finite-precision floating-point numbers to represent and calculate probabilities and other values. As a consequence, their results are affected by rounding errors that may accumulate and interact in hard-to-predict ways. In this paper, we show how to implement fast *and* correct probabilistic model checking by exploiting the ability of current hardware to control the direction of rounding in floating-point calculations. We outline the complications in achieving correct rounding from higher-level programming languages, describe our implementation as part of the Modest Toolset's mcsta model checker, and exemplify the trade-offs between performance and correctness in an extensive experimental evaluation across different operating systems and CPU architectures.

1 Introduction

Given a Markov chain or Markov decision process (MDP [25]) model of a safety- or performance-critical system, probabilistic model checking (PMC) calculates quantitative properties of interest: the probability of (rare or catastrophic) failures, the expected recovery time after service interruption, or the long-run average throughput. These properties involve probabilities or expected costs/rewards of sets of model behaviours, and are often specified in a temporal logic like PCTL [16]. As a formal verification approach, users place great trust in the results delivered by a PMC tool such as Prism [22], Storm [9], ePMC [15], or the Modest Toolset's [18] mcsta. In contrast to classical model checkers for functional, Boolean-valued properties specified in e.g. LTL or CTL [2], a probabilistic model checker is inherently quantitative: the input model contains real-valued probabilities and costs/rewards; PCTL makes comparisons between real-valued constants and probabilities; the most efficient algorithms numerically iterate towards a fixpoint; and the final result itself may well be a real number.

* This work was supported by NWO VENI grant no. 639.021.754 and the EU's Horizon 2020 research and innovation programme under MSCA grant agreement 101008233.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 41–59, 2022.
https://doi.org/10.1007/978-3-030-99527-0_3

Often, we can restrict to rationals, which simplifies the theory and facilitates "exact" algorithms using arbitrary-precision rational number datatypes. These algorithms only work for small models (as shown in the most recent QComp 2020 competition of quantitative verification tools [6]). In this paper, we thus focus on the PMC techniques that scale to large problems: those building upon iterative numerical algorithms, in particular value iteration (VI) [8]. We restrict to probabilistic reachability, i.e. calculating the probability to eventually reach a goal state, as this is the core problem in PMC for MDP. Embedded in the usual recursive CTL algorithm, it allows us to check any (unbounded) PCTL formula.

Starting from a trivial underapproximation of the reachability probability for each state of the model, VI iteratively improves the value of each state based on its successors' values. The true reachability probabilities are the least fixpoint of this procedure, towards which the algorithm converges. For roughly a decade, PMC tools implemented VI by stopping once the relative or absolute difference between subsequent iterations was below a threshold ϵ. Haddad and Monmege [12] showed in 2014[1] that this does not guarantee a difference of $\leq \epsilon$ between the reported and the true probability, putting in question the trust placed in PMC tools. Then variants of VI were developed that provide *sound*, i.e. ϵ-correct, results: interval iteration (II) [3,5,13], sound value iteration (SVI) [26], and optimistic value iteration (OVI) [19]. We focus on II as the prototypical sound algorithm. It additionally iterates on an overapproximation; its stopping criterion is the difference between over- and underapproximation being $\leq \epsilon$.

If all probabilities in an MDP are rational numbers, then the true reachability probability as well as all intermediate values in II are rational, too. Yet implementing II with arbitrary-precision rationals is impractical since the smaller-and-smaller differences between intermediate values end up using excessive computation time and memory. II is thus implemented with fixed-precision (usually 64-bit IEEE 754 *double* precision) floating point numbers. These, however, cannot represent all rationals, so operations must round to nearby representable values. Although II is numerically benign, consisting only of multiplications and additions within $[0, 1]$, the default *round to nearest, ties to even* policy can cause II to deliver incorrect results. Wimmer et al. [29] show an example where PMC tools incorrectly state that a simple PCTL property is satisfied by a small Markov chain due to the underlying numeric difference having disappeared in rounding. We confirmed with current versions of Prism, Storm, and mcsta that the problem persists to today, even when requesting a "sound" algorithm like II. Wimmer et al. propose interval arithmetic to avoid such problems, cautioning that

> *[...] the memory consumption will roughly double, since two numbers for the interval bounds have to be stored [...]. The runtime will be higher by a small factor, because we need to derive lower and upper bounds for the intervals, requiring two model checking runs per sub-formula.* [29, p. 5]

They did not provide an implementation, and we are not aware of any to date.

[1] Wimmer et al. [29] already in 2008 mention this problem in a more general setting, but neither give a concrete counterexample nor propose a solution tailored to PMC.

Our contribution. We present the first PMC implementation that computes *correct* lower and upper bounds on reachability probabilities despite using floating-point arithmetic. We benefit from two developments since Wimmer et al.'s paper of 2008: First, II (published 2014) already uses intervals (though not as Wimmer et al. envisioned), necessarily doubling memory consumption compared to VI (as do SVI and OVI, so it appears an unavoidable cost of soundness). In place of "two model checking runs per sub-formula", we can make the two interleaved computations inside II safe w.r.t. rounding. Second, hardware and programming language support for controlling the rounding direction in floating-point operations has improved, in particular with the AVX-512 instruction set in the newest x86-64 CPUs and widespread compiler support for C99's "floating-point environment" header `fenv.h`. Nevertheless, it is nontrivial to achieve runtime that is only "higher by a small factor". For the analysis of probabilistic systems, the only related use of safe rounding we are aware of is in the SSMT tool SiSAT [27].

Structure. We recap PMC and II (Sect. 2) as well as problems and solutions related to rounding in floating-point arithmetic in Sect. 3. We then present our new approach in Sect. 4, including important implementation aspects. The performance of our approach is crucial to its adoption in tools; thus in Sect. 5 we report on extensive experiments across different software and hardware configurations on models from the Quantitative Verification Benchmark Set (QVBS) [20].

2 Probabilistic Model Checking

We write $\{ x_1 \mapsto y_1, \dots \}$ to denote the function that maps all x_i to y_i. Given a set S, its powerset is 2^S. A (discrete) *probability distribution* over S is a function $\mu \in S \to [0,1]$ with countable *support* $spt(\mu) \stackrel{\text{def}}{=} \{ s \in S \mid \mu(s) > 0 \}$ and $\sum_{s \in spt(\mu)} \mu(s) = 1$. $Dist(S)$ is the set of all probability distributions over S. If $\mu(s) \in \mathbb{Q}$ for all $s \in S$, we call μ a *rational probability distribution*, in $Dist_{\mathbb{Q}}(S)$.

Markov decision processes (MDP) [25] combine the nondeterminism of Kripke structures with the finite random choices of discrete-time Markov chains (DTMC).

Definition 1. *A* Markov decision process *(MDP) is a triple* $M = \langle S, s_I, T \rangle$ *where* S *is a finite set of* states *with* initial state $s_I \in S$ *and* $T \colon S \to 2^{Dist_{\mathbb{Q}}(S)}$ *is the* transition function. $T(s)$ *must be finite and non-empty for all* $s \in S$.

For $s \in S$, an element μ of $T(s)$ is a *transition*, and if $s' \in spt(\mu)$, then the transition has a *branch* to successor state s' with probability $\mu(s')$. If $|T(s)| = 1$ for all $s \in S$, then M is a DTMC.

Example 1. Fig. 1 shows our example MDP M_n^γ, which is actually a DTMC. It is a simplified and parametrised version of the counterexample of Wimmer et al. [29, Fig. 2]. It is parametrised in terms of $n \in \mathbb{N}$ (determining the number of chained states with transitions labelled `b`) and $\gamma \in (0, 0.5)$ (changing some probabilities). We draw transitions as lines to an intermediate node from which

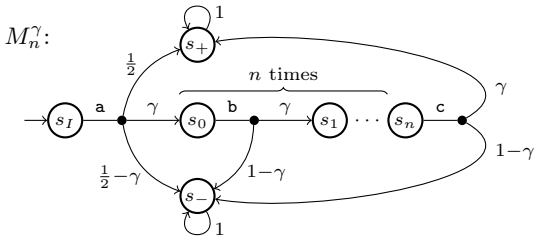


Fig. 1. Example parametrised MDP M_n^γ

probability-labelled branches lead to successor states. We omit the intermediate node for transitions with a single branch, and label some transitions to easily refer to them. M_n^γ has $4+n$ states and transitions, and $7+2n$ branches.

In practice, higher-level modelling languages like MODEST [14] are used to specify MDP. The semantics of an MDP is captured by its *paths*. A path represents a concrete resolution of all nondeterministic and probabilistic choices. Formally:

Definition 2. *A* finite path *is a sequence* $\pi_{\text{fin}} = s_0\,\mu_0\,s_1\,\mu_1 \ldots \mu_{n-1}s_n$ *where* $s_i \in S$ *for all* $i \in \{0,\ldots,n\}$ *and* $\mu_i \in T(s_i) \wedge \mu_i(s_{i+1}) > 0$ *for all* $i \in \{0,\ldots,n-1\}$. *Let* $|\pi_{\text{fin}}| \stackrel{\text{def}}{=} n$ *and* $\text{last}(\pi_{\text{fin}}) \stackrel{\text{def}}{=} s_n$. $\Pi_{fin}(s)$ *is the set of all finite paths starting in* s. *A* path *is an analogous infinite sequence* π, *and* $\Pi(s)$ *is the set of all paths starting in* s. *We write* $s \in \pi$ *if* $\exists\, i\colon s = s_i$.

A scheduler (or *adversary*, *policy* or *strategy*) only resolves the nondeterministic choices of M. For this paper, memoryless deterministic schedulers suffice [4].

Definition 3. *A function* $\mathfrak{s}\colon S \to \mathit{Dist}(S)$ *is a* scheduler *if, for all* $s \in S$, *we have* $\mathfrak{s}(s) \in T(s)$. *The set of all schedulers of* M *is* $\mathfrak{S}(M)$.

We are interested in *reachability probabilities*. Let $M|_{\mathfrak{s}} = \langle S, s_I, T|_{\mathfrak{s}}\rangle$ with $T|_{\mathfrak{s}}(s) = \{\mathfrak{s}(s)\}$ be the DTMC induced by $\mathfrak{s}$ on M. Via the standard cylinder set construction [10, Sect. 2.2] on $M|_{\mathfrak{s}}$, a scheduler induces probability measures $\mathbb{P}_{\mathfrak{s}}^{M,s}$ on measurable sets of paths starting in $s \in S$.

Definition 4. *For state* s *and goal state* $g \in S$, *the maximum and minimum* probability of reaching g *from* s *is defined as* $\mathrm{P}_{\max}^{M,s}(\diamond\, g) = \sup_{\mathfrak{s}\in\mathfrak{S}} \mathbb{P}_{\mathfrak{s}}^{M,s}(\{\pi \in \Pi(s) \mid g \in \pi\})$ *and* $\mathrm{P}_{\min}^{M,s}(\diamond\, g) = \inf_{\mathfrak{s}\in\mathfrak{S}} \mathbb{P}_{\mathfrak{s}}^{M,s}(\{\pi \in \Pi(s) \mid g \in \pi\})$, *respectively.*

The definition extends to sets G of goal states. We omit the superscript for M when it is clear from the context, and if we omit that for s, then $s = s_I$. From now on, whenever we have an MDP with a set of goal states G, we assume w.l.o.g. that all $g \in G$ are absorbing, i.e. every g only has one self-loop transition.

Definition 5. *A* maximal end component *(MEC) of* M *is a maximal (sub-)MDP* $\langle S', T', s_I'\rangle$ *where* $S' \subseteq S$, $T'(s) \subseteq T(s)$ *for all* $s \in S'$, *and the directed graph with vertex set* S' *and edge set* $\{\langle s, s'\rangle \mid \exists\, \mu \in T'(s)\colon \mu(s') > 0\}$ *is strongly connected.*

1 **function** $II(M = \langle S, s_I, T\rangle, G, opt, \epsilon)$
  // Preprocessing
2 **if** $opt = \max$ **then** $M := \texttt{CollapseMECs}(M, G)$ *// collapse MECs*
3 $S_0 := \texttt{Prob0}(M, G, opt)$, $S_1 := \texttt{Prob1}(M, G, opt)$ *// identify 0/1 states*
4 $l := \{ s \mapsto 0 \mid s \in S \setminus S_1 \} \cup \{ s \mapsto 1 \mid s \in S_1 \}$ *// initialise lower vector*
5 $u := \{ s \mapsto 0 \mid s \in S_0 \} \cup \{ s \mapsto 1 \mid s \in S \setminus S_0 \}$ *// initialise upper vector*
  // Iteration
6 **while** $(u(s_I) - l(s_I))/l(s_I) > \epsilon$ **do** *// while relative error > ϵ:*
7 **foreach** $s \in S \setminus (S_0 \cup S_1)$ **do** *// update non-0/1 states:*
8 $l(s) := opt_{\mu \in T(s)} \sum_{s' \in spt(\mu)} \mu(s') \cdot l(s')$ *// iterate lower vector*
9 $u(s) := opt_{\mu \in T(s)} \sum_{s' \in spt(\mu)} \mu(s') \cdot u(s')$ *// iterate upper vector*
10 **return** $\frac{1}{2}(u(s_I) - l(s_I))$

Alg. 1: Interval iteration for probabilistic reachability

2.1 Algorithms

Interval iteration [3,5,12,13] computes reachability probabilities $p(s) = \mathrm{P}^s_{opt}(\diamond G)$, $opt \in \{\max, \min\}$. We show the basic algorithm as Alg. 1. It iteratively refines vectors l and u that map each state to a value in $\mathbb{Q}$ such that, at all times, we have $l(s) \leq p(s) \leq u(s)$. In each iteration, the values in l and u are updated for all relevant states (line 7) via the classic Bellman equations of value iteration (lines 8-9). Their least fixpoint is p, towards which l converges from below. Some preprocessing is needed to ensure that the fixpoint is unique and also u converges towards p: for maximisation, we need to collapse MECs into single states (line 2). This can be be done via graph-based algorithms (see e.g. [7]) that only consider the graph structure of the MDP as in Definition 1 but do not perform calculations with the concrete probability values. For both maximisation and minimisation, we need to identify the sets S_0 and S_1 such that $\forall s \in S_0 \colon p(s) = 0$ and $\forall s \in S_1 \colon p(s) = S_1$ (line 3). This can equally done via graph-based algorithms [10, Algs. 1-4]. We then initialise l and u to trivial under-/overapproximations of p (lines 4-5). Iteration stops when the relative difference between l and u at s_I is at most ϵ (which is often chosen as 10^{-3} or 10^{-6}). The corresponding check in line 6 assumes that division by zero results in $+\infty$, as is the default in IEEE 754. By convergence of l and u towards the fixpoint, II terminates, and we eventually return a value $\hat{p}$ with the guarantee that $p(s_I) \in [(1-\epsilon) \cdot \hat{p}, (1+\epsilon) \cdot \hat{p}]$. This makes II *sound*.

PCTL. The temporal logic PCTL [16] allows us to construct complex branching-time properties. It takes standard CTL [2] and replaces the $\mathtt{A}(\psi)$ ("for all paths ψ holds") and $\mathtt{E}(\psi)$ ("there exists a path for which ψ holds") operators by the probabilistic operator $\mathtt{P}_{\sim c}(\psi)$ for "under all schedulers, the probability of the measurable set of paths for which ψ holds is $\sim c$" where $\sim \in \{<, \leq, >, \geq\}$ and $c \in [0, 1]$. To model-check a PCTL formula on MDP M, we follow the standard recursive CTL model checking algorithm [2, Sect. 6.4] except for the P

operator, which can be reduced to computing reachability probabilities. For the "finally"/"eventually" case $\mathtt{P}_{\sim c}(\mathtt{F}\,\phi)$, we can directly use interval iteration: Let S_ϕ be the set of states recursively determined to satisfy ϕ. Call $\mathtt{II}(M, S_\phi, opt_\sim, \epsilon)$ of Alg. 1 with $opt_\sim = \max$ if $\sim \in \{<, \leq\}$ and $opt_\sim = \min$ otherwise, with two modifications: Change the stopping criterion of line 6 to check the difference for *all* states, and in line 10, return the set $S_\mathtt{P} \stackrel{\text{def}}{=} \{\, s \in S \mid \forall x \in [l(s), u(s)] \colon x \sim c \,\}$. If $\exists s \in S, x \in [l(s), u(s)] \colon x \nsim c$, however, we would need to either abort and report an "unknown" situation, or continue with a reduced ϵ until we can (hopefully eventually) decide the comparison. None of Prism, Storm, and mcsta appear to perform this extra check, though. In this paper, we only use PCTL for non-nested top-level $\mathtt{P}_*(\mathtt{F}\ldots)$ operators; the results are then *true* if $s_I \in S_\mathtt{P}$, should be *unknown* in case the "unknown" situation applies to s_I, and are *false* otherwise.

3 Floating-Point Arithmetic

The current implementations of II (in Prism, Storm, and mcsta) use IEEE 754 double-precision floating-point arithmetic to represent (a) the probabilities of the MDP's branches and (b) the values in l and u. A floating-point number is stored as a *significand* d and an *exponent* e w.r.t. to an agreed-upon *base* b such that it represents the value $d \cdot b^e$. We fix $b = 2$. IEEE 754 *double* precision uses 64 bits in total, of which 1 is a sign bit, 52 are for d, and 11 are for e. Standard alternatives are 32-bit *single* precision (1 sign, 23 bits for d, and 8 for e) and the 80-bit x87 *extended* precision format (with 1 sign bit, 64 for d, and 15 for e). The subset of $\mathbb{Q}$ that can be represented in such a representation is determined by the numbers of bits for d and e. For example, $\frac{1}{2}$ or $\frac{7}{8}$ can be represented exactly in all formats, but $\frac{1}{10}$ cannot. IEEE 754 prescribes that all basic operations (addition, multiplication, etc.) are performed at "infinite precision" with the result *rounded* to a representable number. The default rounding mode is to round to the nearest such number, choosing an even value in case of ties (*round to nearest, ties to even*). In single precision, $\frac{1}{10}$ is thus by default rounded to

$$13421773 \cdot 2^{-27} = 0.100000001490116119384765625.$$

A single rounded operation leads to an error of at most the distance between the two nearest representable numbers. In iterative computations, however, rounding may happen at every step. A striking example of the consequences is the failure of an American Patriot missile battery to intercept an incoming Iraqi Scud missile in February 1992 in Dharan, Saudi Arabia [28], which resulted in 28 fatalities. The Patriot system calculated time in seconds by multiplying its internal clock's value by a rounded binary representation of $\frac{1}{10}$. After 100 hours of continuous operation, this lead to a cumulative rounding error large enough to miscalculate the incoming missile's position by more than half a kilometre [1].

3.1 Errors in Probabilistic Model Checking

II accumulates and multiplies rounded floating-point values in the l and u vectors with potentially already-rounded values representing the rational probabilities of

the model. Using the default rounding mode, how can we be sure that the final result does not miss the true probability by more than half a kilometre, too?

Following Wimmer et al. [29], let us consider MDP M_n^γ of Fig. 1 again, and determine whether $\mathtt{P}_{\leq \frac{1}{2}}(\Diamond \{ s_+ \})$ holds. The model is acyclic, so it is easy to see that

$$p \stackrel{\text{def}}{=} \mathrm{P}_{\max}(\Diamond \{ s_+ \}) = \frac{1}{2} + \gamma^{n+2} > \frac{1}{2}.$$

Let us fix $n = 1$ and $\gamma = 10^{-6}$. Then $p = \frac{1}{2} + 10^{-18}$. This value cannot be represented in double precision, and is by default rounded to 0.5.

We have encoded M_n^γ in the MODEST and PRISM languages, and checked the answers returned by PRISM 4.7, STORM 1.6.4, and `mcsta` 3.1 for the property. The correct result would be *false*. PRISM returns *true* in its default configuration, which uses an unsound algorithm, and *false* when requesting an algorithm with exact rational arithmetic, for which M_n^γ is small enough. If we explicitly request PRISM to use II, then the result depends on the specified ϵ: for $\epsilon \geq 10^{-11}$, we get the correct result of *false*; for *smaller* $\epsilon \leq 10^{-12}$, i.e. *higher precision*, however, we incorrectly get *true*. STORM incorrectly returns *true* in its default configuration as well as when we request a sound algorithm via the `--sound` parameter. Only when using an exact rational algorithm via the `--exact` parameter does STORM correctly return *false*. `mcsta`, when using II (`--alg IntervalIteration`), incorrectly returns *true*, and additionally reports that it computed $[l(s_I), u(s_I)]$ as $[0.5, 0.5]$, thus not including the true value of p. Other algorithms are not immune to the problem, either; for example, `mcsta` also answers *true* when using SVI, OVI, and when solving the MDP as a linear programming problem via the Google OR Tools' GLOP LP solver.

This example shows that using a sound algorithm does not guarantee correct results. The problem is not specific to cases of small probabilities like $\gamma = 10^{-6}$ in the MDP; we can achieve the same effect using arbitrarily higher values of γ if we just increase n a little. Such bounded try-and-retry chains where "normal" probabilities in the model result in very small values during iteration and on the final result are not uncommon in the systems often modelled as MDPs, e.g. backoff schemes in communication protocols and randomised algorithms. In general, tiny differences in probabilities in one place may result in significant changes of the overall reachability probability; for example, in two-dimensional random walks, the long-run behaviour when the probabilities to move forward or backward are both $\frac{1}{2}$ is vastly different from if they are $\frac{1}{2} + \delta$ and $\frac{1}{2} - \delta$, respectively, for any $\delta > 0$.

3.2 On Precision and Rounding Modes

In our concrete example, we may be able to avoid the problem by increasing precision: In the 80-bit extended format supported by all x86-64 CPUs, $\frac{1}{2} + 10^{-18}$ is by default rounded to $5.00000000000000009\ldots \cdot 10^{-1}$, so there is a chance of obtaining *false* unless other rounding during iterations would lose all the difference. Extended precision is used for C's `long double` type by e.g. the GCC compiler; it is thus readily accessible to programmers. It is, however, the most

precise format supported in common CPUs today; if we need more precision, we would have to resort to much slower software implementations using e.g. the GNU MPFR library. Any a-priori fixed precision, however, just shifts the problem to smaller differences, but does not eliminate it.

The more general solution that we propose in this paper is to control the rounding mode of the floating-point operations performed in the II algorithm. In addition to the default *round to nearest, ties to even* mode, the IEEE 754 standard defines three *directed* rounding modes: *round towards zero* (i.e. truncation), *round towards* $+\infty$ (i.e. always round up), and *round towards* $-\infty$ (i.e. always round down). As we will explain in Sect. 4, using the latter gives us an easy way to make the computations inside II *safe*, i.e. guarantee the under- and overapproximation invariants for l and u, respectively. Control of the floating-point rounding mode however appears to be a rarely-used feature of IEEE 754 implementations; consequently the level and style of support for it in CPUs and high-level programming languages is diverse.

3.3 CPU Support for Rounding Modes

Storm and mcsta run exclusively on x86-64 systems (with the upcoming ARM-based systems so far only supported via their x86-64 emulation layers), while Prism additionally supports several other platforms via manual compilation. Thus we focus on x64-64 in this paper as the platform probabilistic model checkers overwhelmingly run on today.

X87 and SSE. All x64-64 CPUs support two instruction sets to perform floating-point operations in double precision: The x87 instruction set, originating from the 8087 floating-point coprocessor, and the SSE instruction set, which includes support for double precision since the Pentium 4's SSE2 extension. Both implement operations according to the IEEE 754 standard. Aside from architectural particularities such as its stack-based approach to managing registers, the x87 instruction set notably includes support for 80-bit extended precision. In fact, by default, it performs all calculations in that extended precision, only rounding to double or single precision when storing values back to 64- or 32-bit memory locations. This has the advantage of reducing the error across sequences of operations, but for high-level languages makes the results depend on the compiler's choices of when to load/store intermediate values in memory vs. keeping them in x87 registers. The SSE instructions only support single and double precision.

Both the x87 and SSE instruction sets support all four rounding modes mentioned above. The rounding mode of operations for x87 and SSE is determined by the current value of the x87 FPU control word stored in the x87 FPU control register or the current value of the SSE MXCSR control register, respectively. That is, to change rounding mode, we need to obtain the current control register value, change the two bits determining rounding mode (with the other bits controlling other aspects of floating-point operations such as the treatment of NaNs), and apply the new value. This is done via the FNSTCW/FLDCW instruction pair on x87, and VSTMXCSR/VLDMXCSR for SSE. Rounding mode

is thus part of the global (per-thread) state, and we must be careful to restore its original configuration when returning to code that does not expect rounding mode changes. Frequent changes of rounding mode thus incur a performance overhead due to the extra instructions that must be executed for every change and their effects on e.g. pipelining.

AVX-512. AVX-512 is the extension to 512 bits of the sequence of *single instruction, multiple data* (SIMD) instruction sets in x84-64 processors that started with SSE. It became available for general-purpose systems in high-end desktop (Skylake-X) and server (Xeon) CPUs in 2017, but it took until the 10th generation of Intel's Core mobile CPUs in 2019 before it was more widely available in end-user systems. It is supposed to appear in AMD CPUs with the upcoming Zen 4 architecture. Aside from its 512-bit SIMD instructions, AVX-512 crucially also includes new instructions for single floating-point values where the operation's rounding mode is specified as part of the instruction itself via the new "EVEX" encoding. Of particular note for implementing II are the new VFMADD($r_1r_2r_3$)SD *fused multiply-add instructions* (the r_i determining how the operand registers are used) that can directly be used for the sums of products in the Bellman equations in lines 8-9 of Alg. 1. Overall, AVX-512 thus makes rounding mode independent of global state, and may improve performance by removing the need for extra instruction sequences to change rounding mode.

3.4 Rounding Modes in Programming Languages

Support for non-default rounding modes is lacking in most high-level programming languages. Java, C#, and Python, for example, do not support them at all. If II is implemented in such a language, there is consequently no hope for a high-performance solution to the rounding problems described earlier.

For C and C++, the C99 and C++11 standards introduced access to the *floating-point environment*. The `fenv.h`/`cfenv` headers include the `fegetround` and `fesetround` functions to query the current rounding mode and change it, respectively. Implementations of these functions on x86-64 read/change both the x87 and SSE control registers accordingly. In the remainder of this paper, we focus on a C implementation, but most statements hold for C++ analogously. The level of support for the C99 floating-point features varies significantly between compilers; it is in particular still incomplete in Clang[2] and GCC [11, Further notes]. Still, both compilers provide access to the `fegetround`/`fesetround` functions (via the associated standard libraries), but GCC in particular is not rounding mode-aware in optimisations. This means that, for example, subexpressions that are evaluated twice, with a change in rounding mode in between, may be compiled by GCC into a single evaluation before the change, with the resulting value stored in a register and reused after the rounding mode change. This can

[2] The documentation as of October 2021 states that C99 support in Clang "is feature-complete except for the C99 floating-point pragmas".

even happen when using the `-frounding-math` option[3]. Programmers thus need to inspect the generated assembly to ensure that no problematic transformations have been made, or try to make them impossible by declaring values `volatile` or inserting inline assembly "barriers".

Overall, C thus provides a standardised way to change x87/SSE rounding mode, but programmers need to be aware of compiler quirks when using these facilities. Support for AVX-512 instructions that include rounding mode bits in C, on the other hand, is only slightly more convenient than programming in assembly as we can use the intrinsics in the `immintrin.h` header; there is no standard higher-level abstraction of this feature in either C or C++.

4 Correctly Rounding Interval Iteration

Let us now change II as in Alg. 1 to consistently round in safe directions at every numeric operation. Given that we can change or specify the rounding mode of all basic floating-point operations on current hardware, we expect that a high-performance implementation can be achieved. First, the preprocessing steps require no changes as they are purely graph-based. The changes to the iteration part of the algorithm are straightforward: In line 6,

$$\textbf{while } (u(s_I) - l(s_I))/l(s_I) > \epsilon \textbf{ do } \ldots,$$

we round the results of the subtraction and of the division towards $+\infty$ to avoid stopping too early. In line 8,

$$l(s) := opt_{\mu \in T(s)} \textstyle\sum_{s' \in spt(\mu)} \mu(s') \cdot l(s'),$$

the multiplications and additions round towards $-\infty$ while the corresponding operations on the upper bound in line 9 round towards $+\infty$. Recall that all probabilities in the MDP are rational numbers, i.e. representable as $\frac{num}{den}$ with $num, den \in \mathbb{N}$. We assume that num and den can be represented exactly in the implementation. Then, in line 8, we calculate the floating-point values for the $\mu(s') = num/den$ by rounding towards $-\infty$. In line 9, we round the result of the corresponding division towards $+\infty$. Finally, instead of returning the middle of the interval in line 10, we return $[l(s_I), u(s_I)]$ so as not to lose any information (e.g. in case the result is compared to a constant as in the example of Sect. 3.1).

With these changes, we obtain an interval guaranteed to contain the true reachability probability if the algorithm terminates. However, rounding away from the theoretical fixpoint in the updates of l and u means that we may reach an effective fixpoint—where l and u no longer change because all newly computed values round down/up to the values from the previous iteration—at a point where the relative difference of $l(s_I)$ and $u(s_I)$ is still above ϵ. This will happen in practice: In QComp 2020 [6], mcsta participated in the *floating-point correct track* by letting VI run until it reached a fixpoint under the default rounding mode with double precision. In 9 of the 44 benchmark instances that mcsta attempted to solve in this way, the difference between this fixpoint and

[3] The documentation as of Oct. 2021 states that `-frounding-math` "does not currently guarantee to disable all GCC optimizations that are affected by rounding mode."

```
1  function SR-SII(M = ⟨S, s_I, T⟩, G, opt, ε)
2      ... (preprocessing as in Alg. 1) ...
3      repeat
4          chg := false
5          fesetround(towards −∞)
6          foreach s ∈ S \ (S_0 ∪ S_1) do
7              l_new := opt_{μ∈T(s)} Σ_{s'∈spt(μ)} μ(s') · l(s')        // iterate lower vector
8              if l_new ≠ l(s) then chg := true
9              l(s) := l_new
10         fesetround(towards +∞)
11         foreach s ∈ S \ (S_0 ∪ S_1) do
12             u_new := opt_{μ∈T(s)} Σ_{s'∈spt(μ)} μ(s') · u(s')        // iterate upper vector
13             if u_new ≠ u(s) then chg := true
14             u(s) := u_new
15     until ¬chg ∨ (u(s_I) − l(s_I))/l(s_I) ≤ ε
16     return [l(s_I), l(s_I)]
```

Alg. 2: Safely rounding sequential interval iteration (SR-SII) for x87 or SSE

the true value was more than the specified ϵ. With safe rounding away from the true fixpoint, this would likely have happened in even more cases.

To ensure termination, we thus need to make one further change to the II of Alg. 1: In each iteration of the **while** loop, we additionally keep track of whether any of the updates to l and u changes the previous value. If not, we end the loop and return the current interval, which will be wider than the requested ϵ relative difference. We refer to II with all of the these modifications as *safely rounding interleaved II* (SR-III) in the remainder of this paper.

4.1 Sequential Interval Iteration

When using the x87 or SSE instruction sets to implement SR-III, we need to insert a call to `fesetround` just before line 8, and another just before line 9. If, for an MDP with n states, we need m iterations of the **while** loop, we will make $2 \cdot n \cdot m$ calls to `fesetround`. This might significantly impact performance for models with many states, or that need many iterations (such as the *haddad-monmege* model of the QVBS, which requires 7 million iterations with $\epsilon = 10^{-6}$ despite only having 41 states). As an alternative, we can rearrange the iteration phase of II as shown in Alg. 2: We first update l for all states (lines 6-9), then u for all states (lines 11-14), with the rounding mode changes in between (lines 5 and 10). We call this variant of II *safely rounding sequential II* (SR-SII). It only needs $2 \cdot m$ calls to `fesetround`, which should improve its performance. However, it also changes the memory access pattern of II with an a priori unknown effect on performance. We write III for II to stress that it is *interleaved*, and SII for Alg. 2 without the safe rounding, in the remainder of this paper.

4.2 Implementation Aspects

We have implemented III, SII, SR-III, and SR-SII in mcsta. While mcsta is written in C#, the new algorithms are (necessarily) written in C, called from the main tool via the P/Invoke mechanism. We used GCC 10.3.0 to compile our implementations on both 64-bit Linux and Windows 10. We manually inspected the disassembly of the generated code to ensure that GCC's optimisations did not interfere with rounding mode changes as described in Sect. 3.4. In a significant architectural change, we modified mcsta's state space exploration and representation code to preserve the exact rational values for the probabilities specified in the model, so that safely-rounded floating-point representations for the $\mu(s')$ can be computed during iteration as described above.

Of each algorithm, we implemented four variants: a *default* one that leaves the choice of instruction set to the compiler and uses `fesetround` to change rounding mode; an *x87* variant that forces floating-point operations to use the x87 instructions by attributing the relevant functions with `target("fpmath=387")` and that changes rounding mode via inline assembly using FNSTCW/FLDCW; an *SSE* variant that forces the SSE instruction set via `target("fpmath=sse")` and uses VSTMXCSR/VLDMXCSR in inline assembly for rounding mode changes; and an *AVX-512* variant that implements all floating-point operations requiring non-default rounding modes via AVX-512 intrinsics, in particular using `_mm_fmadd_round_sd` in the Bellman equations. All variants use double precision; *default* and *SSE* additionally have a single-precision version (which we omit for *x87* since the reduced precision does not speed up the operations we use); and *x87* also provides an 80-bit extended-precision version (however we currently return its results as safely-rounded double-precision values due to the unavailability of a `long double` equivalent in C#, which limits its use outside of performance testing for now). All in all, we thus provide 28 variants of interval iteration for comparison, out of which 14 provide guaranteed correct results.

In particular, the safe rounding makes PMC feasible at 32-bit single precision, which would otherwise be too likely to produce incorrect results. While we expect that this may deliver many results with low precision (but which are correct) due to a rounded fixpoint being reached long before the relative width reaches ϵ, it also halves the memory needed to store l and u, and may speed up computations. At the opposite end, mcsta is now also the first PMC tool that can use 80-bit extended precision, which however doubles the memory needed for l and u since 80-bit `long double` values occupy 16 bytes in memory (with GCC).

5 Experiments

Using our implementation in mcsta, we first tested all variants of the algorithms on M_n^γ in the setting of Sect. 3.1. As expected, and validating the correctness of the approach and its implementation, all SR variants return *unknown*.

We then assembled a set of 31 benchmark instances—combinations of a model, values for its configurable parameters, and a property to check—from

the QVBS covering DTMC, MDP, and probabilistic timed automata (PTA) [24] transformed to MDP by mcsta using the digital clocks approach [23]. These are all the models and probabilistic reachability probabilities from the QVBS supported by mcsta for which the result was not 0 or 1 (then it can be computed via graph-based algorithms) and for which a parameter configuration was available where PMC terminated within our timeout of 120 s but II needed enough time for it to be measured reliably ($\gtrapprox 0.2$ s). We checked each of these benchmarks with all 28 variants of our algorithms using $\epsilon = 10^{-6}$ on different x86-64 systems: **I11w**: an Intel Core i5-1135G7 (up to 4.2 GHz) laptop running Windows 10, this being the only system we had access to with AVX-512 support; **AMDw**: an AMD Ryzen 9 5900X (3.7-4.8 GHz) workstation running Windows 10, representing current AMD CPUs in our evaluation; **I4x**: an Intel Core i7-4790 (3.6-4.0 GHz) workstation running Ubuntu Linux 18.04, representing older-generation Intel desktop hardware; and **IPx**: an Intel Pentium Silver J5005 (1.5-2.8 GHz) compact PC running Ubuntu Linux 18.04, representing a non-Core low-power Intel system. We show a selection of our experimental results in the remainder of this section, mainly from I11w and AMDw. We remark on cases where the other systems (all with Intel CPUs) showed different patterns from I11w.

We present results graphically as scatter plots like in Fig. 2. Each such plot compares two algorithm variants in terms of runtime for the iteration phase of the algorithm only (i.e. we exclude the time for state space exploration and preprocessing). Every point $\langle x, y \rangle$ corresponds to a benchmark instance and indicates that the variant noted on the x-axis took x seconds to solve this instance while the one noted on the y-axis took y seconds. Thus points above the solid diagonal line correspond to instances where the x-axis method was faster; points above (below) the upper (lower) dotted diagonal line are where the x-axis method took less than half (more than twice) as long.

Fig. 2 first shows the performance impact of enabling safe rounding for the standard interleaved algorithm using double precision. The top row shows the behaviour on I11w. We see that runtime is drastically longer in the *default* variant that uses `fesetround`, but only increases by a factor of around 2 if we use the specific inline assembly instructions. We note that GCC includes the code for `fesetround` in the generated `.dll` file on Windows, but in contrast to the assembly methods does not inline it into the callers. Some of the difference may thus be function call overhead. The middle row shows the behaviour on AMDw. Here, *default* is affected just as badly, but the effect on *SEE* is worse while that on *x87* is much lower than on the Intel I11w system. In the bottom row, we show the impact on *default* on the Linux systems (bottom left and bottom middle), which is much lower than on Windows. This is despite GCC implementing `fesetround` as an external library call here. The overhead still markedly differs between the two Intel CPUs, though. Finally, as expected, we see on the bottom right than safe rounding has almost no performance impact when using the AVX-512 instructions.

Seeing the significant impact enabling safe rounding can have, we next show what the sequential algorithm brings to the table, in Fig. 3. On the top left, we

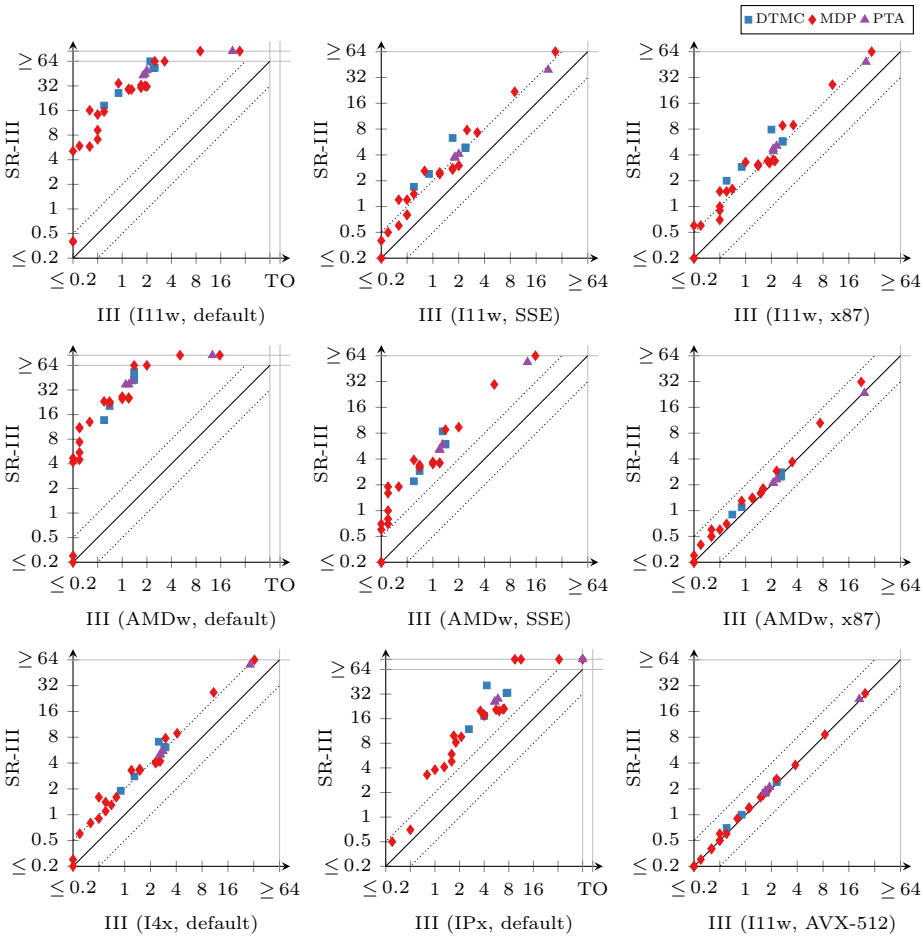


Fig. 2. Performance impact of safe rounding across instruction sets and systems

compare the base algorithms without safe rounding, where SII takes up to twice as long in the worst case. This is likely due to the more cache-friendly memory access pattern of III: we store l and u interleaved for III, so it always operates on two adjacent values at a time. The bottom-left plot confirms that reducing the number of rounding mode changes reduces the overhead of safe rounding to essentially zero. The remaining four plots show the differences between SR-III and SR-SII. In all cases except *x87* on AMDw, SR-III is slower. We thus have that III is fastest but unsafe, SII and SR-SII are equally fast but the latter is safe, and SR-III is safe but tends to be slower on the Intel systems. On the AMD system, SR-III surprisingly wins over SR-SII with *x87*, highlighting that the x87 instruction set in Ryzen 3 must be implemented very differently from SSE.

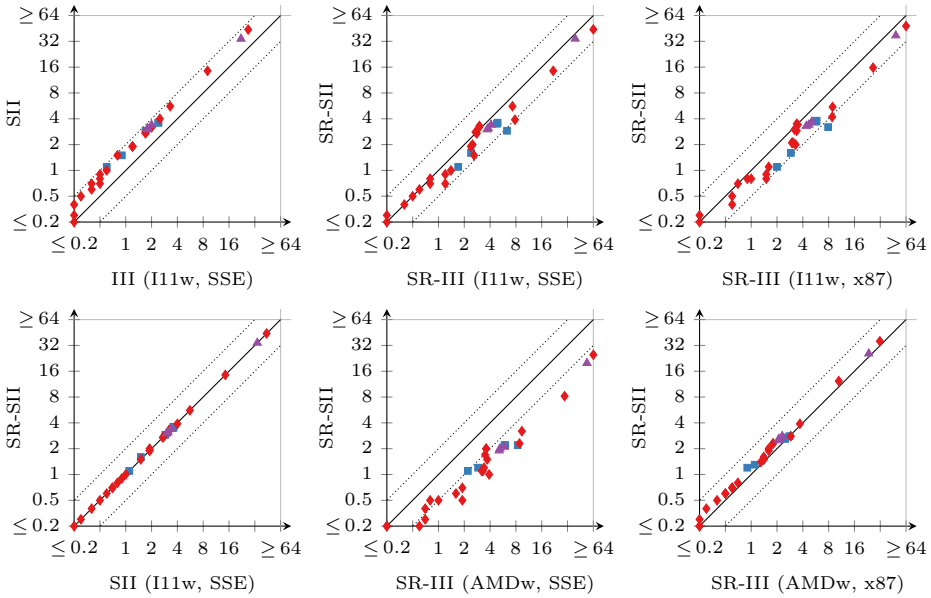


Fig. 3. Performance of interleaved compared to sequential II

We further investigate the impact of the instruction set in Fig. 4. Confirming the patterns we saw so far, SSE is slightly faster than x87 on I11w (and we see similar behaviour on the other Intel systems) but slower by a factor of more than 2 on the AMD CPU. The rightmost plot highlights that AVX-512 is the fastest alternative on the most recent Intel CPUs, which may in part be due to the availability of the fused multiply-add instruction that fits II so well.

All results so far were for double-precision computations. To conclude our evaluation, we show in Fig. 5 that reducing to single precision does not bring the expected performance benefits. We see in the leftmost plot that the overhead

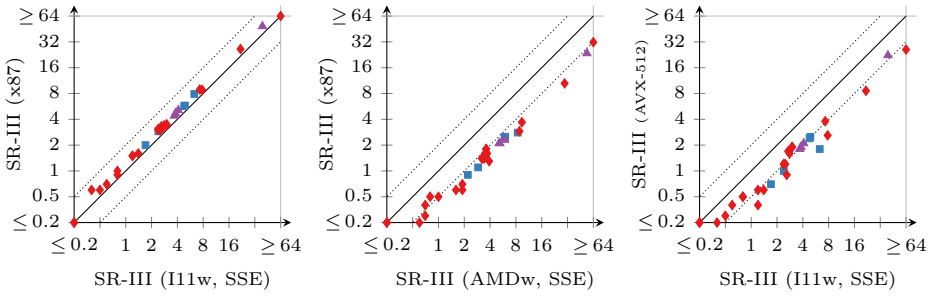


Fig. 4. Performance with different instruction sets

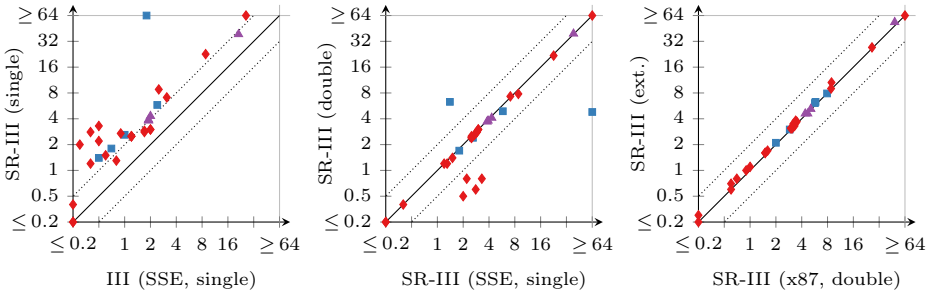


Fig. 5. Performance with different precision settings (on I11w)

of safe rounding has a much higher variance compared to Fig. 2. The detailed tool outputs hint at the reason being that rounding away from the fixpoint occurs in much larger steps with single precision, which significantly slows down or stops the convergence in several instances. The middle plot shows that, aside from the slowly converging outliers, using single precision does not provide a speedup over using doubles. Finally, on the right, we show that the impact of enabling 80-bit extended precision on x87 is minimal.

6 Conclusion

There has been ample research into sound PMC algorithms over the past years, but the problem of errors introduced by naive implementations using default floating-point rounding has been all but ignored. We showed that a solution exists that, while perhaps conceptually simple, faces a number of implementation and performance obstacles. In particular, hardware support for rounding modes is arguably essential to achieve acceptable performance, but difficult to use from C/C++ and impossible to access from most other programming languages. We extensively explored the space of implementation variants, highlighting that performance crucially depends on the combination of the variant, the CPU, and the operating system. Nevertheless, our results show that truly correct PMC is possible today at a small cost in performance, which should all but disappear as AVX-512 is more widely adopted. With our implementation in mcsta, we provide the first PMC tool that combines fast, scalable, *and* correct.

Acknowledgments. This work was triggered by Masahide Kashiwagi's excellent overview of the different ways to change rounding mode as used by his `kv` library for verified numerical computations [21]. The author thanks Anke and Ursula Hartmanns for contributing to the diversity of hardware on which the experiments were performed by providing access to the AMDw and I11w systems.

Data availability. A dataset to replicate the experimental evaluation, including the exact versions of the tools and models used, is archived and available at DOI 10.4121/19074047 [17].

References

1. Arnold, D.N.: Some disasters attributable to bad numerical computing: The Patriot missile failure (2000), https://www-users.cse.umn.edu/~arnold/disasters/patriot.html, last accessed 2021-10-14.
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
3. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for markov decision processes. In: Majumdar, R., Kuncak, V. (eds.) 29th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10426, pp. 160–180. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_8
4. Bianco, A., de Alfaro, L.: Model checking of probabalistic and nondeterministic systems. In: 15th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). Lecture Notes in Computer Science, vol. 1026, pp. 499–513. Springer (1995). https://doi.org/10.1007/3-540-60692-0_70
5. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.F. (eds.) 12th International Symposium on Automated Technology for Verification and Analysis (ATVA). Lecture Notes in Computer Science, vol. 8837, pp. 98–114. Springer (2014). https://doi.org/10.1007/978-3-319-11936-6_8
6. Budde, C.E., Hartmanns, A., Klauck, M., Kretínský, J., Parker, D., Quatmann, T., Turrini, A., Zhang, Z.: On correctness, precision, and performance in quantitative verification – QComp 2020 competition report. In: Margaria, T., Steffen, B. (eds.) 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA). Lecture Notes in Computer Science, vol. 12479, pp. 216–241. Springer (2020). https://doi.org/10.1007/978-3-030-83723-5_15
7. Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: Randall, D. (ed.) Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 1318–1336. SIAM (2011). https://doi.org/10.1137/1.9781611973082.101
8. Chatterjee, K., Henzinger, T.A.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking - History, Achievements, Perspectives. Lecture Notes in Computer Science, vol. 5000, pp. 107–138. Springer (2008). https://doi.org/10.1007/978-3-540-69850-0_7
9. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kuncak, V. (eds.) 29th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10427, pp. 592–600. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_31
10. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M., Issarny, V. (eds.) 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM). Lecture Notes in Computer Science, vol. 6659, pp. 53–113. Springer (2011). https://doi.org/10.1007/978-3-642-21455-4_3
11. Free Software Foundation: Status of C99 features in GCC (2021), https://gcc.gnu.org/c99status.html, as accessed on 2021-10-14.
12. Haddad, S., Monmege, B.: Reachability in MDPs: Refining convergence of value iteration. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) 8th International Workshop

on Reachability Problems (RP). Lecture Notes in Computer Science, vol. 8762, pp. 125–137. Springer (2014). https://doi.org/10.1007/978-3-319-11439-2_10
13. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018). https://doi.org/10.1016/j.tcs.2016.12.003
14. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013). https://doi.org/10.1007/s10703-012-0167-z
15. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: A web-based probabilistic model checker. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) 19th International Symposium on Formal Methods (FM). Lecture Notes in Computer Science, vol. 8442, pp. 312–317. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_22
16. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects Comput. **6**(5), 512–535 (1994). https://doi.org/10.1007/BF01211866
17. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic (artifact). 4TU.Centre for Research Data (2022). https://doi.org/10.4121/19074047
18. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 8413, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_51
19. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) 32nd International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 12225, pp. 488–511. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_26
20. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 11427, pp. 344–350. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_20
21. Kashiwagi, M.: kv – a C++ library for verified numerical computation, http://verifiedby.me/kv/index-e.html, last accessed 2021-10-13.
22. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) 23rd International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47
23. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Des. **29**(1), 33–78 (2006). https://doi.org/10.1007/s10703-006-0005-2
24. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**(1), 101–150 (2002). https://doi.org/10.1016/S0304-3975(01)00046-9
25. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887
26. Quatmann, T., Katoen, J.P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) 30th International Conference on Computer Aided Verifica-

tion (CAV). Lecture Notes in Computer Science, vol. 10981, pp. 643–661. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_37
27. Teige, T., Fränzle, M.: Constraint-based analysis of probabilistic hybrid systems. In: Giua, A., Mahulea, C., Silva, M., Zaytoon, J. (eds.) 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS). IFAC Proceedings Volumes, vol. 42, pp. 162–167. Elsevier (2009). https://doi.org/10.3182/20090916-3-ES-3003.00029
28. United States General Accounting Office: Software problem led to system failure at Dhahran, Saudi Arabia. Report GAO/IMTEC-92-26 (February 1992), https://www-users.cse.umn.edu/~arnold/disasters/GAO-IMTEC-92-96.pdf
29. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: Straube, B., Drutarovský, M., Renovell, M., Gramata, P., Fischerová, M. (eds.) 11th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS). pp. 207–212. IEEE Computer Society (2008). https://doi.org/10.1109/DDECS.2008.4538787

Correlated Equilibria and Fairness in Concurrent Stochastic Games

Marta Kwiatkowska[1], Gethin Norman[2], David Parker[3], and Gabriel Santos[1] (✉)

[1] Department of Computer Science, University of Oxford, Oxford, UK
{marta.kwiatkowska,gabriel.santos}@cs.ox.ac.uk
[2] School of Computing Science, University of Glasgow, Glasgow, UK
gethin.norman@glasgow.ac.uk
[3] School of Computer Science, University of Birmingham, Birmingham, UK
d.a.parker@cs.bham.ac.uk

Abstract. Game-theoretic techniques and equilibria analysis facilitate the design and verification of competitive systems. While algorithmic complexity of equilibria computation has been extensively studied, practical implementation and application of game-theoretic methods is more recent. Tools such as PRISM-games support automated verification and synthesis of zero-sum and (ε-optimal subgame-perfect) social welfare Nash equilibria properties for concurrent stochastic games. However, these methods become inefficient as the number of agents grows and may also generate equilibria that yield significant variations in the outcomes for individual agents. We extend the functionality of PRISM-games to support *correlated equilibria*, in which players can coordinate through public signals, and introduce a novel optimality criterion of *social fairness*, which can be applied to both Nash and correlated equilibria. We show that correlated equilibria are easier to compute, are more equitable, and can also improve joint outcomes. We implement algorithms for both normal form games and the more complex case of multi-player concurrent stochastic games with temporal logic specifications. On a range of case studies, we demonstrate the benefits of our methods.

1 Introduction

Game-theoretic verification techniques can support the modelling and design of systems that comprise multiple agents operating in either a cooperative or competitive manner. In many cases, to effectively analyse these systems we also need to adopt a probabilistic approach to modelling, for example because agents operate in uncertain environments, use faulty hardware or unreliable communication mechanisms, or explicitly employ randomisation for coordination.

In these cases, *probabilistic model checking* provides a convenient unified framework for both formally modelling probabilistic multi-agent systems and specifying their required behaviour. In recent years, progress has been made in this direction for several models, including turn-based and concurrent stochastic

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 60–78, 2022.
https://doi.org/10.1007/978-3-030-99527-0_4

games (TSGs and CSGs), and for multiple temporal logics, such as rPATL [10] and its extensions [24]. Tool support has been developed, in the form of PRISM-games [22], and successfully applied to case studies across a broad range of areas.

Initially, the focus was on *zero-sum* specifications [24], which can be natural for systems whose participants have directly opposing goals, such as the defender and attacker in a security protocol minimising or maximising the probability of a successful attack, respectively. However, agents often have objectives that are distinct but not directly opposing, and may also want to cooperate to achieve these objectives. Examples include network protocols and multi-robot systems.

For these purposes, *Nash equilibria* (NE) have also been integrated into probabilistic model checking of CSGs [24], together with *social welfare* (SW) optimality criterion, resulting in social welfare Nash equilibria (SWNE). An SWNE comprises a strategy for each player in the game where no player has an incentive to deviate unilaterally from their strategy and the sum of the individual objectives over all players is maximised.

One key limitation of SWNE, however, is that, as these techniques are extended to support larger numbers of players [21], the efficiency and scalability of synthesising SWNE is significantly reduced. In addition, simply aiming to maximise the sum of individual objectives may not produce the best performing equilibrium, either collectively or individually; for example, they can offer higher gains for specific players, reducing the incentive of the other players to collaborate and instead motivating them to deviate from the equilibrium.

In this paper, we adopt a different approach and introduce, for the first time within formal verification, both *social fairness* as an optimality criterion and *correlated equilibria*, and the insights required to make these usable in practical applications. Social fairness (SF) is particularly novel, as it is inspired by similar concepts used in economics and distinct from the fairness notions employed in verification. Correlated equilibria (CE) [3], in which players are able to coordinate through public *signals*, are easier to compute than NE and can yield better outcomes. Social fairness, which minimises the differences between the objectives of individual players, can be considered for both CE and NE.

We first investigate these concepts for the simpler case of normal form games, illustrating their differences and benefits. We then extend the approach to the more powerful modelling formalism of CSGs and extend the temporal logic rPATL to formally specify agent objectives. We present algorithms to synthesise equilibria, using linear programming to find CE and a combination of backwards induction or value iteration for CSGs. We implement our approach in the PRISM-games tool [22] and demonstrate significant gains in computation time and that quantifiably more fair and useful strategies can by synthesised for a range of application domains. An extended version of this paper, with the complete model checking algorithm, is available [23].

Related work. Nash equilibria have been considered for concurrent systems in [18], where a temporal logic is proposed whose key operator is a novel path quantifier which asserts that a property holds on all Nash equilibrium computations of the system. There is no stochasticity and correlated equilibria are not

considered. In [2], a probabilistic logic that can express equilibria is formulated, along with complexity results, but no implementation has been provided.

The notion of fairness studied here is inspired by fairness of equilibria from economics [33,34] and aims to minimise the difference between the payoffs, as opposed to maximising the lowest payoff among the players in an NE [25]. Our notion of fairness can be thought of as a constraint applied to equilibria strategies, similar in style to social welfare, and used to select certain equilibria based on optimality. This is distinct from fairness used in verification of concurrent processes, where (strong) fairness refers to a property stating that, whenever a process is enabled infinitely often, it is executed infinitely often. This notion is typically defined as a constraint on infinite execution paths expressible in logics LTL and CTL* and needed to prove liveness properties. For probabilistic models, verification under fairness constraints has been formulated for Markov decision processes and the logic PCTL* [5,4]. For games on graphs, fairness conditions expressed as ω-regular winning conditions can be used to synthesise reactive processes [8]. Algorithms for strong transition fairness for ω-regular games have been recently studied in [6]. Both qualitative and quantitative approaches have been considered for verification under fairness constraints, but no equilibria.

2 Normal Form Games

We start by considering normal form games (NFGs), then define our equilibria concepts for these games, present algorithms and an implementation for computing them, and finally summarise some experimental results.

We first require the following notation. Let $Dist(X)$ denote the set of probability distributions over set X. For any vector $v \in \mathbb{R}^n$, we use $v(i)$ to refer to the ith entry of the vector. For any tuple $x = (x_1, \ldots, x_n) \in X^n$, element $x' \in X$ and $i \leqslant n$, we define the tuples $x_{-i} \stackrel{\text{def}}{=} (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ and $x_{-i}[x'] \stackrel{\text{def}}{=} (x_1, \ldots, x_{i-1}, x', x_{i+1}, \ldots, x_n)$.

Definition 1 (Normal form game). *A (finite, n-person)* normal form game *(NFG) is a tuple* $\mathsf{N} = (N, A, u)$ *where:* $N = \{1, \ldots, n\}$ *is a finite set of players;* $A = A_1 \times \cdots \times A_n$ *and* A_i *is a finite set of actions available to player* $i \in N$*;* $u = (u_1, \ldots, u_n)$ *and* $u_i \colon A \to \mathbb{R}$ *is a utility function for player* $i \in N$*.*

We fix an NFG $\mathsf{N} = (N, A, u)$ for the remainder of this section. In a play of N, each player $i \in N$ chooses an action from the set A_i at the same time. If each player i chooses a_i, then the utility received by player j equals $u_j(a_1, \ldots, a_n)$. We next define the *strategies* for players of N and *strategy profiles* comprising a strategy for each player. We also define *correlated profiles*, which allow the players to coordinate their choices through a (probabilistic) *public signal*.

Definition 2 (Strategy and profile). *A* strategy σ_i *for player* i *is an element of* $\Sigma_i = Dist(A_i)$ *and a* strategy profile σ *is an element of* $\Sigma_{\mathsf{N}} = \Sigma_1 \times \cdots \times \Sigma_n$*.*

For strategy σ_i of player i, the *support* is the set of actions $\{a_i \in A_i \mid \sigma_i(a_i) > 0\}$ and the support of a profile is the product of the supports of the strategies.

Definition 3 (Correlated profile). *A* correlated profile *is a tuple* (τ, ς) *comprising* $\tau \in \mathit{Dist}(D)$*, where* $D = D_1 \times \cdots \times D_n$*,* D_i *is a finite set of* signals *for player* i*, and* $\varsigma = (\varsigma_1, \ldots, \varsigma_n)$*, where* $\varsigma_i \colon D_i \to A_i$*.*

For a correlated profile (τ, ς), the public signal τ is a joint distribution over signals D_i for each player i such that, if player i receives the signal $d_i \in D_i$, then it chooses action $\varsigma_i(d_i)$. We can consider any correlated profile (τ, ς) as a *joint strategy*, i.e., a distribution over $A_1 \times \cdots \times A_n$ where:

$$(\tau, \varsigma)(a_1, \ldots, a_n) = \sum\{\tau(d_1, \ldots, d_n) \mid d_i \in D_i \wedge \varsigma(d_i) = a_i \text{ for all } i \in N\}\,.$$

Conversely, any joint strategy $\tau \in \mathit{Dist}(A_1 \times \cdots \times A_n)$ can be considered as a correlated profile (τ, ς) where $D_i = A_i$ and ς_i is the identity function for $i \in N$.

Any strategy profile σ can be mapped to an equivalent correlated profile (in which τ is the joint distribution $\sigma_1 \times \cdots \times \sigma_n$ and ς_i is the identity function). On the other hand, there are correlated profiles with no equivalent strategy profile. Under profile σ and correlated profile (τ, ς) the expected utilities of player i are:

$$\begin{aligned} u_i(\sigma) &\stackrel{\text{def}}{=} \textstyle\sum_{(a_1,\ldots,a_n)\in A} u_i(a_1, \ldots, a_n) \cdot \left(\prod_{j=1}^{n} \sigma_j(a_j)\right) \\ u_i(\tau, \varsigma) &\stackrel{\text{def}}{=} \textstyle\sum_{(d_1,\ldots,d_n)\in D} \tau(d_1, \ldots, d_n) \cdot u_i(\varsigma_1(d_1), \ldots, \varsigma_n(d_n))\,. \end{aligned}$$

Example 1. Consider the two-player NFG where $A_i = \{a_1^i, a_2^i\}$ and a correlated profile corresponding to the joint distribution $\tau \in \mathit{Dist}(A_1 \times A_2)$ where $\tau(a_1^1, a_2^1) = \tau(a_1^2, a_2^2) = 0.5$. Under this correlated profile the players share a fair coin and both choose their first action if the coin is heads and their second action otherwise. This has no equivalent strategy profile. ■

Optimal equilibria of NFGs. We now introduce the notions of *Nash equilibrium* [27] and *correlated equilibrium* [3], as well as different definitions of optimality for these equilibria: *social welfare* and *social fairness*. Using the notation introduced above for tuples, for any profile σ and strategy $\sigma_i^\star$, the strategy tuple σ_{-i} corresponds to σ with the strategy of player i removed and $\sigma_{-i}[\sigma_i^\star]$ to the profile σ after replacing player i's strategy with $\sigma_i^\star$.

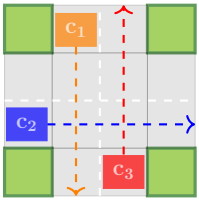
Definition 4 (Best response). *For a profile* σ *and correlated profile* (τ, ς)*, a* best response *for player* i *to* σ_{-i} *and* (τ, ς_{-i}) *are, respectively:*

- *a strategy* $\sigma_i^\star$ *for player* i *such that* $u_i(\sigma_{-i}[\sigma_i^\star]) \geqslant u_i(\sigma_{-i}[\sigma_i])$ *for all* $\sigma_i \in \Sigma_i$*;*
- *a function* $\varsigma_i^\star \colon D_i \to A_i$ *for player* i *such that* $u_i(\tau, \varsigma_{-i}[\varsigma_i^\star]) \geqslant u_i(\tau, \varsigma_{-i}[\varsigma_i])$ *for all functions* $\varsigma_i \colon D_i \to A_i$*.*

Definition 5 (NE and CE). *A strategy profile* $\sigma^\star$ *is a* Nash equilibrium *(NE) and a correlated profile* $(\tau, \varsigma^\star)$ *is a* correlated equilibrium *(CE) if:*

- $\sigma_i^\star$ *is a best response to* $\sigma_{-i}^\star$ *for all* $i \in N$*;*
- $\varsigma_i^\star$ *is a best response to* $(\tau, \varsigma_{-i}^\star)$ *for all* $i \in N$*;*

respectively. We denote by Σ^N *and* Σ^C *the set of NE and CE, respectively.*



α	$u_1(\alpha)$	$u_2(\alpha)$	$u_3(\alpha)$
(pro_1, pro_2, pro_3)	-1000	-1000	-100
(pro_1, pro_2, yld_3)	-1000	-100	-5
(pro_1, yld_2, pro_3)	5	-5	5
(pro_1, yld_2, yld_3)	5	-5	-5
(yld_1, pro_2, pro_3)	-5	-1000	-100
(yld_1, pro_2, yld_3)	-5	5	-5
(yld_1, yld_2, pro_3)	-5	-5	5
(yld_1, yld_2, yld_3)	-10	-10	-10

Fig. 1: Example: Cars at an intersection and the corresponding NFG.

Any NE of N is also a CE, while there can exist CEs that cannot be represented by a strategy profile and therefore are not NEs. For each class of equilibria, NE and CE, we introduce two optimality criteria, the first maximising *social welfare* (SW), defined as the *sum* of the utilities, and the second maximising *social fairness* (SF), which minimises the *difference* between the players' utilities. Other variants of fairness have been considered for NE, such as in [25], where the authors seek to maximise the lowest utility among the players.

Definition 6 (SW and SF). *An equilibrium $\sigma^\star$ is a* social welfare *(SW) equilibrium if the sum of the utilities of the players under $\sigma^\star$ is maximal over all equilibria, while $\sigma^\star$ is a* social fair *(SF) equilibrium if the difference between the player's utilities under $\sigma^\star$ is minimised over all equilibria.*

We can also define the dual concept of *cost equilibria* [24], where players try to minimise, rather than maximise, their expected utilities by considering equilibria of the game $\mathsf{N}^- = (N, A, -u)$ in which the utilities of N are negated.

Example 2. Consider the scenario, based on an example from [32], where three cars meet at an intersection and want to proceed as indicated by the arrows in Figure 1. Each car can either *proceed* or *yield*. If two cars with intersecting paths proceed, then there is an accident. If an accident occurs, the car having the right of way, i.e., the other car is to its right, has a utility of -100 and the car that should yield has a utility of -1000. If a car proceeds without causing an accident, then its utility is 5 and the cars that yield have a utility of -5. If all cars yield, then, since this delays all cars, all have utility -10. The 3-player NFG is given in Figure 1. Considering the different optimal equilibria of the NFG:

- the SWNE and SWCE are the same: for c_2 to yield and c_1 and c_3 to proceed, with the expected utilities $(5, -5, 5)$;
- the SFNE is for c_1 to yield with probability 1, c_2 to yield with probability 0.863636 and c_3 to yield with probability 0.985148, with the expected utilities $(-9.254050, -9.925742, -9.318182)$;
- the SFCE gives a joint distribution where the probability of c_2 yielding and of c_1 and c_3 yielding are both 0.5 with the expected utilities $(0, 0, 0)$.

Modifying u_2 such that $u_2(pro_1, pro_2, pro_3) = -4.5$ to, e.g., represent a reckless driver, the SWNE becomes for c_1 and c_3 to yield and c_2 to proceed with the expected utilities $(-5, 5, -5)$, while the SWCE is still for c_2 to yield and c_1 and c_3 to proceed. The SFNE and SFCE also do not change. ■

Algorithms for computing equilibria. Before we give our algorithm to compute correlated equilibria, we briefly describe the approach of [21,24] for Nash equilibria computation that this paper builds upon. Finding NE in two-player NFGs is in the class of *linear complementarity* problems (LCPs) and we follow the algorithm presented in [24], which reduces the problem to SMT via labelled polytopes [28] by considering the regions of the strategy profile space, iteratively reducing the search space as positive probability assignments are found and added as restrictions on this space. To find SWNE and SFNE, we can enumerate all NE and then find the optimal NE.

When there are more than two players, computing NE values becomes a more complex task, as finding NE within a given support no longer reduces to a linear programming (LP) problem. In [21] we presented an algorithm using support enumeration [31], which exhaustively examines all sub-regions, i.e., supports, of the strategy profile space, one at a time, checking whether that sub-region contains NEs. For each support, finding SWNE can be reduced to a *nonlinear programming problem* [21]. This nonlinear programming problem can be modified to find SFNE in each support, similarly to how the LP problem for SWCEs is modified to find SFCEs below.

In the case of CE we can first find a joint strategy for the players, i.e., a distribution over the action tuples, which, as explained above, can then be mapped to a correlated profile. A SWCE can be found by solving the following LP problem. Maximise: $\sum_{i\in N}\sum_{\alpha\in A} u_i(\alpha)\cdot p_\alpha$ subject to:

$$\sum_{\alpha_{-i}\in A_{-i}}(u_i(\alpha_{-i}[a_i]) - u_i(\alpha_{-i}[a_i'])) \cdot p_{\alpha_{-i}[a_i]} \geqslant 0 \tag{1}$$

$$0 \leqslant p_\alpha \leqslant 1 \tag{2}$$

$$\sum_{\alpha\in A} p_\alpha = 1 \tag{3}$$

for all $i \in N$, $\alpha \in A$, $a_i, a_i' \in A_i$, $\alpha_{-i} \in A_{-i}$ where $A_{-i} \stackrel{\text{def}}{=} \{\alpha_{-i} \mid \alpha \in A\}$. The variables p_α represent the probability of the joint strategy corresponding to the correlated profile selecting the action-tuple α. The above LP has $|A|$ variables, one for each action-tuple, and $\sum_{i\in N}(|A_i|^2 - |A_i|) + |A| + 1$ constraints. Computation of SFCE can be reduced to the following optimisation problem. Minimise $p^{\max} - p^{\min}$ subject to: (1), (2) and (3) together with:

$$p^i = \sum_{\alpha\in A} p_\alpha \cdot u_i(\alpha) \tag{4}$$

$$\left(\wedge_{m\in N} p^i \geqslant p^m\right) \rightarrow (p^{\max} = p^i) \tag{5}$$

$$\left(\wedge_{m\in N} p^i \leqslant p^m\right) \rightarrow (p^{\min} = p^i) \tag{6}$$

for all $i \in N$, $m \neq i$, $\alpha \in A$, $a_j, a_l \in A_i$, $\alpha_{-i} \in A_{-i}$. Again, the variables p_α in the program represent the probability of the players playing the joint action α. The constraint (4) requires p^i to equal the utility of player i. The constraints (5) and (6) set $p^{\max}$ and $p^{\min}$ as the maximum and minimum values within the utilities of the players, respectively. Given we use the constraints (1), (2) and (3), we start with the same number of variables and constraints as needed to compute SWCEs and incur an additional $|N|+2$ variables and $3\cdot|N|$ constraints.

Game	Players	$\lvert A_i \rvert$	$\lvert A \rvert$	NE		CE	
				Supports	SW	SW	SF
Majority voting games	2	4	16	225	0.07	0.02	0.08
		6	36	3,969	0.1	0.02	0.1
		8	64	65,025	0.4	0.03	0.3
		10	100	1,046,529	5.8	0.07	0.7
	3	3	27	343	1.2	0.07	0.1
		4	81	3,375	25.8	0.08	0.3
Covariant games	3	3	27	343	8.7	0.08	1.7
		4	81	3,375	598.5	0.08	2.9
	8	2	256	6,561	TO	0.3	TO
		3	6,561	5,764,801	TO	22.8	TO
	10	2	1,024	59,049	TO	1.2	TO

Table 1: Times (s) for synthesis of equilibria in NFGs (timeout 30 mins).

Implementation. To find SWNE or SFNE of two-player NFGs, we adopt a similar approach to [24], using labelled polytopes to characterise and find NE values through a reduction to SMT in both Z3 [13] and Yices [14]. As an optimised precomputation step, when possible we also search for and filter out *dominated strategies*, which speeds up the computation and reduces solver calls.

For NFGs with more than two players, solving the nonlinear programming problem based on support enumeration has been implemented in [21] using a combination of the SMT solver Z3 [13] and the nonlinear optimisation suite Ipopt [38]. To mitigate the inefficiencies of an SMT solver for such problems, we used Z3 to filter out unsatisfiable support assignments with a timeout and then Ipopt is called to find SWNE values using an interior-point filter line-search algorithm [39]. To speed up the overall computation, the support assignments are analysed in parallel. Computing SFNE increases the complexity of the nonlinear program and, due to the inefficiency in this approach [21], we have not extended the implementation to compute SFNE.

As shown above, computing SWCE for NFGs reduces to solving an LP, and we implement this using either the optimisation solver Gurobi [17] or the SMT solver Z3 [13]. In the case of SFCE, the constraints (5) and (6) include implications, and therefore the problem does not reduce directly to an LP. When using Z3, we can encode these constraints directly as it supports assertions that combine inequalities with logical implications, a feature that linear solvers such as Gurobi do not have. Section 5 discusses implementing SFCE computation in Gurobi. Both solvers support the specification of *lower priority* or *soft* objectives, which makes it possible to have a consistent ordering for the players' payoffs in cases where multiple equilibria exist.

Efficiency and scalability. Table 1 presents experimental results for solving a selection of NFGs randomly generated with GAMUT [29], using Gurobi for SWCE and NE of two-player NFGs, Z3 for SFCE and both Ipopt and Z3 for NFGs of more than two players, and running on a 2.10GHz Intel Xeon Gold with 32GB of JVM memory. For each instance, Table 1 lists the number of players, actions for each player, joint actions and supports that need to be enumerated when finding NE, as well as the time to find SWNEs, SWCEs and SFCEs (the time for finding SFNEs of two-player games is the same as for SWNEs). As the results demonstrate, due to a simpler problem being solved and the fact that we

do not need to enumerate the solutions, computing CEs scales far better than NEs as the number of players and actions increases. Finding NEs in games with more than two players is particularly hard as the constraints are nonlinear. We also see that SFCE computation is slower than SWCE, which is caused by the additional variables and constraints required when finding SFCE and using Z3 rather than Gurobi for the solver.

3 Concurrent Stochastic Games

We now further develop our approach to support concurrent stochastic games (CSGs) [36], in which players repeatedly make simultaneous action choices that cause the game's state to be updated probabilistically. We extend the previously introduced definitions of optimal equilibria to such games, focusing on subgame-perfect equilibria, which are equilibria in every state of a CSG. We then present algorithms to reason about and synthesise such equilibria.

Definition 7 (Concurrent stochastic game). *A* concurrent stochastic multi-player game *(CSG) is a tuple* $\mathsf{G} = (N, S, \bar{S}, A, \Delta, \delta, AP, L)$ *where:*

- $N = \{1, \ldots, n\}$ *is a finite set of players;*
- S *is a finite set of states and* $\bar{S} \subseteq S$ *is a set of initial states;*
- $A = (A_1 \cup \{\bot\}) \times \cdots \times (A_n \cup \{\bot\})$ *and* A_i *is a finite set of actions available to player* $i \in N$ *and* $\bot$ *is an idle action disjoint from the set* $\cup_{i=1}^{n} A_i$*;*
- $\Delta\colon S \to 2^{\cup_{i=1}^{n} A_i}$ *is an action assignment function;*
- $\delta\colon (S \times A) \to Dist(S)$ *is a (partial) probabilistic transition function;*
- AP *is a set of atomic propositions and* $L\colon S \to 2^{AP}$ *is a labelling function.*

For the remainder of this section we fix a CSG G as in Definition 7. The game G starts in one of its initial states $\bar{s} \in \bar{S}$ and, supposing G is in a state s, then each player i of G chooses an action from the set that are available, defined as $A_i(s) \stackrel{\text{def}}{=} \Delta(s) \cap A_i$ if $\Delta(s) \cap A_i$ is non-empty and $A_i(s) \stackrel{\text{def}}{=} \{\bot\}$ otherwise. Supposing each player chooses a_i, then the game transitions to state s' with probability $\delta(s, (a_1, \ldots, a_n))$. To enable quantitative analysis of G we augment it with *reward structures*, which are tuples $r{=}(r_A, r_S)$ of an action reward function $r_A\colon S \times A \to \mathbb{R}$ and state reward function $r_S\colon S \to \mathbb{R}$.

A *path* of G is a sequence $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots$ where $s_k \in S$, $\alpha_k = (a_1^k, \ldots, a_n^k) \in A$, $a_i^k \in A_i(s_k)$ for $i \in N$ and $\delta(s_k, \alpha_k)(s_{k+1}) > 0$ for all $k \geqslant 0$. We denote by $FPaths_{\mathsf{G},s}$ and $IPaths_{\mathsf{G},s}$ the sets of finite and infinite paths starting in state s of G respectively and drop the subscript s when considering all finite and infinite paths of G. As for NFGs, we can define *strategies* of G that resolve the choices of the players. Here, a strategy for player i is a function $\sigma_i\colon FPaths_{\mathsf{G}} \to Dist(A_i \cup \{\bot\})$ such that, if $\sigma_i(\pi)(a_i){>}0$, then $a_i \in A_i(last(\pi))$ where $last(\pi)$ is the final state of π. Furthermore, we can define strategy profiles, correlated profiles and joint strategies analogously to Definitions 2 and 3.

The utility of a player i of G is defined by a random variable $X_i \colon \mathit{IPaths}_{\mathsf{G}} \to \mathbb{R}$ over infinite paths. For a profile[4] σ and state s, using standard techniques [20], we can construct a probability measure $\mathit{Prob}^{\sigma}_{\mathsf{G},s}$ over the paths with initial state s corresponding to σ, denoted $\mathit{IPaths}^{\sigma}_{\mathsf{G},s}$ and the expected value $\mathbb{E}^{\sigma}_{\mathsf{G},s}(X_i)$ of player i's utility from s under σ. Given utilities $X_1, \ldots, X_n$ for all the players of G, we can then define NE and CE (see Definition 5) as well as the restricted classes of SW and SF equilibria as for NFGs (see Definition 6). Following [24,21], we focus on *subgame-perfect* equilibria [30], which are equilibria in *every state* of G.

Nonzero-sum properties. As in [24] (for two-player CSGs) and [21] (for n-player CSGs) we can specify equilibria-based properties using temporal logic. For simplicity, we restrict attention to nonzero-sum properties without nesting, allowing for the specification of NE and CE against either SW or SF optimality.

Definition 8 (Nonzero-sum specifications). *The syntax of nonzero-sum specifications θ for CSGs is given by the grammar:*

$$\begin{aligned} \phi &\coloneqq \langle\!\langle \mathbb{C} \rangle\!\rangle (\star_1, \star_2)_{\mathrm{opt} \sim x}(\theta) \\ \theta &\coloneqq \mathtt{P}[\,\psi\,] + \cdots + \mathtt{P}[\,\psi\,] \;\mid\; \mathtt{R}^{r}[\,\rho\,] + \cdots + \mathtt{R}^{r}[\,\rho\,] \\ \psi &\coloneqq \mathtt{X}\,\mathtt{a} \;\mid\; \mathtt{a}\;\mathtt{U}^{\leqslant k}\;\mathtt{a} \;\mid\; \mathtt{a}\;\mathtt{U}\;\mathtt{a} \\ \rho &\coloneqq \mathtt{I}^{=k} \;\mid\; \mathtt{C}^{\leqslant k} \;\mid\; \mathtt{F}\;\mathtt{a} \end{aligned}$$

where $\mathbb{C} = C_1 {:} \cdots {:} C_m$, $C_1, \ldots, C_m$ are coalitions of players such that $C_i \cap C_j = \varnothing$ for all $1 \leqslant i \neq j \leqslant m$ and $\cup_{i=1}^{m} C_i = N$, $(\star_1, \star_2) \in \{\mathrm{NE}, \mathrm{CE}\} \times \{\mathrm{SW}, \mathrm{SF}\}$, $\mathrm{opt} \in \{\min, \max\}$, $\sim \in \{<, \leqslant, \geqslant, >\}$, $x \in \mathbb{Q}$, r is a reward structure, $k \in \mathbb{N}$ and $\mathtt{a}$ is an atomic proposition.

The nonzero-sum formulae of Definition 8 extend the logic of in [24,21] in that we can now specify the type of equilibria, NE or CE, and optimality criteria, SW or SF. A probabilistic formula $\langle\!\langle C_1 {:} \cdots {:} C_m \rangle\!\rangle (\star_1, \star_2)_{\max \sim x}(\mathtt{P}[\,\psi_1\,] + \cdots + \mathtt{P}[\,\psi_m\,])$ is true in a state if, when the players form the coalitions $C_1, \ldots, C_m$, there is a subgame-perfect equilibrium of type $\star_1$ meeting the optimality criterion $\star_2$ for which the *sum* of the values of the objectives $\mathtt{P}[\,\psi_1\,], \ldots, \mathtt{P}[\,\psi_m\,]$ for the coalitions $C_1, \ldots, C_m$ satisfies $\sim x$. The objective ψ_i of coalition C_i is either a next ($\mathtt{X}\,\mathtt{a}$), bounded until ($\mathtt{a}_1\;\mathtt{U}^{\leqslant k}\;\mathtt{a}_2$) or until ($\mathtt{a}_1\;\mathtt{U}\;\mathtt{a}_2$) formula, with the usual equivalences, e.g., $\mathtt{F}\;\mathtt{a} \equiv \mathtt{true}\;\mathtt{U}\;\mathtt{a}$.

For a reward formula $\langle\!\langle C_1 {:} \cdots {:} C_m \rangle\!\rangle (\star_1, \star_2)_{\mathrm{opt} \sim x}(\mathtt{R}^{r_1}[\,\rho_1\,] + \cdots + \mathtt{R}^{r_m}[\,\rho_m\,])$ the meaning is similar; however, here the objective of coalition C_i refers to a reward formula ρ_i with respect to reward structure r_i and this formula is either a bounded instantaneous reward ($\mathtt{I}^{=k}$), bounded accumulated reward ($\mathtt{C}^{\leqslant k}$) or reachability reward ($\mathtt{F}\;\mathtt{a}$).

For formulae of the form $\langle\!\langle C_1 {:} \cdots {:} C_m \rangle\!\rangle (\star_1, \star_2)_{\min \sim x}(\theta)$, the dual notions of cost equilibria are considered. We also allow *numerical* queries of the form $\langle\!\langle C_1 {:} \cdots {:} C_m \rangle\!\rangle (\star_1, \star_2)_{\mathrm{opt}=?}(\theta)$, which return the sum of the optimal subgame-perfect equilibrium's values.

[4] We can also construct such a probability measure and expected value given a correlated profile or joint strategy.

Model checking nonzero-sum specifications. Similarly to [24,21], to allow model checking of nonzero-sum properties we consider a restricted class of CSGs. We make the following assumption, which can be checked using graph algorithms with time complexity quadratic in the size of the state space [1].

Assumption 1. *For each subformula* $\mathtt{P}[\,\mathtt{a}_1\ \mathtt{U}\ \mathtt{a}_2\,]$*, a state labelled* $\neg\mathtt{a}_1 \vee \mathtt{a}_2$ *is reached with probability 1 from all states under all strategy profiles and correlated profiles. For each subformula* $\mathtt{R}^r[\,\mathtt{F}\ \mathtt{a}\,]$*, a state labelled* $\mathtt{a}$ *is reached with probability 1 from all states under all strategy profiles and correlated profiles.*

We now show how to compute the optimal values of a nonzero-sum formula $\phi = \langle\!\langle C_1 : \cdots : C_m \rangle\!\rangle(\star_1, \star_2)_{\mathrm{opt}\sim x}(\theta)$ when opt = max. The case when opt = min can be computed by negating all utilities and maximising.

The model checking algorithm broadly follows those presented in [24,21], with the differences described below. The problem is reduced to solving an m-player *coalition game* $\mathsf{G}^{\mathcal{C}}$ where $\mathcal{C} = \{C_1, \ldots, C_m\}$ and the choices of each player i in $\mathsf{G}^{\mathcal{C}}$ correspond to the choices of the players in coalition C_i in G. Formally, we have the following definition in which, without loss of generality, we assume $\mathcal{C}$ is of the form $\{\{1, \ldots, n_1\}, \{n_1{+}1, \ldots n_2\}, \ldots, \{n_{m-1}{+}1, \ldots n_m\}\}$ and let $j_{\mathcal{C}}$ denote player j's position in its coalition.

Definition 9 (Coalition game). *For CSG* $\mathsf{G} = (N, S, \bar{S}, A, \Delta, \delta, AP, L)$ *and partition* $\mathcal{C} = \{C_1, \ldots, C_m\}$ *of the players into* m *coalitions, we define the* coalition game $\mathsf{G}^{\mathcal{C}} = (\{1, \ldots, m\}, S, \bar{S}, A^{\mathcal{C}}, \Delta^{\mathcal{C}}, \delta^{\mathcal{C}}, AP, L)$ *as an* m*-player CSG where:*

- $A^{\mathcal{C}} = (A^{\mathcal{C}}_1 \cup \{\bot\}) \times \cdots \times (A^{\mathcal{C}}_m \cup \{\bot\})$*;*
- $A^{\mathcal{C}}_i = \left(\prod_{j \in C_i} (A_j \cup \{\bot\}) \setminus \{(\bot, \ldots, \bot)\}\right)$ *for all* $1 \leqslant i \leqslant m$*;*
- *for any* $s \in S$ *and* $1 \leqslant i \leqslant m$*:* $a^{\mathcal{C}}_i \in \Delta^{\mathcal{C}}(s)$ *if and only if either* $\Delta(s) \cap A_j = \varnothing$ *and* $a^{\mathcal{C}}_i(j_{\mathcal{C}}) = \bot$ *or* $a^{\mathcal{C}}_i(j_{\mathcal{C}}) \in \Delta(s)$ *for all* $j \in C_i$*;*
- *for any* $s \in S$ *and* $(a^{\mathcal{C}}_1, \ldots, a^{\mathcal{C}}_m) \in A^{\mathcal{C}}$*:* $\delta^{\mathcal{C}}(s, (a^{\mathcal{C}}_1, \ldots, a^{\mathcal{C}}_m)) = \delta(s, (a_1, \ldots, a_n))$ *where for* $i \in M$ *and* $j \in C_i$ *if* $a^{\mathcal{C}}_i = \bot$*, then* $a_j = \bot$ *and otherwise* $a_j = a^{\mathcal{C}}_i(j_{\mathcal{C}})$*.*

If all the objectives in θ are finite-horizon, *backward induction* [35,27] can be applied to compute (precise) optimal equilibria values with respect to the criterion $\star_2$ and equilibria type $\star_1$. On the other hand, if all the objectives are infinite-horizon, *value iteration* [9] can be used to approximate optimal equilibria values and, when there is a combination of objectives, the game under study is modified in a standard manner to make all objectives infinite-horizon.

Backward induction and value iteration over the CSG $\mathsf{G}^{\mathcal{C}}$ both work by iteratively computing new values for each state s of $\mathsf{G}^{\mathcal{C}}$. The values for each state, in each iteration, are found by computing optimal equilibria values of an NFG N whose utility function is derived from the outgoing transition probabilities from s in the CSG and the values computed for successor states of s in the previous iteration. The difference here, with respect to [21], is that the NFGs are solved for the additional equilibria and optimality conditions considered in this paper, which we compute using the algorithms presented in Section 2.

Algorithm for probabilistic until. Because of space limitations, we only present here the details of value iteration for (unbounded) probabilistic until, i.e.,

for $\phi = \langle\langle C_1 : \cdots : C_m \rangle\rangle(\star_1, \star_2)_{\max\sim x}(\theta)$ where $\theta = \mathtt{P}[\,\mathsf{a}_1^1 \;\mathtt{U}\; \mathsf{a}_2^1\,] + \cdots + \mathtt{P}[\,\mathsf{a}_1^m \;\mathtt{U}\; \mathsf{a}_2^m\,]$. The complete model checking algorithm can be found in [23].

Following [21], we use $\mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, \theta, n)$ to denote the vector of computed values, at iteration n, in state s of $\mathsf{G}^{\mathcal{C}}$ for optimality criterion $\star_2$ (SW or SF), equilibria type $\star_1$ (NE or CE) and (until) objectives θ. We also use $\mathbf{1}_m$ and $\mathbf{0}_m$ to denote a vector of size m whose entries all equal to 1 or 0, respectively. For any set of states S', atomic proposition a and state s we let $\eta_{S'}(s)$ equal 1 if $s \in S'$ and 0 otherwise, and $\eta_{\mathsf{a}}(s)$ equal 1 if $\mathsf{a} \in L(s)$ and 0 otherwise.

Each step of value iteration also keeps track of two sets $D, E \subseteq M$, where $M = \{1, \ldots, m\}$ are the players of $\mathsf{G}^{\mathcal{C}}$. We use D for the subset of players that have already reached their goal (by satisfying a_2^i) and E for the players who can no longer can satisfy their goal (having reached a state that fails to satisfy a_1^i). It can then be ensured that their payoffs no longer change and are set to 1 or 0, respectively. In these cases, we effectively consider a modified game where, although the payoffs for these players are set, we still need to take their strategies into account in order to guarantee an optimal equilibrium.

Optimal values for all states s in the CSG $\mathsf{G}^{\mathcal{C}}$ can be computed as the following limit: $\mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, \theta) = \lim_{n\to\infty} \mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, \theta, n)$, where $\mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, \theta, n) = \mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, \varnothing, \varnothing, \theta, n)$ and, for any $D, E \subseteq M$ such that $D \cap E = \varnothing$:

$$\mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, D, E, \theta, n) = \begin{cases} (\eta_D(1), \ldots, \eta_D(m)) & \text{if } D \cup E = M \\ (\eta_{\mathsf{a}_2^1}(s), \ldots, \eta_{\mathsf{a}_2^m}(s)) & \text{else if } n = 0 \\ \mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, D \cup D', E, \theta, n) & \text{else if } D' \neq \varnothing \\ \mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s, \star_1, \star_2, D, E \cup E', \theta, n) & \text{else if } E' \neq \varnothing \\ \mathit{val}(\mathsf{N}, \star_1, \star_2) & \text{otherwise} \end{cases}$$

where $D' = \{l \in M \backslash (D \cup E) \mid \mathsf{a}_2^l \in L(s)\}$, $E' = \{l \in M \backslash (D \cup E) \mid \mathsf{a}_1^l \notin L(s) \text{ and } s \in L(\mathsf{a}_2^l)\}$ and $\mathit{val}(\mathsf{N}, \star_1, \star_2)$ equals optimal values of the NFG $\mathsf{N} = (M, A^{\mathcal{C}}, u)$ with respect to the criterion $\star_2$ and of equilibria type $\star_1$ in which for any $1 \leqslant l \leqslant m$ and $\alpha \in A^{\mathcal{C}}$:

$$u_l(\alpha) = \begin{cases} 1 & \text{if } l \in D \\ 0 & \text{else if } l \in E \\ \sum_{s' \in S} \delta^{\mathcal{C}}(s, \alpha)(s') \cdot v_{n-1}^{s',l} & \text{otherwise} \end{cases}$$

and $(v_{n-1}^{s',1}, v_{n-1}^{s',2}, \ldots, v_{n-1}^{s',m}) = \mathtt{V}_{\mathsf{G}^{\mathcal{C}}}(s', \star_1, \star_2, D, E, \theta, n-1)$ for all $s' \in S$.

Since this paper considers equilibria for any number of coalitions (in particular, for more than two), the above follows the algorithm of [21] in the way that it keeps track of the coalitions that have satisfied their objective (D) or can no longer do so (E). By contrast the CSG algorithm of [24] was limited to two coalitions, which enabled the exploitation of efficient MDP analysis techniques for such coalitions. As explained in [21], in such a scenario we cannot reduce the analysis from an n-coalition game to an $(n-1)$-coalition game, as otherwise we would give one of the remaining coalitions additional power (the action choices of the coalition that has satisfied their objective or can no longer do so), which would therefore give this coalition an advantage over the other coalitions.

Strategy synthesis. As in [24,21] we can extend the model checking algorithm to perform *strategy synthesis*, generating a witness (i.e., a profile or joint strategy) representing the corresponding optimal equilibrium. This is achieved by storing the profile or joint strategy for the NFG solved in each state. Both the profiles and joint strategies require finite memory and are probabilistic. Memory is required as choices change after a path formula becomes true or a target is reached and to keep track of the step bound in finite-horizon properties. Randomisation is required for both NE and CE of NFGs.

Correctness and complexity. The correctness of the algorithm follows directly from [24,21], as changing the class of equilibria or optimality criterion does not change the proof. The complexity of the algorithm is linear in the formula size and value iteration requires finding optimal NE or CE for an NFG in each state of the model. Computing NEs of an NFG with two (or more) players is PPAD-complete [12,11], while finding optimal CEs of an NFG is in P [15].

4 Case Studies and Experimental Results

We have developed an implementation of our techniques for equilibria synthesis on CSGs, described above, building on top of the PRISM-games [22] model checker. Our implementation extends the tool's existing support for construction and analysis of CSGs, which is contained within its sparse matrix based "explicit" engine written in Java. We have considered a range of CSG case studies (supplementary material can be found at [40]). Below, we summarise the efficiency and scalability of our approach, again running on a 2.10GHz Intel Xeon Gold with 32GB JVM memory, and then describe our findings on individual case studies.

Efficiency and scalability. Table 2 summarises the performance of our implementation on the case studies that we have considered. It shows the statistics for each CSG, and the time taken to build it and perform equilibria synthesis, for several different variants (NE vs. CE, SW vs. SF). Comparing the efficiency of synthesising SWNE and SWCE, we see that the latter is typically much faster. For two-player NE, the social fairness variant is no more expensive to compute as we enumerate all NEs. For CE, which uses Z3 rather than Gurobi for finding SF, we note that, although Z3 is able to find optimal equilibria, it is not primarily developed as an optimisation suite, and therefore generally performs poorly in comparison with Gurobi. The benefits of the social fair equilibria, in terms of the values yielded for individual players, are discussed in the in-depth coverage of the different case studies below.

Aloha. In this case study, introduced in [24], a number of users try to send packets using the slotted Aloha protocol. We suppose that each user has one packet to send and, in a time slot, if k users try and send their packet, then the probability that each packet is successfully sent is q/k where $q \in [0, 1]$. If a user fails to send a packet, then the number of slots it waits before resending the packet is set according to Aloha's exponential backoff scheme. The scheme requires that each user maintains a backoff counter, which it increases each time

Case study & property [parameters]	Players	$\star_1,\star_2$	Param. values	CSG statistics States	CSG statistics Trans.	Constr. time(s)	Verif. time (s)
Aloha $(\star_1,\star_2)_{\min=?}(\mathtt{R}^{time}[\mathtt{F}\ \mathtt{s}_i])$ $[b_{max}, q]$	2	NE,SW	4,0.8	2,778	6,285	0.1	2.2
		CE,SW					2.1
		NE,SF					2.1
		CE,SF					23.3
	3	CE,SW	4,0.8	107,799	355,734	3.0	80.1
		CE,SF					114.6
	4	NE,SW	2,0.8	68,689	161,904	1.9	1042.9
		CE,SW					58.8
Aloha $(\star_1,\star_2)_{\max=?}(\mathtt{P}_{\max=?}[\mathtt{F}\ \mathtt{s}_i \wedge t \leqslant D])$ $[b_{max}, q, D]$	4	NE,SW	2,0.8,8	159,892	388,133	3.9	1027.5
		CE,SW					224.5
	5	CE,SW	2,0.8,8	1,797,742	5,236,655	54.5	4,936.8
		CE,SF					TO
Power control $(\star_1,\star_2)_{\max=?}(\mathtt{R}^{r}[\mathtt{F}\ \mathtt{e}_i])$ $[pow_{max}, e_{max}, q_{fail}]$	2	NE,SW	8,40,0.2	32,812	260,924	1.2	564.5
		NE,SF					566.3
		CE,SW					177.9
	3	CE,SW	5,15,0.2	42,156	740,758	3.5	147.0
		CE,SF					TO
Public good $(\star_1,\star_2)_{\max=?}(\mathtt{R}^{c}[\mathtt{I}^{=r_{max}}])$ $[f, r_{max}]$	3	NE,SW	2.5,3	16,202	35,884	0.8	27.5
		CE,SW					1.9
	4	NE,SW	3,3	391,961	923,401	13.0	71.9
		CE,SW					35.3
	5	CE,SW	4,2	59,294	118,342	3.1	5.2
Investors $(\star_1,\star_2)_{\max=?}(\mathtt{R}^{prof}[\mathtt{F}\ \mathtt{cin}_i])$ $[p_{bar}, months]$	2	CE,SW	0.2,8	71,731	315,804	2.4	47.5
		CE,SF					2,401.9
	3	CE,SW	0.2,5	83,081	462,920	3.6	79.3
		CE,SF					861.2

Table 2: Statistics for a set of CSG verification instances (timeout 2 hours).

there is a packet failure (up to $b_{\max}$) and, if the counter equals k and a failure occurs, randomly chooses the slots to wait from $\{0, 1, \ldots, 2^k-1\}$.

We suppose that the objective of each user is to minimise the expected time to send their packet, which is represented by the nonzero-sum formula $\langle\langle usr_1 : \cdots : usr_m \rangle\rangle (\star_1,\star_2)_{\min=?}(\mathtt{R}^{time}[\mathtt{F}\ \mathtt{s}_1] + \cdots + \mathtt{R}^{time}[\mathtt{F}\ \mathtt{s}_m])$. Synthesising optimal strategies for this specification, we find that the cases for SWNE and SWCE coincide (although SWCE returns a joint strategy for the players, this joint strategy can be separated to form a strategy profile). This profile requires one user to try and send first, and then for the remaining users to take turns to try and send afterwards. If a user fails to send, then they enter backoff and allow all remaining users to try and send before trying to send again. There is no gain to a user in trying to send at the same time as another, as this will increase the probability of a sending failure, and therefore the user having to spend time in backoff before getting to try again. For SFNE, which has only been implemented for the two-player case, the two users follow identical strategies, which involve randomly deciding whether to wait or transmit, unless they are the only user that has not transmitted, and then they always try to send when not in backoff. In the case of SFCE, users can employ a shared probabilistic signal to coordinate which user sends next. Initially, this is a uniform choice over the users, but as time progresses the signal favours the users with lower backoff counters as these users have had fewer opportunities to send their packet previously.

In Figure 2 we have plotted the optimal values for the players, where SW_i correspond to the optimal values (expected times to send their packets) for player

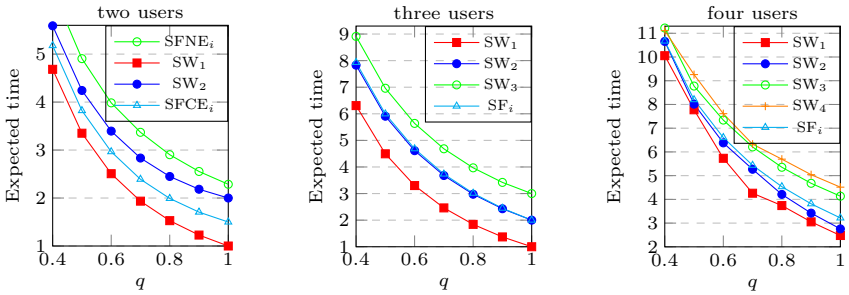


Fig. 2: Aloha: $\langle\langle usr_1 : \cdots : usr_m \rangle\rangle(\star_1, \star_2)_{\min=?}(\mathtt{R}^{time}[\,\mathtt{F}\ \mathtt{s}_1\,] + \cdots + \mathtt{R}^{time}[\,\mathtt{F}\ \mathtt{s}_m\,])$

i for both SWNE and SWCE for the cases of two, three and four users. We see that the optimal values for the different users under SFNE and SFCE coincide, while under SWNE and SWCE they are different for each user (with the user sending first having the lowest and the user sending last the highest). Comparing the sum of the SWNE (and SWCE) values and that of the SFCE values, we see a small decrease in the sum of less than 2% of the total, while for SFNE there is a greater difference as the players cannot coordinate, and hence try and send at the same time.

Power control. This case study is based on a model of power control in cellular networks from [7]. In the network there are a number of users that each have a mobile phone. The phones emit signals that the users can strengthen by increasing the phone's power level up to a bound ($pow_{\max}$). A stronger signal can improve transmission quality, but uses more energy and lowers the quality of the transmissions of other phones due to interference. We use the extended model from [22], which adds a probability of failure (q_{fail}) when a power level is increased and assumes each phone has a limited battery capacity ($e_{\max}$). There is a reward structure associated with each phone representing transmission quality, which is dependent on both the phone's power level and the power levels of other phones due to interference. We consider the nonzero-sum property $\langle\langle p_1 : \cdots : p_m \rangle\rangle(\star_1, \star_2)_{\max=?}(\mathtt{R}^{r_1}[\,\mathtt{F}\ \mathtt{e}_1\,] + \cdots + \mathtt{R}^{r_m}[\,\mathtt{F}\ \mathtt{e}_m\,])$, where each user tries to maximise their expected reward before their phone's battery is depleted.

In Figure 3 we have presented the expected rewards of the players under the synthesised SWCE and SFCE joint strategies. When performing strategy synthesis, in the case of two users the SWNE and SWCE yield the same profile in which, when the users' batteries are almost depleted, one user tries to increase their phone's power level and, if successful, in the next step, the second user then tries to increase their phone's power level. Since the first user's phone battery is depleted when the second tries to increase, this increase does not cause any interference. On the other hand, if the first user fails to increase their power level, then both users increase their battery levels. For the SFCE, the users can coordinate and flip a coin as to which user goes first: as demonstrated by Figure 3 this yields equal rewards for the users, unlike the SWCE. In the case of three users, the SWNE and SWCE differ (we were only able to synthesise SWNE for $pow_{\max} = 2$ as for larger values the computation had not completed within

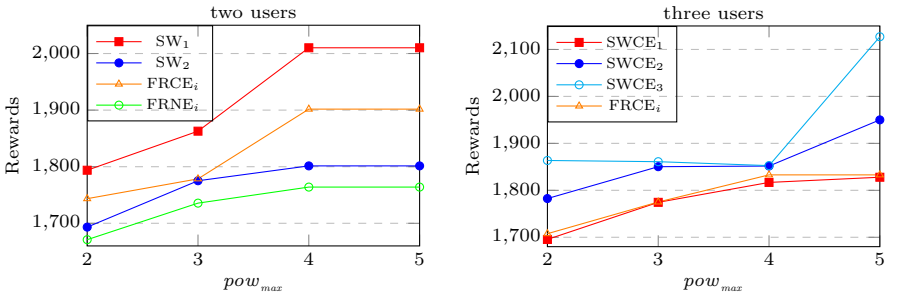


Fig. 3: Power control: $\langle\langle p_1{:}\cdots{:}p_m\rangle\rangle(\star_1,\star_2)_{\max=?}(\mathtt{R}^{r_1}[\,\mathtt{F}\ \mathtt{e}_1\,]+\cdots+\mathtt{R}^{r_m}[\,\mathtt{F}\ \mathtt{e}_m\,])$

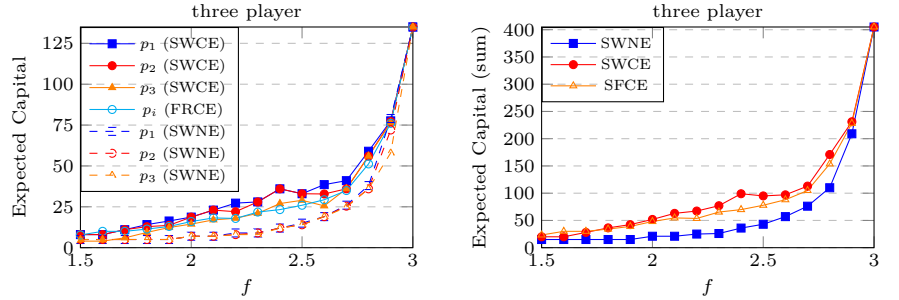


Fig. 4: Public good: $\langle\langle p_1{:}\cdots{:}p_m\rangle\rangle(\star_1,\star_2)_{\max=?}(\mathtt{R}^{c_1}[\,\mathtt{I}^{=r_{max}}\,]+\cdots+\mathtt{R}^{c_m}[\,\mathtt{I}^{=r_{max}}\,])$

the timeout), again users take turns to try and increase their phone's power level. However, here if the users are unsuccessful the SWCE can coordinate as to which user goes next trying to increase their phone's battery level. Through this coordination, the users' rewards can be increased as the battery level of at most one phone increases at a time, which limits interference. On the other hand, for the SWNE users must decide independently whether to increase their phone's battery level and they each randomly decide whether to do so or not.

Public good. We next consider a variant of a *public good* game [19], based on the one presented in [22] for the two-player case. In this game a number of players each receive an initial amount of capital (e_{init}) and, in each of r_{max} months, can invest none, half or all of their current capital. The total invested by the players in a month is multiplied by a factor f and distributed equally among the players before the start of the next month. The aim of the players is to maximise their expected capital which is represented by the formula: $\langle\langle p_1{:}\cdots{:}p_m\rangle\rangle(\star_1,\star_2)_{\max=?}(\mathtt{R}^{c_1}[\,\mathtt{I}^{=r_{max}}\,]+\cdots+\mathtt{R}^{c_m}[\,\mathtt{I}^{=r_{max}}\,])$.

Figure 4 plots, for the three-player model, both the expected capital of individual players and the total expected capital after three months for the SWNE, SWCE and SFNE as the parameter f varies. As the results demonstrate the players benefit, both as individuals and as a population, by coordinating through a correlated strategy. In addition, under the SFCE, all players receive the same expected capital with only a small decrease in the sum from that of the SWCE.

Investors. The final case study concerns a concurrent multi-player version of futures market investor model of [26], in which a number of investors (the players)

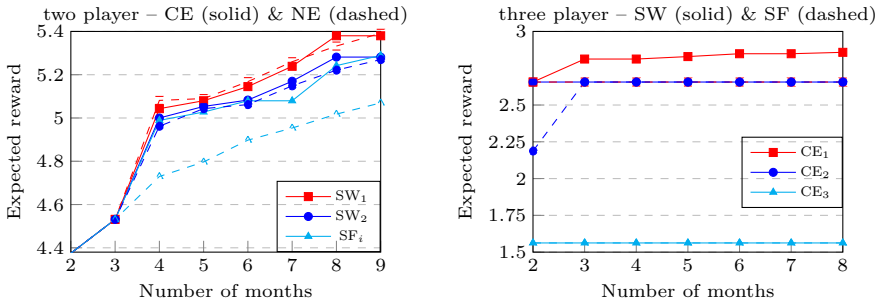


Fig. 5: Investors: $\langle\!\langle inv_1 : \cdots : inv_m \rangle\!\rangle (\star_1, \star_2)_{\max=?} (\mathsf{R}^{pf_1}[\mathsf{F}\ \mathsf{cin}_1] + \cdots + \mathsf{R}^{pf_m}[\mathsf{F}\ \mathsf{cin}_m])$

interact with a probabilistic stock market. In successive months, the investors choose whether to invest, wait or cash in their shares, while at the same time the market decides with probability *pbar* to bar each investor, with the restriction that an investor cannot be barred two months in a row or in the first month, and then the values of shares and cap on values are updated probabilistically.

We consider both two- and three-player models, where each investor tries to maximise its individual profit represented by the following nonzero-sum property: $\langle\!\langle inv_1 : \cdots : inv_m \rangle\!\rangle (\star_1, \star_2)_{\max=?} (\mathsf{R}^{pf_1}[\mathsf{F}\ \mathsf{cin}_1] + \cdots + \mathsf{R}^{pf_m}[\mathsf{F}\ \mathsf{cin}_m])$. In Figure 5 we have plotted the different optimal values for NE and CE of the two-player game and the different optimal values for CE of the three-player game (the computation of NE values timed out for the three player case). As the results demonstrate, again we see that the coordination that CEs offer can improve the returns of the players and that, although considering social fairness does decrease the returns of some players, this is limited, particularly for CEs.

5 Conclusions

We have presented novel techniques for game-theoretic verification of probabilistic multi-agent systems, focusing on correlated equilibria and a notion of social fairness. We began with the simpler case of normal form games and then extended this to concurrent stochastic games, and used temporal logic to formally specify equilibria. We proposed algorithms for equilibrium synthesis, implemented them and illustrated their benefits, in terms of efficiency and fairness, on case studies from a range of application domains.

Future work includes exploring the use of further game-theoretic topics within this area, such as techniques for mechanism design or other concepts such as Stackelberg equilibria. We plan to implement SFCE computation in Gurobi using the *big-M method* [16] to encode implications and techniques from [37] to encode conjunctions, which should yield a significant speed-up in their computation.

Acknowledgements. This project was funded by the ERC under the European Union's Horizon 2020 research and innovation programme (FUN2MODEL, grant agreement No. 834115).

References

1. de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D. thesis, Stanford University (1997)
2. Aminof, B., Kwiatkowska, M., B. Maubert, B., Murano, A., Rubin, S.: Probabilistic strategy logic. In: Proc. IJCAI'19. pp. 32–38 (2019)
3. Aumann, R.: Subjectivity and correlation in randomized strategies. Journal of Mathematical Economics **1**(1), 67–96 (1974)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
5. Baier, C., Kwiatkowska, M.: Model checking for a probabilistic branching time logic with fairness. Distributed Computing **11**(3), 125–155 (1998)
6. Banerjee, T., Majumdar, R., Mallik, K., Schmuck, A.K., Soudjani, S.: Fast symbolic algorithms for omega-regular games under strong transition fairness. Tech. Rep. MPI-SWS-2020-007r, Max Planck Institute (2021)
7. Brenguier, R.: PRALINE: A tool for computing Nash equilibria in concurrent games. In: Sharygina, N., Veith, H. (eds.) Proc. CAV'13. LNCS, vol. 8044, pp. 890–895. Springer (2013), lsv.fr/Software/praline/
8. Chatterjee, K., Fijalkow, N.: A reduction from parity games to simple stochastic games. EPTCS **54**, 74–86 (2011)
9. Chatterjee, K., Henzinger, T.: Value iteration. In: 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer (2008)
10. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Methods in System Design **43**(1), 61–92 (2013)
11. Chen, X., Deng, X., Teng, S.H.: Settling the complexity of computing two-player Nash equilibria. J. ACM **56**(3) (2009)
12. Daskalakis, C., Goldberg, P., Papadimitriou, C.: The complexity of computing a Nash equilibrium. Communications of the ACM **52**(2), 89–97 (2009)
13. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008), github.com/Z3Prover/z3
14. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Proc CAV'14. LNCS, vol. 8559, pp. 737–744. Springer (2014), yices.csl.sri.com
15. Gilboa, I., Zemel, E.: Nash and correlated equilibria: Some complexity considerations. Games and Economic Behavior **1**(1), 80–93 (1989)
16. Griva, I., Nash, S., Sofer, A.: Linear and Nonlinear Optimization: Second Edition. CUP (01 2009)
17. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021), www.gurobi.com
18. Gutierrez, J., Harrenstein, P., Wooldridge, M.J.: Reasoning about equilibria in game-like concurrent systems. In: Proc. 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14) (2014)
19. Hauser, O., Hilbe, C., Chatterjee, K., Nowak, M.: Social dilemmas among unequals. Nature **572**, 524–527 (2019)
20. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains. Springer (1976)
21. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Multi-player equilibria verification for concurrent stochastic games. In: Gribaudo, M., Jansen, D., Remke, A. (eds.) Proc. QEST'20. LNCS, Springer (2020)
22. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: PRISM-games 3.0: Stochastic game verification with concurrency, equilibria and time. In: Proc. CAV'20. pp. 475–487. LNCS, Springer (2020)

23. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Correlated equilibria and fairness in concurrent stochastic games (2022), arXiv:2201.09702
24. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Automatic verification of concurrent stochastic systems. Formal Methods in System Design pp. 1–63 (2021)
25. Littman, M., Ravi, N., Talwar, A., Zinkevich, M.: An efficient optimal-equilibrium algorithm for two-player game trees. In: Proc. UAI'06. pp. 298–305. AUAI Press (2006)
26. McIver, A., Morgan, C.: Results on the quantitative mu-calculus qMu. ACM Trans. Computational Logic **8**(1) (2007)
27. von Neumann, J., Morgenstern, O., Kuhn, H., Rubinstein, A.: Theory of Games and Economic Behavior. Princeton University Press (1944)
28. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.: Algorithmic Game Theory. CUP (2007)
29. Nudelman, E., Wortman, J., Shoham, Y., Leyton-Brown, K.: Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. In: Proc. AAMAS'04. pp. 880–887. ACM (2004), gamut.stanford.edu
30. Osborne, M., Rubinstein, A.: An Introduction to Game Theory. OUP (2004)
31. Porter, R., Nudelman, E., Shoham, Y.: Simple search methods for finding a Nash equilibrium. In: Proc. AAAI'04. pp. 664–669. AAAI Press (2004)
32. Prisner, E.: Game Theory Through Examples. Mathematical Association of America, 1 edn. (2014)
33. Rabin, M.: Incorporating fairness into game theory and economics. The American Economic Review **83**(5), 1281–1302 (1993)
34. Rabin, M.: Fairness in repeated games. working paper 97–252, University of California at Berkeley (1997)
35. Schwalbe, U., Walker, P.: Zermelo and the early history of game theory. Games and Economic Behavior **34**(1), 123–137 (2001)
36. Shapley, L.: Stochastic games. PNAS **39**, 1095–1100 (1953)
37. Stevens, S., Palocsay, S.: Teaching use of binary variables in integer linear programs: Formulating logical conditions. INFORMS Transactions on Education **18**(1), 28–36 (2017)
38. Wächter, A.: Short tutorial: Getting started with IPOPT in 90 minutes. In: Combinatorial Scientific Computing. No. 09061 in Dagstuhl Seminar Proceedings, Leibniz-Zentrum für Informatik (2009), github.com/coin-or/Ipopt
39. Wächter, A., Biegler, L.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical Programming **106**(1), 25–57 (2006)
40. Supporting material, www.prismmodelchecker.org/files/tacas22equ/

Omega Automata


A Direct Symbolic Algorithm for Solving Stochastic Rabin Games

Tamajit Banerjee[1], Rupak Majumdar[2], Kaushik Mallik[2] ✉, Anne-Kathrin Schmuck[2] ✉, and Sadegh Soudjani[3]

[1] IIT Delhi, New Delhi, India
[2] MPI-SWS, Kaiserslautern, Germany
[3] Newcastle University, Newcastle upon Tyne, UK

Abstract. We consider turn-based stochastic 2-player games on graphs with ω-regular winning conditions. We provide a direct symbolic algorithm for solving such games when the winning condition is formulated as a Rabin condition. For a stochastic Rabin game with k pairs over a game graph with n vertices, our algorithm runs in $O(n^{k+2}k!)$ symbolic steps, which improves the state of the art.
We have implemented our symbolic algorithm, along with performance optimizations including parallellization and acceleration, in a BDD-based synthesis tool called Fairsyn. We demonstrate the superiority of Fairsyn compared to the state of the art on a set of synthetic benchmarks derived from the VLTS benchmark suite and on a control system benchmark from the literature. In our experiments, Fairsyn performed significantly faster with up to *two* orders of magnitude improvement in computation time.

1 Introduction

Symbolic algorithms for 2-player graph games are at the heart of many problems in the automatic synthesis of correct-by-construction hardware, software, and cyber-physical systems from logical specifications. The problem has a rich pedigree, going back to Church [10] and a sequence of seminal results [6,31,17,30,13,14,34,21]. A chain of reductions can be used to reduce the synthesis problem for ω-regular specifications to finding winning strategies in 2-player games on graphs, for which (symbolic) algorithms are known (see, e.g., [29,14,34,27]). These algorithms form the basis for algorithmic reactive synthesis.

For systems under uncertainty, it is also essential to capture non-determinism quantitatively using probability distributions [5,18,22,25]. Turn-based *stochastic* 2-player games [3,9], also known as $2^1/_2$-player games, generalize 2-player graph games with an additional category of "random" vertices: Whenever the game reaches a random vertex, a random process picks one of the outgoing edges according to a probability distribution. The *qualitative* winning problem asks whether a vertex of the game graph is almost surely winning for Player 0. Stochastic Rabin games were studied by Chatterjee et al. [7], who showed that the problem is NP-complete and that winning strategies can be restricted to

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 81–98, 2022.
https://doi.org/10.1007/978-3-030-99527-0_5

be pure (non-randomized) and memoryless. Moreover, they showed a reduction from qualitative winning in an n-vertex k-pair stochastic Rabin game to an $O\left(n(k+1)\right)$-vertex $(k+1)$-pair (deterministic) Rabin game, resulting in an $O\left((n(k+1))^{k+2}(k+1)!\right)$ algorithm. In contrast, we provide a direct $O(n^{k+2}k!)$ symbolic algorithm for the problem.

Our new direct symbolic algorithm is obtained in the following way. We replace the probabilistic transitions with transitions of the environment constrained by *extreme fairness* as described by Pnueli [28]. Extreme fairness is specified via a special set of Player 1 vertices, called *live vertices*. A run is extremely fair if whenever a live vertex is visited infinitely often, *every* outgoing edge from this vertex is taken infinitely often. As our first contribution, we show that to solve a qualitative stochastic Rabin game, we can equivalently solve a (deterministic) Rabin game over the same game graph by interpreting random vertices of the stochastic game as live vertices.

As our second contribution we prove a direct symbolic algorithm to solve (deterministic) Rabin games with live vertices, which we call *extremely fair adversarial Rabin games*. In particular, we show a surprisingly simple syntactic transformation that modifies well-known symbolic fixpoint algorithm for solving 2-player Rabin games on graphs (*without* live vertices), such that the modified fixpoint solves the extremely fair adversarial version of the game.

To appreciate the simplicity of our modification, let us consider the well-known fixpoint algorithms for Büchi and co-Büchi games—particular classes of Rabin games—given by the following μ-calculus formula:

$$\begin{array}{ll} \textbf{Büchi:} & \nu Y.\ \mu X.\ \left(G \cap \mathrm{Cpre}(Y)\right) \cup \left(\mathrm{Cpre}(X)\right), \\ \textbf{Co-Büchi:} & \mu X.\ \nu Y.\ \left(G \cup \mathrm{Cpre}(X)\right) \cap \left(\mathrm{Cpre}(Y)\right). \end{array}$$

where $\mathrm{Cpre}(\cdot)$ denotes the controllable predecessor operator and G denotes the set of goal states that should be visited recurrently. In the presence of strong transition fairness, the new algorithm becomes

$$\begin{array}{ll} \textbf{Büchi:} & \nu Y.\ \mu X.\ \left(G \cap \mathrm{Cpre}(Y)\right) \cup \left(\mathrm{Apre}(Y, X)\right), \\ \textbf{Co-Büchi:} & \nu W.\ \mu X.\ \nu Y.\ \left(G \cup \mathrm{Apre}(W, X)\right) \cap \left(\mathrm{Cpre}(Y)\right). \end{array}$$

The only syntactic change (highlighted in blue) we make is to substitute the controllable predecessor for the μ variable X by a new *almost sure predecessor operator* $\mathrm{Apre}(Y, X)$ incorporating also the previous ν variable Y; if the fixpoint starts with a μ variable (with no previous ν variable), like for co-Büchi games, we introduce one additional ν variable in the front. For the general class of Rabin specifications, with a more involved fixpoint and with arbitrarily high nesting depth depending on the number of Rabin pairs, we need to perform this substitution for every such $\mathrm{Cpre}(\cdot)$ operator for every μ variable.

We prove the correctness of this syntactic fixpoint transformation for solving Rabin games [31,27] in this paper. It can be shown that the same syntactic transformation may be used to obtain fixpoint algorithms for qualitative solution of stochastic games with other popular ω-regular objectives, namely Reachability, Safety, (generalized) Büchi, (generalized) co-Büchi, Rabin-chain, parity, and

GR(1). Owing to page constraints, these additional fixpoints are only discussed in the extended version [4] of this paper, where we also generalize all results presented in this paper to a weaker notion of fairness, called transition fairness. In a nutshell, these results show that one can solve games with live vertices *while retaining* the algorithmic characteristics and implementability of known symbolic fixpoint algorithms that do not consider fairness assumptions.

We have implemented our symbolic algorithm for solving stochastic Rabin games in a symbolic BDD-based reactive synthesis tool called Fairsyn. Fairsyn additionally uses parallellization and a fixpoint acceleration technique [23] to boost performance. We evaluate our tool on two case studies, one using synthetic benchmarks derived from the VLTS benchmark suite [15] and the other from controller synthesis for stochastic control systems [12]. We show that Fairsyn scales well on these case studies, and outperforms the state-of-the-art methods by up to two orders of magnitude.

All the technical proofs, the fixpoints for various other specifications, and an additional benchmark taken from the software engineering literature [8] can be found in the extended version of this paper under a slighly more relaxed setting of the problem (transition fairness instead of extreme fairness) [4].

2 Preliminaries

Notation: We write $\mathbb{N}_0$ to denote the set of natural numbers including zero. Given $a, b \in \mathbb{N}_0$, we write $[a; b]$ to denote the set $\{n \in \mathbb{N}_0 \mid a \leq n \leq b\}$. By definition, $[a; b]$ is an empty set if $a > b$. For any set $A \subseteq U$ defined on the universe U, we write $\overline{A}$ to denote the complement of A. Given an alphabet A, we use the notation A^* and A^ω to denote respectively the set of all finite words and the set of all infinite words formed using the letters of the alphabet A. Let A and B be two sets and $R \subseteq A \times B$ be a relation. For any element $a \in A$, we use the notation $R(a)$ to denote the set $\{b \in B \mid (a, b) \in R\}$.

$2^1/_2$-player game graph: We consider usual turn-based stochastic games, also known as $2^1/_2$-player games, played between Player 0, Player 1, and a third player representing *environmental randomness* which is treated as a "half player." Formally, a $2^1/_2$-*player game graph* is a tuple $\mathcal{G} = \langle V, V_0, V_1, V_r, E \rangle$ where (i) V is a finite set of vertices, (ii) V_0, V_1, and V_r are subsets of V which form a partition of V, and (iii) $E \subseteq V \times V$ is the set of directed edges. The vertices in V_r are called *random vertices*, and the edges originating in a random vertex are called *random edges*, denoted as E_r. A $2^1/_2$-player game graph with no random vertices (i.e. $V_r = \emptyset$) is called a 2-*player game graph*. A $2^1/_2$-player game graph with $V_1 = \emptyset$ is called a $1^1/_2$-*player game graph* (also known as Markov Decision Processes or MDPs). A $2^1/_2$-player game graph with $V = V_r$ is known as a *Markov chain*.

Strategies: A (deterministic) strategy of Player 0 is a function $\rho_0 \colon V^*V_0 \to V$ with $\rho_0(wv) \in E(v)$ for every $wv \in V^*V_0$. Likewise, a strategy of Player 1 is a function $\rho_1 \colon V^*V_1 \to V$ with $\rho_1(wv) \in E(v)$ for every $wv \in V^*V_1$. We denote the set of strategies of Player i by Π_i. A strategy ρ_i of Player i ($i \in \{0, 1\}$) is *memoryless* if for every $w_1 v, w_2 v \in V^*V_i$, we have $\rho_i(w_1 v) = \rho_i(w_2 v)$. In this

paper we restrict attention to *deterministic* strategies, as randomized strategies are no more powerful than deterministic ones for $2^1/_2$-player Rabin games [7].

Plays: Consider an infinite sequence of vertices[4] $\pi = v^0 v^1 v^2 \ldots \in V^\omega$. The sequence π is called a *play* over $\mathcal{G}$ starting at the vertex v^0 if for every $i \in \mathbb{N}_0$, we have $v^i \in V$ and $(v^i, v^{i+1}) \in E$. A play is *finite* if it is of the form $v^0 v^1 \ldots v^n$ for some finite $n \in \mathbb{N}_0$. Let $\rho_0 \in \Pi_0$ and $\rho_1 \in \Pi_1$ be a pair of strategies for the two players, and $v^0 \in V$ be a given initial vertex. For every finite play $\pi = v^0 v^1 \ldots v^n$, the next vertex v^{n+1} is obtained as follows: If $v^n \in V_0$ then $v^{n+1} = \rho_0(v^0 \ldots v^n)$; if $v^n \in V_1$ then $v^{n+1} = \rho_1(v^0 \ldots v^n)$; and if $v^n \in V_r$ then v^{n+1} is chosen uniformly at random from the set $E_r(v^n)$. The uniform probability distribution over the random edges is without loss of generality for the problem considered in this paper; we will come back to this after setting up the problem statement. Every play generated in this way by fixing ρ_0, ρ_1, and v^0 is called a *play compliant with* ρ_0 *and* ρ_1 *that starts at vertex* v^0. The random choice in the random vertices induces a probability measure $P_{v^0}^{\rho_0,\rho_1}$ on the sample space of plays.[5] This is in contrast to 2-*player* games, where for any choice of $\rho_0 \in \Pi_0$, $\rho_1 \in \Pi_1$, and $v^0 \in V$, the resulting compliant play is *unique*.

Winning Conditions: A *winning condition* φ is a set of infinite plays over $\mathcal{G}$, i.e., $\varphi \subseteq V^\omega$, where the game graph $\mathcal{G}$ will always be clear from the context. We adopt Linear Temporal Logic (LTL) notation for describing winning conditions. The atomic propositions for the LTL formulas are sets of vertices, i.e., elements of the set 2^V. We use the standard symbols for the Boolean and the temporal operators: "$\neg$" for *negation*, "$\wedge$" for *conjunction*, "$\vee$" for *disjunction*, "$\rightarrow$" for *implication*, "$\mathcal{U}$" for *until* ($A \,\mathcal{U}\, B$ means "the play remains inside the set A until it moves to the set B"), "$\bigcirc$" for *next* ($\bigcirc A$ means "the next vertex is in the set A"), "$\Diamond$" for *eventually* ($\Diamond A$ means "the play will eventually visit a vertex from the set A"), and "$\Box$" for *always* ($\Box A$ means "the play will only visit vertices from the set A"). The syntax and semantics of LTL can be found in standard textbooks [3]. By slightly abusing notation, we use φ interchangeably to denote both the LTL formula and the set of plays satisfying φ. Hence, we write $\pi \in \varphi$ to denote the satisfaction of the formula φ by the play π.

Rabin Winning Conditions: A *Rabin* winning condition is expressed using a set of k *Rabin pairs* $\mathcal{R} = \{\langle G_1, R_1\rangle, \ldots, \langle G_k, R_k\rangle\}$, where k is any positive integer and $G_i, R_i \subseteq V$ for all $i \in [1; k]$. We say that $\mathcal{R}$ has the index set $P = [1; k]$. A play π satisfies the *Rabin condition* $\mathcal{R}$ if π satisfies the LTL formula

$$\varphi \coloneqq \bigvee_{i \in P} \left(\Diamond \Box \overline{R}_i \wedge \Box \Diamond G_i\right). \tag{2}$$

Almost Sure Winning: Let $\mathcal{G}$ be $2^1/_2$-player game graph, $\rho_0 \in \Pi_0$ and $\rho_1 \in \Pi_1$ be a pair of strategies, $v^0 \in V$ be an initial vertex, and φ be an ω-regular

[4] In our convention for denoting vertices, superscripts (ranging over $\mathbb{N}_0$) will denote the position of a vertex within a given sequence/play, whereas subscripts, either 0, 1, or r, will denote the membership of a vertex in the sets V_0, V_1, or V_r respectively.

[5] The unique measure $P_{v^0}^{\rho_0,\rho_1}$ is obtained through Carathéodory's extension theorem by extending the pre-measure on every infinite extension—called the cylinder set—of every finite play; see [3, pp. 757] for details.

specification over the vertices of $\mathcal{G}$. Then $P_{v^0}^{\rho_0,\rho_1}(\varphi)$ denotes the probability of satisfaction of φ by the plays compliant with ρ_0 and ρ_1 and starting at v^0. The set of *almost sure winning states* of Player 0 for the specification φ is defined as the set $\mathcal{W}^{a.s.} \subseteq V$ such that for every $v^0 \in \mathcal{W}^{a.s.}$ the following holds: $\sup_{\rho_0 \in \Pi_0} \inf_{\rho_1 \in \Pi_1} P_{v^0}^{\rho_0,\rho_1}(\varphi) = 1$. It is known [7, Thm. 4] that there is an optimal (deterministic) memoryless strategy $\rho_0^* \in \Pi_0$—called the *optimal almost sure winning strategy*—such that for every $v^0 \in \mathcal{W}^{a.s.}$ it holds that $\inf_{\rho_1 \in \Pi_1} P_{v^0}^{\rho_0^*,\rho_1}(\varphi) = 1$.

We extend the notion of winning to 2-player games as follows. Fix a 2-player game graph $\mathcal{G} = \langle V, V_0, V_1, \emptyset, E \rangle$ and an ω-regular specification φ over V. Player 0 wins the game from a vertex $v^0 \in V$ if Player 0 has a strategy ρ_0 such that for every Player 1 strategy ρ_1, the unique resulting play starting at v^0 is in φ. The winning region $\mathcal{W} \subseteq V$ is the set of vertices from which Player 0 wins the game. It is known that Player 0 has a memoryless strategy ρ_0^*—called the *optimal winning strategy*—such that for every Player 1 strategy $\rho_1 \in \Pi_1$ and for every initial vertex $v^0 \in \mathcal{W}$, the resulting unique compliant play is in φ [19].

3 Problem Statement and Outline

Given a $2^1/_2$-player game graph $\mathcal{G}$ and a Rabin specification φ as in (2), we consider the problem of solving the induced qualitative reactive synthesis problem. That is, we want to compute the set of almost sure winning states $\mathcal{W}^{a.s.}$ of $\mathcal{G}$ w.r.t. φ and the corresponding optimal memoryless winning strategy ρ_0^* of Player 0. This problem was solved by Chatterjee et al. [7] via a reduction from qualitative winning in the original $2^1/_2$-player Rabin game to winning in a larger (deterministic) 2-player Rabin game with an additional Rabin pair.

Instead of inflating the game graph and introducing an extra Rabin pair at the cost of more expensive computation, we propose a direct and computationally more efficient symbolic algorithm over the original game graph $\mathcal{G}$. We get this algorithm by interpreting the random vertices of $\mathcal{G}$ as special Player 1 vertices, called *live vertices*, which are subject to an extreme fairness assumption: along every play, if a live vertex v is visited infinitely often, then all outgoing transitions of v are also taken infinitely often. This re-interpretation results in a 2-player Rabin game with special *live Player* 1 *vertices* that are subjected to extreme fairness assumptions on Player 1's behavior. We call such games *extremely fair adversarial (*2*-player) Rabin games*. The correctness of our symbolic algorithm then follows from the two main results of our paper.

(I) We show that qualitative winning in a $2^1/_2$-player Rabin game $\mathcal{G}$ is equivalent to winning in the extremely fair adversarial (2-player) Rabin game $\mathcal{G}^\ell$ obtained from $\mathcal{G}$. Moreover, the winning strategy ρ_0 of Player 0 in $\mathcal{G}^\ell$ is also the optimal almost sure winning strategy in $\mathcal{G}$ for φ (see Thm. 1 in Sec. 4).

(II) We give a direct symbolic algorithm to compute the set of winning states, along with the Player 0 winning strategy for extremely fair adversarial (2-player) Rabin games (see Thm. 2 in Sec. 5).

Both contributions are discussed in detail in Sec. 4 and Sec. 5, respectively. Even though, for convenience, we have assumed a uniform probability distribution over the random edges, our contributions are valid for any arbitrary probability distribution. This follows from the established fact that the qualitative analysis of $2^1/_2$-player games does not depend on the precise probability values but only on the supports of the distributions [7].

We conclude the paper by an experimental evaluation in Sec. 6.

4 From Randomness to Extreme Fairness

In this section, we show that qualitative winning in $2^1/_2$-player Rabin games is equivalent to winning in extremely fair adversarial (2-player) Rabin games over the same underlying game graph. While it is known [16, Thm. 11.1] that the reduction of random vertices to extreme fairness is sound and complete for liveness winning conditions[6] we extend this connection to arbitrary Rabin winning conditions in this section, and therefore to the entire class of ω-regular specifications. We start with a formal definition of extremely fair adversarial games and the connection between randomness and extreme fairness, before stating our main result in Thm. 1.

Extremely Fair Adversarial Games: Let $\mathcal{G} = \langle V, V_0, V_1, \emptyset, E\rangle$ be a 2-*player game graph with live vertices* $V^\ell \subseteq V_1$, denoted using the tuple $\mathcal{G}^\ell = \langle \mathcal{G}, V^\ell\rangle$. The set of edges originating from the live vertices are called the *live edges*, and is denoted as $E^\ell := (V^\ell \times V) \cap E$. A play π over $\mathcal{G}^\ell$ is *extremely fair* with respect to V^ℓ if it satisfies the following LTL formula:

$$\alpha := \bigwedge_{(v,v')\in E^\ell} (\Box\Diamond v \rightarrow \Box\Diamond(v \wedge \bigcirc v')) \,. \tag{3}$$

Given $\mathcal{G}^\ell$ and an ω-regular winning condition φ over V, Player 0 wins the *extremely fair adversarial game* over $\mathcal{G}^\ell$ for φ from a vertex $v^0 \in V$ if Player 0 wins the game over $\mathcal{G}^\ell$ for the winning condition $\alpha \rightarrow \varphi$ from v^0.

Randomness as Extreme Fairness: Let $\mathcal{G} = \langle V, V_0, V_1, V_r, E\rangle$ be a $2^1/_2$-player game graph. Then we say that $\mathcal{G}$ induces the 2-player game graph with live vertices $\mathcal{G}^\ell := \langle\langle V, V_0, V_1 \cup V_r, \emptyset, E\rangle, V_r\rangle$. Intuitively, we interpret every random vertex of $\mathcal{G}$ as a live Player 1 vertex in $\mathcal{G}^\ell$. Obviously, this reinterpretation does not change the structure of the underlying graph specified by V and E.

Soundness of the Reduction: It remains to show that the almost sure winning set and the optimal almost sure winning strategy of Player 0 in $\mathcal{G}$ for φ is the same as the winning state set and the winning strategy of Player 0 in $\mathcal{G}^\ell$ for φ. This is formalized in the following theorem when φ is given as a Rabin condition. The proof essentially shows that the random vertices of $\mathcal{G}$ simulate the live vertices of $\mathcal{G}^\ell$, and vice versa; details are in the extended version [4, App. B.6, pp. 61].

[6] An LTL formula φ over V describes a *liveness property* if *every* finite play π over $\mathcal{G}$ allows for a continuation π' s.t. $\pi\pi' \in \varphi$.

Theorem 1. *Let $\mathcal{G}$ be a $2^1/_2$-player game graph with vertex set V, $\varphi \subseteq V^\omega$ be a Rabin winning condition as in (2), and $\mathcal{G}^\ell$ be the 2-player game graph with live edges induced by $\mathcal{G}$. Let $\mathcal{W} \subseteq V$ be the set of vertices from which Player 0 wins the extremely fair adversarial game over $\mathcal{G}^\ell$ with respect to φ, and $\mathcal{W}^{a.s.}$ be the almost sure winning set of Player 0 in the $2^1/_2$-player game $\mathcal{G}$ with respect to φ. Then, $\mathcal{W} = \mathcal{W}^{a.s.}$. Moreover, an optimal almost sure winning strategy in $\mathcal{G}^\ell$ is also an optimal winning strategy in $\mathcal{G}$, and vice versa.*

5 Extremely Fair Adversarial Rabin Games

This section presents our main result, which is a symbolic fixpoint algorithm that computes the winning region of Player 0 in the extremely fair adversarial game over $\mathcal{G}^\ell$ with respect to any ω-regular property formalized as a Rabin winning condition. This new symbolic fixpoint algorithm has multiple unique features.
(I) It works directly over $\mathcal{G}^\ell$, without requiring any pre-processing step to reduce $\mathcal{G}^\ell$ to a "normal" 2-player game with larger set of vertices.
(II) Our new fixpoint algorithm is obtained from the algorithm of Piterman et al. [27] by a simple syntactic change. We simply replace all controllable predecessor operators over least fixpoint variables by a new *almost sure predecessor operator* invoking the preceding maximal fixpoint variable. This makes the proof of our new fixpoint algorithm conceptually simple (see Sec. 5.3).

At a higher level, we make a simple yet efficient syntactic transformation of the fixpoint to incorporate the fairness assumption on the live vertices, without introducing any extra computational complexity. Most remarkably, this transformation also works *directly* for fixpoint algorithms for reachability, safety, Büchi, (generalized) co-Büchi, Rabin-chain, and parity games, as these can be formalized as particular instances of a Rabin game. Moreover, it also works for generalized Rabin, generalized Büchi, and GR(1) games. Owing to page constrains, these additional cases are described in the extended version [4].

5.1 Preliminaries on Symbolic Computations over Game Graphs

Set Transformers: Our goal is to develop symbolic fixpoint algorithms to characterize the winning region of an extremely fair adversarial game over a game graph with live edges. As a first step, given $\mathcal{G}^\ell$, we define the required symbolic transformers of sets of states. We define the existential, universal, and controllable predecessor operators as follows. For $S \subseteq V$, we have

$$\mathrm{Pre}_0^{\exists}(S) \coloneqq \{v \in V_0 \mid E(v) \cap S \neq \emptyset\}, \tag{4a}$$

$$\mathrm{Pre}_1^{\forall}(S) \coloneqq \{v \in V_1 \mid E(v) \subseteq S\}, \text{ and} \tag{4b}$$

$$\mathrm{Cpre}(S) \coloneqq \mathrm{Pre}_0^{\exists}(S) \cup \mathrm{Pre}_1^{\forall}(S). \tag{4c}$$

Intuitively, the controllable predecessor operator $\mathrm{Cpre}(S)$ computes the set of all states that can be controlled by Player 0 to stay in S after one step regardless

of the strategy of Player 1. Additionally, we define two operators which take advantage of the fairness assumption on the live vertices. Given two sets $S, T \subseteq V$, we define the live-existential and almost sure predecessor operators:

$$\mathrm{Lpre}^{\exists}(S) \coloneqq \{v \in V^{\ell} \mid E(v) \cap S \neq \emptyset\}, \quad \text{and} \tag{5a}$$

$$\mathrm{Apre}(S,T) \coloneqq \mathrm{Cpre}(T) \cup \left(\mathrm{Lpre}^{\exists}(T) \cap \mathrm{Pre}_1^{\forall}(S)\right). \tag{5b}$$

Intuitively, the almost sure predecessor operator[7] $\mathrm{Apre}(S,T)$ computes the set of all states that can be controlled by Player 0 to stay in T (via $\mathrm{Cpre}(T)$) *as well as* all Player 1 states in V^{ℓ} that (a) will *eventually* make progress towards T if Player 1 obeys its fairness-assumptions encoded in α (via $\mathrm{Lpre}^{\exists}(T)$) and (b) will never leave S in the "meantime" (via $\mathrm{Pre}_1^{\forall}(S)$). All the used set transformers are monotonic with respect to set inclusion. Further, $\mathrm{Cpre}(T) \subseteq \mathrm{Apre}(S,T)$ always holds, $\mathrm{Cpre}(T) = \mathrm{Apre}(S,T)$ if $V^{\ell} = \emptyset$, and $\mathrm{Apre}(S,T) \subseteq \mathrm{Cpre}(S)$ if $T \subseteq S$.

Fixpoint Algorithms in the μ-calculus: We use μ-calculus [20] as a convenient logical notation to define a symbolic algorithm (i.e., an algorithm that manipulates sets of states rather than individual states) for computing a set of states with a particular property over a given game graph $\mathcal{G}$. The formulas of the μ-calculus, interpreted over a 2-player game graph $\mathcal{G}$, are given by the grammar

$$\varphi \ ::= \ p \mid X \mid \varphi \cup \varphi \mid \varphi \cap \varphi \mid \mathit{pre}(\varphi) \mid \mu X.\varphi \mid \nu X.\varphi$$

where p ranges over subsets of V, X ranges over a set of formal variables, *pre* ranges over monotone set transformers in $\{\mathrm{Pre}_0^{\exists}, \mathrm{Pre}_1^{\forall}, \mathrm{Cpre}, \mathrm{Lpre}^{\exists}, \mathrm{Apre}\}$, and μ and ν denote, respectively, the least and the greatest fixed point of the functional defined as $X \mapsto \varphi(X)$. Since the operations $\cup$, $\cap$, and the set transformers *pre* are all monotonic, the fixed points are guaranteed to exist. A μ-calculus formula evaluates to a set of states over $\mathcal{G}$, and the set can be computed by induction over the structure of the formula, where the fixed points are evaluated by iteration. We omit the (standard) semantics of formulas (see [20]).

5.2 The Symbolic Algorithm

We now present our new symbolic fixpoint algorithm to compute the winning region of Player 0 in the extremely fair adversarial game over $\mathcal{G}^{\ell}$ with respect to a Rabin winning condition $\mathcal{R}$. A detailed correctness proof can be found in the extended version [4, App. B.3, pp. 40].

Theorem 2. *Let $\mathcal{G}^{\ell} = \langle \mathcal{G}, V^{\ell} \rangle$ be a game graph with live edges and $\mathcal{R}$ be a Rabin condition over $\mathcal{G}$ with index set $P = [1;k]$. Further, let Z^* denote the fixed point of the following μ-calculus expression:*

$$\nu Y_{p_0}.\mu X_{p_0}. \bigcup_{p_1 \in P} \nu Y_{p_1}.\mu X_{p_1}. \bigcup_{p_2 \in P_{\setminus 1}} \nu Y_{p_2}.\mu X_{p_2}. \ \ldots \bigcup_{p_k \in P_{\setminus k-1}} \nu Y_{p_k}.\mu X_{p_k}. \left[\bigcup_{j=0}^{k} \mathcal{C}_{p_j}\right], \tag{6a}$$

[7] We will justify the naming of this operator later in Rem. 1.

$$\textit{where} \quad \mathcal{C}_{p_j} := \left(\bigcap_{i=0}^{j} \overline{R}_{p_i}\right) \cap \left[\left(G_{p_j} \cap \mathrm{Cpre}(Y_{p_j})\right) \cup \left(\mathrm{Apre}(Y_{p_j}, X_{p_j})\right)\right], \quad (6b)$$

with[8] $p_0 = 0$, $G_{p_0} := \emptyset$ *and* $R_{p_0} := \emptyset$ *as well as* $P_{\setminus i} := P \setminus \{p_1, \ldots, p_i\}$. *Then* Z^* *is equivalent to the winning region* $\mathcal{W}$ *of Player* 0 *in the extremely fair adversarial game over* $\mathcal{G}^\ell$ *for the winning condition* φ *in* (2). *Moreover, the fixpoint algorithm runs in* $O(n^{k+2}k!)$ *symbolic steps, and a memoryless winning strategy for Player* 0 *can be extracted from it.*

5.3 Proof Outline

Given a Rabin winning condition over a "normal" 2-player game, [27] provided a symbolic fixpoint algorithm which computes the winning region for Player 0. The fixpoint algorithm in their paper is almost identical to our fixpoint algorithm in (6): it only differs in the last term of the constructed $\mathcal{C}$-terms in (6b). [27] defines the term $\mathcal{C}_{p_j}$ as

$$\left(\bigcap_{i=0}^{j} \overline{R}_{p_i}\right) \cap \left[\left(G_{p_j} \cap \mathrm{Cpre}(Y_{p_j})\right) \cup \left(\mathrm{Cpre}(X_{p_j})\right)\right].$$

Intuitively, a single term $\mathcal{C}_{p_j}$ computes the set of states that always remain within $Q_{p_j} := \bigcap_{i=0}^{j} \overline{R}_{p_i}$ while always re-visiting G_{p_j}. That is, given the simpler (local) winning condition

$$\psi := \Box Q \wedge \Box\Diamond G \quad (7)$$

for two sets $Q, G \subseteq V$, the set

$$\nu Y.\ \mu X.\ Q \cap [(G \cap \mathrm{Cpre}(Y)) \cup (\mathrm{Cpre}(X))] \quad (8)$$

is known to define exactly the states of a "normal" 2-player game $\mathcal{G}$ from which Player 0 has a strategy to win the game with winning condition ψ [26]. Such games are typically called *Safe Büchi Games*. The key insight in the proof of Thm. 2 is to show that the new definition of $\mathcal{C}$-terms in (6b) via the new *almost sure predecessor operator* Apre actually computes the winning state sets of *extremely fair adversarial* safe Büchi games. Subsequently, we generalize this intuition to the fixpoint for the Rabin games.

Fair Adversarial Safe Büchi Games: The following theorem characterizes the winning states in an extremely fair adversarial safe Büchi game.

Theorem 3. *Let* $\mathcal{G}^\ell = \langle \mathcal{G}, V^\ell \rangle$ *be a game graph with live vertices and* $Q, G \subseteq V$ *be two state sets over* $\mathcal{G}$. *Further, let*

$$Z^* := \nu Y.\ \mu X.\ Q \cap [(G \cap \mathrm{Cpre}(Y)) \cup (\mathrm{Apre}(Y, X))]. \quad (9)$$

Then Z^* *is equivalent to the winning region of Player* 0 *in the extremely fair adversarial game over* $\mathcal{G}^\ell$ *for the winning condition* ψ *in* (7). *Moreover, the fixpoint algorithm runs in* $O(n^2)$ *symbolic steps, and a memoryless winning strategy for Player* 0 *can be extracted from it.*

[8] The Rabin pair $\langle G_{p_0}, R_{p_0} \rangle = \langle \emptyset, \emptyset \rangle$ in (6) is artificially introduced to make the fixpoint representation more compact. It is not part of $\mathcal{R}$.

Intuitively, the fixpoints in (8) and (9) consist of two parts: (a) A minimal fixpoint over X which computes (for any fixed value of Y) the set of states that can *reach* the "target state set" $T := Q \cap G \cap \mathrm{Cpre}(Y)$ while staying inside the safe set Q, and (b) a maximal fixpoint over Y which ensures that the only states considered in the target T are those that allow to re-visit a state in T while staying in Q.

By comparing (8) and (9) we see that our syntactic transformation only changes part (a). Hence, in order to prove Thm. 3 it essentially remains to show that this transformation works for the even simpler *safe reachability games*.

Extremely Fair Adversarial Safe Reachability Games: A safe reachability condition is a tuple $\langle T, Q\rangle$ with $T, Q \subseteq V$ and a play π satisfies the *safe reachability condition* $\langle T, Q\rangle$ if π satisfies the LTL formula

$$\psi := Q \,\mathcal{U}\, T. \tag{10}$$

A safe reachability game is often called a *reach-while-avoid* game, where the safe sets are specified by an unsafe set $R := \overline{Q}$ that needs to be avoided. Their extremely fair adversarial version is formalized in the following theorem and proved in the extended version [4, Thm. 3.3].

Theorem 4. *Let $\mathcal{G}^\ell = \langle \mathcal{G}, V^\ell\rangle$ be a game graph with live edges and $\langle T, Q\rangle$ be a safe reachability winning condition. Further, let*

$$Z^* := \nu Y.\ \mu X.\ T \cup (Q \cap \mathrm{Apre}(Y, X)). \tag{11}$$

Then Z^* is equivalent to the winning region of Player 0 in the extremely fair adversarial game over $\mathcal{G}^\ell$ for the winning condition ψ in (10). Moreover, the fixpoint algorithm runs in $O(n^2)$ symbolic steps, and a memoryless winning strategy for Player 0 can be extracted from it.

To gain some intuition on the correctness of Thm. 4, let us recall that the fixpoint for safe reachability games *without* live edges is given by:

$$\mu X.\ T \cup (Q \cap \mathrm{Cpre}(X)). \tag{12}$$

Intuitively, the fixpoint computation in (12) is initialized with $X^0 = \emptyset$ and computes a sequence $X^0, X^1, \ldots, X^k$ of increasing sets until $X^k = X^{k+1}$. We say that v has rank r if $v \in X^r \setminus X^{r-1}$. All states contained in X^r allow Player 0 to force the play to reach T in at most $r - 1$ steps while staying in Q. The corresponding Player 0 strategy ρ_0 is known to be winning w.r.t. (10) and along every play π compliant with ρ_0, the path π remains in Q and the rank is always decreasing.

To see why the same strategy is also *sound* in the extremely fair adversarial safe reachability game $\mathcal{G}^\ell$, first recall that for vertices $v \notin V^\ell$ of $\mathcal{G}^\ell$, the operator $\mathrm{Apre}(X, Y)$ simplifies to $\mathrm{Cpre}(X)$. With this, we see that for every $v \notin V^\ell$ a Player 0 winning strategy $\widetilde{\rho}_0$ in $\mathcal{G}^\ell$ can always force plays to stay in Q and to decrease their rank, similar to ρ_0. Then every play π compliant with such a strategy $\widetilde{\rho}_0$ and visiting a vertex in V^ℓ only finitely often satisfies (10).

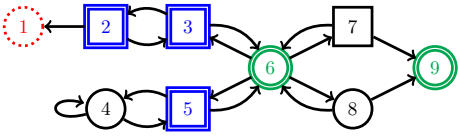


Fig. 1. Fair adversarial game graph discussed in Ex. 1 and Ex. 2 with Player 0 and Player 1 vertices being indicated by circles and squares, respectively. The live vertices are $V^\ell = \{2, 3, 5\}$ (double square, blue), the target vertices are $G = \{6, 9\}$ (double circle, green), and the unsafe vertices are $\overline{Q} = \{1\}$ (red,dotted).

The only interesting case for soundness of Thm. 4 is therefore every play π that visits states in V^ℓ infinitely often. However, as the number of vertices is finite, we only have a finite number of ranks and hence a certain vertex $v \in V^\ell$ with a finite rank r needs to get visited by π infinitely often. From the definition of Apre, we know that only states $v \in V^\ell$ are contained in X^r if v has an outgoing edge reaching X^k with $k < r$. Because of the extreme fairness condition, reaching v infinitely often implies that also a state with rank k s.t. $k < r$ will get visited infinitely often. As $X^1 = T$ we can show by induction that T is eventually visited along π while π always remains in Q until then.

In order to prove *completeness* of Thm. 4 we need to show that all states in $V \setminus Z^*$ are losing for Player 0. Here, again the reasoning is equivalent to the "normal" safe reachability game for $v \notin V^\ell$. For live vertices $v \in V^\ell$, we see that v is not added to Z^* via Apre if $v \notin T$ and either (i) none of its outgoing edges make progress towards T or (ii) some of its outgoing edges leave Z^*. One can therefore construct a Player 1 strategy that for (i)-vertices always choose an arbitrary transition and thereby never makes progress towards T (also if v is visited infinitely often), and for (ii)-vertices ensures that they are only visited once on plays which remain in Q. This ensures that (ii)-vertices never make progress towards T via their possibly existing rank-decreasing edges.

In the extended version [4], we have provided a detailed soundness and completeness proof of Thm. 4 along with the respective Player 0 and Player 1 strategy construction. In addition, there we also proved Thm. 3 using a reduction to Thm. 4 for every iteration over Y.

Example 1 (Extremely Fair adversarial safe reachability game). We consider an extremely fair adversarial safe reachability game over the game graph depicted in Fig. 1 with target vertex set $T = G = \{6, 9\}$ and safe vertex set $Q = V \setminus \{1\}$.

We denote by Y^m the m-th iteration over the fixpoint variable Y in (11), where $Y^0 = V$. Further, we denote by X^{mi} the set computed in the i-th iteration over the fixpoint variable X in (11) during the computation of Y^m where $X^{m0} = \emptyset$. We further have $X^{m1} = T = \{6, 9\}$ as $\text{Apre}(\cdot, \emptyset) = \emptyset$. Now we compute

$$\begin{aligned} X^{12} &= T \cup (Q \cap \text{Apre}(Y^0, X^{11})) \\ &= \{6, 9\} \cup (V \setminus \{1\} \cap [\underbrace{\text{Cpre}(X^{11})}_{\{7,8\}} \cup \underbrace{(\text{Lpre}^{\exists}(X^{11}) \cap \text{Pre}_1^{\forall}(V))}_{\{3,5\}}]) = \{3, 5, 6, 7, 8, 9\}. \end{aligned} \tag{13}$$

We observe that the only vertices added to X via the Cpre term are 7 and 8. The live vertices 3 and 5 are added due to their outgoing edges leading to the target vertex 6. The additional requirement $\mathrm{Pre}_1^{\forall}(V)$ in $\mathrm{Apre}(Y^0, X^{11})$ is trivially satisfied for all vertices at this point as $Y^0 = V$ and can therefore be ignored. Doing one more iteration over X we see that now vertex 4 gets added via the Cpre term (as it is a Player 0 vertex that allows progress towards 5) and vertex 2 is added via the Apre term (as it is live and allows progress to 3). The iteration over X terminates with $Y^1 = X^{1*} = V \setminus \{1\}$.

Re-iterating over X for Y^1 gives $X^{22} = X^{12} = \{3, 5, 6, 7, 8, 9\}$ as before. However, now vertex 2 does not get added to X^{23} because vertex 2 has an edge leading to $V \setminus Y^1 = \{1\}$. Therefore the iteration over X terminates with $Y^2 = X^{2*} = V \setminus \{1, 2\}$. When we now re-iterate over X for Y^2 we see that vertex 3 is not added to X^{32} any more, as vertex 3 has a transition to $V \setminus Y^2 = \{1, 2\}$. Therefore the iteration over X now terminates with $Y^3 = X^{3*} = V \setminus \{1, 2, 3\}$. Now re-iterating over X does not change the vertex set anymore and the fixed-point terminates with $Y^* = Y^3 = V \setminus \{1, 2, 3\}$.

We note that the fixpoint expression (12) for "normal" safe reachability games terminates after two iterations over X with $X^* = \{6, 7, 8, 9\}$, as vertices 7 and 8 are the only vertex added via the Cpre operator in (13). Due to the stricter notion of Cpre requiring that *all* outgoing edges of Player 0 vertices make process towards the target, (12) does not require an outer largest fixed-point over Y to "trap" the play in a set of vertices which allow progress when "waiting long enough". This "trapping" required in (11) via the outer fixpoint over Y actually fails for vertices 2 and 3 (as they are excluded from the winning set of (11)). Here, Player 1 can enforce to "escape" to the unsafe vertex 1 in two steps before 2 and 3 are visited infinitely often (which would imply progress towards 6 via the existing live edges).

We see that the winning region in the "normal" game is much smaller than the winning region for the extremely fair adversarial game, as adding live transitions restricts the strategy choices of Player 1, making it easier for Player 0 to win.

Example 2 (Extremely fair adversarial safe Büchi game). We now consider an extremely fair adversarial safe Büchi game over the game graph depicted in Fig. 1 with target set $G = \{6, 9\}$ and safe set $Q = V \setminus \{1\}$.

We first observe that we can rewrite the fixpoint in (9) as

$$\nu Y.\ \mu X.\ [Q \cap G \cap \mathrm{Cpre}(Y)] \cup [Q \cap (\mathrm{Apre}(Y, X))]\,. \tag{14}$$

Using (14) we see that for $Y^0 = V$ we can define $T^0 := Q \cap G \cap \mathrm{Cpre}(V) = G = \{6, 9\}$. Therefore the first iteration over X is equivalent to (13) and terminates with $Y^1 = X^{1*} = V \setminus \{1\}$.

Now, however, we need to re-compute T for the next iteration over X and obtain $T^1 = Q \cap G \cap \mathrm{Cpre}(Y^1) = V \setminus \{1\} \cap \{6, 9\} \cap V \setminus \{1, 2, 9\} = \{6\}$. This re-computation of T^1 checks which target vertices are repeatedly reachable, as required by the Büchi condition. As vertex 9 has no outgoing edge trivially it cannot be reached repeatedly.

With this, we see that for the next iteration over X we only have one target vertex $T^1 = \{6\}$. Unlike the safe reachability case in Ex. 1, the vertex 7 cannot be added to X^{22}, since Player 1 can always decide to take the edge towards 9 from 7, and therefore prevents repeated visit of a target state. Vertices 2 and 3 get eliminated for the same reason as in the safe reachability game within the second and third iteration over Y. The overall fixpoint computation therefore terminates with $Y^* = Y^3 = \{4, 5, 6, 8\}$.

Proof of Thm. 2: The proof of Thm. 2 essentially follows from the same arguments as in the soundness proof of the Rabin fixpoint for 2-player game by Piterman et al. [27], which utilizes Thm. 4 and Thm. 3 at all suitable places. In [4, App. A, pp. 29], we illustrate the steps of the Rabin fixpoint in (6) using a simple extremely fair adversarial Rabin game with two Rabin pairs.

Remark 1. We remark that the fixpoint (11), as well as the Apre operator, are similar in structure to the solution of almost surely winning states in concurrent reachability games [1]. In concurrent games, the fixpoint captures the largest set of states in which the game can be trapped while maintaining a positive probability of reaching the target. In our case, the fixpoint captures the largest set of states in which Player 0 can keep the game while ensuring a visit to the target either directly or through some of the edges from the live vertices. The commonality justifies our notation and terminology for Apre.

Remark 2. [2] studied fair CTL and LTL model checking where the fairness condition is given by exteme fairness with *all* vertices of the transition system being live. They show that CTL model checking under this all-live fairness condition, can be syntactically transformed to *non-fair* CTL model checking. A similar transformation is possible for fair model checking of Büchi, Rabin, and Streett formulas. The correctness of their transformation is based on reasoning similar to our Apre operator. For example, a state satisfies the CTL formula $\forall\Diamond p$ under fairness iff all paths starting from the state either eventually visits p or always visits states from which a visit to p is possible.

Complexity Analysis of (6): For Rabin games with k Rabin pairs, Piterman et al. [27] proposed a fixpoint formula with alternation depth $2k+1$. Using the accelerated fixpoint computation technique of Long et al. [23], they deduce a bound of $O(n^{k+1}k!)$ symbolic steps. We can apply the same acceleration technique to our fixpoint (6), yielding a complexity upper bound of $O(n^{k+2}k!)$ symbolic steps. (The additional complexity is because of an additional outermost ν-fixpoint.)

6 Experimental Evaluation

We developed a C++-based tool Fairsyn[9], which implements the symbolic fair adversarial Rabin fixpoint from Eq. (6) using Binary Decision Diagrams (BDD).

Repository URL: https://gitlab.mpi-sws.org/kmallik/synthesis-with-edge-fairness

Fairsyn has a single-threaded and a multi-threaded version, which respectively use the CUDD BDD library [32] and the Sylvan BDD library [11]. In both, we used a fixpoint acceleration procedure that "warm-starts" the inner fixpoints by exploiting a monotonicity property (detailed in the extended version [4]).

We demonstrate the effectiveness of our proposed symbolic algorithm for $2^1/_2$-player Rabin games using a set of synthetic benchmark experiments derived from the VLTS benchmark suite (Sec. 6.1) and a controller synthesis experiment for a stochastic dynamical system (Sec. 6.2); in the extended version [4], we include an additional software engineering benchmark example from the literature. In all of these examples, Fairsyn significantly outperformed the state-of-the-art.

The experiments in Sec. 6.1 were performed using the multi-threaded Fairsyn on a computer equipped with a 3 GHz Intel Xeon E7 v2 processor with 48 CPU cores and 1.5 TiB RAM. The experiments in Sec. 6.2 were performed using the single-threaded Fairsyn on a Macbook Pro (2015) laptop equipped with a 2.7 GHz Dual-Core Intel Core i5 processor with 16 GiB RAM.

6.1 The VLTS Benchmark Experiments

We present a collection of synthetic benchmarks for empirical evaluation of the merits of our direct symbolic algorithm compared to the one using the reduction to 2-player games [7]; in the following, we refer the latter as the *indirect approach*. Like our direct algorithm, the indirect approach has been implemented in Fairsyn and benefits from the same Sylvan-based parallel BDD-library and accelerated fixpoint solution technique. We collect the first 20 transition systems from the Very Large Transition Systems (VLTS) benchmark suite [15]; their descriptions can be found in the VLTS benchmark website. For each of them, we randomly generated instances of $2^1/_2$-player Rabin games with up to 3 Rabin pairs using the following procedure: (i) we labeled a given fraction of the vertices as random vertices, (ii) we equally partitioned the remaining vertices into system and environment vertices, and (iii) for every set in $\mathcal{R} = \{\langle G_1, R_1\rangle, \ldots, \langle G_k, R_k\rangle\}$, we randomly selected up to 5% of all vertices to be contained in the set. All the vertices in (i), (ii), and (iii) were selected randomly. In these examples, the number of vertices ranged from 289–164,865, the number of BDD variables ranged from 9–18, and the number of transitions from 1224–2,621,480.

In Fig. 2, we compare the running times of Fairsyn and the indirect approach. On the left scatter plot, every point corresponds to one instance of the randomly generated benchmarks, where the X and the Y coordinates represent the running time for Fairsyn and the indirect approach respectively. The solid red line indicates the exact same performance for both methods, whereas the dashed red line indicates an order of magnitude performance improvement for Fairsyn compared to the indirect approach. Observe that Fairsyn was faster by up to two orders of magnitude for the majority of the cases. In the experiments, the memory footprint of Fairsyn and the indirect approach was similar.

In the right plot, the X-axis corresponds to the proportion of random vertices within the set of vertices in percentage: 0% corresponds to a 2-player game and 100% corresponds to a Markov chain. The Y-axis corresponds to the running

time normalized with respect to the running time for the 0% case. We observe that Fairsyn was insensitive to the change of proportion of the random vertices. On the other hand, the indirect approach took longer time for larger proportion of random vertices, because for every random vertex it adds $3k + 2$ additional vertices, thus causing a linear blowup in the size of the game graph. The big variations in the time differences of the two approaches are due to the varying size of the experiments: The larger a game graph is, the larger is the difference. Interestingly, for both Fairsyn and the indirect method, there is a dip in the running time when all the vertices are random (i.e. the 100% case), which is possibly due to faster computation of the Cpre and Apre operators and faster convergence of the fixpoint algorithm, owing to the absence of Player 0 and Player 1 vertices.

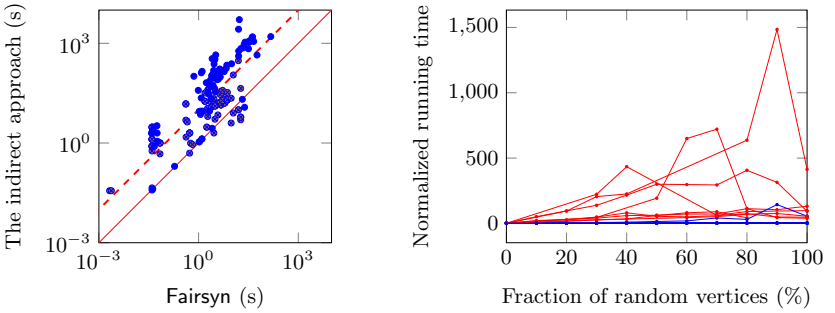


Fig. 2. LEFT: Comparison of running time of Fairsyn and the indirect approach on the VLTS benchmarks. All axes are in log-scale. **RIGHT:** Sensitivity of normalized running time w.r.t. variation of the proportion of random vertices. The blue and the red lines correspond to different instances of Fairsyn and the indirect approach respectively.

6.2 Synthesis for Stochastically Perturbed Dynamical Systems

Synthesizing verified symbolic controllers for continuous dynamical systems is an active area in cyber-physical systems research [33]. We consider a stochastically perturbed dynamical system model, called the bistable switch [12], which is an important model studied in molecular biology. The system model, call it Σ, has a continuous and compact two-dimensional state space $X = [0, 4] \times [0, 4] \in \mathbb{R}^2$ and a finite input space $U = \{-0.5, 0, 0.5\} \times \{-0.5, 0, 0.5\}$. Suppose for any given time $k \in \mathbb{N}$, $x_1(k), x_2(k)$ are the two states, $u_1(k), u_2(k)$ are the two inputs, and $w_1(k), w_2(k)$ are a pair of statistically independent noise samples drawn from a pair of distributions with bounded supports $W_1 = [-0.4, -0.2]$, $W_2 = [-0.4, -0.2]$ respectively. Then the states of Σ in the next time instant are:

$$x_1(k+1) = x_1(k) + 0.05\,(-1.3x_1(k) + x_2(k)) + u_1(k) + w_1(k), \tag{15}$$

$$x_2(k+1) = x_2(k) + 0.05\left(\frac{(x_1(k))^2}{(x_1(k))^2 + 1} - 0.25x_2(k)\right) + u_2(k) + w_2(k).$$

A controller C for Σ is a function $C\colon X \to U$ mapping the state $x(k)$ at any time instant k to a suitable control input $u(k)$. Then applying (15) repeatedly

Table 1. Performance comparison between Fairsyn and StochasticSynthesis (abbreviated as SS) [12] on a comparable implementation of the abstraction (uniform grid-based abstraction). Col. 1 shows the size of the resulting $2^1/_2$-player game graph (computed using the algorithm given in [24]), Col. 2 and 3 compare the total synthesis times and Col. 4 and 5 compare the peak memory footprint (as measured using the "time" command) for Fairsyn and SS respectively. "OoM" stands for out-of-memory.

# vertices in $2^1/_2$-game abstraction	Total synthesis time		Peak memory footprint	
	Fairsyn	SS	Fairsyn	SS
3.8×10^3	0.4 s	30 s	66 MiB	156 MiB
2.2×10^4	8.2 s	55 s	72 MiB	1 GiB
1.1×10^5	1 min 23 s	16 min 1 s	108 MiB	81 GiB
6.6×10^5	5 min 27 s	OoM	166 MiB	126 GiB
4.3×10^6	41 min 7 s	OoM	517 MiB	127 GiB

with $u(k) = C(x(k))$, starting with an initial state $(x_1(0), x_2(0)) = x(0) = x_{\mathsf{init}}$, gives us an infinite sequence of states $(x(0), x(1), x(2), \ldots)$ called a *path*. For a fixed controller C and for a given initial state x_{init}, we obtain a probability measure $P^C_{x_{\mathsf{init}}}$ on the sample space of paths of Σ, in a way similar to how we obtained the probability measure $P^{\rho_0,\rho_1}_{v_0}$ over infinite plays of $2^1/_2$-player games.

Let $\varphi \subseteq X^\omega$ be a Rabin specification, defined using a finite predicate over X. We extend the notion of almost sure winning for control systems in the obvious way: A state $x \in X$ of Σ is almost sure winning if there is a controller C such that $P^C_x(\varphi) = 1$. The controller synthesis problem asks to compute an optimal controller C^* such that for every almost sure winning state x, $P^{C^*}_x(\varphi) = 1$.

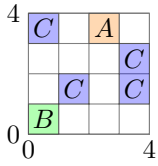


Fig. 3. Predicates over X.

Majumdar et al. [24] show that this synthesis problem can be approximately solved by lifting the system Σ to a finite $2^1/_2$-player game. We used Fairsyn to solve the resulting $2^1/_2$-player Rabin games obtained for the controller synthesis problem for Σ in (15) and for the following specification given in LTL using the predicates A, B, C, D as shown in Fig. 3: $\varphi := (\square\lozenge B \rightarrow \lozenge C) \wedge (\lozenge A \rightarrow \square\neg C)$.

In Table 1, we compare the performance of Fairsyn against the state-of-the-art algorithm for solving this problem, which is implemented in the tool called StochasticSynthesis (SS) [12]. It can be observed that Fairsyn significantly outperforms SS for every abstraction of different coarseness considered here.

Acknowledgments:
R. Majumdar and K. Mallik are funded through the DFG project 389792660 TRR 248–CPEC, A.-K. Schmuck is funded through the DFG project (SCHM 3541/1-1), and S. Soudjani is funded through the EPSRC New Investigator Award CodeCPS (EP/V043676/1).

References

1. de Alfaro, L., Henzinger, T.A., Kupferman, O.: Concurrent reachability games. In: 39th Annual Symposium on Foundations of Computer Science, FOCS. pp. 564–575.

IEEE Computer Society (1998)
2. Aminof, B., Ball, T., Kupferman, O.: Reasoning about systems with transition fairness. In: 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. LNCS, vol. 3452, pp. 194–208. Springer (2004)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
4. Banerjee, T., Majumdar, R., Kaushik, M., Schmuck, A.K., Soudjani, S.: Fast symbolic algorithms for omega-regular games under strong transition fairness (2021), https://www.mpi-sws.org/tr/2020-007.pdf
5. Belta, C., Yordanov, B., Gol, E.A.: Formal methods for discrete-time dynamical systems, vol. 15. Springer (2017)
6. Buchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society **138**, 295–311 (1969)
7. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: The complexity of stochastic Rabin and Streett games. In: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP). Lecture Notes in Computer Science, vol. 3580, pp. 878–890. Springer (2005)
8. Chatterjee, K., De Alfaro, L., Faella, M., Majumdar, R., Raman, V.: Code aware resource management. Formal Methods in System Design **42**(2), 146–174 (2013)
9. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Quantitative stochastic parity games. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 121–130. Society for Industrial and Applied Mathematics (2004)
10. Church, A.: Logic, arithmetic, and automata. Proceedings of the International Congress of Mathematicians, 1962 pp. 23–35 (1963)
11. van Dijk, T., van de Pol, J.: Sylvan: Multi-core decision diagrams. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 677–691. Springer (2015)
12. Dutreix, M., Huh, J., Coogan, S.: Abstraction-based synthesis for stochastic systems with omega-regular objectives. arXiv preprint arXiv:2001.09236 (2020)
13. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. In: FoCS. vol. 88, pp. 328–337 (1988)
14. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: FoCS. vol. 91, pp. 368–377 (1991)
15. Garavel, H., Descoubes, N.: Very large transition systems (2003), http://cadp.inria.fr/resources/vlts/
16. van Glabbeek, R., Höfner, P.: Progress, justness, and fairness. ACM Comput. Surv. **52**(4) (2019)
17. Gurevich, Y., Harrington, L.: Trees, automata, and games. In: Proceedings of the fourteenth annual ACM symposium on Theory of computing. pp. 60–65 (1982)
18. Kamgarpour, M., Summers, S., Lygeros, J.: Control design for property specifications on stochastic hybrid systems. Hybrid Systems: Computation and Control pp. 303–312 (April 2013)
19. Klarlund, N.: Progress measures, immediate determinacy, and a subset construction for tree automata. Annals of Pure and Applied Logic **69**(2-3), 243–268 (1994)
20. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27**(3), 333 – 354 (1983), international Colloquium on Automata, Languages and Programming (ICALP)
21. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05). pp. 531–540. IEEE (2005)

22. Laurenti, L., Lahijanian, M., Abate, A., Cardelli, L., Kwiatkowska, M.: Formal and efficient synthesis for continuous-time linear stochastic hybrid processes. IEEE Transactions on Automatic Control (2020)
23. Long, D.E., Browne, A., Clarke, E.M., Jha, S., Marrero, W.R.: An improved algorithm for the evaluation of fixpoint expressions. In: International Conference on Computer Aided Verification. pp. 338–350. Springer (1994)
24. Majumdar, R., Mallik, K., Schmuck, A.K., Soudjani, S.: Symbolic qualitative control for stochastic systems via finite parity games. In: ADHS 2021 (2021)
25. Majumdar, R., Mallik, K., Soudjani, S.: Symbolic controller synthesis for Büchi specifications on stochastic systems. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control. pp. 1–11 (2020)
26. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Annual Symposium on Theoretical Aspects of Computer Science. pp. 229–242. Springer Berlin Heidelberg (1995)
27. Piterman, N., Pnueli, A.: Faster solutions of Rabin and Streett games. In: 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06). pp. 275–284 (2006)
28. Pnueli, A.: On the extremely fair treatment of probabilistic algorithms. In: Proceedings of the fifteenth annual ACM symposium on Theory of computing. pp. 278–290 (1983)
29. Pnueli, A., Rosner, R.: A framework for the synthesis of reactive modules. In: Vogt, F.H. (ed.) International Conference on Concurrency, Proceedings. LNCS, vol. 335, pp. 4–17. Springer (1988)
30. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Annual ACM Symposium on Principles of Programming Languages. pp. 179–190. ACM Press (1989)
31. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the American Mathematical Society **141**, 1–35 (1969)
32. Somenzi, F.: Cudd 3.0.0 (2019), https://github.com/ivmai/cudd
33. Tabuada, P.: Verification and control of hybrid systems: a symbolic approach. Springer Science & Business Media (2009)
34. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998)


Practical Applications of the Alternating Cycle Decomposition

Antonio Casares[1](✉), Alexandre Duret-Lutz[2], Klara J. Meyer[3], Florian Renkin[2], and Salomon Sickert[4]*

[1] LaBRI, Université de Bordeaux, France, antonio.casares-santos@labri.fr
[2] LRDE, EPITA, France, adl@lrde.epita.fr, frenkin@lrde.epita.fr
[3] Independent Researcher, email@klarameyer.de
[4] School of Computer Science and Engineering, The Hebrew University, Israel, salomon.sickert@mail.huji.ac.il

Abstract. In 2021, Casares, Colcombet, and Fijalkow introduced the Alternating Cycle Decomposition (ACD) to study properties and transformations of Muller automata. We present the first practical implementation of the ACD in two different tools, Owl and Spot, and adapt it to the framework of Emerson-Lei automata, i.e., ω-automata whose acceptance conditions are defined by Boolean formulas. The ACD provides a transformation of Emerson-Lei automata into parity automata with strong optimality guarantees: the resulting parity automaton is minimal among those automata that can be obtained by duplication of states. Our empirical results show that this transformation is usable in practice. Further, we show how the ACD can generalize many other specialized constructions such as deciding typeness of automata and degeneralization of generalized Büchi automata, providing a framework of practical algorithms for ω-automata.

1 Introduction

Automata over infinite words have many applications, including verification and synthesis of reactive systems with specifications given in formalisms such as *Linear Temporal Logic* (LTL) [27, 23, 11, 12, 2, 29]. The synthesis problem from LTL specifications asks, given an LTL formula φ, to build a controller that processes an input word letter by letter, producing an output word, such that the combined input-output-word satisfies φ. The automata-theoretic approach to this problem (first introduced by Pnueli and Rosner [27]) consists of building a deterministic ω-automaton $\mathcal{A}$ equivalent to the LTL specification φ, then construct a game from $\mathcal{A}$ in which the opponent chooses the input letters for the automaton, and finally solve this game and obtain a controller from a winning strategy (whenever such a strategy exists). The automaton $\mathcal{A}$ can use different kinds of acceptance conditions (Rabin, Emerson-Lei, Muller, parity...) and

* Salomon Sickert is supported in part by the Deutsche Forschungsgemeinschaft (DFG) under project number 436811179, and in part funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No. 787367 (PaVeS)

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 99–117, 2022.
https://doi.org/10.1007/978-3-030-99527-0_6

thus we obtain games with different winning conditions. Among these games, parity games are the easiest to solve and there are highly-developed techniques for parity games solvers. Thus it is common practice to transform the automaton $\mathcal{A}$ to a parity one (for which we might need to augment the state space of the automaton). The top-ranked tools in the SyntComp competitions [17], Strix [23] (winner in editions 2018, 2019, 2020 and 2021) and `ltlsynt` [26], use this approach, producing a transition-based Emerson-Lei automata (TELA) as an intermediate step before constructing the parity automaton. For this reason, optimal and efficient procedures to transform Emerson-Lei automata into parity automata are of great importance.

Emerson-Lei (EL) acceptance conditions (first defined by Emerson and Lei [10], and reinvented in the HOA format [3]) are arbitrary positive Boolean formulas over the primitives $\mathsf{Inf}(c)$ and $\mathsf{Fin}(c)$ where c's are *colors* from a set Γ. A run is accepting if the set of colors $\mathcal{F} \subseteq 2^\Gamma$ seen infinitely often is a satisfying assignment to the EL acceptance condition (see Section 2 for a formal definition). Note that an explicit representation of all satisfying assignments is comparable to the *Muller condition* [15, Section 1.3.2]. Since the Boolean structure of LTL formulas can be mimicked by the Emerson-Lei acceptance conditions, a translation of LTL formulas to Emerson-Lei automata is particularly convenient.

Many algorithms to transform Emerson-Lei and Muller automata to parity have been proposed. In essence they all transform an automaton by turning each original state q into multiple states of the form (q, r) where r records some information about the current run, and transitions leaving (q, r) otherwise have a one-to-one mapping with those leaving q. Definition 3 calls this a *locally bijective morphism*, and we like to refer to those as *algorithms that duplicate states*. For instance in the *Later Appearance Record (LAR)* [16], r is a list of all colors ordered by most recent appearance, producing therefore a blow-up of $|\Gamma|!$ in the state-space of the automaton. The *State Appearance Record (SAR)* [24, 22] is a variation of this idea for state-based conditions, and the *Color Appearance Record (CAR)* [28] is a variation for the Emerson-Lei condition. The *Index Appearance Record (IAR)* [24, 22, 20] is a specialized construction for Rabin and Streett conditions, where r is now an ordering of pair indices. These algorithms have no particular insights about the input acceptance condition, such as inclusion or redundancies between colors (or pairs). In the Zielonka-tree transformation [31], r is a reference to a branch in a tree representation of a Muller condition. That tree representation is tailored to the condition and allows such simplifications compared to previous methods (it can be proven to be always better [6, 25]). While none of these algorithms use the structure of the input automaton to optimize the produced automata, some heuristics have been proposed [28, 25, 21].

In 2021, inspired by the *Zielonka tree*, Casares et al. introduced the Alternating Cycle Decomposition (ACD) of a Muller automaton [6]. Simply put, the ACD is a forest, i.e., a list of trees, that captures how accepting and rejecting cycles interleave in the automaton. They use the ACD to transform Muller automata into parity automata, and they prove a strong optimality result: the resulting automaton uses an optimal number of colors and has a minimal number of states

among those parity automata that can be obtained by duplicating states of the original one (see Theorem 1 for a formal statement). The main novelty of this transformation is that it does not only take into account the structure of both the acceptance condition and the automaton, but it exactly captures how they interact with each other. Moreover, Casares et al. [6] show that we can obtain some other valuable information about a Muller automaton from its ACD: for example the ACD can be used to decide *typeness*, i.e, if we can relabel it with another acceptance condition (parity, Rabin, Streett...). Their approach is primarily theoretical and puts the emphasis on how the ACD can be useful to obtain new results concerning Muller automata, but little is said about the costs of computing the ACD or the applicability of the transformation in practice.

Contributions. In this paper, we show that the ACD is practical. We adapt the definition of the ACD to Emerson-Lei automata and the HOA format [3]. We implement the ACD and the associated transformation in two tools: Owl [18] and Spot [9], providing baselines for efficient implementations of these structures. We show that the ACD gives a usable and useful method to transform Emerson-Lei automata into parity ones, improving upon any previous transformation in terms of the size of the output parity automaton. We extend the ACD to produce state-based automata, and show that the ACD generally beats traditional degeneralization-based procedures. Our implementation can also use the ACD to check typeness of deterministic automata.

Structure of the paper. We begin by providing some common definitions in Section 2. In Section 3, we define the Alternating Cycle Decomposition, adapting the definition of Casares et al. [6] to Emerson-Lei automata, and we provide an algorithm to compute it. In Section 5, we study the transformation of Emerson-Lei automata into parity ones using the ACD and we show experimental results obtained by comparing the ACD-transform implemented in Spot and Owl with other commonly used transformations. In Section 6 we show experimental results in the particular case of degeneralization of generalized Büchi automata. In Section 7 we discuss the utility of the ACD to decide typeness of automata.

2 Preliminaries

We denote by $|A|$ the cardinality of a set A and by 2^A its power set. For a finite alphabet Σ, we write Σ^* and Σ^ω for the sets of finite and infinite words, respectively, over Σ. The empty word is denoted by ε. Given $v \in \Sigma^*, w \in \Sigma^\omega$, we denote their concatenation by $v \cdot w$ and we write $v \sqsubseteq w$ if v is a prefix of w. We note $\inf(w)$ the set of letters that occur infinitely often in w. Given a map $\sigma\colon A \to B$ and a subset $A' \subseteq A$, we denote $\sigma|_{A'}$ the restriction of σ to A'. We extend σ to A^* and A^ω component-wise and we denote these extensions by σ whenever no confusion arises.

A (directed, edge-colored) *graph* is a pair $G = (V, E)$ where V is a finite set of vertices and $E \subseteq V \times \Gamma \times V$ is a finite set of Γ-colored edges. Note that with

Table 1: Encoding of common acceptance conditions into Emerson-Lei conditions. The variables $c, c_0, c_1, \ldots$ stand for arbitrary colors from the set Γ.

(B)	Büchi	$\mathsf{Inf}(c)$
(GB)	generalized Büchi	$\bigwedge_i \mathsf{Inf}(c_i)$
(C)	co-Büchi	$\mathsf{Fin}(c)$
(GC)	generalized co-Büchi	$\bigvee_i \mathsf{Fin}(c_i)$
(R)	Rabin	$\bigvee_i (\mathsf{Fin}(c_{2i}) \wedge \mathsf{Inf}(c_{2i+1}))$
(S)	Streett	$\bigwedge_i (\mathsf{Inf}(c_{2i}) \vee \mathsf{Fin}(c_{2i+1}))$
(P)	parity min even	$\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge (\mathsf{Inf}(2) \vee (\mathsf{Fin}(3) \wedge \ldots)))$
	parity min odd	$\mathsf{Fin}(0) \wedge (\mathsf{Inf}(1) \vee (\mathsf{Fin}(2) \wedge (\mathsf{Inf}(3) \vee \ldots)))$

this definition one can have multiple differently colored edges from a vertex v to a vertex u. A graph $G' = (V', E')$ is a *subgraph* of G (written $G' \subseteq G$) if $V' \subseteq V$ and $E' \subseteq E$. A graph $G = (V, E)$ is *strongly connected* if for every pair of vertices $(v, u) \in V^2$ there is a path from v to u. A *strongly connected component* (SCC) of a graph G is a maximal strongly connected subgraph of G.

Emerson-Lei acceptance conditions. Let $\Gamma = \{0, \ldots, n-1\}$ be a finite set of n integers called *colors*, from now on also written $\Gamma = \{0, 1, \ldots\}$ in our examples. We define the set $\mathbb{EL}(\Gamma)$ of *acceptance conditions* according to the following grammar, where c stands for any color in Γ:

$$\alpha ::= \top \mid \bot \mid \mathsf{Inf}(c) \mid \mathsf{Fin}(c) \mid (\alpha \wedge \alpha) \mid (\alpha \vee \alpha)$$

Acceptance conditions are interpreted over subsets of Γ. For $C \subseteq \Gamma$ we define the satisfaction relation $C \models \alpha$ inductively according to the following semantics:

$$C \models \top \qquad C \models \mathsf{Inf}(c) \text{ iff } c \in C \qquad C \models \alpha_1 \wedge \alpha_2 \text{ iff } C \models \alpha_1 \text{ and } C \models \alpha_2$$
$$C \not\models \bot \qquad C \models \mathsf{Fin}(c) \text{ iff } c \notin C \qquad C \models \alpha_1 \vee \alpha_2 \text{ iff } C \models \alpha_1 \text{ or } C \models \alpha_2$$

We denote by $\neg\alpha$ the negation of the acceptance condition α, i.e., $\mathsf{Fin}(m)$ becomes $\mathsf{Inf}(m)$, and vice-versa, $\wedge$ becomes $\vee$, etc. We assume that constants are propagated, i.e., a formula is either $\top$, $\bot$, or does not contain $\top$ and $\bot$.

Table 1 shows how common acceptance conditions can be encoded into Emerson-Lei conditions. Note that colors may appear multiple times; for instance $(\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0))$ is a Rabin condition.

Emerson-Lei automata. A *transition-based Emerson-Lei automaton* (TELA) is a tuple $\mathcal{A} = (Q, \Sigma, Q_0, \Delta, \Gamma, \alpha)$, where Q is a finite set of states, Σ is a finite input alphabet, $Q_0 \subseteq Q$ is a non-empty set of initial states, Γ is a set of colors, $\Delta \subseteq Q \times \Sigma \times 2^\Gamma \times Q$ is a finite set of transitions, and $\alpha \in \mathbb{EL}(\Gamma)$ is an Emerson-Lei condition. The *graph of* $\mathcal{A}$ is the directed edge-colored graph $G_\mathcal{A} = (Q, E)$ where the edges $E = \{(q, C, q') : \exists a \in \Sigma.\ (q, a, C, q') \in \Delta\}$ are obtained from Δ by removing Σ. We denote the transition $(q, a, C, q') \in \Delta$ and the edge $(q, C, q') \in E$ by $q \xrightarrow{a:C} q'$ and $q \xrightarrow{C} q'$, respectively. Further, we might omit a or C if they are

clear from the context. We denote by γ the projection of Δ or E to the set of colors Γ. Given a word $w = a_0 \cdot a_1 \cdot a_2 \cdots \in \Sigma^\omega$, a *run over* w in $\mathcal{A}$ is a sequence $\varrho = (q_0, a_0, C_0, q_1) \cdot (q_1, a_1, C_1, q_2) \cdots \in \Delta^\omega$ such that $q_0 \in Q_0$. The *output* of the run ϱ, is the word $\gamma(\varrho) \in (2^\Gamma)^\omega$. A run ϱ is *accepting* if $\inf(\gamma(\varrho)) \vDash \alpha$. A word $w \in \Sigma^\omega$ is *accepted* (or *recognized*) by $\mathcal{A}$ if there exists an accepting run over w in $\mathcal{A}$. We denote $\mathcal{L}(\mathcal{A})$ the set of words accepted by $\mathcal{A}$. Two automata $\mathcal{A}$, $\mathcal{A}'$ are *equivalent* if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. The size of an automaton, written $|\mathcal{A}|$, is the cardinality of its set of states. A state $q \in Q$ is *reachable* if there is a path from some state in Q_0 to q in $G_\mathcal{A}$.

An automaton $\mathcal{A}$ is *deterministic* if Q_0 is a singleton and for every $q \in Q$ and $a \in \Sigma$ there is at most one transition from q labeled with a, $q \xrightarrow{a:C} q' \in \Delta$.

We will use automata with acceptance defined over transitions (instead of stated-based acceptance) by default. However, in Sections 5 and 6 we will also discuss transformations towards automata with state-based acceptance.

If the acceptance condition of an automaton is represented as a condition of kind X (cf. Table 1), we call it an *X-automaton*. We assume that each transition of a parity-automaton is colored with exactly one color; this can be achieved by substituting the set C in a transition $q \xrightarrow{a:C} q'$ by $\min C$ (if $C \neq \emptyset$) or by $\{|\Gamma|+1\}$ if $C = \emptyset$. (If C is a singleton we will omit the brackets in the notation).

Labeled trees. A *tree* is a non-empty prefix-closed set $T \subseteq \mathbb{N}^*$ whose elements are called *nodes*. It is partially ordered by the prefix relation; if $x \sqsubseteq y$ we say that x is an *ancestor* of y and y is a *descendant* of x (we add the adjective "strict" if moreover $x \neq y$). The empty string ε is the *root* of the tree. The set of *children* of a node $x \in T$ is $\mathit{Children}_T(x) = \{x \cdot i \in T \ : \ i \in \mathbb{N}\}$. The set of leaves of T is $\mathit{Leaves}(T) = \{x \in T \ : \ \mathit{Children}_T(x) = \emptyset\}$. Nodes belonging to a same set $\mathit{Children}_T(x)$ are called *siblings*, and they are ordered from left to right by increasing value of their last component. If A is a set of labels, an *A-labeled tree* is a pair $\langle T, \eta \rangle$ of a tree T and a map $\eta \colon T \to A$. The *depth* of a node x is $\mathit{Depth}(x) = |x|$. The *height* of T is $\mathit{Height}(T) = \max_{x \in T} \mathit{Depth}(x)$.

3 The Alternating Cycle Decomposition

The *Alternating Cycle Decomposition* (ACD), proposed by Casares et al. [6], is a generalization of the Zielonka tree. The ACD of an automaton $\mathcal{A}$ is a forest, a collection of trees, labeled with accepting and rejecting cycles of the automaton. For each SCC of $\mathcal{A}$ we have a unique tree and the labeling of each tree alternates between accepting and rejecting cycles. Thus the ACD captures the complexity of the cycle structure of each SCC. We present now the definition of the ACD adapted to TELA.

For the rest of this section, let $\mathcal{A} = (Q, \Sigma, Q_0, \Delta, \Gamma, \alpha)$ be a TELA and let $G_\mathcal{A} = (Q, E)$ be the associated graph with edges colored by $\gamma \colon E \to 2^\Gamma$. We lift γ to sets and define $\gamma(E') = \bigcup_{e \in E'} \gamma(e)$ for every subset $E' \subseteq E$.

Definition 1. *A* cycle *of $\mathcal{A}$ is a subset of edges $\ell \subseteq E$ forming a closed path in $G_\mathcal{A}$. A cycle ℓ is* accepting (resp. rejecting) *if $\gamma(\ell) \models \alpha$ (resp. $\gamma(\ell) \not\models \alpha$). The set of* states *of a cycle ℓ is $States(\ell) = \{q \in Q : \text{some } e \in \ell \text{ passes through } q\}$. The set of cycles of $\mathcal{A}$ is denoted $\mathcal{C}ycles(\mathcal{A})$. It is (partially) ordered by set inclusion.*

Definition 2 ([6]). *Let $S_1, \ldots, S_k$ be an enumeration of the strongly connected components of $G_\mathcal{A}$. The* Alternating Cycle Decomposition *of $\mathcal{A}$, denoted $\mathcal{ACD}(\mathcal{A})$, is a collection of k $\mathcal{C}ycles(\mathcal{A})$-labeled trees $\langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$ with $\mathcal{T}_i = \langle T_i, \eta_i \rangle$ such that:*

- *$\eta_i(\varepsilon)$ is the set of edges of S_i, for $i = 1, \ldots, k$.*
- *If $x \in T_i$ and $\eta_i(x)$ is an accepting cycle, then x has a child in $\mathcal{T}_i$ for each maximal element in $\{\ell \in \mathcal{C}ycles(\mathcal{A}) : \ell \subseteq \eta_i(x) \text{ and } \ell \text{ is rejecting}\}$. In this case, we say that x is a* round node.
- *If $x \in T_i$ and $\eta_i(x)$ is a rejecting cycle, then x has a child in $\mathcal{T}_i$ for each maximal element in $\{\ell \in \mathcal{C}ycles(\mathcal{A}) : \ell \subseteq \eta_i(x) \text{ and } \ell \text{ is accepting}\}$. In this case, we say that x is a* square node.

If $q \in Q$ is a state belonging to the SCC S_i in $\mathcal{A}$, we define the *tree associated to q* as the subtree $\mathcal{T}_q = \langle T_q, \eta_q \rangle$ given by:

$$T_q = \{\varepsilon\} \cup \{x \in T_i : q \in States(\eta_i(x))\}, \quad \eta_q = \eta_i|_{T_q}.$$

Remark 1. We provide examples online at https://spot.lrde.epita.fr/ipynb/zlktree.html and an executable copy of this notebook is included in the artifact [8].

4 An Efficient Computation of the ACD

In this section we give an algorithm to compute the Alternating Cycle Decomposition of an Emerson-Lei automaton $\mathcal{A}$, implemented in Owl [18] and Spot [9]. This can be done by first computing an SCC-decomposition of $\mathcal{G}_\mathcal{A}$ which gives us the labels of the roots of the trees $\langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$, and then recursively computing the children of the nodes of each tree, following the definition of $\mathcal{ACD}(\mathcal{A})$. Algorithm 1 shows how to compute the children of a given node and uses notation we introduce now.

Let $C \subseteq \Gamma$ be a subset of colors and let $\mathcal{S} = (Q_S, E_S) \subseteq G_\mathcal{A}$ be a subgraph. We define the *projection of $\mathcal{S}$ on C*, denoted $\mathcal{S}_{\downarrow C} = (Q_S, E'_S)$, as the subgraph of $\mathcal{S}$ obtained by removing the edges $e \in E_S$ such that $\gamma(e) \not\subseteq C$, that is, $E'_S = \{(q, D, q') \in E_S : D \subseteq C\}$. We write $Colors(\mathcal{S}) = \bigcup_{e \in E_S} \gamma(e)$. We say that $\mathcal{S}' \subseteq \mathcal{S}$ is an *C-strongly connected component* in $\mathcal{S}$ (C-SCC) if it is an SCC of $\mathcal{S}$ and $Colors(\mathcal{S}') = C$. Further, $\max_\subseteq$ is the *set* of all maximal elements according to the partial order defined by $\subseteq$.

Note that Algorithm 1 uses Algorithm 2, which simplifies the Emerson-Lei conditions before passing the formula to a **Max-SAT** function (a SAT-solver that computes maximal satisfying assignments, e.g., by clause blocking) [4]. This preprocessing ensures that the ACD for Rabin or Streett acceptance conditions can be constructed without making use of the general purpose algorithm for computing maximal satisfying assignments.

Algorithm 1 Computing the children of a node.

Input: A cycle $\mathcal{S} = \eta_i(x)$ corresponding to the label of a node x of $\mathcal{ACD}(\mathcal{A})$.
Output: The set of labels for the children of x, $(\mathcal{S}_1, \ldots, \mathcal{S}_k)$.
function `Compute-Children`$(\mathcal{S})$
    $children \leftarrow \emptyset$, $C \leftarrow Colors(\mathcal{S})$
    if $C \vDash \alpha$ **then** ▷ Maximal subsets $D \subseteq C$ such that $D \vDash \alpha \Leftrightarrow C \nvDash \alpha$
        $\{C_1, \ldots, C_k\} \leftarrow$ `Max-Satisfying-Subsets`$(C, \neg\alpha)$
    else
        $\{C_1, \ldots, C_k\} \leftarrow$ `Max-Satisfying-Subsets`(C, α)
    for $D \in \{C_1, \ldots, C_k\}$ **do**
        for $\mathcal{S}' \in$ SCCs of $\mathcal{S}_{\downarrow D}$ **do** ▷ These might not be D-SCC in $\mathcal{S}$
            if $Colors(\mathcal{S}') \vDash \alpha \Leftrightarrow D \vDash \alpha$ **then**
                $children \leftarrow children \cup \{\mathcal{S}'\}$
            else
                $children \leftarrow children \cup$ `Compute-Children`$(\mathcal{S}')$
    return $\max_{\subseteq} children$ ▷ Remove from *children* non-maximal cycles

Algorithm 2 The subprocedure `Max-Satisfying-Subsets`.

Input: A subset of colors $C \subseteq \Gamma$ and an EL condition $\alpha \in \mathbb{EL}(\Gamma)$.
Output: $\max_{\subseteq}\{D \subseteq C : D \vDash \alpha\}$.
function `Max-Satisfying-Subsets`(C, α)
    if $C \vDash \alpha$ **then**
        return $\{C\}$
    $\alpha \leftarrow \alpha[\text{if } c \in C \text{ then } c \text{ else } \bot]$ ▷ Replace colors not in C by false
    $L \leftarrow \{c \in C : \neg c \text{ does not occur in } \alpha\}$
    if $L \neq \emptyset$ **then**
        $\alpha \leftarrow \alpha[\text{if } c \in L \text{ then } \top \text{ else } c]$ ▷ Replace colors in L by true
        $\{C_1, \ldots, C_k\} \leftarrow$ `Max-Satisfying-Subsets`$(C \setminus L, \alpha)$
        return $\{C_1 \cup L, \ldots, C_k \cup L\}$
    if $\alpha = \neg c_1 \vee \cdots \vee \neg c_n$ **then**
        return $\{\{c_1, \ldots, c_n\} \setminus \{c_i\} : 1 \leq i \leq n\}\}$
    return Max-SAT(α)

Memoization. To optimize the construction of the ACD and to avoid duplicated recursive calls, we perform two kinds of memoization: First, we memoize the results of calling Algorithm 2 from Algorithm 1. (Thus we implicitly construct a Zielonka DAG for α.) Second, we memoize the recursive calls to Algorithm 1: this is useful, as distinct nodes in the ACD can be labeled by the same cycles.

5 From Emerson-Lei to Parity Automata

In this section we describe the transformation from TELA to parity automata using the Alternating Cycle Decomposition [6]. This transformation provides strong optimality guarantees: the resulting parity automaton has minimal size

among those that can be produced without merging states from the TELA and it uses an optimal number of colors (Theorem 1). We also show that this transformation can be adapted to produce state-based automata. Note that in this case we loose the first optimality guarantee.

5.1 The ACD Transformation

Let $\mathcal{A} = (Q, \Sigma, Q_0, \Delta, \Gamma, \alpha)$ be a TELA and let $\mathcal{ACD}(\mathcal{A}) = \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$. We introduce the following notation that will allow us to move in the ACD.

Given a transition $e = q \xrightarrow{a:C} q'$ such that both q and q' belong to the i-th SCC of $\mathcal{A}$ and a node $x \in T_i$, we define $\mathit{Support}(x, e)$ to be the least ancestor z of x in $\mathcal{T}_i$ such that $e \in \eta_i(z)$. If $\mathit{Support}(x, e) \neq x$ and it is not a leaf in $\mathcal{T}_{q'}$, let z' be the only child of $\mathit{Support}(x, e)$ that is an ancestor of x, and let $y_1, \ldots, y_s$ be an enumeration from left to right of the nodes in $\mathit{Children}_{T_{q'}}(\mathit{Support}(x, e))$. We define $\mathit{NextBranch}(x, e)$ as:

$$\begin{cases} \mathit{Support}(x, e), & \text{if } \mathit{Support}(x, e) = x \text{ or if } \mathit{Support}(x, e) \text{ is a leaf in } \mathcal{T}_{q'}, \\ y_1, & \text{if } z' = y_s, \\ y_{j+1}, & \text{if } z' = y_j,\ 1 \leq j < s. \end{cases}$$

We define a parity automaton $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})} = (P, \Sigma, P_0, \Delta_P, \Gamma_P, \beta)$ (*ACD transform of* $\mathcal{A}$) equivalent to $\mathcal{A}$ as follows:

States. The states of $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}$ are of the form (q, x), for $q \in Q$ and x a leaf of the tree associated to q. Initial states are of the form (q_0, x) with $q_0 \in Q_0$ is an initial state in $\mathcal{A}$ and x is the leftmost leaf on its corresponding tree.

$$P = \bigcup_{q \in Q} \{q\} \times \mathit{Leaves}(\mathcal{T}_q),\ P_0 = \{(q_0, x) : q_0 \in Q_0,\ x \text{ the leftmost leaf in } \mathcal{T}_{q_0}\}.$$

Transitions. For each transition $e = q \xrightarrow{a:C} q'$ in Δ and each state $(q, x) \in P$, let us define a transition $(q, x) \xrightarrow{a:p} (q', y)$ in Δ_P as follows: first, q' is the destination state for the original transition. If q and q' are not in the same SCC then y is defined as the leftmost leaf in $\mathcal{T}_{q'}$ and $p = 1$ (except if all $\mathcal{T}_i$ have height 1 and a rounded root: in that case $p = 0$). Otherwise, if both q and q' belong to the i-th SCC of $\mathcal{A}$, then the destination leaf y is the leftmost descendant of $\mathit{NextBranch}(x, e)$ in $\mathcal{T}_{q'}$.
We define the color p of the transition as $\mathit{Depth}(\mathit{Support}(x, e))$, if the root of $\mathcal{T}_i$ is a round node ($\eta_i(\varepsilon) \vDash \alpha$), or as $\mathit{Depth}(\mathit{Support}(x, e)) + 1$ otherwise. We remark that in this way, p is even if and only if $\eta_i(z) \vDash \alpha$.

Parity condition. The condition β is a parity min even condition (cf. Table 1).

Remark 2. If the color 0 does not appear on any transition then we shift all colors by -1 and replace β by a *parity min odd* condition.

Proposition 1 ([6]). *The automaton* $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}$ *recognizes* $\mathcal{L}(\mathcal{A})$.

Remark 3. The ACD transformation preserves many properties (determinism, completeness, good-for-gameness, unambiguity...) of the automaton $\mathcal{A}$, see [6].

Remark 4. Since the number of colors used by $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}$ is at most the height of a tree in $\mathcal{ACD}(\mathcal{A})$, we obtain that $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}$ never uses more colors than $|\Gamma|+1$. Furthermore, since the TELA does not require all transitions to have a color, we can omit the maximal one and produce an automaton with at most $|\Gamma|$ colors.

In order to state the optimality of this transformation we introduce the notion of *locally bijective morphisms* of automata. Given an automaton $\mathcal{A} = (Q, \Sigma, Q_0, \Delta, \Gamma, \alpha)$ and $q \in Q$, we denote $Out_{\mathcal{A}}(q)$ the set of outgoing transitions of q, i.e., $Out_{\mathcal{A}}(q) = \{q \xrightarrow{a:C} q' \in \Delta \,:\, a \in \Sigma, C \subseteq \Gamma, q' \in Q\}$.

Definition 3 ([6]). *Let $\mathcal{A} = (Q, \Sigma, Q_0, \Delta, \Gamma, \alpha)$ and $\mathcal{A}' = (Q', \Sigma, Q'_0, \Delta', \Gamma', \alpha')$ be two EL automata over Σ. A* locally bijective morphism *from $\mathcal{A}$ to $\mathcal{A}'$ (denoted $\varphi\colon \mathcal{A} \to \mathcal{A}'$) is a pair of maps $\varphi_Q\colon Q \to Q'$, $\varphi_\Delta : \Delta \to \Delta'$ such that:*

- *$\varphi_Q|_{Q_0}$ is a bijection between Q_0 and Q'_0.*
- *$\varphi_\Delta\big(q_1 \xrightarrow{a:C} q_2\big) = \varphi_Q(q_1) \xrightarrow{a:C'} \varphi_Q(q_2)$ for some $C' \subseteq \Gamma'$.*
- *For every $q \in Q$, $\varphi_\Delta|_{Out_{\mathcal{A}}(q)}$ is a bijection between $Out_{\mathcal{A}}(q)$ and $Out_{\mathcal{A}'}(\varphi_Q(q))$*
- *For every run $\varrho \in \Delta^\omega$ in $\mathcal{A}$, ϱ is accepting iff $\varphi_\Delta(\varrho)$ is accepting in $\mathcal{A}'$.*

Theorem 1 ([6]). *Let $\mathcal{A}$ be an Emerson-Lei automaton, and let $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}$ be the parity automaton obtained by applying the ACD transformation. Then,*

- *There is a locally bijective morphism $\varphi : \mathcal{P}_{\mathcal{ACD}(\mathcal{A})} \to \mathcal{A}$.*
- *If $\mathcal{P}'$ is a parity automaton admitting a locally bijective morphism to $\mathcal{A}$, then $|\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}| \leq |\mathcal{P}'|$.*
- *If $\mathcal{P}'$ is a parity automaton recognizing $\mathcal{L}(\mathcal{A})$, $\mathcal{P}'$ uses at least as many colors as $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}$.*

Note that all state-duplicating constructions mentioned in the introduction create locally bijective morphisms. Thus the above theorem shows that the ACD transformation duplicates the least number of states.

5.2 Experimental Results

Figures 1 and 2 compare four different paritization procedures applied to 1065 TELA generated[5] from LTL formulas from the Synthesis Competition. These automata have between 2 and 55 colors (mean 5.92, median 5) and between 1 and 245761 states (mean 2023.20, median 20). Automata with fewer than 2 colors have been ignored since they are trivial to paritize.

The procedures are Owl's and Spot's implementation of ACD transform, as well as Spot's implementation of the Zielonka Tree transform [6], and Spot's previous paritization function (called `to_parity`) [28]. We refer the reader to Section 8 for information about the used versions. Two dotted lines on the sides

[5] We used `ltl2tgba -G -D` from Spot, and `ltl2dela` from Owl.

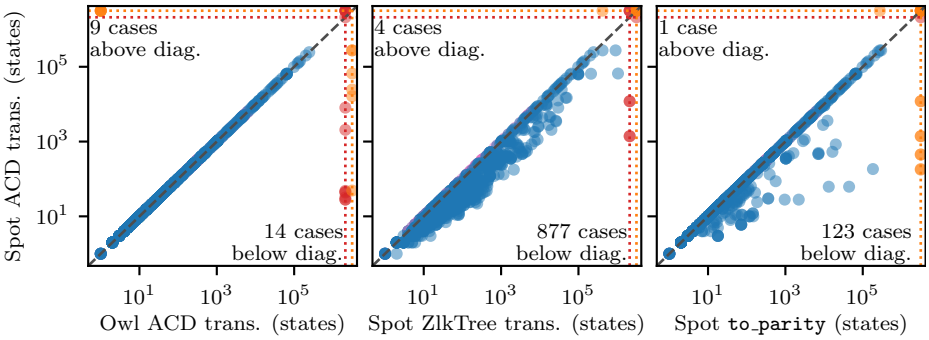


Fig. 1: Comparison of the output size of the four paritization procedures.

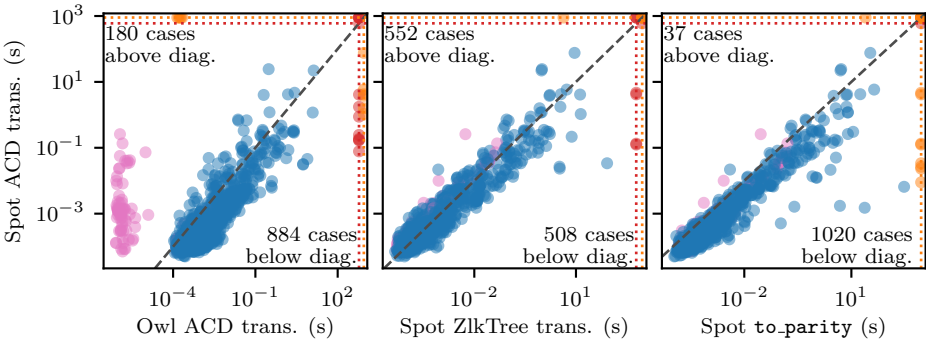


Fig. 2: Time spent performing these four paritization procedures.

of the plots hold cases that did not finish within 500 seconds (red, inner line), or where the tool reported an error[6] (orange, outer line). Pink dots represent input automata that already have parity acceptance: for those, running the ACD transform still makes sense as it will produce an output with a minimal number of colors. However, Owl's implementation, which mostly cares about reducing the number of states, uses a shortcut and will return the input automaton unmodified in this case: this explains the pink cloud on the left of Figure 2.

Owl's and Spot's implementations of the ACD transform produce automata with the same size, as expected. The cases that are not on the diagonal all correspond to timeouts or tool errors. The Zielonka Tree transform, which does not take the automaton structure into consideration, produces automata that are on the average 2.11 times bigger (median 1.60), while its runtime is on the average 6.55 times slower (median 0.97). Lastly, Spot's `to_parity` function is not far from the optimal size given by ACD transform: on the average its output is 3.28 times larger, but the median of that size ratio is 1.00. Similarly, it is on the average 15.94 times slower, but with a median of 1.04.

[6] Either "out-of-memory", or "too many colors" as Spot is restricted to 32 colors.

5.3 ACD Transformation Towards State-Based Parity Automata

Sometimes it is desired to obtain an automaton with the acceptance defined over states. A *state-based parity automaton* is a tuple $\mathcal{A} = (Q, \Sigma, Q_0, \Delta, \phi\colon Q \to \mathbb{N})$ where (Q, Σ, Q_0, Δ) is the underlying structure defined as for transition-based automata in Section 2 (with the only difference that $\Delta \subseteq Q \times \Sigma \times Q$ now), and $\phi\colon Q \to \mathbb{N}$ is a map associating colors to states. A run over $\mathcal{A}$ is *accepting* if the minimal color visited infinitely often is even.

Let $\mathcal{A}$ be a TELA with $\mathcal{ACD}(\mathcal{A}) = \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$. We define an equivalent state-based parity automaton $\mathcal{P}_{sb\text{-}\mathcal{ACD}(\mathcal{A})} = (P, \Sigma, P_0, \Delta_P, \phi\colon P \to \mathbb{N})$ as follows:

States. States are of the form (q, x), for $q \in Q$ and $x \in T_q$ (now the second component corresponds to a node of the ACD that is not necessarily a leaf). The set of initial states is the same as for $\mathcal{P}_{\mathcal{ACD}(\mathcal{A})}$:

$$P = \bigcup_{q \in Q} \{q\} \times T_q\,, \quad P_0 = \{(q_0, x) \,:\, q_0 \in Q_0,\, x \text{ the leftmost leaf in } \mathcal{T}_{q_0}\}.$$

Transitions. For each transition $e = q \xrightarrow{a:C} q' \in \Delta$ and $(q, x) \in P$ we define one transition $(q, x) \xrightarrow{a} (q', y) \in \Delta_P$. To specify the destination node y, we distinguish two cases:

Suppose that x is a leaf in $\mathcal{T}_q$. If $\mathit{NextBranch}(x, e)$ is not the leftmost child of $\mathit{Support}(x, e)$ in $\mathcal{T}_{q'}$, then y is the leftmost leaf below $\mathit{NextBranch}(x, e)$ in $\mathcal{T}_{q'}$ (as in the transition-based case). If $\mathit{NextBranch}(x, e)$ is the leftmost child (a "lap" around $\mathit{Support}(x, e)$ is finished), then we set $y = \mathit{Support}(x, e)$.

If x is not a leaf in $\mathcal{T}_q$, the destination y is determined exactly as if the transition started in (q, x') for x' the leftmost leaf in T_q under x.

Parity condition. $\phi((q, x)) = \mathit{Depth}(x)$, if the root of $\mathcal{T}_q$ is a round node, and $\phi((q, x)) = \mathit{Depth}(x) + 1$ otherwise.

Note that we do not have the same optimality guarantee as in the transition-based case: If x is not a leaf in its corresponding tree, then the states of the form $(q, x) \in P$ are not necessarily reachable in $\mathcal{P}_{sb\text{-}\mathcal{ACD}(\mathcal{A})}$. We only need to add those that can be reached from the initial state. However, the set of reachable states does depend on the ordering of the children in the trees of the ACD, and therefore the size of the final automaton depends on this ordering.

We propose a heuristic to order the children of nodes in $\mathcal{ACD}(\mathcal{A})$. Let $\mathcal{T}_i$ be a tree in $\mathcal{ACD}(\mathcal{A})$ and $x \in T_i$. We define:

$$D_i(x) = \{q' \in Q \,:\, q \xrightarrow{a} q' \notin \eta_i(x),\ \text{for some } q \in \mathit{States}(\eta_i(x)), a \in \Sigma\}.$$

The heuristic consists in ordering the children of a node $\mathcal{T}_i$ by decreasing $|D_i(x)|$. Experiments involving transformations towards state-based automata and testing this heuristic can be found in Section 6.2.

6 Degeneralization of Generalized Büchi Automata

The transformation of generalized-Büchi automata with n colors into Büchi automata (with a single color) is known as "*degeneralization*" and has been a very common processing step between algorithms that translate temporal-logic formulas into generalized-Büchi automata, and model-checking algorithms that (used to) only work with Büchi automata. While it initially consisted in making 2^n copies of the GBA [30, Appendix B] to remember the set of colors that had yet to be seen, degeneralization to state-based Büchi acceptance can be done using only $n+1$ copies once an arbitrary order of colors has been selected [13]. A similar construction to transition-based Büchi acceptance requires only n copies of the original automaton. Different orders of colors may lead to a different numbers of reachable states in the Büchi automaton. Some tools even attempted to start the degeneralization in different copies to reduce the number of reachable states [14]. Nowadays, an implementation such as the degeneralization of Spot implements several SCC-based optimizations [2] to reduce the number of output states, but is still sensitive to the arbitrary order selected for colors.

6.1 Transition-based Degeneralization

This order-sensitivity of the degeneralization, even in its transition-based variant, makes a striking difference with ACD. When applied to a generalized Büchi automaton that has some accepting and rejecting paths, the ACD-transform produces an automaton with acceptance $\mathsf{Inf}(0)\vee\mathsf{Fin}(1)$. Since all transitions are either labeled by 0 or 1, color 1 is superfluous[7] and the condition can be reduced to $\mathsf{Inf}(0)$. In this context, ACD-transform therefore gives us a transition-based Büchi automaton by duplicating the fewest number of states (Theorem 1(2)).

It can be seen that the cycling around the different children of the ACD (whose ordering is arbitrary) performed during ACD-transform is similar to the process used in traditional degeneralization. What makes the latter sensitive to color ordering is that it only "sees" one transition at a time, while the ACD provides a view of the cycles. For instance a degeneralization would process the sequence $(x)\xrightarrow{0}(y)\xrightarrow{1}(z)$ differently from the sequence $(x)\xrightarrow{1}(y)\xrightarrow{0}(z)$ depending on the order in which colors are expected to be encountered. However, if there is no other transition reaching or leaving (y) the two colors will always be seen together so their order should not matter: the two transitions belong to the same node of the ACD. The *propagation of colors* [28] is a related preprocessing step that can improve the degeneralization by propagating all colors common to the incoming transitions of a state to its outgoing transitions and vice-versa. It would turn the previous situation into $(x)\xrightarrow{0,1}(y)\xrightarrow{0,1}(z)$ making the color order selected by the degeneralization irrelevant (in this case).

A comparison of the output size of the traditional degeneralization implemented in Spot (which includes several optimizations learned over the years)

[7] In an automaton with "parity min" acceptance where all transitions are colored, the maximal color can always be omitted and replaced by the empty set.

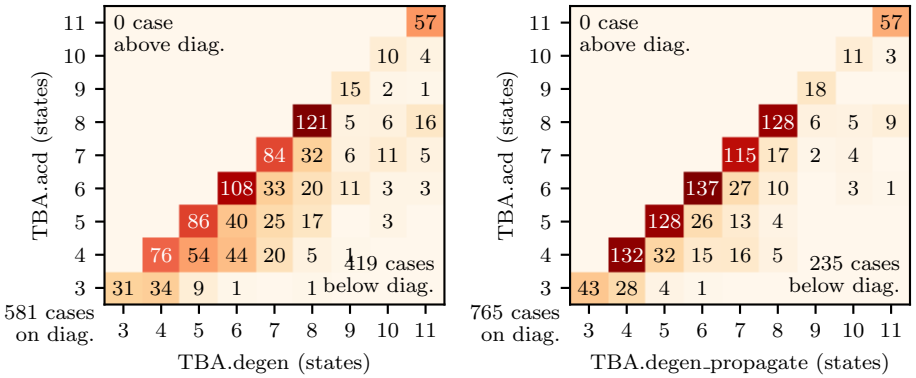


Fig. 3: Two-dimensional histogram of the sizes of 1000 automata, degeneralized to transition-based Büchi automata, using Spot's degeneralization function (with or without propagation of colors), or using ACD-transform.

against that of ACD-transform is given in the left plot of Figure 3. Unsurprisingly, because of ACD-transform's optimality, there are no cases where ACD loses to Spot's transition-based degeneralization. The use of the *propagation of colors* (right of the plot) is an improvement (the non-optimal cases dropped from 419 to 235) but not a cure.

Remark 5. The input automata used in this section and the next one is a set of 1000 randomly generated, minimal, deterministic, transition-based generalized Büchi automata, with 3 or 4 states and 2 or 3 colors. The reason for using such small minimal automata is to be able to use a SAT-based minimization [1] on the degeneralized state-based output in the next section to estimate how large the gap between an optimal and our procedure is.

6.2 State-based degeneralization

If ACD is used to produce a state-based output, as explained in Subsection 5.3, the obtained automaton is not guaranteed to be minimal with respect to locally bijective morphisms. In this case we can obtain a weaker optimality result:

Proposition 2. *Let $\mathcal{A}$ be a generalized Büchi automaton, and let $\mathcal{B}_{sb-\mathcal{ACD}(\mathcal{A})}$ be the state-based Büchi automaton obtained by applying the ACD state-based transformation. If $\mathcal{B}'$ be is a state-based Büchi automaton admitting a locally bijective morphism to $\mathcal{A}$, then $|\mathcal{B}_{sb-\mathcal{ACD}(\mathcal{A})}| \leq |\mathcal{B}'| + |\mathcal{A}|$.*

Proof. Let $\mathcal{B}'$ be a state-based Büchi automaton admitting a locally bijective morphism to $\mathcal{A}$. We can transform it into a transition-based Büchi automaton $\mathcal{B}'_{trans}$ by setting the transitions leaving accepting states to be accepting. This automaton has the same size than $\mathcal{B}'$ and it also accepts a locally bijective morphism to $\mathcal{A}$. Therefore, by Theorem 1, we have that $|\mathcal{B}_{\mathcal{ACD}(\mathcal{A})}| \leq |\mathcal{B}'_{trans}| =$

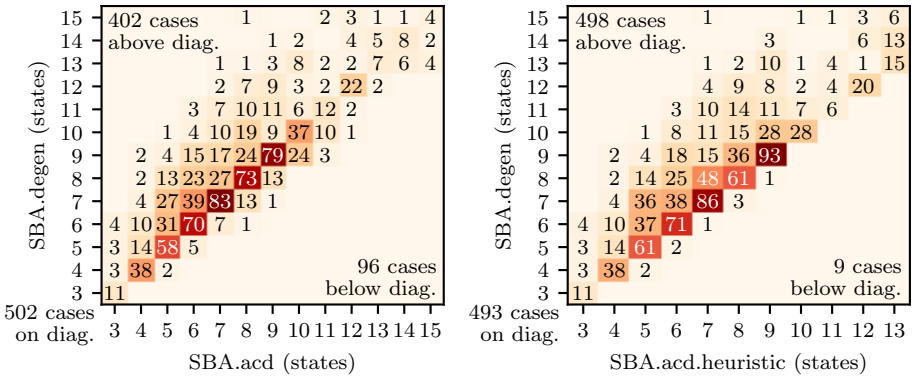


Fig. 4: Comparison of three ways to degeneralize to state-based Büchi: (acd, acd.heuristic) using the state-based version of ACD-transform with or without heuristic, and (degen) classical degeneralization.

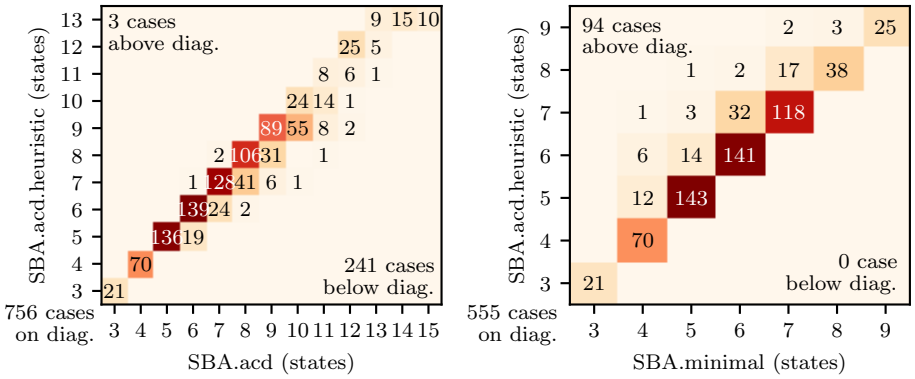


Fig. 5: Effect of the heuristic for ordering children of the ACD, and comparison to the minimal degeneralized automata (when known).

$|\mathcal{B}'|$, where $\mathcal{B}_{\mathcal{ACD}(\mathcal{A})}$ is the transition-based automaton obtained applying the ACD-transformation. We claim that $|\mathcal{B}_{sb-\mathcal{ACD}(\mathcal{A})}| \leq |\mathcal{B}_{\mathcal{ACD}(\mathcal{A})}| + |\mathcal{A}|$ (therefore implying that $|\mathcal{B}_{sb-\mathcal{ACD}(\mathcal{A})}| \leq |\mathcal{B}'| + |\mathcal{A}|$). Indeed, the set of states of $\mathcal{B}_{sb-\mathcal{ACD}(\mathcal{A})}$ is the union of the set of states of $\mathcal{B}_{\mathcal{ACD}(\mathcal{A})}$ and a subset of nodes of the form (q, ε), where ε is the root of T_q. There are at most $|\mathcal{A}|$ nodes of this form. □

Figure 4 compares three ways to perform state-based degeneralization. The ACD comes in two variants, with or without the heuristic of Section 5.3, and it is compared against the state-based degeneralization of Spot.

Figure 5 shows how the heuristic variant compares to the one without, and how it compares with the size of a minimal DBA, when its size could be computed in reasonable time (in 649 cases). Note that there might not be a local bijective

morphism between the input automaton and the minimal DBA computed this way, nonetheless these minimal size automata can serve as a reference point to estimate the quality of a degeneralization. Compared to this subset of minimal DBA, the average number of additional states produced by the state-based ACD is 0.17 with heuristics, and 0.33 without. Comparatively, Spot's degeneralization has an average of 1.21 extra states.

7 Deciding Typeness

We highlight now how the ACD can be used to decide *typeness* of deterministic TELA. This problem, first introduced by Krishnan and Brayton [19], consists of deciding whether we can replace the acceptance condition of a given automaton by another (hopefully simpler) without changing the transition structure and preserving the language (see Table 1 for a list of common acceptance conditions).

Let $\mathcal{A} = (Q, \Sigma, Q_0, \Delta, \Gamma, \alpha)$ be a TELA. We say that $\mathcal{A}$ is *X-type*, for $X \in \{\mathrm{B}, \mathrm{C}, \mathrm{GB}, \mathrm{GC}, \mathrm{P}, \mathrm{R}, \mathrm{S}\}$, if there is an X-automaton over the same structure, $\mathcal{A}' = (Q, \Sigma, Q_0, \Delta', \Gamma', \beta)$ (where Δ and Δ' only differ on the coloring of the transitions), such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ and β belongs to X. We emphasize that we permit to use a different set of colors Γ' in $\mathcal{A}'$. Some conditions can always be rewritten as conditions of other kinds (for example, Büchi conditions can be expressed as parity ones, so being B-type implies being P-type). We should not confuse this notion with the expressive power of deterministic automata using these conditions. For example, both deterministic parity automata and Rabin automata recognize all ω-regular languages, but there are Rabin automata that are not parity-type. Further, we say that an automaton $\mathcal{A}$ is *weak* if for every SCC $\mathcal{S}$ of $\mathcal{A}$, all cycles in $\mathcal{S}$ are accepting or all of them are rejecting.

The following result shows that the ACD is a sufficient data structure for deciding typeness for many common acceptance conditions. We remark that the second item adds to the results of Casares et al. [7] (this statement only holds if transitions of automata are labeled with subsets of colors, which is not allowed in their model).

Proposition 3 ([7, Section 5.2]). *Let $\mathcal{A}$ be a deterministic TELA such that all its states $q \in Q$ are reachable and let $\mathcal{ACD}(\mathcal{A}) = \langle \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle$ be its Alternating Cycle Decomposition. Then the following statements hold:*

1. *$\mathcal{A}$ is Rabin-type (resp. Streett type) if and only if for every $q \in Q$, every round node (resp. square node) of $\mathcal{T}_q$ has at most one child in $\mathcal{T}_q$. It is parity-type if and only if it is both Rabin and Streett-type.*
2. *$\mathcal{A}$ is generalized Büchi-type (resp. generalized co-Büchi-type) if and only if for every $1 \leq i \leq k$, $Height(\mathcal{T}_i) \leq 2$ and in case of equality, the root of T_i is a round node (resp. square node).*
3. *$\mathcal{A}$ is weak if and only if for every $1 \leq i \leq k$, $Height(\mathcal{T}_i) = 1$.*

Also, the least number of colors used by a deterministic parity automaton recognizing $\mathcal{L}(\mathcal{A})$ is $\max_{1 \leq i \leq k} Height(T_i) + \nu$, where $\nu = 0$ if the root of all trees of maximal height have the same shape (round or square), and $\nu = 1$ otherwise.

If one of the previous conditions holds, then $\mathcal{ACD}(\mathcal{A})$ also provides an effective procedure to relabel $\mathcal{A}$ with the corresponding acceptance condition.

Remark 6. The ACD gives a typeness result for each SCC of the automaton, which allows to simplify the acceptance condition of each of them independently. Further, implications from right to left in Proposition 3 also hold for non-deterministic automata.

Proposition 3 provides an effective procedure to check typeness of TELA: we just have to build the ACD and verify that it has the appropriate shape. Spot's implementation of ACD has options to abort the construction as soon as it detects that the shape is wrong. Moreover, if an automaton is parity-type, the ACD provides a method to relabel the automaton with a minimal number of colors. Finally, if the automaton already has parity acceptance, the ACD transformation boils down to the algorithm of Carton and Maceiras [5].

8 Availability

The ACD and the transformations based on it are currently implemented in two open-source tools: Spot 2.10 [9] and Owl 21.0 [18]. (The original developments were independent before the authors met and worked on this joint paper.)

In Spot 2.10, the ACD can be played with using the Python bindings. The `acd` class implements the decomposition, and will render it as an interactive forest of nodes that can be clicked to highlight the relevant cycles in the input automaton. The `acd_transform()` and `acd_transform_sbacc()` implements the transition-based and state-based variant of the paritization procedure. Additionally, the `acd` class has options to heuristically order the children to favor the state-based construction, or to abort the construction as soon as it is clear that the ACD does not have Rabin or Street shape (in case one wants to use it to establish typeness of automata). All these features are illustrated at https://spot.lrde.epita.fr/ipynb/zlktree.html. In the future, ACD will be used more by the rest of Spot, and will be one option of the `ltlsynt` tool (for LTL synthesis).

In Owl, the ACD transformation is available through the `aut2parity` command. This command reads an automaton in the HOA format [3] using arbitrary acceptance, and produces a parity automaton in the same format. The tool Strix [23], which builds upon Owl, gained in version 21.0.0 the option to use the ACD-construction as an intermediate step.

Instructions to reproduce all experiments and included in the artifact [8].

9 Conclusion

We have shown that ACD is more than a theoretically-appealing construction: our two implementations show that the construction is very usable in practice, and provide a baseline for further improvements. We have also shown that ACD is a Swiss-army knife for ω-automata in the sense that it can generalize and replace several specific constructions (paritization, degeneralization, typeness checks).

References

1. Baarir, S., Duret-Lutz, A.: Mechanizing the minimization of deterministic generalized Büchi automata. In: Proceedings of the 34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems (FORTE'14), Lecture Notes in Computer Science, vol. 8461, pp. 266–283, Springer (Jun 2014), https://doi.org/10.1007/978-3-662-43613-4_17
2. Babiak, T., Badie, T., Duret-Lutz, A., Křetínský, M., Strejček, J.: Compositional approach to suspension and other improvements to LTL translation. In: Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13), Lecture Notes in Computer Science, vol. 7976, pp. 81–98, Springer (Jul 2013), https://doi.org/10.1007/978-3-642-39176-7_6
3. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, pp. 479–486, Springer International Publishing (2015)
4. Battiti, R., , Protasi, M.: Handbook of Combinatorial Optimization: Volume 1–3, chap. Approximate Algorithms and Heuristics for MAX-SAT, pp. 77–148. Springer US (1998), ISBN 978-1-4613-0303-9, https://doi.org/10.1007/978-1-4613-0303-9_2
5. Carton, O., Maceiras, R.: Computing the Rabin index of a parity automaton. Informatique théorique et applications **33**(6), 495–505 (1999), URL http://www.numdam.org/item/ITA_1999__33_6_495_0/
6. Casares, A., Colcombet, T., Fijalkow, N.: Optimal transformations of games and automata using Muller conditions. In: Bansal, N., Merelli, E., Worrell, J. (eds.) Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP'21), Leibniz International Proceedings in Informatics (LIPIcs), vol. 198, pp. 123:1–123:14, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021), https://doi.org/10.4230/LIPIcs.ICALP.2021.123
7. Casares, A., Colcombet, T., Fijalkow, N.: Optimal transformations of muller conditions. Extended version of [6], on ArXiv. (2021), https://arxiv.org/abs/2011.13041
8. Casares, A., Duret-Lutz, A., Meyer, K.J., Renkin, F., Sickert, S.: Artifact for the paper "Practical applications of the alternating cycle decomposition". https://doi.org/10.5281/zenodo.5572613 (2021)
9. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16), Lecture Notes in Computer Science, vol. 9938, pp. 122–129, Springer (Oct 2016), https://doi.org/10.1007/978-3-319-46520-3_8
10. Emerson, E.A., Lei, C.L.: Modalities for model checking (extended abstract): Branching time strikes back. In: Proceedings of the 12th ACM symposium on Principles of Programming Languages (POPL'85), pp. 84–96, ACM (1985), https://doi.org/10.1145/318593.318620
11. Esparza, J., Křetínský, J., Raskin, J.F., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17), Lecture Notes in Computer Science, vol. 10205, pp. 426–442, Springer-Verlag (2017), https://doi.org/10.1007/978-3-662-54577-5_25
12. Esparza, J., Křetínský, J., Sickert, S.: A unified translation of linear temporal logic to ω-automata. J. ACM **67**(6) (Oct 2020), https://doi.org/10.1145/3417995

13. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), Lecture Notes in Computer Science, vol. 2102, pp. 53–65, Springer-Verlag (2001), https://doi.org/10.1007/3-540-44585-4_6
14. Giannakopoulou, D., Lerda, F.: From states to transitions: Improving translation of LTL formulæ to Büchi automata. In: Peled, D., Vardi, M. (eds.) Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02), Lecture Notes in Computer Science, vol. 2529, pp. 308–326, Springer-Verlag, Houston, Texas (Nov 2002)
15. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata Logics, and Infinite Games. Springer, Berlin, Heidelberg (2002), https://doi.org/10.1007/3-540-36387-4
16. Gurevich, Y., Harrington, L.: Trees, automata, and games. In: Proceedings of the 14th annual ACM symposium on Theory of computing (STOC'82), pp. 60–65 (1982), https://doi.org/10.1145/800070.802177
17. Jacobs, S., Bloem, R., Colange, M., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, P.J., Michaud, T., Sakr, M., Sickert, S., Tentrup, L., Walker, A.: The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR **abs/1904.07736** (2019), URL http://arxiv.org/abs/1904.07736
18. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA'18), Lecture Notes in Computer Science, vol. 11138, pp. 543–550, Springer (2018), https://doi.org/10.1007/978-3-030-01090-4_34
19. Krishnan, Sriram C.and Puri, A., Brayton, R.K.: Deterministic ω automata vis-a-vis deterministic buchi automata. In: Algorithms and Computation, pp. 378–386, Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
20. Křetínský, J., Meggendorfer, T., Waldmann, C., Weininger, M.: Index appearance record for transforming Rabin automata into parity automata. In: Legay, A., Margaria, T. (eds.) Proceedings of the 23st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17), Lecture Notes in Computer Science, vol. 10205, pp. 443–460 (2017), https://doi.org/10.1007/978-3-662-54577-5_26
21. Křetínský, J., Meggendorfer, T., Waldmann, C., Weininger, M.: Index appearance record with preorders. Acta Informatica (2021), https://doi.org/10.1007/s00236-021-00412-y
22. Löding, C.: Optimal bounds for transformations of ω-automata. In: Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'99), Lecture Notes in Computer Science, vol. 1738, pp. 97–109, Springer (1999), https://doi.org/10.1007/3-540-46691-6_8
23. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica pp. 3–36 (2020), https://doi.org/10.1007/s00236-019-00349-3
24. Löding, C.: Methods for the Transformation of ω-Automata: Complexity and Connection to Second Order Logic. Master's thesis, Institute of Computer Science and Applied Mathematics Christian-Albrechts-University of Kiel (1998), URL https://old.automata.rwth-aachen.de/users/loeding/diploma_loeding.pdf
25. Meyer, P., Sickert, S.: On the optimal and practical conversion of Emerson-Lei automata into parity automata. Unpublished manuscript, obsoleted by the work of Casares et al. [6]. (2021)

26. Michaud, T., Colange, M.: Reactive synthesis from LTL specification with Spot. In: Proceedings of the 7th Workshop on Synthesis, SYNT@CAV 2018, Electronic Proceedings in Theoretical Computer Science (2018)
27. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'89), pp. 179—190 (1989), https://doi.org/10.1145/75277.75293
28. Renkin, F., Duret-Lutz, A., Pommellet, A.: Practical "paritizing" of Emerson-Lei automata. In: Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA'20), Lecture Notes in Computer Science, vol. 12302, pp. 127–143, Springer (Oct 2020), https://doi.org/10.1007/978-3-030-59152-6_7
29. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Logics for Concurrency: Structure versus Automata, volume 1043 of Lecture Notes in Computer Science, pp. 238–266, Springer-Verlag (1996)
30. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the 1st Symposium on Logic in Computer Science (LICS'86), pp. 332–344, IEEE Computer Society Press (Jun 1986)
31. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science **200**(1), 135–183 (1998), https://doi.org/10.1016/S0304-3975(98)00009-7

Sky Is Not the Limit

Tighter Rank Bounds for Elevator Automata in Büchi Automata Complementation

Vojtěch Havlena, Ondřej Lengál✉, and Barbora Šmahlíková
ihavlena@fit.vut.cz, lengal@vut.cz, xsmahl00@vut.cz

Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

Abstract. We propose several heuristics for mitigating one of the main causes of combinatorial explosion in rank-based complementation of Büchi automata (BAs): unnecessarily high bounds on the ranks of states. First, we identify *elevator automata*, which is a large class of BAs (generalizing semi-deterministic BAs), occurring often in practice, where ranks of states are bounded according to the structure of strongly connected components. The bounds for elevator automata also carry over to general BAs that contain elevator automata as a sub-structure. Second, we introduce two techniques for refining bounds on the ranks of BA states using data-flow analysis of the automaton. We implement out techniques as an extension of the tool Ranker for BA complementation and show that they indeed greatly prune the generated state space, obtaining significantly better results and outperforming other state-of-the-art tools on a large set of benchmarks.

1 Introduction

Büchi automata (BA) complementation has been a fundamental problem underlying many applications since it was introduced in 1962 by Büchi [8,17] as an essential part of a decision procedure for a fragment of the second-order arithmetic. BA complementation has been used as a crucial part of, e.g., termination analysis of programs [13,20,10] or decision procedures for various logics, such as S1S [8], the first-order logic of Sturmian words [33], or the temporal logics ETL and QPTL [38]. Moreover, BA complementation also underlies BA inclusion and equivalence testing, which are essential instruments in the BA toolbox. Optimal algorithms, whose output asymptotically matches the lower bound of $(0.76n)^n$ [43] (potentially modulo a polynomial factor), have been developed [37,1]. For a successful real-world use, asymptotic optimality is, however, not enough and these algorithms need to be equipped with a range of optimizations to make them behave better than the worst case on BAs occurring in practice.

In this paper, we focus on the so-called *rank-based* approach to complementation, introduced by Kupferman and Vardi [24], further improved with the help of Friedgut [14], and finally made optimal by Schewe [37]. The construction stores in a macrostate partial information about all runs of a BA $\mathcal{A}$ over some word α. In addition to tracking states that $\mathcal{A}$ can be in (which is sufficient, e.g., in the determinization of NFAs), a macrostate also stores a guess of the rank of each of the tracked states in the *run DAG* that captures all these runs. The guessed ranks impose restrictions on how the future of a state might look like (i.e., when $\mathcal{A}$ may accept). The number of macrostates in the complement

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 118–136, 2022.
https://doi.org/10.1007/978-3-030-99527-0_7

depends combinatorially on the maximum rank that occurs in the macrostates. The constructions in [24,14,37] provides only coarse bounds on the maximum ranks.

A way of decreasing the maximum rank has been suggested in [15] using a PSPACE (and, therefore, not really practically applicable) algorithm (the problem of finding the optimal rank is PSPACE-complete). In our previous paper [19], we have identified several basic optimizations of the construction that can be used to refine the *tight-rank upper bound* (TRUB) on the maximum ranks of states. In this paper, we push the applicability of rank-based techniques much further by introducing two novel lightweight techniques for refining the TRUB, thus significantly reducing the generated state space.

Firstly, we introduce a new class of the so-called *elevator automata*, which occur quite often in practice (e.g., as outputs of natural algorithms for translating LTL to BAs). Intuitively, an elevator automaton is a BA whose strongly connected components (SCCs) are all either inherently weak[1] or deterministic. Clearly, the class substantially generalizes the popular inherently weak [6] and semi-deterministic BAs [11,3,4]). The structure of elevator automata allows us to provide tighter estimates of the TRUBs, not only for elevator automata *per se*, but also for BAs where elevator automata occur as a sub-structure (which is even more common). Secondly, we propose a lightweight technique, inspired by data flow analysis, allowing to propagate rank restriction along the skeleton of the complemented automaton, obtaining even tighter TRUBs. We also extended the optimal rank-based algorithm to transition-based BAs (TBAs).

We implemented our optimizations within the RANKER tool [18] and evaluated our approach on thousands of hard automata from the literature (15 % of them were elevator automata that were not semi-deterministic, and many more contained an elevator sub-structure). Our techniques drastically reduce the generated state space; in many cases we even achieved exponential improvement compared to the optimal procedure of Schewe and our previous heuristics. The new version of RANKER gives a smaller complement in the majority of cases of hard automata than other state-of-the-art tools.

2 Preliminaries

Words, functions. We fix a finite nonempty alphabet Σ and the first infinite ordinal $\omega = \{0, 1, \ldots\}$. For $n \in \omega$, by $[n]$ we denote the set $\{0, \ldots, n\}$. For $i \in \omega$ we use $\llbracket i \rrbracket$ to denote the largest even number smaller of equal to i, e.g., $\llbracket 42 \rrbracket = \llbracket 43 \rrbracket = 42$. An (infinite) word α is represented as a function $\alpha\colon \omega \to \Sigma$ where the i-th symbol is denoted as α_i. We abuse notation and sometimes also represent α as an infinite sequence $\alpha = \alpha_0\alpha_1 \ldots$ We use Σ^ω to denote the set of all infinite words over Σ. For a (partial) function $f\colon X \to Y$ and a set $S \subseteq X$, we define $f(S) = \{f(x) \mid x \in S\}$. Moreover, for $x \in X$ and $y \in Y$, we use $f \triangleleft \{x \mapsto y\}$ to denote the function $(f \setminus \{x \mapsto f(x)\}) \cup \{x \mapsto y\}$.

Büchi automata. A (nondeterministic transition/state-based) *Büchi automaton* (BA) over Σ is a quadruple $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$ where Q is a finite set of *states*, $\delta\colon Q \times \Sigma \to 2^Q$ is a *transition function*, $I \subseteq Q$ is the sets of *initial* states, and $Q_F \subseteq Q$ and $\delta_F \subseteq \delta$ are the sets of *accepting states* and *accepting transitions* respectively. We sometimes treat δ as a set of transitions $p \xrightarrow{a} q$, for instance, we use $p \xrightarrow{a} q \in \delta$ to denote that $q \in \delta(p, a)$.

[1] An SCC is inherently weak if it either contains no accepting states or, on the other hand, all cycles of the SCC contain an accepting state.

Moreover, we extend δ to sets of states $P \subseteq Q$ as $\delta(P,a) = \bigcup_{p\in P} \delta(p,a)$, and to sets of symbols $\Gamma \subseteq \Sigma$ as $\delta(P,\Gamma) = \bigcup_{a\in\Gamma} \delta(P,a)$. We define the inverse transition function as $\delta^{-1} = \{p \xrightarrow{a} q \mid q \xrightarrow{a} p \in \delta\}$. The notation $\delta|_S$ for $S \subseteq Q$ is used to denote the restriction of the transition function $\delta \cap (S \times \Sigma \times S)$. Moreover, for $q \in Q$, we use $\mathcal{A}[q]$ to denote the BA $(Q, \delta, \{q\}, Q_F \cup \delta_F)$.

A *run* of $\mathcal{A}$ from $q \in Q$ on an input word α is an infinite sequence $\rho\colon \omega \to Q$ that starts in q and respects δ, i.e., $\rho_0 = q$ and $\forall i \geq 0\colon \rho_i \xrightarrow{\alpha_i} \rho_{i+1} \in \delta$. Let $\inf_Q(\rho)$ denote the states occurring in ρ infinitely often and $\inf_\delta(\rho)$ denote the transitions occurring in ρ infinitely often. The run ρ is called *accepting* iff $\inf_Q(\rho) \cap Q_F \neq \emptyset$ or $\inf_\delta(\rho) \cap \delta_F \neq \emptyset$.

A word α is accepted by $\mathcal{A}$ from a state $q \in Q$ if there is an accepting run ρ of $\mathcal{A}$ from q, i.e., $\rho_0 = q$. The set $\mathcal{L}_\mathcal{A}(q) = \{\alpha \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } \alpha \text{ from } q\}$ is called the *language* of q (in $\mathcal{A}$). Given a set of states $R \subseteq Q$, we define the language of R as $\mathcal{L}_\mathcal{A}(R) = \bigcup_{q\in R} \mathcal{L}_\mathcal{A}(q)$ and the language of $\mathcal{A}$ as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_\mathcal{A}(I)$. We say that a state $q \in Q$ is *useless* iff $\mathcal{L}_\mathcal{A}(q) = \emptyset$. If $\delta_F = \emptyset$, we call $\mathcal{A}$ *state-based* and if $Q_F = \emptyset$, we call $\mathcal{A}$ *transition-based*. In this paper, we fix a BA $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$.

3 Complementing Büchi automata

In this section, we describe a generalization of the rank-based complementation of state-based BAs presented by Schewe in [37] to our notion of transition/state-based BAs. Proofs can be found in [16].

3.1 Run DAGs

First, we recall the terminology from [37] (which is a minor modification of the one in [24]), which we use in the paper. Let the *run DAG* of $\mathcal{A}$ over a word α be a DAG (directed acyclic graph) $\mathcal{G}_\alpha = (V, E)$ containing vertices V and edges E such that

- $V \subseteq Q \times \omega$ s.t. $(q,i) \in V$ iff there is a run ρ of $\mathcal{A}$ from I over α with $\rho_i = q$,
- $E \subseteq V \times V$ s.t. $((q,i),(q',i')) \in E$ iff $i' = i+1$ and $q' \in \delta(q, \alpha_i)$.

Given $\mathcal{G}_\alpha$ as above, we will write $(p,i) \in \mathcal{G}_\alpha$ to denote that $(p,i) \in V$. A vertex $(p,i) \in V$ is called *accepting* if p is an accepting state and an edge $((q,i),(q',i')) \in E$ is called *accepting* if $q \xrightarrow{\alpha_i} q'$ is an accepting transition. A vertex $v \in \mathcal{G}_\alpha$ is *finite* if the set of vertices reachable from v is finite, *infinite* if it is not finite, and *endangered* if it cannot reach an accepting vertex or an accepting edge.

We assign ranks to vertices of run DAGs as follows: Let $\mathcal{G}_\alpha^0 = \mathcal{G}_\alpha$ and $j = 0$. Repeat the following steps until the fixpoint or for at most $2n+1$ steps, where $n = |Q|$.

- Set $rank_\alpha(v) \leftarrow j$ for all finite vertices v of $\mathcal{G}_\alpha^j$ and let $\mathcal{G}_\alpha^{j+1}$ be $\mathcal{G}_\alpha^j$ minus the vertices with the rank j.
- Set $rank_\alpha(v) \leftarrow j+1$ for all endangered vertices v of $\mathcal{G}_\alpha^{j+1}$ and let $\mathcal{G}_\alpha^{j+2}$ be $\mathcal{G}_\alpha^{j+1}$ minus the vertices with the rank $j+1$.
- Set $j \leftarrow j+2$.

For all vertices v that have not been assigned a rank yet, we assign $rank_\alpha(v) \leftarrow \omega$.

We define the *rank of* α, denoted as $rank(\alpha)$, as $\max\{rank_\alpha(v) \mid v \in \mathcal{G}_\alpha\}$ and the *rank of* $\mathcal{A}$, denoted as $rank(\mathcal{A})$, as $\max\{rank(w) \mid w \in \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})\}$.

Lemma 1. *If* $\alpha \notin \mathcal{L}(\mathcal{A})$*, then* $rank(\alpha) \leq 2|Q|$.

3.2 Rank-Based Complementation

In this section, we describe a construction for complementing BAs developed in the work of Kupferman and Vardi [24]—later improved by Friedgut, Kupferman, and Vardi [14], and by Schewe [37]—extended to our definition of BAs with accepting states and transitions (see [19] for a step-by-step introduction). The construction is based on the notion of tight level rankings storing information about levels in run DAGs. For a BA $\mathcal{A}$ and $n = |Q|$, a *(level) ranking* is a function $f\colon Q \to [2n]$ such that $f(Q_F) \subseteq \{0, 2, \ldots, 2n\}$, i.e., f assigns even ranks to accepting states of $\mathcal{A}$. For two rankings f and f' we define $f \overset{a}{\underset{S}{\ominus\!\!\rightarrow}} f'$ iff for each $q \in S$ and $q' \in \delta(q, a)$ we have $f'(q') \leq f(q)$ and for each $q'' \in \delta_F(q, a)$ it holds $f'(q'') \leq \lfloor\!\lfloor f(q) \rfloor\!\rfloor$. The set of all rankings is denoted by $\mathcal{R}$. For a ranking f, the *rank* of f is defined as $rank(f) = \max\{f(q) \mid q \in Q\}$. We use $f \leq f'$ iff for every state $q \in Q$ we have $f(q) \leq f'(q)$ and we use $f < f'$ iff $f \leq f'$ and there is a state $q \in Q$ with $f(q) < f'(q)$. For a set of states $S \subseteq Q$, we call f to be *S-tight* if (i) it has an odd rank r, (ii) $f(S) \supseteq \{1, 3, \ldots, r\}$, and (iii) $f(Q \setminus S) = \{0\}$. A ranking is *tight* if it is Q-tight; we use $\mathcal{T}$ to denote the set of all tight rankings.

The original rank-based construction [24] uses macrostates of the form (S, O, f) to track all runs of $\mathcal{A}$ over α. The f-component contains guesses of the ranks of states in S (which is obtained by the classical subset construction) in the run DAG and the O-set is used to check whether all runs contain only a finite number of accepting states. Friedgut, Kupferman, and Vardi [14] improved the construction by having f consider only tight rankings. Schewe's construction [37] extends the macrostates to (S, O, f, i) with $i \in \omega$ representing a particular even rank such that O tracks states with rank i. At the cut-point (a macrostate with $O = \emptyset$) the value of i is changed to $i + 2$ modulo the rank of f. Macrostates in an accepting run hence iterate over all possible values of i. Formally, the complement of $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$ is given as the (state-based) BA $\textsc{Schewe}(\mathcal{A}) = (Q', \delta', I', Q'_F \cup \emptyset)$, whose components are defined as follows:

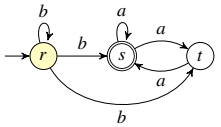
- $Q' = Q_1 \cup Q_2$ where
  - $Q_1 = 2^Q$ and
  - $Q_2 = \{(S, O, f, i) \in 2^Q \times 2^Q \times \mathcal{T} \times \{0, 2, \ldots, 2n-2\} \mid f \text{ is } S\text{-tight}, O \subseteq S \cap f^{-1}(i)\}$,
- $I' = \{I\}$,
- $\delta' = \delta_1 \cup \delta_2 \cup \delta_3$ where
  - $\delta_1\colon Q_1 \times \Sigma \to 2^{Q_1}$ such that $\delta_1(S, a) = \{\delta(S, a)\}$,
  - $\delta_2\colon Q_1 \times \Sigma \to 2^{Q_2}$ such that $\delta_2(S, a) = \{(S', \emptyset, f, 0) \mid S' = \delta(S, a), f \text{ is } S'\text{-tight}\}$, and
  - $\delta_3\colon Q_2 \times \Sigma \to 2^{Q_2}$ such that $(S', O', f', i') \in \delta_3((S, O, f, i), a)$ iff
    - $S' = \delta(S, a)$,
    - $f \overset{a}{\underset{S}{\ominus\!\!\rightarrow}} f'$,
    - $rank(f) = rank(f')$,
    - and
      - if $O = \emptyset$ then $i' = (i + 2) \mod (rank(f') + 1)$ and $O' = f'^{-1}(i')$, and
      - if $O \neq \emptyset$ then $i' = i$ and $O' = \delta(O, a) \cap f'^{-1}(i)$; and
- $Q'_F = \{\emptyset\} \cup ((2^Q \times \{\emptyset\} \times \mathcal{T} \times \omega) \cap Q_2)$.

We call the part of the automaton with states from Q_1 the *waiting* part (denoted as Waiting), and the part corresponding to Q_2 the *tight* part (denoted as Tight).

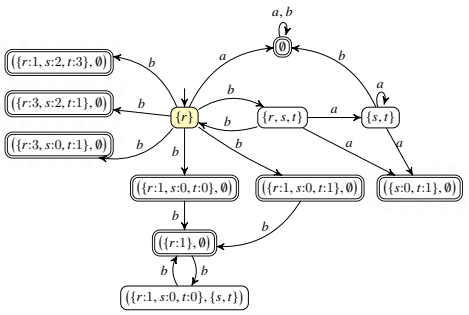
Theorem 2. *Let $\mathcal{A}$ be a BA. Then $\mathcal{L}(\textsc{Schewe}(\mathcal{A})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

The space complexity of Schewe's construction for BAs matches the theoretical lower bound $O((0.76n)^n)$ given by Yan [43] modulo a quadratic factor $O(n^2)$. Note that our extension to BAs with accepting transitions does not increase the space complexity of the construction.

Example 3. Consider the BA $\mathcal{A}$ over $\{a, b\}$ given in Fig. 1a. A part of $\textsc{Schewe}(\mathcal{A})$ is shown in Fig. 1b (we use $(\{s{:}0, t{:}1\}, \emptyset)$ to denote the macrostate $(\{s, t\}, \emptyset, \{s \mapsto 0, t \mapsto 1\}, 0)$). We omit the i-part of each macrostate since the corresponding values are 0 for all macrostates in the figure. Useless states are covered by grey stripes. The full automaton contains even more transitions from $\{r\}$ to useless macrostates of the form $(\{r{:}\cdot, s{:}\cdot, t{:}\cdot\}, \emptyset)$. □



(a) BA $\mathcal{A}$ over $\{a, b\}$



(b) A part of $\textsc{Schewe}(\mathcal{A})$

Fig. 1: Schewe's complementation

From the construction of $\textsc{Schewe}(\mathcal{A})$, we can see that the number of states is affected mainly by sizes of macrostates and by the maximum rank of $\mathcal{A}$. In particular, the upper bound on the number of states of the complement with the maximum rank r is given in the following lemma.

Lemma 4. *For a BA $\mathcal{A}$ with sufficiently many states n such that $rank(\mathcal{A}) = r$ the number of states of the complemented automaton is bounded by $2^n + \frac{(r+m)^n}{(r+m)!}$ where $m = \max\{0, 3 - \lceil \frac{r}{2} \rceil\}$.*

From Lemma 1 we have that the rank of $\mathcal{A}$ is bounded by $2|Q|$. Such a bound is often too coarse and hence $\textsc{Schewe}(\mathcal{A})$ may contain many redundant states. Decreasing the bound on the ranks is essential for a practical algorithm, but an optimal solution is PSpace-complete [15]. The rest of this paper therefore proposes a framework of lightweight techniques for decreasing the maximum rank bound and, in this way, significantly reducing the size of the complemented BA.

3.3 Tight Rank Upper Bounds

Let $\alpha \notin \mathcal{L}(\mathcal{A})$. For $\ell \in \omega$, we define the ℓ-th *level* of $\mathcal{G}_\alpha$ as $level_\alpha(\ell) = \{q \mid (q, \ell) \in \mathcal{G}_\alpha\}$. Furthermore, we use f_ℓ^α to denote the ranking of level ℓ of $\mathcal{G}_\alpha$. Formally,

$$f_\ell^\alpha(q) = \begin{cases} rank_\alpha((q, \ell)) & \text{if } q \in level_\alpha(\ell), \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

We say that the ℓ-th level of $\mathcal{G}_\alpha$ is *tight* if for all $k \geq \ell$ it holds that (i) f_k^α is tight, and (ii) $rank(f_k^\alpha) = rank(f_\ell^\alpha)$. Let $\rho = S_0 S_1 \ldots S_{\ell-1}(S_\ell, O_\ell, f_\ell, i_\ell) \ldots$ be a run on a word

α in Schewe($\mathcal{A}$). We say that ρ is a *super-tight run* [19] if $f_k = f_k^\alpha$ for each $k \geq \ell$. Finally, we say that a mapping $\mu\colon 2^Q \to \mathcal{R}$ is a *tight rank upper bound (TRUB) wrt* α iff

$$\exists \ell \in \omega\colon level_\alpha(\ell) \text{ is tight} \wedge (\forall k \geq \ell\colon \mu(level_\alpha(k)) \geq f_k^\alpha). \tag{2}$$

Informally, a TRUB is a ranking that gives a conservative (i.e., larger) estimate on the necessary ranks of states in a super-tight run. We say that μ is a TRUB iff μ is a TRUB wrt all $\alpha \notin \mathcal{L}(\mathcal{A})$. We abuse notation and use the term TRUB also for a mapping $\mu'\colon 2^Q \to \omega$ if the mapping $inner(\mu')$ is a TRUB where $inner(\mu')(S) = \{q \mapsto m \mid m = \mu'(S) \dot{-} 1 \text{ if } q \in Q_F \text{ else } m = \mu'(S)\}$ for all $S \in 2^Q$. ($\dot{-}$ is the *monus* operator, i.e., minus with negative results saturated to zero.) Note that the mappings $\mu_t = \{S \mapsto (2|S \setminus Q_F| \dot{-} 1)\}_{S \in 2^Q}$ and $inner(\mu_t)$ are trivial TRUBs.

The following lemma shows that we can remove from Schewe($\mathcal{A}$) macrostates whose ranking is not covered by a TRUB (in particular, we show that the reduced automaton preserves super-tight runs).

Lemma 5. *Let μ be a TRUB and $\mathcal{B}$ be a BA obtained from Schewe($\mathcal{A}$) by replacing all occurrences of Q_2 by $Q_2' = \{(S, O, f, i) \mid f \leq \mu(S)\}$. Then, $\mathcal{L}(\mathcal{B}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$.*

4 Elevator Automata

In this section, we introduce *elevator automata*, which are BAs having a particular structure that can be exploited for complementation and semi-determinization; elevator automata can be complemented in $O(16^n)$ (cf. Lemma 10) space instead of $2^{O(n \log n)}$, which is the lower bound for unrestricted BAs, and semi-determinized in $O(2^n)$ instead of $O(4^n)$ (cf. [16]). The class of elevator automata is quite general: it can be seen as a substantial generalization of semi-deterministic BAs (SDBAs) [11,5]. Intuitively, an elevator automaton is a BA whose strongly connected components are all either deterministic or inherently weak.

Let $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$. $C \subseteq Q$ is a *strongly connected component* (SCC) of $\mathcal{A}$ if for any pair of states $q, q' \in C$ it holds that q is reachable from q' and q' is reachable from q. C is *maximal* (MSCC) if it is not a proper subset of another SCC. An MSCC C is *trivial* iff $|C| = 1$ and $\delta|_C = \emptyset$. The *condensation* of $\mathcal{A}$ is the DAG $cond(\mathcal{A}) = (\mathcal{M}, \mathcal{E})$ where $\mathcal{M}$ is the set of $\mathcal{A}$'s MSCCs and $\mathcal{E} = \{(C_1, C_2) \mid \exists q_1 \in C_1, \exists q_2 \in C_2, \exists a \in \Sigma\colon q_1 \xrightarrow{a} q_2 \in \delta\}$. An MSCC is *non-accepting* if it contains no accepting state and no accepting transition, i.e., $C \cap Q_F = \emptyset$ and $\delta|_C \cap \delta_F = \emptyset$. The *depth* of $(\mathcal{M}, \mathcal{E})$ is defined as the number of MSCCs on the longest path in $(\mathcal{M}, \mathcal{E})$.

We say that an SCC C is *inherently weak accepting* (IWA) iff *every cycle* in the transition diagram of $\mathcal{A}$ restricted to C contains an accepting state or an accepting transition. C is *inherently weak* if it is either non-accepting or IWA, and $\mathcal{A}$ is inherently weak if all of its MSCCs are inherently weak. $\mathcal{A}$ is *deterministic* iff $|I| \leq 1$ and $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. An SCC $C \subseteq Q$ is *deterministic* iff $(C, \delta|_C, \emptyset, \emptyset)$ is deterministic. $\mathcal{A}$ is a *semi-deterministic BA* (SDBA) if $\mathcal{A}[q]$ is deterministic for every $q \in Q_F \cup \{p \in Q \mid s \xrightarrow{a} p \in \delta_F, s \in Q, a \in \Sigma\}$, i.e., whenever a run in $\mathcal{A}$ reaches an accepting state or an accepting transition, it can only continue deterministically.

$\mathcal{A}$ is an *elevator (Büchi) automaton* iff for every MSCC C of $\mathcal{A}$ it holds that C is (i) deterministic, (ii) IWA, or (iii) non-accepting. In other words, a BA is an elevator automaton iff every nondeterministic SCC of $\mathcal{A}$ that contains an accepting state or transition is inherently weak. An example of an elevator automaton obtained from the LTL formula $\mathsf{GF}(a \vee \mathsf{GF}(b \vee \mathsf{GF}c))$ is shown in Fig. 2. The BA consists of three connected deterministic components. Note that the automaton is neither semi-deterministic nor unambiguous.

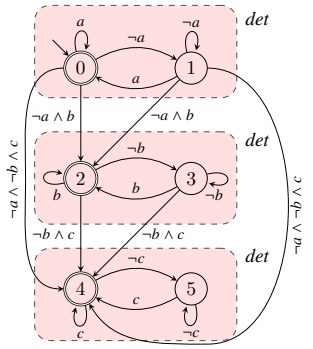


Fig. 2: The BA for LTL formula $\mathsf{GF}(a \vee \mathsf{GF}(b \vee \mathsf{GF}c))$ is elevator

The rank of an elevator automaton $\mathcal{A}$ does not depend on the number of states (as in general BAs), but only on the number of MSCCs and the depth of $cond(\mathcal{A})$. In the worst case, $\mathcal{A}$ consists of a chain of deterministic components, yielding the upper bound on the rank of elevator automata given in the following lemma.

Lemma 6. *Let $\mathcal{A}$ be an elevator automaton such that its condensation has the depth d. Then $rank(\mathcal{A}) \leq 2d$.*

4.1 Refined Ranks for Elevator Automata

Notice that the upper bound on ranks provided by Lemma 6 can still be too coarse. For instance, for an SDBA with three linearly ordered MSCCs such that the first two MSCCs are non-accepting and the last one is deterministic accepting, the lemma gives us an upper bound on the rank 6, while it is known that every SDBA has the rank at most 3 (cf. [5]). Another examples might be two deterministic non-trivial MSCCs connected by a path of trivial MSCCs, which can be assigned the same rank.

Instead of refining the definition of elevator automata into some quite complex list of constraints, we rather provide an algorithm that performs a traversal through $cond(\mathcal{A})$ and assigns each MSCC a label of the form $\boxed{type{:}rank}$ that contains (i) a type and (ii) a bound on the maximum rank of states in the component. The types of MSCCs that we consider are the following:

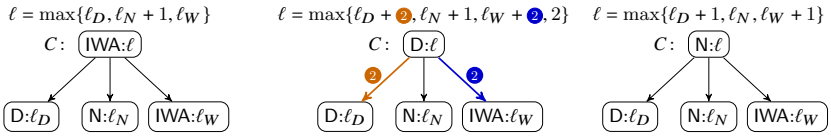
T**:** *trivial* components,
IWA**:** *inherently weak accepting* components,
D**:** *deterministic* (potentially accepting) components, and
N**:** *non-accepting* components.

Note that the type in an MSCC is not given *a priori* but is determined by the algorithm (this is because for deterministic non-accepting components, it is sometimes better to treated them as D and sometimes as N, depending on their neighbourhood). In the following, we assume that $\mathcal{A}$ is an elevator automaton without useless states and, moreover, all accepting conditions on states and transitions not inside non-trivial MSCCs are removed (any BA can be easily transformed into this form).

We start with terminal MSCCs C, i.e., MSCCs that cannot reach any other MSCC:

T1: If C is IWA, then we label it with $\boxed{\text{IWA:0}}$.

T2: Else if C is deterministic accepting, we label it with $\boxed{\text{D:2}}$.



(a) C is IWA (b) C is D (c) C is N

Fig. 3: Rules for assigning types and rank bounds to MSCCs. The symbols 2 and 2 are interpeted as 0 if all the corresponding edges from the components having rank ℓ_D and ℓ_W, respectively, are deterministic; otherwise they are interpreted as 2. Transitions between two components C_1 and C_2 are deterministic if the BA $(C, \delta|_C, \emptyset, \emptyset)$ is deterministic for $C = \delta(C_1, \Sigma) \cap (C_1 \cup C_2)$.

(Note that the previous two options are complete due to our requirements on the structure of $\mathcal{A}$.) When all terminal MSCCs are labelled, we proceed through $cond(\mathcal{A})$, inductively on its structure, and label non-terminal components C based on the rules defined below.

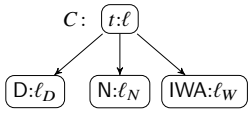


Fig. 4: Structure of elevator ranking rules

The rules are of the form that uses the structure depicted in Fig. 4, where children nodes denote already processed MSCCs. In particular, a child node of the form $k{:}\ell_k$ denotes an aggregate node of *all* siblings of the type k with ℓ_k being the maximum rank of these siblings. Moreover, we use $\text{typemax}\{e_D, e_N, e_W\}$ to denote the type $j \in \{\mathsf{D}, \mathsf{N}, \mathsf{IWA}\}$ for which $e_j = \max\{e_D, e_N, e_W\}$ where e_i is an expression containing ℓ_i (if there are more such types, j is chosen arbitrarily). The rules for assigning a type t and a rank ℓ to C are the following:

I1: If C is trivial, we set $t = \text{typemax}\{\ell_D, \ell_N, \ell_W\}$ and $\ell = \max\{\ell_D, \ell_N, \ell_W\}$.
I2: Else if C is IWA, we use the rule in Fig. 3a.
I3: Else if C is deterministic accepting, we use the rule in Fig. 3b.
I4: Else if C is deterministic and non-accepting, we try both rules from Figs. 3b and 3c and pick the rule that gives us a smaller rank.
I5: Else if C is nondeterministic and non-accepting, we use the rule in Fig. 3c.

Then, for every MSCC C of $\mathcal{A}$, we assign each of its states the rank of C. We use $\chi\colon Q \to \omega$ to denote the rank bounds computed by the procedure above.

Lemma 7. *χ is a TRUB.*

Using Lemma 5, we can now use χ to prune states during the construction of Schewe($\mathcal{A}$), as shown in the following example.

Example 8. As an example, consider the BA $\mathcal{A}$ in Fig. 1a. The set of MSCCs with their types is given as

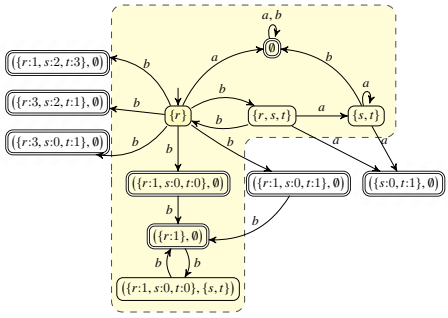


Fig. 5: A part of Schewe($\mathcal{A}$). The TRUB computed by elevator rules is used to prune states outside the yellow area.

$\{\{r\}{:}\mathsf{N}, \{s,t\}{:}\mathsf{IWA}\}$ showing that BA $\mathcal{A}$ is an elevator. Using the rules **T1** and **I4** we get the TRUB $\chi = \{r{:}1, s{:}0, t{:}0\}$. The TRUB can be used to prune the generated states as shown in Fig. 5. □

4.2 Efficient Complementation of Elevator Automata

In Section 4.1 we proposed an algorithm for assigning ranks to MSCCs of an elevator automaton $\mathcal{A}$. The drawback of the algorithm is that the maximum obtained rank is not bounded by a constant but by the depth of the condensation of $\mathcal{A}$. We will, however, show that it is actually possible to change $\mathcal{A}$ by at most doubling the number of states and obtain an elevator BA with the rank at most 3.

Intuitively, the construction copies every non-trivial MSCC C with an accepting state or transition into a component $C^\bullet$, copies all transitions going into states in C to also go into the corresponding states in $C^\bullet$, and, finally, removes all accepting conditions from C. Formally, let $\mathcal{A} = (Q, \delta, I, Q_F \cup \delta_F)$ be a BA. For $C \subseteq Q$, we use $C^\bullet$ to denote a unique copy of C, i.e., $C^\bullet = \{q^\bullet \mid q \in C\}$ s.t. $C^\bullet \cap Q = \emptyset$. Let $\mathcal{M}$ be the set of MSCCs of $\mathcal{A}$. Then, the *deelevated* BA $\textsc{DeElev}(\mathcal{A}) = (Q', \delta', I', Q'_F \cup \delta'_F)$ is given as follows:

- $Q' = Q \cup Q^\bullet$,
- $\delta' : Q' \times \Sigma \to 2^{Q'}$ where for $q \in Q$
  - $\delta'(q,a) = \delta(q,a) \cup (\delta(q,a))^\bullet$ and
  - $\delta'(q^\bullet, a) = (\delta(q,a) \cap C)^\bullet$ for $q \in C \in \mathcal{M}$;
- $I' = I$, and
- $Q'_F = Q^\bullet_F$ and $\delta'_F = \{q^\bullet \xrightarrow{a} r^\bullet \mid q \xrightarrow{a} r \in \delta_F\} \cap \delta'$.

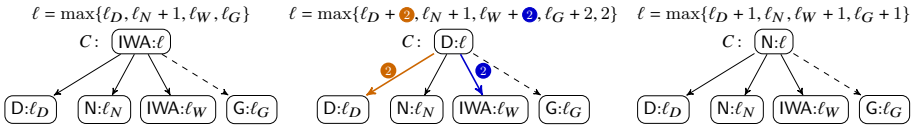
It is easy to see that the number of states of the deelevated automaton is bounded by $2|Q|$. Moreover, if $\mathcal{A}$ is elevator, so is $\textsc{DeElev}(\mathcal{A})$. The construction preserves the language of $\mathcal{A}$, as shown by the following lemma.

Lemma 9. *Let $\mathcal{A}$ be a BA. Then, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\textsc{DeElev}(\mathcal{A}))$.*

Moreover, for an elevator automaton $\mathcal{A}$, the structure of $\textsc{DeElev}(\mathcal{A})$ consists of (after trimming useless states) several non-accepting MSCCs with copied terminal deterministic or IWA MSCCs. Therefore, if we apply the algorithm from Section 4.1 on $\textsc{DeElev}(\mathcal{A})$, we get that its rank is bounded by 3, which gives the following upper bound for complementation of elevator automata.

Lemma 10. *Let $\mathcal{A}$ be an elevator automaton with suffficiently many states n. Then the language $\Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$ can be represented by a BA with at most $O(16^n)$ states.*

The complementation through $\textsc{DeElev}(\mathcal{A})$ gives a better upper bound than the rank refinement from Section 4.1 applied directly on $\mathcal{A}$, however, based on our experience, complementation through $\textsc{DeElev}(\mathcal{A})$ behaves worse in many real-world instances. This poor behaviour is caused by the fact that the complement of $\textsc{DeElev}(\mathcal{A})$ can have a larger Waiting and macrostates in Tight can have larger S-components, which can yield more generated states (despite the rank bound 3). It seems that the most promising approach would to be a combination of the approaches, which we leave for future work.



(a) C is IWA (b) C is D (c) C is N

Fig. 6: Rules assigning types and rank bounds for non-elevator automata.

4.3 Refined Ranks for Non-Elevator Automata

The algorithm from Section 4.1 computing a TRUB for elevator automata can be extended to compute TRUBs even for general non-elevator automata (i.e., BAs with nondeterministic accepting components that are not inherently weak). To achieve this generalization, we extend the rules for assigning types and ranks to MSCCs of elevator automata from Section 4.1 to take into account general non-deterministic components. For this, we add into our collection of MSCC types *general* components (denoted as G). Further, we need to extend the rules for terminal components with the following rule:

T3: Otherwise, we label C with $\boxed{\mathsf{G}{:}2|C \setminus Q_F|}$.

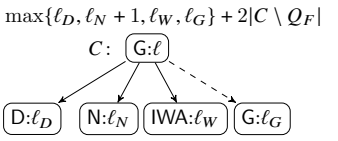


Fig. 7: C is G

Moreover, we adjust the rules for assigning a type t and a rank ℓ to C to the following (the rule **I1** is the same as for the case of elevator automata):

I2–I5: *(We replace the corresponding rules for their counterparts including general components from Fig. 6).*

I6: Otherwise, we use the rule in Fig. 7.

Then, for every MSCC C of a BA $\mathcal{A}$, we assign each of its states the rank of C. Again, we use $\chi : Q \to \omega$ to denote the rank bounds computed by the adjusted procedure above.

Lemma 11. *χ is a TRUB.*

5 Rank Propagation

In the previous section, we proposed a way, how to obtain a TRUB for elevator automata (with generalization to general automata). In this section, we propose a way of using the structure of $\mathcal{A}$ to refine a TRUB using a propagation of values and thus reduce the size of Tight. Our approach uses *data flow analysis* [32] to reason on how ranks and rankings of macrostates of Schewe($\mathcal{A}$) can be decreased based on the ranks and rankings of the *local neighbourhood* of the macrostates. We, in particular, use a special case of *forward analysis* working on the *skeleton* of Schewe($\mathcal{A}$), which is defined as the BA $\mathcal{K}_\mathcal{A} = (2^Q, \delta', \emptyset, \emptyset)$ where $\delta' = \{R \xrightarrow{a} S \mid S = \delta(R, a)\}$ (note that we are only interested in the structure of $\mathcal{K}_\mathcal{A}$ and

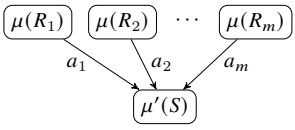


Fig. 8: Rank propagation flow

not its language; also notice the similarity of $\mathcal{K}_{\mathcal{A}}$ with WAITING). Our analysis refines a rank/ranking estimate $\mu(S)$ for a macrostate S of $\mathcal{K}_{\mathcal{A}}$ based on the estimates for its predecessors $R_1, \ldots, R_m$ (see Fig. 8). The new estimate is denoted as $\mu'(S)$.

More precisely, $\mu\colon 2^Q \to \mathbb{V}$ is a function giving each macrostate of $\mathcal{K}_{\mathcal{A}}$ a value from the domain $\mathbb{V}$. We will use the following two value domains: (i) $\mathbb{V} = \omega$, which is used for estimating *ranks* of macrostates (in the *outer macrostate analysis*) and (ii) $\mathbb{V} = \mathcal{R}$, which is used for estimating *rankings* within macrostates (in the *inner macrostate analysis*). For each of the analyses, we will give the *update function* $up\colon (2^Q \to \mathbb{V}) \times (2^Q)^{m+1} \to \mathbb{V}$, which defines how the value of $\mu(S)$ is updated based on the values of $\mu(R_1), \ldots, \mu(R_m)$. We then construct a system with the following equation for every $S \in 2^Q$:

$$\mu(S) = up(\mu, S, R_1, \ldots, R_m) \quad \text{where } \{R_1, \ldots, R_m\} = \delta'^{-1}(S, \Sigma). \tag{3}$$

We then solve the system of equations using standard algorithms for data flow analysis (see, e.g., [32, Chapter 2]) to obtain the fixpoint μ^*. Our analyses have the important property that if they start with μ_0 being a TRUB, then μ^* will also be a TRUB.

As the initial TRUB, we can use a trivial TRUB or any other TRUB (e.g., the output of elevator state analysis from Section 4).

5.1 Outer Macrostate Analysis

We start with the simpler analysis, which is the *outer macrostate analysis*, which only looks at sizes of macrostates. Recall that the rank r of every super-tight run in SCHEWE($\mathcal{A}$) does not change, i.e., a super tight run stays in WAITING as long as needed so that when it jumps to TIGHT, it takes the rank r and never needs to decrease it. We can use this fact to decrease the maximum rank of a macrostate S in $\mathcal{K}_{\mathcal{A}}$. In particular, let us consider all cycles going through S. For each of the cycles c, we can bound the maximum rank of a super-tight run going through c by $2m - 1$ where m is the smallest number of non-accepting states occurring in any macrostate on c (from the definition, the rank of a tight ranking does not depend on accepting states). Then we can infer that the maximum rank of any super-tight run going through S is bounded by the maximum rank of any of the cycles going through S (since S can never assume a higher rank in any super-tight run). Moreover, the rank of each cycle can also be estimated in a more precise way, e.g. using our elevator analysis.

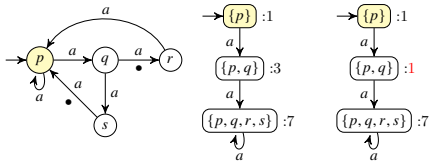
Since the number of cycles in $\mathcal{K}_{\mathcal{A}}$ can be large[2], instead of their enumeration, we employ data flow analysis with the value domain $\mathbb{V} = \omega$ (i.e, for every macrostate S of $\mathcal{K}_{\mathcal{A}}$, we remember a bound on the maximum rank of S) and the following update function:

$$up_{out}(\mu, S, R_1, \ldots, R_m) = \min\{\mu(S), \max\{\mu(R_1), \ldots, \mu(R_m)\}\}. \tag{4}$$

Intuitively, the new bound on the maximum rank of S is taken as the smaller of the previous bound $\mu(S)$ and the largest of the bounds of all predecessors of S, and the new value is propagated forward by the data flow analysis.

[2] $\mathcal{K}_{\mathcal{A}}$ can be exponentially larger than $\mathcal{A}$ and the number of cycles in $\mathcal{K}_{\mathcal{A}}$ can be exponential to the size of $\mathcal{K}_{\mathcal{A}}$, so the total number of cycles can be double-exponential.

Example 12. Consider the BA $\mathcal{A}_{ex}$ in Fig. 9a. When started from the initial TRUB $\mu_0 = \{\{p\} \mapsto 1, \{p,q\} \mapsto 3, \{p,q,r,s\} \mapsto 7\}$ (Fig. 9b), outer macrostate analysis decreases the maximum rank estimate for $\{p,q\}$ to 1, since $\min\{\mu_0(\{p,q\}, \max\{\mu_0(\{p\})\}\} = \min\{3,1\} = 1$. The estimate for $\{p,q,r,s\}$ is not affected, because $\min\{7, \max\{1,7\}\} = 7$ (Fig. 9c). □



(a) $\mathcal{A}_{ex}$ (b) μ_0 (c) μ^*_{out}

Fig. 9: Example of outer macrostate analysis. (a) $\mathcal{A}_{ex}$ (• denotes accepting transitions). The initial TRUB μ_0 in (b) is refined to μ^*_{out} in (c).

Lemma 13. *If μ is a TRUB, then $\mu \lhd \{S \mapsto up_{out}(\mu, S, R_1, \ldots, R_m)\}$ is a TRUB.*

Corollary 14. *When started with a TRUB μ_0, the outer macrostate analysis terminates and returns a TRUB μ^*_{out}.*

5.2 Inner Macrostate Analysis

Our second analysis, called *inner macrostate analysis*, looks deeper into super-tight runs in Schewe($\mathcal{A}$). In particular, compared with the outer macrostate analysis from the previous section—which only looks at the *ranks*, i.e., the bounds on the numbers in the rankings—, inner macrostate analysis looks at how the *rankings* assign concrete values to the *states* of $\mathcal{A}$ *inside the macrostates*.

Inner macrostate analysis is based on the following. Let ρ be a super-tight run of Schewe($\mathcal{A}$) on $\alpha \notin \mathcal{L}(\mathcal{A})$ and (S, O, f, i) be a macrostate from Tight. Because ρ is super-tight, we know that the rank $f(q)$ of a state $q \in S$ is bounded by the ranks of the predecessors of q. This holds because in super-tight runs, the ranks are only *as high as necessary*; if the rank of q were higher than the ranks of its predecessors, this would mean that we may wait in Waiting longer and only jump to q with a lower rank later.

Let us introduce some necessary notation. Let $f, f' \in \mathcal{R}$ be rankings (i.e., $f, f' \colon Q \to \omega$). We use $f \sqcup f'$ to denote the ranking $\{q \mapsto \max\{f(q), f'(q)\} \mid q \in Q\}$, and $f \sqcap f'$ to denote the ranking $\{q \mapsto \min\{f(q), f'(q)\} \mid q \in Q\}$. Moreover, we define *max-succ-rank*$^a_S(f) = \max_{\leq}\{f' \in \mathcal{R} \mid f \mathrel{\ominus\!\!\rightarrow}^a_S f'\}$ and a function $dec \colon \mathcal{R} \to \mathcal{R}$ such that $dec(\theta)$ is the ranking θ' for which

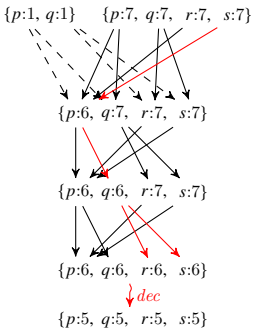
$$\theta'(q) = \begin{cases} \theta(q) \dot{-} 1 & \text{if } \theta(q) = rank(\theta) \text{ and } q \notin Q_F, \\ \llbracket \theta(q) \dot{-} 1 \rrbracket & \text{if } \theta(q) = rank(\theta) \text{ and } q \in Q_F, \\ \theta(q) & \text{otherwise.} \end{cases} \tag{5}$$

Intuitively, *max-succ-rank*$^a_S(f)$ is the (pointwise) maximum ranking that can be reached from macrostate S with ranking f over a (it is easy to see that there is a unique such maximum ranking) and $dec(\theta)$ decreases the maximum ranks in a ranking θ by one (or by two for even maximum ranks and accepting states).

The analysis uses the value domain $\mathbb{V} = \mathcal{R}$ (i.e., each macrostate of $\mathcal{K}_{\mathcal{A}}$ is assigned a ranking giving an upper bound on the rank of each state in the macrostate) and the update function up_{in} given in the right-hand side of the page. Intuitively, up_{in}

updates $\mu(q)$ for every $q \in S$ to hold the maximum rank compatible with the ranks of its predecessors. We note line Line 6, which makes use of the fact that we can only consider tight rankings (whose rank is odd), so we can decrease the estimate using the function dec defined above.

```
1 up_in(μ, S, R_1, ..., R_m):
2   foreach 1 ≤ i ≤ m and a ∈ Σ do
3     if δ(R_i, a) = S then
4       g_i^a ← max-succ-rank^a_{R_i}(μ(R_i))
5   θ ← μ(S) ⊓ ⨆{g_i^a | g_i^a is defined};
6   if rank(θ) is even then θ ← dec(θ);
7   return θ;
```



Example 15. Let us continue in Section 5.1 and perform inner macrostate analysis starting with the TRUB $\{\{p{:}1\}, \{p{:}1, q{:}1\}, \{p{:}7, q{:}7, r{:}7, s{:}7\}\}$ obtained from μ^*_{out}. We show three iterations of the algorithm for $\{p, q, r, s\}$ in the right-hand side (we do not show $\{p, q\}$ except the first iteration since it does not affect intermediate steps). We can notice that in the three iterations, we could decrease the maximum rank estimate to $\{p{:}6, q{:}6, r{:}6, s{:}6\}$ due to the accepting transitions from r and s. In the last of the three iterations, when all states have the even rank 6, the condition on Line 6 would become true and the rank of all states would be decremented to 5 using dec. Then, again, the accepting transitions from r and s would decrease the rank of p to 4, which would be propagated to q and so on. Eventually, we would arrive to the TRUB $\{p{:}1, q{:}1, r{:}1, s{:}1\}$, which could not be decreased any more, since $\{p{:}1, q{:}1\}$ forces the ranks of r and s to stay at 1. □

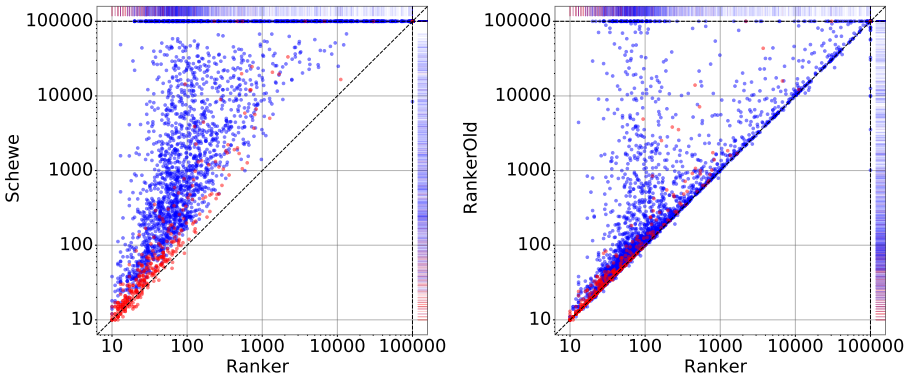
Lemma 16. *If μ is a TRUB, then $\mu \lhd \{S \mapsto up_{in}(\mu, S, R_1, \ldots, R_m)\}$ is a TRUB.*

Corollary 17. *When started with a TRUB μ_0, the inner macrostate analysis terminates and returns a TRUB μ^*_{in}.*

6 Experimental Evaluation

Used tools and evaluation environment. We implemented the techniques described in the previous sections as an extension of the tool Ranker [18] (written in C++). Speaking in the terms of [19], the heuristics were implemented on top of the Ranker$_{\text{MaxR}}$ configuration (we refer to this previous version as Ranker$_{\text{Old}}$). We tested the correctness of our implementation using Spot's `autcross` on all BAs in our benchmark. We compared modified Ranker with other state-of-the-art tools, namely, Goal [41] (implementing Piterman [34], Schewe [37], Safra [36], and Fribourg [1]), Spot 2.9.3 [12] (implementing Redziejowski's algorithm [35]), Seminator 2 [4], LTL2dstar 0.5.4 [23], and Roll [26]. All tools were set to the mode where they output an automaton with the standard state-based Büchi acceptance condition. The experimental evaluation was performed on a 64-bit GNU/Linux Debian workstation with an Intel(R) Xeon(R) CPU E5-2620 running at 2.40 GHz with 32 GiB of RAM and using a timeout of 5 minutes.

Datasets. As the source of our benchmark, we use the two following datasets: (i) `random` containing 11,000 BAs over a two letter alphabet used in [40], which were randomly



(a) RANKER vs SCHEWE (b) RANKER vs RANKER$_{\text{OLD}}$

Fig. 10: Comparison of the state space generated by our optimizations and other rank-based procedures (horizontal and vertical dashed lines represent timeouts). Blue data points are from `random` and red data points are from `LTL`. Axes are logarithmic.

generated via the Tabakov-Vardi approach [39], starting from 15 states and with various different parameters; (ii) `LTL` with 1,721 BAs over larger alphabets (up to 128 symbols) used in [4], which were obtained from LTL formulae from literature (221) or randomly generated (1,500). We preprocessed the automata using RABIT [30] and SPOT's `autfilt` (using the `--high` simplification level), transformed them to state-based acceptance BAs (if they were not already), and converted to the HOA format [2]. From this set, we removed automata that were (i) semi-deterministic, (ii) inherently weak, (iii) unambiguous, or (iv) have an empty language, since for these automata types there exist more efficient complementation procedures than for unrestricted BAs [5,4,6,28]. In the end, we were left with **2,592** (`random`) and **414** (`LTL`) *hard* automata. We use `all` to denote their union (**3,006** BAs). Of these hard automata, 458 were elevator automata.

6.1 Generated State Space

In our first experiment, we evaluated the effectiveness of our heuristics for pruning the generated state space by comparing the sizes of complemented BAs without postprocessing. This use case is directed towards applications where postprocessing is irrelevant, such as inclusion or equivalence checking of BAs.

We focused on a comparison with two less optimized versions of the rank-based complementation procedure: SCHEWE (the version "Reduced Average Outdegree" from [37] implemented in GOAL under `-m rank -tr -ro`) and its optimization RANKER$_{\text{OLD}}$. The scatter plots in Fig. 10 compare the numbers of states of automata generated by RANKER and the other algorithms and the upper part of Table 1 gives summary statistics. Observe that our optimizations from this paper drastically reduced the generated search space compared with both SCHEWE and RANKER$_{\text{OLD}}$ (the mean for SCHEWE is lower than for RANKER$_{\text{OLD}}$ due to its much higher number of timeouts); from Fig. 10b we can see that the improvement was in many cases *exponential* even when compared with our previous optimizations in RANKER$_{\text{OLD}}$. The median (which is a more meaningful indicator with the presence of timeouts) decreased by 44 % w.r.t. RANKER$_{\text{OLD}}$, and we also reduced the

Table 1: Statistics for our experiments. The upper part compares various optimizations of the rank-based procedure (no postprocessing). The lower part compares RANKER to other approaches (with postprocessing). The left-hand side compares sizes of complement BAs and the right-hand side runtimes of the tools. The **wins** and **losses** columns give the number of times when RANKER was strictly better and worse. The values are given for the three datasets as "`all` (`random` : `LTL`)". Approaches in GOAL are labelled with ⚽.

method	mean	median	wins	losses	mean runtime [s]	median runtime [s]	timeouts
RANKER	3812 (4452 : 207)	79 (93 : 26)			7.83 (8.99 : 1.30)	0.51 (0.84 : 0.04)	279 (276 : 3)
RANKER$_{\text{OLD}}$	7398 (8688 : 358)	141 (197 : 29)	2190 (2011 : 179)	111 (107 : 4)	9.37 (10.73 : 1.99)	0.61 (1.04 : 0.04)	365 (360 : 5)
SCHEWE ⚽	4550 (5495 : 665)	439 (774 : 35)	2640 (2315 : 325)	55 (1 : 54)	21.05 (24.28 : 7.80)	6.57 (7.39 : 5.21)	937 (928 : 9)
RANKER	47 (52 : 18)	22 (27 : 10)			7.83 (8.99 : 1.30)	0.51 (0.84 : 0.04)	279 (276 : 3)
PITERMAN ⚽	73 (82 : 22)	28 (34 : 14)	1435 (1124 : 311)	416 (360 : 56)	7.29 (7.39 : 6.65)	5.99 (6.04 : 5.62)	14 (12 : 2)
SAFRA ⚽	83 (91 : 30)	29 (35 : 17)	1562 (1211 : 351)	387 (350 : 37)	14.11 (15.05 : 8.37)	6.71 (6.92 : 5.79)	172 (158 : 14)
SPOT	75 (85 : 15)	24 (32 : 10)	1087 (936 : 151)	683 (501 : 182)	0.86 (0.99 : 0.06)	0.02 (0.02 : 0.02)	13 (13 : 0)
FRIBOURG ⚽	91 (104 : 13)	23 (31 : 9)	1120 (1055 : 65)	601 (376 : 225)	17.79 (19.53 : 7.22)	9.25 (10.15 : 5.48)	81 (80 : 1)
LTL2DSTAR	73 (82 : 21)	28 (34 : 13)	1465 (1195 : 270)	465 (383 : 82)	3.31 (3.84 : 0.11)	0.04 (0.05 : 0.02)	136 (130 : 6)
SEMINATOR 2	79 (91 : 15)	21 (29 : 10)	1266 (1131 : 135)	571 (367 : 204)	9.51 (11.25 : 0.08)	0.22 (0.39 : 0.02)	363 (362 : 1)
ROLL	18 (19 : 14)	10 (9 : 11)	2116 (1858 : 258)	569 (443 : 126)	31.23 (37.85 : 7.28)	8.19 (12.23 : 2.74)	1109 (1106 : 3)

number of timeouts by 23 %. Notice that the numbers for the `LTL` dataset do not differ as much as for `random`, witnessing the easier structure of the BAs in `LTL`.

6.2 Comparison with Other Complementation Techniques

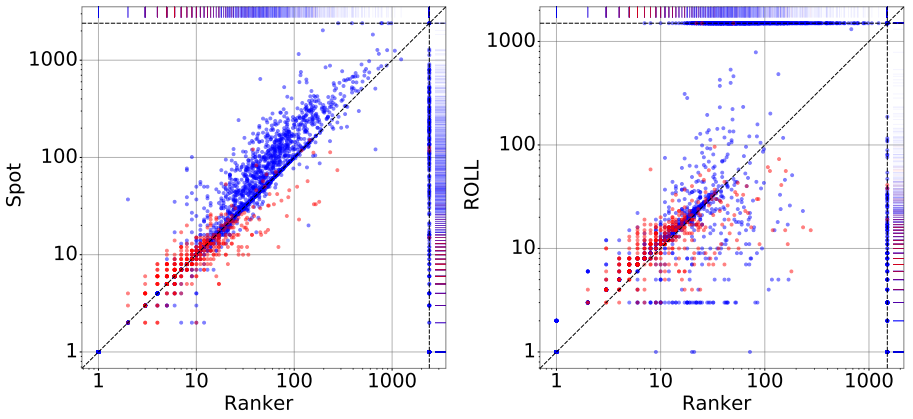
In our second experiment, we compared the improved RANKER with other state-of-the-art tools. We were comparing sizes of output BAs, therefore, we postprocessed each output automaton with `autfilt` (simplification level `--high`). Scatter plots are given in Fig. 11, where we compare RANKER with SPOT (which had the best results on average from the other tools except ROLL) and ROLL, and summary statistics are in the lower part of Table 1. Observe that RANKER has by far the lowest mean (except ROLL) and the third lowest median (after SEMINATOR 2 and ROLL, but with less timeouts). Moreover, comparing the numbers in columns **wins** and **losses** we can see that RANKER gives strictly better results than other tools (**wins**) more often than the other way round (**losses**).

In Fig. 11a see that indeed in the majority of cases RANKER gives a smaller BA than SPOT, especially for harder BAs (SPOT, however, behaves slightly better on the simpler BAs from `LTL`). The results in Fig. 11b do not seem so clear. ROLL uses a learning-based approach—more heavyweight and completely orthogonal to any of the other tools—and can in some cases output a tiny automaton, but does not scale, as observed by the number of timeouts much higher than any other tool. It is, therefore, positively surprising that RANKER could in most of the cases still obtain a much smaller automaton than ROLL.

Regarding runtimes, the prototype implementation in RANKER is comparable to SEMINATOR 2, but slower than SPOT and LTL2DSTAR (SPOT is the fastest tool). Implementations of other approaches clearly do not target speed. We note that the number of timeouts of RANKER is still higher than of some other tools (in particular PITERMAN, SPOT, FRIBOURG); further state space reduction targeting this particular issue is our future work.

7 Related Work

BA complementation remains in the interest of researchers since their first introduction by Büchi in [8]. Together with a hunt for efficient complementation techniques, the effort has been put into establishing the lower bound. First, Michel showed that the lower bound is $n!$ (approx. $(0.36n)^n$) [31] and later Yan refined the result to $(0.76n)^n$ [43].



(a) RANKER vs SPOT (b) RANKER vs ROLL

Fig. 11: Comparison of the complement size obtained by RANKER and other state-of-the-art tools (horizontal and vertical dashed lines represent timeouts). Axes are logarithmic.

The complementation approaches can be roughly divided into several branches. *Ramsey-based complementation*, the very first complementation construction, where the language of an input automaton is decomposed into a finite number of equivalence classes, was proposed by Büchi and was further enhanced in [7]. *Determinization-based complementation* was presented by Safra in [36] and later improved by Piterman in [34] and Redziejowski in [35]. Various optimizations for determinization of BAs were further proposed in [29]. The main idea of this approach is to convert an input BA into an equivalent deterministic automaton with different acceptance condition that can be easily complemented (e.g. Rabin automaton). The complemented automaton is then converted back into a BA (often for the price of some blow-up). *Slice-based complementation* tracks the acceptance condition using a reduced abstraction on a run tree [42,21]. *A learning-based approach* was introduced in [27,26]. Allred and Ultes-Nitsche then presented a novel optimal complementation algorithm in [1]. For some special types of BAs, e.g., deterministic [25], semi-deterministic [5], or unambiguous [28], there exist specific complementation algorithms. *Semi-determinization based complementation* converts an input BA into a semi-deterministic BA [11], which is then complemented [4].

Rank-based complementation, studied in [24,15,14,37,22], extends the subset construction for determinization of finite automata by storing additional information in each macrostate to track the acceptance condition of all runs of the input automaton. Optimizations of an alternative (sub-optimal) rank-based construction from [24] going through *alternating Büchi automata* were presented in [15]. Furthermore, the work in [22] introduces an optimization of SCHEWE, in some cases producing smaller automata (this construction is not compatible with our optimizations). As shown in [9], the rank-based construction can be optimized using simulation relations. We identified several heuristics that help reducing the size of the complement in [19], which are compatible with the heuristics in this paper.

Acknowledgements. We thank anonymous reviewers for their useful remarks that helped us improve the quality of the paper. This work was supported by the Czech Science Foundation project 20-07487S and the FIT BUT internal project FIT-S-20-6427.

References

1. Allred, J.D., Ultes-Nitsche, U.: A simple and optimal complementation algorithm for Büchi automata. In: Proceedings of the Thirty third Annual IEEE Symposium on Logic in Computer Science (LICS 2018). pp. 46–55. IEEE Computer Society Press (July 2018)
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The Hanoi omega-automata format. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_31
3. Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.H.: Complementing semi-deterministic büchi automata. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 770–787. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
4. Blahoudek, F., Duret-Lutz, A., Strejček, J.: Seminator 2 can complement generalized Büchi automata via improved semi-determinization. In: Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV'20). Lecture Notes in Computer Science, vol. 12225, pp. 15–27. Springer (Jul 2020)
5. Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.: Complementing semi-deterministic Büchi automata. In: Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 770–787. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_49
6. Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2083, pp. 611–625. Springer (2001). https://doi.org/10.1007/3-540-45744-5_50
7. Breuers, S., Löding, C., Olschewski, J.: Improved Ramsey-based Büchi complementation. In: Proc. of FOSSACS'12. pp. 150–164. Springer (2012)
8. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. of International Congress on Logic, Method, and Philosophy of Science 1960. Stanford Univ. Press, Stanford (1962)
9. Chen, Y., Havlena, V., Lengál, O.: Simulations in rank-based Büchi automata complementation. In: Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11893, pp. 447–467. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_23
10. Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 135–150. ACM (2018). https://doi.org/10.1145/3192366.3192405
11. Courcoubetis, C., Yannakakis, M.: Verifying temporal properties of finite-state probabilistic programs. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 338–345. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21950
12. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Automated Technology for Verification and Analysis. pp. 122–129. Springer International Publishing, Cham (2016)

13. Fogarty, S., Vardi, M.Y.: Büchi complementation and size-change termination. In: Proc. of TACAS'09. pp. 16–30. Springer (2009)
14. Friedgut, E., Kupferman, O., Vardi, M.: Büchi complementation made tighter. International Journal of Foundations of Computer Science **17**, 851–868 (2006)
15. Gurumurthy, S., Kupferman, O., Somenzi, F., Vardi, M.Y.: On complementing nondeterministic Büchi automata. In: Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings. LNCS, vol. 2860, pp. 96–110. Springer (2003). https://doi.org/10.1007/978-3-540-39724-3_10
16. Havlena, V., Lengál, O., Smahlíková, B.: Sky is not the limit: Tighter rank bounds for elevator automata in Büchi automata complementation (technical report). CoRR **abs/2110.10187** (2021), https://arxiv.org/abs/2110.10187
17. Havlena, V., Lengál, O., Šmahlíková, B.: Deciding S1S: Down the rabbit hole and through the looking glass. In: Proceedings of NETYS'21. pp. 215–222. No. 12754 in LNCS, Springer Verlag (2021). https://doi.org/10.1007/978-3-030-91014-3_15
18. Havlena, V., Lengál, O., Šmahlíková, B.: Ranker (2021), https://github.com/vhavlena/ranker
19. Havlena, V., Lengál, O.: Reducing (To) the Ranks: Efficient Rank-Based Büchi Automata Complementation. In: Proc. of CONCUR'21. LIPIcs, vol. 203, pp. 2:1–2:19. Schloss Dagstuhl, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.2, iSSN: 1868-8969
20. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Proc. of CAV'14. pp. 797–813. Springer (2014)
21. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Proc. of ICALP'08. pp. 724–735. Springer (2008)
22. Karmarkar, H., Chakraborty, S.: On minimal odd rankings for Büchi complementation. In: Proc. of ATVA'09. LNCS, vol. 5799, pp. 228–243. Springer (2009). https://doi.org/10.1007/978-3-642-04761-9_18
23. Klein, J., Baier, C.: On-the-fly stuttering in the construction of deterministic *omega* -automata. In: Proc. of CIAA'07. LNCS, vol. 4783, pp. 51–61. Springer (2007). https://doi.org/10.1007/978-3-540-76336-9_7
24. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001). https://doi.org/10.1145/377978.377993
25. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. J. Comput. Syst. Sci. **35**(1), 59–71 (1987). https://doi.org/10.1016/0022-0000(87)90036-5
26. Li, Y., Sun, X., Turrini, A., Chen, Y., Xu, J.: ROLL 1.0: ω-regular language learning library. In: Proc. of TACAS'19. LNCS, vol. 11427, pp. 365–371. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_23
27. Li, Y., Turrini, A., Zhang, L., Schewe, S.: Learning to complement Büchi automata. In: Proc. of VMCAI'18. pp. 313–335. Springer (2018)
28. Li, Y., Vardi, M.Y., Zhang, L.: On the power of unambiguity in Büchi complementation. In: Proc. of GandALF'20. EPTCS, vol. 326, pp. 182–198. Open Publishing Association (2020). https://doi.org/10.4204/EPTCS.326.12
29. Löding, C., Pirogov, A.: New optimizations and heuristics for determinization of büchi automata. In: Automated Technology for Verification and Analysis. pp. 317–333. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_18
30. Mayr, R., Clemente, L.: Advanced automata minimization. In: Proc. of POPL'13. pp. 63–74 (2013)
31. Michel, M.: Complementation is more difficult with automata on infinite words. CNET, Paris **15** (1988)
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6

33. Oei, R., Ma, D., Schulz, C., Hieronymi, P.: Pecan: An automated theorem prover for automatic sequences using büchi automata. CoRR **abs/2102.01727** (2021), https://arxiv.org/abs/2102.01727
34. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: Proc. of LICS'06. pp. 255–264. IEEE (2006)
35. Redziejowski, R.R.: An improved construction of deterministic omega-automaton using derivatives. Fundam. Informaticae **119**(3-4), 393–406 (2012). https://doi.org/10.3233/FI-2012-744
36. Safra, S.: On the complexity of ω-automata. In: Proc. of FOCS'88. pp. 319–327. IEEE (1988)
37. Schewe, S.: Büchi complementation made tight. In: Proc. of STACS'09. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl (2009). https://doi.org/10.4230/LIPIcs.STACS.2009.1854
38. Sistla, A.P., Vardi, M.Y., Wolper, P.: The Complementation Problem for Büchi Automata with Applications to Temporal Logic. Theoretical Computer Science **49**(2-3), 217–237 (1987)
39. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Proc. of LPAR'05. pp. 396–411. Springer (2005)
40. Tsai, M.H., Fogarty, S., Vardi, M.Y., Tsay, Y.K.: State of Büchi complementation. In: Implementation and Application of Automata. pp. 261–271. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
41. Tsai, M.H., Tsay, Y.K., Hwang, Y.S.: GOAL for games, omega-automata, and logics. In: Computer Aided Verification. pp. 883–889. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
42. Vardi, M.Y., Wilke, T.: Automata: From logics to algorithms. Logic and Automata **2**, 629–736 (2008)
43. Yan, Q.: Lower bounds for complementation of ω-automata via the full automata technique. In: Automata, Languages and Programming. pp. 589–600. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

On-The-Fly Solving for Symbolic Parity Games

Maurice Laveaux[1](✉), Wieger Wesselink[1], and Tim A.C. Willemse[1,2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] ESI (TNO), Eindhoven, The Netherlands
{m.laveaux, j.w.wesselink, t.a.c.willemse}@tue.nl

Abstract. Parity games can be used to represent many different kinds of decision problems. In practice, tools that use parity games often rely on a specification in a higher-order logic from which the actual game can be obtained by means of an exploration. For many of these decision problems we are only interested in the solution for a designated vertex in the game. We formalise how to use on-the-fly solving techniques during the exploration process, and show that this can help to decide the winner of such a designated vertex in an incomplete game. Furthermore, we define partial solving techniques for incomplete parity games and show how these can be made resilient to work directly on the incomplete game, rather than on a set of *safe* vertices. We implement our techniques for symbolic parity games and study their effectiveness in practice, showing that speed-ups of several orders of magnitude are feasible and overhead (if unavoidable) is typically low.

1 Introduction

A parity game is a two-player game with an ω-regular winning condition, played by players $\Diamond$ ('even') and $\Box$ ('odd') on a directed graph. The true complexity of solving parity games is still a major open problem, with the most recent breakthroughs yielding algorithms running in quasi-polynomial time, see, *e.g.*, [18,7]. Apart from their intriguing status, parity games pop up in various fundamental results in computer science (*e.g.*, in the proof of decidability of a monadic second-order theory). In practice, parity games provide an elegant, uniform framework to encode many relevant decision problems, which include model checking problems, synthesis problems and behavioural equivalence checking problems.

Often, a decision problem that is encoded as a parity game, can be answered by determining which of the two players wins a designated vertex in the game graph. Depending on the characteristics of the game, it may be the case that only a fraction of the game is relevant for deciding which player wins a vertex. For instance, deciding whether a transition system satisfies an invariant can be encoded by a simple, solitaire (*i.e.*, single player) parity game. In such a game, player $\Box$ wins all vertices that are sinks (*i.e.*, have no successors), and all states leading to such sinks, so checking whether sinks are reachable from a designated vertex suffices to determine whether this vertex is won by $\Box$, too. Clearly, as soon as a sink is detected, any further inspection of the game becomes irrelevant.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 137–155, 2022.
https://doi.org/10.1007/978-3-030-99527-0_8

A complicating factor is that in practice, the parity games that encode decision problems are not given explicitly. Rather, they are specified in some higher-order logic such as a parameterised Boolean equation system, see, *e.g.* [11]. Exploring the parity game from such a higher-order specification is, in general, time-and memory-consuming. To counter this, symbolic exploration techniques have been proposed, see *e.g.* [19]. These explore the game graph on-the-fly and exploit efficient symbolic data structures such as LDDs [13] to represent sets of vertices and edges. Many parity game solving algorithms can be implemented quite effectively using such data structures [20,28,29], so that in the end, exploring the game graph often remains the bottleneck.

In this paper, we study how to combine the exploration of a parity game and the on-the-fly solving of the explored part, with the aim to speed-up the overall solving process. The central problem when performing on-the-fly solving during the exploration phase is that we have to deal with incomplete information when determining the winner for a designated vertex. Moreover, in the symbolic setting, the exploration order may be unpredictable when advanced strategies such as *chaining* and *saturation* [9] are used.

To formally reason about all possible exploration strategies and the artefacts they generate, we introduce the concept of an *incomplete parity game*, and an ordering on these. Incomplete parity games are parity games where for some vertices not all outgoing edges are necessarily known. In practice, these could be identified by, *e.g.*, the *todo* queue in a classical breadth-first search. The extra information captured by an incomplete parity game allows us to characterise the *safe* set for a given player α. This is a set of vertices for which it can be established that if player α wins the vertex, then she cannot lose the vertex if more information becomes available. We prove an optimality result for safe sets, which, informally, states that a safe set for player α is also the largest set with this property (see Theorem 1).

The vertices won by player α in an α-safe set can be determined using a standard parity game solving algorithm such as, *e.g.*, Zielonka's recursive algorithm [31] or Priority Promotion [2]. However, these algorithms may be less efficient as on-the-fly solvers. For this reason, we study three symbolic *partial* solvers: *solitaire winning cycle* detection, *forced winning cycle* detection and *fatal attractors* [17]. In particular cases, first determining the safe set for a player and only subsequently solving the game using one of these partial solvers will incur an additional overhead. As a final result, we therefore prove that all these solvers can be (modified to) run on the incomplete game as a whole, rather than on the safe set of a player (see Propositions 1-3).

As a proof of concept, we have implemented an (open source) symbolic tool for the mCRL2 toolset [6], that explores a parity game specified by a parameterised Boolean equation system and solves these games on-the-fly. We report on the effectiveness of our implementation on typical parity games stemming from, *e.g.*, model checking and equivalence checking problems, showing that it can speed up the process with several orders of magnitude, while adding low overhead if the entire game is needed for solving.

Related Work. Our work is related to existing techniques for solving symbolic parity games such as [20,19], as we extend these existing methods with on-the-fly solving. Naturally, our work is also related to existing work for on-the-fly model checking. This includes work for on-the-fly (explicit) model checking of regular alternation-free modal mu-calculus formulas [23] and work for on-the-fly symbolic model checking of RCTL [1]. Compared to these our method is more general as it can be applied to the full modal mu-calculus (with data), which subsumes RCTL and the alternation-free subset. Optimisations such as the observation that checking LTL formulas of type AG reduces to reachability checks [14] are a special case of our methods and partial solvers. Furthermore, our methods are not restricted to model checking problems only and can be applied to any parity game, including decision problems such as equivalence checking [8]. Furthermore, our method is agnostic to the exploration strategy employed.

Structure of the paper. In Section 2 we recall parity games. In Section 3 we introduce incomplete parity games and show how partial solving can be applied correctly. In Section 4 we present several partial solvers that we employ for on-the-fly solving. Finally, in Section 5 we discuss the implementation of these techniques and apply them to several practical examples. The omitted proofs for the supporting lemmas can be found in [22].

2 Preliminaries

A parity game is an infinite-duration, two-player game that is played on a finite directed graph. The objective of the two players, called *even* (denoted by $\Diamond$) and *odd* (denoted by $\Box$), is to win vertices in the graph.

Definition 1. *A* parity game *is a directed graph* $G = (V, E, p, (V_\Diamond, V_\Box))$*, where*

- V *is a finite set of vertices, partitioned in sets* $V_\Diamond$ *and* $V_\Box$ *of vertices owned by* $\Diamond$ *and* $\Box$*, respectively;*
- $E \subseteq V \times V$ *is the edge relation;*
- $p : V \to \mathbb{N}$ *is a function that assigns a* priority *to each node.*

Henceforth, let $G = (V, E, p, (V_\Diamond, V_\Box))$ be an arbitrary parity game. Throughout this paper, we use α to denote an arbitrary player and $\bar{\alpha}$ denotes the opponent. We write vE to denote the set of successors $\{w \in V \mid (v, w) \in E\}$ of vertex v. The set $\mathsf{sinks}(G)$ is defined as the largest set $U \subseteq V$ satisfying for all $v \in U$ that $vE = \emptyset$; *i.e.*, $\mathsf{sinks}(G)$ is the set of all sinks: vertices without successors. If we are only concerned with the sinks of player α, we write $\mathsf{sinks}_\alpha(G)$; *i.e.*, $\mathsf{sinks}_\alpha(G) = V_\alpha \cap \mathsf{sinks}(G)$. We write $G \cap U$, for $U \subseteq V$, to denote the subgame $(U, (U \times U) \cap E, p\restriction_U, (V_\Diamond \cap U, V_\Box \cap U))$, where $p\restriction_U(v) = p(v)$ for all vertices $v \in U$.

Example 1. Consider the graph depicted in Figure 1, representing a parity game. Diamond-shaped vertices are owned by player $\Diamond$, whereas box-shaped vertices are owned by player $\Box$. The priority of a vertex is written inside the vertex. Vertex u_1 is a sink owned by player $\Box$. $\square$

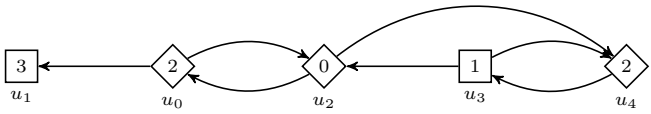


Fig. 1. An example parity game

Plays and strategies. The game is played as follows. Initially, a token is placed on a vertex of the graph. The owner of a vertex on which the token resides gets to decide the successor vertex (if any) that the token is moved to next. A maximal sequence of vertices (*i.e.*, an infinite sequence or a finite sequence ending in a sink) visited by the token by following this simple rule is called a *play*. A finite play π is won by player $\Diamond$ if the sink in which it ends is owned by player $\Box$, and it is won by player $\Box$ if the sink is owned by player $\Diamond$. An infinite play π is won by player $\Diamond$ if the minimal priority that occurs infinitely often along π is even, and it is won by player $\Box$ otherwise.

A strategy $\sigma_\alpha : V^* V_\alpha \to V$ for player α is a partial function that prescribes where player α moves the token next, given a sequence of vertices visited by the token. A play $v_0\, v_1\, \ldots$ is *consistent* with a strategy σ if and only if $\sigma(v_0 \ldots v_i) = v_{i+1}$ for all i for which $\sigma(v_0 \ldots v_i)$ is defined. Strategy σ_α is winning for player α in vertex v if all plays consistent with σ_α and starting in v are won by α. Player α wins vertex v if and only if she has a winning strategy σ_α for vertex v. The *parity game solving problem* asks to compute the set of vertices $W_\Diamond$, won by player $\Diamond$ and the set $W_\Box$, won by player $\Box$. Note that since parity games are *determined* [31,24], every vertex is won by one of the two players. That is, the sets $W_\Diamond$ and $W_\Box$ partition the set V.

Example 2. Consider the parity game depicted in Figure 1. In this game, the strategy $\sigma_\Diamond$, partially defined as $\sigma_\Diamond(\pi u_0) = u_2$ and $\sigma_\Diamond(\pi u_2) = u_0$, for arbitrary π, is winning for player $\Diamond$ in u_0 and u_2. Player $\Box$ wins vertex u_3 using strategy $\sigma_\Box(\pi u_3) = u_4$, for arbitrary π. Note that player $\Diamond$ is always forced to move the token from u_4 to u_3. Vertex u_1 is a sink, owned by player $\Box$, and hence, won by player $\Diamond$. □

Dominions. A strategy σ_α is said to be *closed* on a set of vertices $U \subseteq V$ iff every play, consistent with σ_α and starting in a vertex $v \in U$ remains in U. If player α has a strategy that is closed on U, we say that the set U is α-closed. A *dominion* for player α is a set of vertices $U \subseteq V$ such that player α has a strategy σ_α that is closed on U and which is winning for α. Note that the sets $W_\Diamond$ and $W_\Box$ are dominions for player $\Diamond$ and player $\Box$, respectively, and, hence, every vertex won by player α must belong to an α-dominion.

Example 3. Reconsider the parity game of Figure 1. Observe that player $\Box$ has a closed strategy on $\{u_3, u_4\}$, which is also winning for player $\Box$. Hence, the set $\{u_3, u_4\}$ is an $\Box$-dominion. Furthermore, the set $\{u_2, u_3, u_4\}$ is $\Diamond$-closed. However, none of the strategies for which $\{u_2, u_3, u_4\}$ is closed for player $\Diamond$ is winning for her; therefore $\{u_2, u_3, u_4\}$ is not an $\Diamond$-dominion. □

Predecessors, control predecessors and attractors. Let $U \subseteq V$ be a set of vertices. We write $\mathsf{pre}(G, U)$ to denote the set of predecessors $\{v \in V \mid \exists u \in U : u \in vE\}$ of U in G. The control predecessor set of U for player α in G, denoted $\mathsf{cpre}_\alpha(G, U)$, contains those vertices for which α is able to *force* entering U in one step. It is defined as follows:

$$\mathsf{cpre}_\alpha(G, U) = (V_\alpha \cap \mathsf{pre}(G, U)) \cup (V_{\bar{\alpha}} \setminus (\mathsf{pre}(G, V \setminus U) \cup \mathsf{sinks}(G)))$$

Note that both pre and cpre are monotone operators on the complete lattice $(2^V, \subseteq)$. The α-attractor to U in G, denoted $\mathsf{Attr}_\alpha(G, U)$, is the set of vertices from which player α can force play to reach a vertex in U:

$$\mathsf{Attr}_\alpha(G, U) = \mu Z.(U \cup \mathsf{cpre}_\alpha(G, Z))$$

The α-attractor to U can be computed by means of a fixed point iteration, starting at U and adding α-control predecessors in each iteration until a stable set is reached. We note that the α-attractor to an α-dominion D is again an α-dominion.

Example 4. Consider the parity game G of Figure 1 once again. The $\Diamond$-control predecessors of $\{u_2\}$ is the set $\{u_0\}$. Note that since player $\Box$ can avoid moving to u_2 from vertex u_3 by moving to vertex u_4, vertex u_3 is not among the $\Diamond$-control predecessors of $\{u_2\}$. The $\Diamond$-attractor to $\{u_2\}$ is the set $\{u_0, u_2\}$, which is the largest set of vertices for which player $\Diamond$ has a strategy to force play to the set of vertices $\{u_2\}$. □

3 Incomplete Parity Games

In many practical applications that rely on parity game solving, the parity game is gradually constructed by means of an exploration, often starting from an 'initial' vertex. This is, for instance, the case when using parity games in the context of model checking or when deciding behavioural preorders or equivalences. For such applications, it may be profitable to combine exploration and solving, so that the costly exploration can be terminated when the winner of a particular vertex of interest (often the initial vertex) has been determined. The example below, however, illustrates that one cannot naively solve the parity game constructed so far.

Example 5. Consider the parity game G in Figure 2, consisting of all vertices and only the solid edges. This game could, for example, be the result of an exploration starting from u_4. Then $G \cap \{u_0, u_1, u_2, u_3, u_4, u_5\}$ is a subgame for which we can conclude that all vertices form an $\Diamond$-dominion. However, after exploring the dotted edges, player $\Box$ can escape to vertex u_4 from vertex u_5. Consequently, vertices u_4 and u_5 are no longer won by player $\Diamond$ in the extended game. Furthermore, observe that the additional edge from u_3 to u_5 does not affect the previously established fact that player $\Diamond$ wins this vertex. □

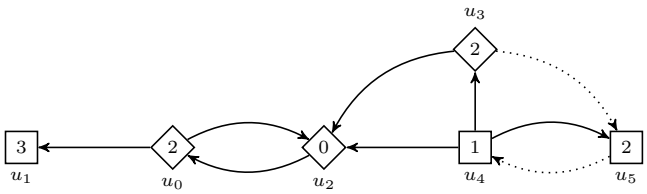


Fig. 2. A parity game where the dotted edges are not yet known.

To facilitate reasoning about games with incomplete information, we first introduce the notion of an *incomplete* parity game.

Definition 2. *An* incomplete *parity game is a structure* $\mathcal{G} = (G, I)$*, where* G *is a parity game* $(V, E, p, (V_\Diamond, V_\Box))$*, and* $I \subseteq V$ *is a set of vertices with potentially unexplored successors. We refer to the set* I *as the set of* incomplete vertices*; the set* $V \setminus I$ *is the set of* complete vertices*.*

Observe that $(G, \emptyset)$ is a 'standard' parity game. We permit ourselves to use the notation for parity game notions such as plays, strategies, dominions, *etcetera* also in the context of incomplete parity games. In particular, for $\mathcal{G} = (G, I)$, we will write $\mathsf{pre}(\mathcal{G}, U)$ and $\mathsf{Attr}_\alpha(\mathcal{G}, U)$ to indicate $\mathsf{pre}(G, U)$ and $\mathsf{Attr}_\alpha(G, U)$, respectively. Furthermore, we define $\mathcal{G} \cap U$ as the structure $(G \cap U, I \cap U)$.

Intuitively, while exploring a parity game, we extend the set of vertices and edges by exploring the incomplete vertices. Doing so gives rise to potentially new incomplete vertices. At each stage in the exploration, the incomplete parity game extends incomplete parity games explored in earlier stages. We formalise the relation between incomplete parity games, abstracting from any particular order in which vertices and edges are explored.

Definition 3. *Let* $\mathcal{G} = ((V, E, p, (V_\Diamond, V_\Box)), I)$, $\mathcal{G}' = ((V', E', p', (V'_\Diamond, V'_\Box)), I')$ *be incomplete parity games. We write* $\mathcal{G} \sqsubseteq \mathcal{G}'$ *iff the following conditions hold:*

(1) $V \subseteq V'$, $V_\Diamond \subseteq V'_\Diamond$ *and* $V_\Box \subseteq V'_\Box$*;*
(2) $E \subseteq E'$ *and* $((V \setminus I) \times V) \cap E' \subseteq E$*;*
(3) $p = p'{\restriction_V}$*;*
(4) $I' \cap V \subseteq I$

Conditions (1) and (3) are self-explanatory. Condition (2) states that on the one hand, no edges are lost, and, on the other hand, E' can only add edges from vertices that are incomplete: for complete vertices, E' specifies no new successors. Finally, condition (4) captures that the set of incomplete vertices I' cannot contain vertices that were previously complete. We note that the ordering $\sqsubseteq$ is reflexive, anti-symmetric and transitive.

Example 6. Suppose that $\mathcal{G} = (G, I)$ is the incomplete parity game depicted in Figure 2, where G is the game with all vertices and only the solid edges, and $I = \{u_3, u_5\}$. Then $\mathcal{G} \sqsubseteq \mathcal{G}'$, where $\mathcal{G}' = (G', I')$ is the incomplete parity game where G' is the depicted game with all vertices and both the solid edges and dotted edges, and $I' = \emptyset$. □

Let us briefly return to Example 5. We concluded that the winner of vertex u_4 (and also u_5) changed when adding new information. The reason is that player $\Box$ has a strategy to reach an *incomplete* vertex owned by her. Such an incomplete vertex may present an opportunity to escape from plays that would be non-winning otherwise. On the other hand, the incomplete vertex u_3 has already been sufficiently explored to allow for concluding that this vertex is won by player $\Diamond$, even if more successors are added to u_3. This suggests that for some subset of vertices, we can decide their winner in an incomplete parity game and preserve that winner in all future extensions of the game. We formally characterise this set of vertices in the definition below.

Definition 4. *Let* $= (G, I)$*, with* $G = (V, E, p, (V_\Diamond, V_\Box))$ *be an incomplete parity game. The* α-safe *vertices for* , *denoted by* $\mathsf{safe}_\alpha(\ )$*, is the set* $V \setminus \mathit{Attr}_{\bar{\alpha}}(G, V_{\bar{\alpha}} \cap I)$.

Example 7. Consider the incomplete parity game of Example 6 once more. We have $\mathsf{safe}_\Diamond(\ ) = \{u_0, u_1, u_2, u_3\}$ and $\mathsf{safe}_\Box(\ ) = \{u_0, u_1, u_2, u_4, u_5\}$. □

In the remainder of this section, we show that it is indeed the case that while exploring a parity game, one can only safely determine the winners in the sets $\mathsf{safe}_\Box(\ )$ and $\mathsf{safe}_\Diamond(\ )$, respectively. More specifically, we claim (Lemma 1) that all α-dominions found in $\mathsf{safe}_\alpha(\ )$ are preserved in extensions of the game, and (Lemma 2) the winner of vertices not in $\mathsf{safe}_\alpha(\ )$ are not necessarily won by the same player in extensions of the game.

Lemma 1. *Given two incomplete games* *and* $'$ *such that* $\sqsubseteq\ '$*. Any* α*-dominion in* $\cap\, \mathsf{safe}_\alpha(\ )$ *is also an* α*-dominion in* $'$.

Example 8. Recall that in Example 7, we found that $\mathsf{safe}_\Diamond(\ ) = \{u_0, u_1, u_2, u_3\}$. Observe that in the incomplete parity game of Example 6, restricted to vertices $\{u_0, u_1, u_2, u_3\}$, all vertices are won by player $\Diamond$, and, hence, $\{u_0, u_1, u_2, u_3\}$ is an $\Diamond$-dominion. Following Lemma 1 we can indeed conclude that this remains an $\Diamond$-dominion in all extensions of , and, in particular, for the (complete) parity game $'$ of Example 6. □

Lemma 2. *Let* *be an incomplete parity game. Suppose that* W *is an* α*-dominion in* *. If* $W \not\subseteq \mathsf{safe}_\alpha(\ )$*, then there is an (incomplete) parity game* $'$ *such that* $\sqsubseteq\ '$ *and all vertices in* $W \setminus \mathsf{safe}_\alpha(\ )$ *are won by* $\bar{\alpha}$.

As a corollary of the above lemma, we find that α-dominions that contain vertices outside of the α-safe set are not guaranteed to be dominions in all extensions of the incomplete parity game.

Corollary 1. *Let* *be an incomplete parity game. Suppose that* W *is an* α*-dominion in* *. If* $W \not\subseteq \mathsf{safe}_\alpha(\ )$*, then there is an (incomplete) parity game* $'$ *such that* $\sqsubseteq\ '$ *and* W *is* not *an* α*-dominion in* $'$.

The theorem below summarises the two previous results, claiming that the sets $\mathsf{safe}_\Diamond(\ )$ and $\mathsf{safe}_\Box(\ )$ are the optimal subsets that can be used safely when combining solving and the exploration of a parity game.

Theorem 1. *Let* $= (G, I)$*, with* $G = (V, E, p, (V_\Diamond, V_\Box))$*, be an incomplete parity game. Define* W_α *as the union of all* α*-dominions in* $\cap \mathsf{safe}_\alpha(\ )$*, and let* $W_? = V \setminus (W_\Diamond \cup W_\Box)$*. Then* $W_?$ *is the largest set of vertices* v *for which there are incomplete parity games* $^\alpha$ *and* $^{\bar\alpha}$ *such that* $\sqsubseteq\ ^\alpha$ *and* $\sqsubseteq\ ^{\bar\alpha}$ *and* v *is won by* α *in* $^\alpha$ *and* v *is won by* $\bar\alpha$ *in* $^{\bar\alpha}$*.*

Proof. Let , with $G = (V, E, p, (V_\Diamond, V_\Box))$ be an incomplete parity game. Pick a vertex $v \in W_?$. Suppose that in G, vertex $v \in W_?$ is won by player α. Let $^\alpha =$. Then $\sqsubseteq\ ^\alpha$ and v is also won by α in $^\alpha$.

Next, we argue that there must be a game $^{\bar\alpha}$ such that $\sqsubseteq\ ^{\bar\alpha}$ and v is won by $\bar\alpha$ in $^{\bar\alpha}$. Since $v \in W_?$ is won by player α in G, v must belong to an α-dominion in G. Towards a contradiction, assume that $v \in \mathsf{safe}_\alpha(\ )$. Then there must also be a α-dominion containing v in $G \cap \mathsf{safe}_\alpha(\ )$, since $\bar\alpha$ cannot escape the set $\mathsf{safe}_\alpha(\ )$. But then $v \in W_\alpha$. Contradiction, so $v \notin \mathsf{safe}_\alpha(\ )$. So, v must be part of an α-dominion D in G such that $D \not\subseteq \mathsf{safe}_\alpha(\ )$. By Lemma 2, we find that there is an incomplete parity game $^{\bar\alpha}$ such that $\sqsubseteq\ ^{\bar\alpha}$ and all vertices in $D \setminus \mathsf{safe}_\alpha(\ )$, and vertex $v \in D$ in particular, are won by $\bar\alpha$ in $^{\bar\alpha}$.

Finally, we argue that $W_?$ cannot be larger. Pick a vertex $v \notin W_?$. Then there must be some player α such that $v \in W_\alpha$, and, consequently, there must be an α-dominion $D \subseteq \cap \mathsf{safe}_\alpha(\ )$ such that $v \in D$. But then by Lemma 1, we find that v is won by α in all incomplete parity games $'$ such that $\sqsubseteq\ '$. □

4 On-the-fly Solving

In the previous section we saw that for any solver solve_α, which accepts a parity game as input and returns an α-dominion W_α, a correct on-the-fly solving algorithm can be obtained by computing $W_\alpha = \mathsf{solve}_\alpha(\ \cap \mathsf{safe}_\alpha(\ ))$ while exploring an (incomplete) parity game . While this approach is clearly sound, computing the set of safe vertices can be expensive for large state spaces and potentially wasteful when no dominions are found afterwards. We next introduce *safe attractors* which, we show, can be used to search for specific dominions without first computing the α-safe set of vertices.

4.1 Safe Attractors

We start by observing that the α-attractor to a set U in an incomplete parity game does not make a distinction between the set of complete and incomplete vertices. Consequently, it may wrongly conclude that α has a strategy to force play to U when the attractor strategy involves incomplete vertices owned by $\bar\alpha$. We thus need to make sure that such vertices are excluded from consideration. This can be achieved by considering the set of *unsafe* vertices $V_{\bar\alpha} \cap I$ as potential vertices that can be used by the other player to escape. We define the safe α-attractor as the least fixed point of the *safe* control predecessor. The latter is defined as follows:

$$\mathsf{spre}_\alpha(\ , U) = (V_\alpha \cap \mathsf{pre}(\ , U)) \cup (V_{\bar\alpha} \setminus (\mathsf{pre}(\ , V \setminus U) \cup \mathsf{sinks}(\ ) \cup I))$$

Lemma 3. *Let be an incomplete parity game. For all vertex sets* $X \subseteq \mathsf{safe}_\alpha(\ )$ *it holds that* $\mathsf{cpre}_\alpha(\ \cap \mathsf{safe}_\alpha(\ ), X) = \mathsf{spre}_\alpha(\ , X)$.

The safe α-attractor to U, denoted $\mathsf{SAttr}_\alpha(\ , U)$, is the set of vertices from which player α can force to *safely* reach U in :

$$\mathsf{SAttr}_\alpha(\ , U) = \mu Z.(U \cup \mathsf{spre}_\alpha(\ , Z))$$

Lemma 4. *Let be an incomplete parity game, and* $X \subseteq \mathsf{safe}_\alpha(\ )$. *Then* $\mathit{Attr}_\alpha(\ \cap \mathsf{safe}_\alpha(\ ), X) = \mathit{SAttr}_\alpha(\ , X)$.

In particular, we can conclude the following:

Corollary 2. *Let be an incomplete parity game, and* $X \subseteq \mathsf{safe}_\alpha(\ )$ *be an* α*-dominion. Then* $\mathit{SAttr}_\alpha(\ , X)$ *is an* α*-dominion for all* $'$ *satisfying* $\sqsubseteq\ '$.

One application of the above corollary is the following: since on-the-fly solving is typically performed repeatedly, previously found dominions can be expanded by computing the safe α-attractor towards these already solved vertices. Another corollary is the following, which states that complete sinks can be safely attracted towards.

Corollary 3. *Let* $= (G, I)$ *be an incomplete parity game and let* $'$ *be such that* $\sqsubseteq\ '$. *Then* $\mathit{SAttr}_\alpha(\ , \mathit{sinks}_{\bar{\alpha}}(\ ) \setminus I)$ *is an* α*-dominion in* $'$.

4.2 Partial Solvers

In practice, a full-fledged solver, such as Zielonka's algorithm [31] or one of the Priority Promotion variants [2], may be costly to run often while exploring a parity game. Instead, cheaper partial solvers may be used that search for a dominion of a particular shape. We study three such partial solvers in this section, with a particular focus on solvers that lend themselves for parity games that are represented symbolically using, *e.g.*, BDDs [5], MDDs [25] or LDDs [13]. For the remainder of this section, we fix an arbitrary incomplete parity game $= ((V, E, p, (V_\Diamond, V_\Box)), I)$.

Winning solitaire cycles. A simple cycle in can be represented by a finite sequence of distinct vertices $v_0\, v_1\, \ldots\, v_n$ satisfying $v_0 \in v_n E$. Such a cycle is an α*-solitaire cycle* whenever all vertices on that cycle are owned by player α.

Observe that if all vertices on an α-solitaire cycle have a priority that is of the same parity as the owner α, then all vertices on that cycle are won by player α. Formally, these are thus cycles through vertices in the set $P_\alpha \cap V_\alpha$, where $P_\Diamond = \{v \in V \setminus \mathsf{sinks}(\ ) \mid p(v) \mod 2 = 0\}$ and $P_\Box = \{v \in V \setminus \mathsf{sinks}(\ ) \mid p(v) \mod 2 = 1\}$. Let $\mathcal{C}^\alpha_{\mathsf{sol}}(\ )$ represent the largest set of α-solitaire winning cycles. Then $\mathcal{C}^\alpha_{\mathsf{sol}}(\ ) = \nu Z.(P_\alpha \cap V_\alpha \cap \mathsf{pre}(\ , Z))$.

Proposition 1. *The set* $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$ *is an* α*-dominion and we have* $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \subseteq \mathit{safe}_{\alpha}(\ )$.

Proof. We first prove that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \subseteq \mathsf{safe}_{\alpha}(\ )$. We show, by means of an induction on the fixed point approximants A_i of the attractor, that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \cap \mathsf{Attr}_{\bar{\alpha}}(\ , V_{\bar{\alpha}} \cap I) = \emptyset$. The base case follows immediately, as $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \cap A_0 = \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \cap \emptyset = \emptyset$. For the induction, we assume that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \cap A_i = \emptyset$; we show that also $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \cap ((V_{\bar{\alpha}} \cap I) \cup \mathsf{cpre}_{\bar{\alpha}}(\ , A_i)) = \emptyset$. First, observe that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \subseteq V_{\alpha}$; hence, it suffices to prove that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \cap (V_{\alpha} \setminus (\mathsf{pre}(\ , V \setminus A_i) \cup \mathsf{sinks}(\ )) = \emptyset$. But this follows immediately from the fact that for every vertex $v \in \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$, we have $v \in P_{\alpha} \cap V_{\alpha} \cap \mathsf{pre}(\ , \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ))$; more specifically, we have $vE \cap \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ ) \neq \emptyset$ for all $v \in \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$.

The fact that $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$ is an α-dominion follows from the fact that for every vertex $v \in \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$, there is some $w \in vE \cap \mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$. This means that player α must have a strategy that is closed on $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$. Since all vertices in $\mathcal{C}^{\alpha}_{\mathsf{sol}}(\ )$ are of the priority that is beneficial to α, this closed strategy is also winning for α. □

Observe that winning solitaire cycles can be computed without first computing the α-safe set. Parity games that stand to profit from detecting winning solitaire cycles are those originating from verifying safety properties.

Winning forced cycles. In general, a cycle in $\mathsf{safe}_{\alpha}(\ )$, through vertices in $P_{\Diamond}$ can contain vertices of both players, providing player □ an opportunity to break the cycle if that is beneficial to her. Nevertheless, if breaking a cycle always inadvertently leads to another cycle through $P_{\Diamond}$, then we may conclude that all vertices on these cycles are won by player ◇. We call these cycles *winning forced cycles* for player ◇. A dual argument applies to cycles through $P_{\Box}$. Let $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$ represent the largest set of vertices that are on winning forced cycles for player α. More formally, we define $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ) = \nu Z.(P_{\alpha} \cap \mathsf{safe}_{\alpha}(\ ) \cap \mathsf{cpre}_{\alpha}(\ , Z))$.

Lemma 5. *The set* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$ *is an* α*-dominion and we have* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ) \subseteq \mathit{safe}_{\alpha}(\ )$.

A possible downside of the above construction is that it again requires to first compute $\mathsf{safe}_{\alpha}(\ )$, which, in particular cases, may incur an additional overhead. Instead, we can compute the same set using the safe control predecessor. We define $\mathcal{C}^{\alpha}_{\mathsf{s-for}}(\ ) = \nu Z.(P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ , Z))$.

Proposition 2. *We have* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ) = \mathcal{C}^{\alpha}_{\mathsf{s-for}}(\ )$.

Proof. Let $\tau(Z) = P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ , Z)$. We use set inclusion to show that $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$ is indeed a fixed point of τ.

- *ad* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ) \subseteq \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. Pick a vertex $v \in \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$. By definition of $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$, we have $v \in P_{\alpha} \cap \mathsf{safe}_{\alpha}(\ ) \cap \mathsf{cpre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. Observe that $\mathsf{safe}_{\alpha}(\ ) \cap \mathsf{cpre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )) = \mathsf{safe}_{\alpha}(\ ) \cap \mathsf{cpre}_{\alpha}(\ \cap \mathsf{safe}_{\alpha}(\ ), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. But then, since $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ) \subseteq \mathsf{safe}_{\alpha}(\ )$, we find, by Lemma 3, that $\mathsf{cpre}_{\alpha}(\ \cap \mathsf{safe}_{\alpha}(\ ), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )) = \mathsf{spre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. Hence, $v \in P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )) = \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$.
- *ad* $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ) \supseteq \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. Again pick a vertex $v \in \tau(\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. Then $v \in P_{\alpha} \cap \mathsf{spre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. Since $\mathcal{C}^{\alpha}_{\mathsf{for}}(\ ) \subseteq \mathsf{safe}_{\alpha}(\ )$, by Lemma 3, we again have $\mathsf{spre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )) = \mathsf{cpre}_{\alpha}(\ \cap \mathsf{safe}_{\alpha}(\ ), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. But then it must be the case that $v \in \mathsf{safe}_{\alpha}(\ )$. Moreover, $\mathsf{cpre}_{\alpha}(\ \cap \mathsf{safe}_{\alpha}(\ ), \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )) \subseteq \mathsf{cpre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ ))$. So $v \in P_{\alpha} \cap \mathsf{safe}_{\alpha}(\ ) \cap \mathsf{cpre}_{\alpha}(\ , \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )) = \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$.

We show next that for any $Z = \tau(Z)$, we have $Z \subseteq \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$. Let Z be such. We first show that for every $v \in Z \cap V_\alpha$, there is some $w \in vE \cap Z$, and for every $v \in Z \cap V_{\bar{\alpha}}$, we have $v \notin \mathsf{sinks}(\ )$, $v \notin I$ and $vE \subseteq Z$. Pick $v \in Z \cap V_\alpha$. Then $v \in \tau(Z) \cap V_\alpha = P_\alpha \cap V_\alpha \cap \mathsf{spre}_\alpha(\ , Z) \subseteq \mathsf{pre}(\ , Z)$. But then $vE \cap Z \neq \emptyset$. Next, let $v \in Z \cap V_{\bar{\alpha}}$. Then $v \in \tau(Z) \cap V_{\bar{\alpha}} = P_\alpha \cap V_{\bar{\alpha}} \cap \mathsf{spre}_\alpha(\ , Z) \subseteq V_{\bar{\alpha}} \setminus (\mathsf{pre}(\ , V \setminus Z) \cup \mathsf{sinks}(\ ) \cup I)$. So $v \notin \mathsf{pre}(\ , V \setminus Z) \cup \mathsf{sinks}(\ ) \cup I$. Consequently, $vE \subseteq Z$, $v \notin \mathsf{sinks}(\ )$ and $v \notin I$.

Since for every $v \in Z \cap V_\alpha$, we have $vE \cap Z \neq \emptyset$, there must be a strategy for player α to move to another vertex in Z. Let σ be this strategy. Moreover, since for all $v \in Z \cap V_{\bar{\alpha}}$ we have $vE \subseteq Z$, we find that σ is closed on Z and since $Z \cap \mathsf{sinks}(\ ) = \emptyset$, strategy σ induces forced cycles. Moreover, since $Z \subseteq P_\alpha$, we can conclude that all vertices in Z are on winning forced cycles.

Finally, we must argue that $Z \subseteq \mathsf{safe}_\alpha(\ )$. But this follows from the fact that $Z \cap V_{\bar{\alpha}} \cap I = \emptyset$, and, hence, also $Z \cap \mathsf{Attr}_{\bar{\alpha}}(\ , V_{\bar{\alpha}} \cap I) = \emptyset$. Since Z is contained within $P_\alpha \cap \mathsf{safe}_\alpha(\ )$, we find that $Z \subseteq \mathcal{C}^{\alpha}_{\mathsf{for}}(\ )$. □

Fatal attractors. Both solitaire cycles and forced cycles utilise the fact that the parity winning condition becomes trivial if the only priorities that occur on a play are of the parity of a single player. Fatal attractors [17] were originally conceived to solve parts of a game using algorithms that have an appealing worst-case running time; for a detailed account, we refer to [17]. While *ibid.* investigates several variants, the main idea behind a fatal attractor is that it identifies cycles in which the priorities are non-decreasing until the dominating priority of the attractor is (re)visited. We focus on a simplified (and cheaper) variant of the `psolB` algorithm of [17], which is based on the concept of a *monotone* attractor, which, in turn, relies on the monotone control predecessor defined below, where $P^{\geq c} = \{v \in V \mid p(v) \geq c\}$:

$$\mathsf{Mcpre}_\alpha(\ , Z, U, c) = P^{\geq c} \cap \mathsf{cpre}_\alpha(\ , Z \cup U)$$

The monotone attractor for a given priority is then defined as the least fixed point of the monotone control predecessor for that priority, formally $\mathsf{MAttr}_\alpha(\ , U, c) = \mu Z.\mathsf{Mcpre}_\alpha(\ , Z, U, c)$. A *fatal* attractor for priority c is then the largest set of vertices closed under the monotone attractor for priority c; *i.e.*, $\mathcal{F}^\alpha(\ , c) = \nu Z.(P^{=c} \cap \mathsf{safe}_\alpha(\ ) \cap \mathsf{MAttr}_\alpha(\ \cap \mathsf{safe}_\alpha(\ ), Z, c))$, where $P^{=c} = P^{\geq c} \setminus P^{\geq c+1}$.

Lemma 6 (See [17], Theorem 2). *For even c, we have that $\mathit{MAttr}_\Diamond(\ \cap \mathit{safe}_\alpha(\ ), \mathcal{F}^\Diamond(\ , c), c) \subseteq \mathit{safe}_\Diamond(\ )$ and $\mathit{MAttr}_\Diamond(\ \cap \mathit{safe}_\alpha(\ ), \mathcal{F}^\Diamond(\ , c), c)$ is an $\Diamond$-dominion. If c is odd then we have $\mathit{MAttr}_\Box(\ \cap \mathit{safe}_\alpha(\ ), \mathcal{F}^\Box(\ , c), c) \subseteq \mathit{safe}_\Box(\ )$ and $\mathit{MAttr}_\Box(\ \cap \mathit{safe}_\alpha(\ ), \mathcal{F}^\Box(\ , c), c)$ is an $\Box$-dominion.*

Our simplified version of the `psolB` algorithm, here dubbed $\texttt{solB}^-$ computes fatal attractors for all priorities in descending order, accumulating $\Diamond$ and $\Box$-dominions and extending these dominions using a standard $\Diamond$ or $\Box$-attractor. This can be implemented using a simple loop over these priorities.

In line with the previous solvers, we can also modify this solver to employ a safe monotone control predecessor, which uses a construction that is similar

in spirit to that of the safe control predecessor. Formally, we define the safe monotone control predecessor as follows:

$$\mathsf{sMcpre}_\alpha(\ ,Z,U,c) = P^{\geq c} \cap \mathsf{spre}_\alpha(\ ,Z \cup U)$$

The corresponding safe monotone α-attractor, denoted $\mathsf{sMAttr}_\alpha(\ ,U,c)$, is defined as follows: $\mathsf{sMAttr}_\alpha(\ ,U,c) = \mu Z.\mathsf{sMcpre}_\alpha(\ ,Z,U,c)$. We define the *safe* fatal attractor for priority c as the set $\mathcal{F}^\alpha_{\mathsf{s}}(\ ,c) = \nu Z.(P^{=c} \cap \mathsf{sMAttr}_\alpha(\ ,Z,c))$.

Proposition 3. *Let be an incomplete parity game. We have* $\mathcal{F}^\Diamond_{\mathsf{s}}(\ ,c) = \mathcal{F}^\Diamond(\ ,c)$ *for even* c *and for odd* c *we have* $\mathcal{F}^\Box_{\mathsf{s}}(\ ,c) = \mathcal{F}^\Box(\ ,c)$.

Similar to algorithm $\mathtt{solB}^-$, the algorithm $\mathtt{solB}^-_s$ computes safe fatal attractors for priorities in descending order and collects the safe-α-attractor extended dominions obtained this way.

5 Experimental Results

We experimentally evaluate the techniques of Section 4. For this, we use games stemming from practical model checking and equivalence checking problems. Our experiments are run, single-threaded, on an Intel Xeon 6136 CPU @ 3 GHz PC. The sources for these experiments can be obtained from the downloadable artefact [21].

5.1 Implementation

We have implemented a symbolic exploration technique for parity games in the mCRL2 toolset [6]. Our tool exploits techniques such as *read* and *write* dependencies [20,4], and uses sophisticated exploration strategies such as *chaining* and *saturation* [9]. We use MDD-like data structures [25] called *List Decision Diagrams (LDDs)*, and the corresponding Sylvan implementation [13], to represent parity games symbolically. Sylvan also offers efficient implementations for set operations and relational operations, such as predecessors, facilitating the implementation of attractor computations, the described (partial) solvers, and a full solver based on Zielonka's recursive algorithm [31], which remains one of the most competitive algorithms in practice, both explicitly and symbolically [28,12]. For the attractor set computation we have also implemented chaining to determine (multi-)step α-predecessors more efficiently.

For all three on-the-fly solving techniques of Section 4, we have implemented 1) a variant that runs the standard (partial) solver on the α-safe subgame and removes the found dominion using the standard attractor (within that subgame), and 2) a variant that uses (partial) solvers with the safe attractors. Moreover, we also conduct experiments using the full solver running on an α-safe subgame. An important design aspect is to decide how the exploration and the on-the-fly solving should interleave. For this we have implemented a time based heuristic that keeps track of the time spent on solving and exploration steps. The time

measurements are used to ensure that (approximately) ten percent of total time is spent on solving by delaying the next call to the solver. We do not terminate the partial solver when it requires more time, and thus it is only approximate. As a result of this heuristic, cheap solvers will be called more frequently than more expensive (and more powerful) ones, which may cause the latter to explore larger parts of the game graph.

5.2 Cases

Table 1 provides an overview of the models and a description of the property that is being checked. The properties are written in the modal μ-calculus with data [15]. For the equivalence checking case we have mutated the original model to introduce a defect. For each property, we indicate the *nesting depth* (ND) and *alternation depth* [10] and whether the parity game is *solitaire* (Yes/No). The nesting depth indicates how many different priorities occur in the resulting game; for our encoding this is at most ND+2 (the additional ones encode constants '*true*' and '*false*'). The alternation depth is an indication of a game's complexity due to alternating priorities.

Table 1. Models and formulas.

Model	Ref.	Prop.	Result	ND	AD	Sol.	Description
SWP	[30]	1	false	1	1	Y	No error transition
		2	false	3	3	N	Infinitely often enabled then infinitely often taken
WMS	[27]	1	false	1	1	Y	Job failed to be done
		2	false	1	1	Y	No zombie jobs
		3	true	3	2	Y	A job can become alive again infinitely often
		4	false	2	2	N	Branching bisimulation with a mutation
BKE	[3]	1	true	1	1	Y	No secret leaked
		2	false	2	1	N	No deadlock
CCP	[26]	1	false	2	1	N	No deadlock
		2	false	2	1	N	After access there is always accessover possible
PDI	n/a	1	true	2	1	N	Controller reaches state before it can connect again
		2	false	2	1	N	Connection impermissible can always happen or we establish a connection
		3	false	3	1	N	When connected move to not ready for connection and do not establish a connection until it is allowed again
		4	true	2	1	N	The interlocking moves to the state connection closed before it is allowed to succesfully establish a connection

We use MODEL-i to indicate the parity game belonging to model MODEL and property i. Models SWP, BKE and CCP are protocol specifications. The model PDI is a specification of a EULYNX SCI-LX SysML interface model that is used for a train interlocking system. Finally, WMS is the specification of a workload management system used at CERN. Using tools in mCRL2 [6], we have converted each model and property combination into a so-called parameterised Boolean equation systems [16], a higher-level logic that can be used to represent the underlying parity game.

Parity games SWP-1, WMS-1, WMS-2 and BKE-1 encode typical safety properties where some action should not be possible. In terms of the alternation-free modal mu-calculus with regular expressions, such properties are of the shape

$[\mathsf{true}^*.\mathsf{a}]\mathsf{false}$. These properties are violated exactly when the vertex encoding '*false*' can be reached. Parity games SWP-2, WMS-3 and WMS-4 are more complex properties with alternating priorities, where WMS-4 encodes branching bisimulation using the theory presented in [8]. The parity games BKE-2 and CCP-1 encode a 'no deadlock' property given by a formula which states that after every path there is at least one outgoing transition. Finally, CCP-2 and all PDI cases contain formulas with multiple fixed points that yield games with multiple priorities but no (dependent) alternation.

Table 2. Experiments with parity games where on-the-fly solving cannot terminate early. All run times are in seconds. The number of vertices is given in millions. Memory is given in gigabytes. Bold-faced numbers indicate the lowest value.

Game	Strategy	Vertices (10^6)	Explore (s)	Solve (s)	Total (s)	Mem (GB)
BKE-1	full	40	640	65	**705**	14
	solitaire	40/40	629/615	153/100	782/715	15/15
	cycles	40/40	635/644	149/160	785/804	15/15
	fatal	40/40	624/625	152/164	776/789	15/15
	partial	40	651	147	798	15
PDI-1	full	114	27	0.1	**28**	2
	solitaire	114/114	28/27	4/0	33/**28**	2/2
	cycles	114/114	29/28	7/7	36/35	2/2
	fatal	114/114	28/28	4/7	32/35	2/2
	partial	114	28	9	37	2
PDI-4	full	474	286	0	**287**	2
	solitaire	474/474	284/281	46/14	331/295	2/2
	cycles	474/474	284/287	92/91	376/378	2/2
	fatal	474/474	285/283	80/91	365/374	2/2
	partial	474	286	64	350	2

5.3 Results

In Tables 2 and 3 we compare the on-the-fly solving strategies presented in Section 4. In the 'Strategy' column we indicate the on-the-fly solving strategy that is used. Here *full* refers to a complete exploration followed by solving with the Zielonka recursive algorithm. We use *solitaire* to refer to solitaire winning cycle detection, *cycles* for forced winning cycle detection, *fatal* to refer to fatal attractors and finally *partial* for on-the-fly solving with a Zielonka solver on safe regions. For solvers with a standard variant and a variant that utilises the safe attractors the first number indicates the result of applying the (standard) solver on *safe* vertices, and the second number (following the slash '/') indicates the result when using the solver that utilises safe attractors.

The column 'Vertices' indicates the number of vertices explored in the game. In the next columns we indicate the time spent on exploring and solving specifically and the total time in seconds. We exclude the initialisation time that is common to all experiments. Finally, the last column indicates memory used by the tool in gigabytes. We report the average of 5 runs and have set a timeout (indicated by ‡) at 1200 seconds per run. Table 2 contains all benchmarks that require a full exploration of the game graph, providing an indication of the over-

Table 3. Experiments with parity games in which *at least one* partial solver terminates early. All run times are in seconds. The number of vertices is given in millions. For solvers with two variants the first number indicates the result of applying the solver on *safe* vertices, and following the slash '/' the result when using the solver that uses safe attractors. Memory is given in gigabytes. Bold-faced numbers indicate the lowest value.

Game	Strategy	Vertices (10^6)	Explore (s)	Solve (s)	Total (s)	Mem (GB)
SWP-1	full	13304	‡	n/a	‡	‡
	solitaire	15.1/0.4	8.5/1.4	27.3/0.1	35.8/**1.5**	2.8/1.5
	cycles	25.2/0.9	12.3/1.8	42.7/1.0	55.0/2.8	3.2/1.5
	fatal	15.1/0.4	9.0/1.3	29.4/0.4	38.4/1.7	3.1/1.5
	partial	27.1	13.1	50.4	63.5	3.6
SWP-2	full	1987	‡	n/a	‡	‡
	solitaire	1631/1987	‡/‡	163/11	‡/‡	‡/‡
	cycles	1774/1774	‡/‡	154/91	‡/‡	‡/‡
	fatal	0.007/0.007	0.9/0.9	0.4/0.2	1.3/**1.0**	1.4/1.2
	partial	0.007	0.9	0.4	1.3	1.4
WMS-1	full	270	2.8	0.4	3.3	0.2
	solitaire	270/240	2.8/2.5	0.8/0.4	3.6/**2.9**	0.3/0.2
	cycles	270/270	2.9/3.2	0.8/8.0	3.7/11.2	0.3/0.5
	fatal	270/270	2.6/3.2	0.8/8.5	3.4/11.7	0.3/0.5
	partial	270	2.7	0.8	3.5	0.3
WMS-2	full	317	3.3	0.3	3.6	0.2
	solitaire	7/7	0.2/0.2	1.0/0.5	1.2/**0.8**	0.1/0.1
	cycles	7/66	0.2/0.8	1.0/2.7	1.2/3.4	0.1/0.2
	fatal	7/66	0.2/0.7	1.0/2.9	1.3/3.6	0.1/0.2
	partial	7	0.2	1.1	1.3	0.1
WMS-3	full	317	2.6	0.1	2.7	0.2
	solitaire	317/317	2.6/2.6	0.4/0.3	3.1/2.9	0.2/0.2
	cycles	317/317	2.7/2.7	0.4/0.6	3.1/3.3	0.2/0.2
	fatal	5/1	0.2/0.1	0.5/0.1	0.7/**0.2**	0.1/0.1
	partial	5	0.2	0.3	0.5	0.1
WMS-4	full	366	‡	n/a	‡	‡
	solitaire	0.03/0.03	38/38	0.8/0.1	39/38	2/2
	cycles	0.03/0.03	37/37	0.8/0.3	38/**37**	2/2
	fatal	0.03/0.03	37/37	0.8/0.3	38/**37**	2/2
	partial	0.03	37	0.7	38	2
BKE-2	full	119	942	36.5	979	28
	solitaire	0.0007/0.0001	0.2/0.1	0.0/0.0	**0.2**/**0.2**	0.9/0.9
	cycles	0.0007/0.0003	0.2/0.2	0.0/0.0	**0.2**/**0.2**	0.9/0.9
	fatal	0.0007/0.0003	0.2/0.2	0.0/0.0	**0.2**/**0.2**	0.9/0.9
	partial	0.0007	0.2	0.0	**0.2**	0.9
CCP-1	full	0.4	28	4.2	32	2
	solitaire	0.003/0.003	1.0/1.0	0.1/0.1	**1.1**/**1.1**	2/2
	cycles	0.003/0.003	1.0/1.0	0.1/0.1	**1.1**/**1.1**	2/2
	fatal	0.006/0.003	1.3/1.1	0.1/0.1	1.4/1.2	1.5/1.5
	partial	0.003	1.0	0.1	**1.1**	1.5
CCP-2	full	0.9	35	33	68	1.7
	solitaire	0.02/0.007	1.6/1.1	0.2/0.0	1.8/**1.1**	1.5/1.5
	cycles	0.02/0.007	1.9/1.1	0.2/0.1	2.1/1.2	1.5/1.5
	fatal	0.02/0.007	1.6/1.2	0.2/0.1	1.8/1.3	1.5/1.5
	partial	0.02	1.6	0.2	1.8	1.5
PDI-2	full	229	31	12	43	2
	solitaire	229/229	33/32	34/12	67/45	2/2
	cycles	30/30	15/14	3/5	**17**/19	2/2
	fatal	30/30	15/15	3/5	18/19	2/2
	partial	123	23	29	51	2
PDI-3	full	436	228	8	236	2
	solitaire	436/436	230/228	36/32	266/260	2/2
	cycles	78/162	65/102	19/64	84/166	2/2
	fatal	75/84	64/67	19/23	**83**/90	2/2
	partial	110	82	30	112	2

head in cases where this is unavoidable; Table 3 contains all benchmarks where *at least one* of the partial solvers allows exploration to terminate early.

For games SWP-1, WMS-1, WMS-2 in Table 3 we find that *solitaire*, and in particular the safe attractor variant, is able to determine the solution the fastest. Also, for all entries in Table 2 this is the solver with the least overhead. Next, we observe that for cases such as WMS-1 and PDI-3 using the safe attractor variants of the solvers can be detrimental. Our observation is that first computing safe sets (especially using chaining) can be quick when most vertices are owned by one player and one priority and the computation of the safe attractor, which uses the more difficult safe control predecessor is more involved in such cases. There are also cases WMS-3, WMS-4, CCP-1 and CCP-2 where the safe attractor variants are faster and these cases all have multiple priorities. In cases where these solvers are slow (for example PDI-3) we also observe that more states are explored before termination, because the earlier mentioned time based heuristic results in calling the solver significantly less frequently.

For parity games SWP-2 and WMS-3 only *fatal* and *partial* are able to find a solution early, which shows that more powerful partial solvers can be useful. From Table 2 and the cases in which the safe attractor variants perform poorly we learn that the partial solvers can, as expected, cause overhead. This overhead is in our benchmarks on average 30 percent, but when it terminates early it can be very beneficial, achieving speed-ups of up to several orders of magnitude.

6 Conclusion

In this work we have developed the theory to reason about on-the-fly solving of parity games, independent of the strategy that is used to explore games. We have introduced the notion of *safe* vertices, shown their correctness, proven an optimality result, and we have studied partial solvers and shown that these can be made to run without determining the safe vertices first; which can be useful for on-the-fly solving. Finally, we have demonstrated the practical purpose of our method and observed that solitaire winning cycle detection with safe attractors is almost always beneficial with minimal overhead, but also that more powerful partial solvers can be useful.

Based on our experiments, one can make an educated guess which partial solver to select in particular cases; we believe that this selection could even be steered by analysing the parameterised Boolean equation system representing the parity game. It would furthermore be interesting to study (practical) improvements for the safe attractors, and their use in Zielonka's recursive algorithm.

Acknowledgements We would like to thank Jeroen Meijer and Tom van Dijk for their help regarding the Sylvan library when implementing our prototype. This work was supported by the TOP Grants research programme with project number 612.001.751 (AVVA), which is (partly) financed by the Dutch Research Council (NWO).

References

1. Beer, I., Ben-David, S., Landver, A.: On-the-fly model checking of RCTL formulas. In: Hu, A., Vardi, M. (eds.) CAV. LNCS, vol. 1427, pp. 184–194. Springer (1998). https://doi.org/10.1007/BFb0028744
2. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving parity games via priority promotion. Formal Methods Syst. Des. **52**(2), 193–226 (2018). https://doi.org/10.1007/s10703-018-0315-1
3. Blom, S., Groote, J.F., Mauw, S., Serebrenik, A.: Analysing the BKE-security protocol with μCRL. Electron. Notes Theor. Comput. Sci. **139**(1), 49–90 (2005). https://doi.org/10.1016/j.entcs.2005.09.005
4. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P.B. (eds.) CAV. LNCS, vol. 6174, pp. 354–359. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_31
5. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
6. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In: Vojnar, T., Zhang, L. (eds.) TACAS. LNCS, vol. 11428, pp. 21–39. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_2
7. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: Hatami, H., McKenzie, P., King, V. (eds.) STOC. pp. 252–263. ACM (2017). https://doi.org/10.1145/3055399.3055409
8. Chen, T., Ploeger, B., van de Pol, J., Willemse, T.A.C.: Equivalence checking for infinite systems using parameterized Boolean equation systems. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR. LNCS, vol. 4703, pp. 120–135. Springer (2007). https://doi.org/10.1007/978-3-540-74407-8_9
9. Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. Int. J. Softw. Tools Technol. Transf. **8**(1), 4–25 (2006). https://doi.org/10.1007/s10009-005-0188-7
10. Cleaveland, R., Klein, M., Steffen, B.: Faster model checking for the modal mu-calculus. In: von Bochmann, G., Probst, D.K. (eds.) CAV. LNCS, vol. 663, pp. 410–422. Springer (1992). https://doi.org/10.1007/3-540-56496-9_32
11. Cranen, S., Luttik, B., Willemse, T.A.C.: Proof graphs for parameterised Boolean equation systems. In: D'Argenio, P.R., Melgratti, H.C. (eds.) CONCUR. LNCS, vol. 8052, pp. 470–484. Springer (2013). https://doi.org/10.1007/978-3-642-40184-8_33
12. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: Beyer, D., Huisman, M. (eds.) TACAS. LNCS, vol. 10805, pp. 291–308. Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_16
13. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int. J. Softw. Tools Technol. Transf. **19**(6), 675–696 (2017). https://doi.org/10.1007/s10009-016-0433-2
14. Eiríksson, Á.T., McMillan, K.L.: Using formal verification/analysis methods on the critical path in system design: A case study. In: Wolper, P. (ed.) CAV. LNCS, vol. 939, pp. 367–380. Springer (1995). https://doi.org/10.1007/3-540-60045-0_63
15. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. Sci. Comput. Program. **56**(3), 251–273 (2005). https://doi.org/10.1016/j.scico.2004.08.002

16. Groote, J.F., Willemse, T.A.C.: Parameterised Boolean equation systems. Theor. Comput. Sci. **343**(3), 332–369 (2005). https://doi.org/10.1016/j.tcs.2005.06.016
17. Huth, M., Kuo, J.H., Piterman, N.: Fatal attractors in parity games. In: Pfenning, F. (ed.) FOSSACS. LNCS, vol. 7794, pp. 34–49. Springer (2013). https://doi.org/10.1007/978-3-642-37075-5_3
18. Jurdziński, M., Lazić, R.: Succinct progress measures for solving parity games. In: LICS. pp. 1–9. IEEE Computer Society (2017). https://doi.org/10.1109/LICS.2017.8005092
19. Kant, G., van de Pol, J.: Efficient instantiation of parameterised Boolean equation systems to parity games. In: Wijs, A., Bosnacki, D., Edelkamp, S. (eds.) GRAPHITE. EPTCS, vol. 99, pp. 50–65 (2012). https://doi.org/10.4204/EPTCS.99.7
20. Kant, G., van de Pol, J.: Generating and solving symbolic parity games. In: Bosnacki, D., Edelkamp, S., Lluch-Lafuente, A., Wijs, A. (eds.) GRAPHITE. EPTCS, vol. 159, pp. 2–14 (2014). https://doi.org/10.4204/EPTCS.159.2
21. Laveaux, M.: Downloadable sources for the case study (2022). https://doi.org/10.5281/zenodo.5896966
22. Laveaux, M., Wesselink, W., Willemse, T.A.C.: On-the-fly solving for symbolic parity games. CoRR **abs/2201.09607** (2022), https://arxiv.org/abs/2201.09607
23. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Sci. Comput. Program. **46**(3), 255–281 (2003). https://doi.org/10.1016/S0167-6423(02)00094-1
24. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic **65**(2), 149–184 (1993). https://doi.org/10.1016/0168-0072(93)90036-D
25. Miller, D.M.: Multiple-valued logic design tools. In: ISMVL. pp. 2–11. IEEE Computer Society (1993). https://doi.org/10.1109/ISMVL.1993.289589
26. Pang, J., Fokkink, W.J., Hofman, R.F.H., Veldema, R.: Model checking a cache coherence protocol of a java DSM implementation. J. Log. Algebraic Methods Program. **71**(1), 1–43 (2007). https://doi.org/10.1016/j.jlap.2006.08.007
27. Remenska, D., Willemse, T.A.C., Verstoep, K., Templon, J., Bal, H.E.: Using model checking to analyze the system behavior of the LHC production grid. Future Gener. Comput. Syst. **29**(8), 2239–2251 (2013). https://doi.org/10.1016/j.future.2013.06.004
28. Sanchez, L., Wesselink, W., Willemse, T.A.C.: A comparison of BDD-based parity game solvers. In: Orlandini, A., Zimmermann, M. (eds.) GandALF. EPTCS, vol. 277, pp. 103–117 (2018). https://doi.org/10.4204/EPTCS.277.8
29. Stasio, A.D., Murano, A., Vardi, M.Y.: Solving parity games: Explicit vs symbolic. In: Câmpeanu, C. (ed.) CIAA. LNCS, vol. 10977, pp. 159–172. Springer (2018). https://doi.org/10.1007/978-3-319-94812-6_14
30. Tanenbaum, A.S., Wetherall, D.: Computer networks, 5th Edition. Pearson (2011), https://www.worldcat.org/oclc/698581231
31. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998). https://doi.org/10.1016/S0304-3975(98)00009-7

Equivalence Checking

Distributed Coalgebraic Partition Refinement

Fabian Birkmann, Hans-Peter Deifel✉,*, and Stefan Milius**

Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
{fabian.birkmann,hans-peter.deifel,stefan.milius}@fau.de

Abstract. Partition refinement is a method for minimizing automata and transition systems of various types. Recently we have developed a partition refinement algorithm and the tool CoPaR that is generic in the transition type of the input system and matches the theoretical run time of the best known algorithms for many concrete system types. Genericity is achieved by modelling transition types as functors on sets and systems as coalgebras. Experimentation has shown that memory consumption is a bottleneck for handling systems with a large state space, while running times are fast. We have therefore extended an algorithm due to Blom and Orzan, which is suitable for a distributed implementation to the coalgebraic level of genericity, and implemented it in CoPaR. Experiments show that this allows to handle much larger state spaces. Running times are low in most experiments, but there is a significant penalty for some.

1 Introduction

Minimization is an important and basic algorithmic task on state-based systems, concerned with reducing the state space as much as possible while retaining the system's behaviour. It is used for equivalence checking of systems and as a subtask in model checking tools in order to handle larger state spaces and thus mitigate the state-explosion problem.

We focus on the task of identifying behaviourally equivalent states modulo bisimilarity. For classic labelled transitions systems this notion obeys the principle 'states s and t are bisimilar if for every transition $s \xrightarrow{a} s'$, there exists a transition $t \xrightarrow{a} t'$ with s' and t' bisimilar', and symmetrically for transitions from t. Bisimilarity is a rather fine-grained branching-time notion of equivalence (cf. [17]); it is widely used and preserves all properties expressible as μ-calculus formulas. Moreover, it has been generalized to yield equivalence notions for many other types of state-based systems and automata.

Due to the above principle, bisimilarity is defined by a fixed point, to be understood as a greatest fixed point and is hence approximable from above. This is used by *partition refinement* algorithms: The initial partition considers all states tentatively equivalent is then iteratively refined using observations

* Supported by the Deutsche Forschungsgemeinschaft (DFG) within the Research and Training Group 2475 "Cybercrime and Forensic Computing" (393541319/GRK2475/1-2019)

** Supported by Deutsche Forschungsgemeinschaft (DFG) under project MI 717/7-1.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 159–177, 2022.
https://doi.org/10.1007/978-3-030-99527-0_9

about the states until a fixed point is reached. Consequently, such procedures run in polynomial time and can also be efficiently implemented, in contrast to coarser system equivalences such as trace equivalence and language equivalence of nondeterministic systems which are PSPACE-complete [23]. This makes minimization under bisimilarity interesting even in cases where the main equivalence is linear-time, such as for automata.

Efficient partition refinement algorithms exist for various systems: Kanellakis and Smolka provide a minimization algorithm with run time $\mathcal{O}(m \cdot n)$ for labelled transition systems with n states and m transitions. Even faster algorithms have been developed over the past 50 years for many types of systems. For example, Hopcroft's algorithm for minimizing deterministic automata has run time in $\mathcal{O}(n \cdot \log n)$ [21]; it was later generalized to variable input alphabets, with run time $\mathcal{O}(n \cdot |A| \cdot \log n)$ [18,24]. The Paige-Tarjan algorithm minimizes transition systems in time $\mathcal{O}((m+n) \cdot \log n)$ [31], and generalizations to labelled transition systems have the same time complexity [13, 22, 36]. For the minimization of weighted systems (a.k.a. *lumping*), Valmari and Franchescini [38] have developed a simple $\mathcal{O}((m+n) \cdot \log n)$ algorithm for systems with rational weights. Buchholz [10] gave an algorithm for weighted automata, and Högberg et al. [20] one for (bottom-up) weighted trees automata, both with run time in $\mathcal{O}(m \cdot n)$.

In previous work [16, 42], we have provided an efficient partition refinement algorithm, which is generic in the system type, captures all the above system types, and matches or, in some cases even improves on the run time complexity of the respective specialized algorithms. Subsequently, we have shown how to extend the generic complexity analysis to weighted tree automata and implemented the algorithm in the tool CoPaR [11, 41], again matching the previous best run time complexity and improving it in the case of weighted tree automata with weights from a non-cancellative monoid. The algorithm is based on ideas of Paige and Tarjan, which leads to its efficiency. Genericity is achieved by modelling state based systems as coalgebras, following the paradigm of universal coalgebra [34], in which the transitions structure of systems is encapsulated by a set functor. The algorithm and tool are *modular* in the sense that functors can be built from a preimplemented set of basic functors by standard set constructions such as cartesian product, disjoint union and functor composition. The tool then automatically derives a parser for input coalgebras of the composed type and provides a corresponding partition refinement implementation off the shelf. In addition, new basic functors F may easily be added to the set of basic functors by implementing a simple refinement interface for them plus a parser for encoded F-coalgebras. Our experiments with the tool have shown that run time scales well with the size of systems. However, memory usage becomes a bottleneck with growing system size, a problem that has previously also been observed by Valmari [37] for partition refinement. One strategy to address this is to distribute the algorithm across multiple computers, which store and process only a part of the state space and communicate via message passing. For ordinary labelled transition systems and Markov systems this has been investigated in a series of papers by Blom and Orzan [4–9] who were also motivated to mitigate the memory bottleneck of sequential partition refinement algorithms.

Our contribution in this paper is an extension of CoPaR by an efficient distributed partition algorithm in coalgebraic generality. Like in Blom and Orzan's work, our algorithm is a distributed version of a simple but effective algorithm called "the naive method" [23], or "the final chain algorithm" in coalgebraic generality [25,42]. We first generalize signature refinement introduced by Blom and Orzan to the level of coalgebras. We also combine generalized signatures (Section 3) with the previous encodings of set functors and their coalgebras [11,41] via the new notion of a signature interface (Definition 3.1). This is a key idea to make coalgebraic signature refinement and the final chain algorithm implementable in a tool like CoPaR. In addition, we demonstrate how signature interfaces of functors can be combined (Construction 3.3 and Proposition 3.4) along standard functor constructions. This yields a similar modularity principle than for the previous sequential algorithm. However, this is a new feature for signature refinement and also, to our knowledge, for the final chain algorithm. Consequently, our distributed, modular and generic implementation of the final chain algorithm is new (already as sequential algorithm).

We also provide experiments demonstrating its scalability and show that much larger state spaces can indeed be handled. Our benchmarks include weighted tree automata for non-cancellative monoids, a type of system for which our previous sequential implementation is heavily limited by its memory requirements. For those systems the running times of the distributed algorithm are even faster then those of the sequential algorithm. In a second set of benchmarks stemming from the PRISM benchmark suite [27] we again show that larger systems can now be handled; however, for some of these there is a penalty in run time.

Related work. Balcazar et al. [1] have proved that the problem of bisimilarity checking for labelled transition systems is P-complete, which implies that it is hard to parallelize efficiently. Nevertheless, parallel algorithms have been proposed by Rajasekaran and Lee [33]. These are designed for shared memory machines and hence do not distribute RAM requirements over multiple machines.

Symbolic techniques are an orthogonal approach to reduce memory usage of partition refinement algorithms and have been explored e.g. by Wimmer et al. [40] and van Dijk and de Pol [15].

Two other orthogonal extensions of the generic coalgebraic minimization and CoPaR have been presented in recent work. First a non-trivial extension computes (1) reachable states and (2) the transition structure of the minimized systems [12]. Second, Wißmann et al. [43] have shown how to compute distinguishing formulas in a Hennessy-Milner style logic for a pair of behaviourally inequivalent states.

2 Preliminaries

Our algorithmic framework and the tool CoPaR [41,42] are based on modelling state-based systems abstractly as *coalgebras* for a (set) *functor* that encapsulates the transition type, following the paradigm of *universal coalgebra* [34]. We now recall some standard notations for sets and maps and basic notions and examples in coalgebra. We fix a singleton set $1 = \{*\}$; for every set X we have a unique map $!\colon X \to 1$ and the identity map $\mathsf{id}_X\colon X \to X$. We denote composition of

maps by $(-)\cdot(-)$, in applicative order. Given maps $f\colon X \to A$, $g\colon X \to B$ we define $\langle f, g\rangle\colon X \to A \times B$ by $\langle f, g\rangle(x) = (f(x), g(x))$. The type of transitions of states in a system is modelled by a set functor F. Informally, F assigns to every set X a set FX of structured collections of elements of X, and an F-coalgebra is a map $c\colon S \to FS$ which assigns to every state $s \in S$ in a system a structured collection $c(s) \in FS$ of successor states of s. The functor F also determines a canonical notion of behavioural equivalence of states of a coalgebra; this arises by stipulating that morphisms of coalgebras are behaviour preserving maps.

Definition 2.1. A *functor* $F\colon \mathsf{Set} \to \mathsf{Set}$ assigns to each set X a set FX and to each map $f\colon X \to Y$ a map $Ff\colon FX \to FY$, preserving identities and composition ($F\mathsf{id}_X = \mathsf{id}_{FX}$, $F(g\cdot f) = Fg\cdot Ff$). An *F-coalgebra* (S, c) consists of a set S of *states* and a *transition structure* $c\colon S \to FS$. A *morphism* $h\colon (S, c) \to (S', c')$ of F-coalgebras is a map $h\colon S \to S'$ that preserves the transition structure, i.e. $Fh\cdot c = c'\cdot h$. Two states $s, t \in S$ of a coalgebra $c\colon S \to FS$ are *behaviourally equivalent* ($s \sim t$) if there exists a coalgebra morphism h with $h(s) = h(t)$.

Example 2.2. We mention several types of systems which are instances of the general notion of coalgebra and the ensuing notion of behavioural equivalence. All these are possible input systems for our tool CoPaR.
(1) Transition systems. The *finite powerset* functor $\mathcal{P}_\omega$ maps a set X to the set $\mathcal{P}_\omega X$ of all *finite* subsets of X, and a map $f\colon X \to Y$ to the map $\mathcal{P}_\omega f = f[-]\colon \mathcal{P}_\omega X \to \mathcal{P}_\omega Y$ taking direct images. Coalgebras for $\mathcal{P}_\omega$ are finitely branching (unlabelled) transition systems. Two states are behaviourally equivalent iff they are (strongly) bisimilar in the sense of Milner [29,30] and Park [32]. Similarly, finitely branching labelled transition systems with label alphabet A are coalgebras for the functor $FX = \mathcal{P}_\omega(A \times X)$.
(2) Deterministic automata. For an input alphabet A, the functor given by $FX = 2 \times X^A$, where $2 = \{0, 1\}$, sends a set X to the set of pairs of boolean values and functions $A \to X$. An F-coalgebra (S, c) is a deterministic automaton (without an initial state). For each state $s \in S$, the first component of $c(s)$ determines whether s is a final state, and the second component is the successor function $A \to S$ mapping each input letter $a \in A$ to the successor state of s under input letter a. States $s, t \in S$ are behaviourally equivalent iff they accept the same language in the usual sense.
(3) Weighted tree automata simultaneously generalize tree automata and weighted (word) automata. Inputs of such automata stem from a finite *signature* Σ, i.e. a finite set of input symbols, each with a prescribed natural number, its *arity*. *Weights* are taken from a commutative monoid $(M, +, 0)$. A (bottom-up) *weighted tree automaton* (WTA) (over M with inputs from Σ) consists of a finite set S of states, an output map $f\colon S \to M$, and for each $k \geq 0$, a transition map $\mu_k\colon \Sigma_k \to M^{S^k \times S}$, where Σ_k denotes the set of k-ary input symbols in Σ; the maximum arity of symbols in Σ is called the *rank*.

Every signature Σ gives rise to its associated *polynomial functor*, also denoted Σ, which assigns to a set X the set $\coprod_{n\in\mathbb{N}} \Sigma_n \times X^n$, where $\coprod$ denotes disjoint union (coproduct). Further, for a given monoid $(M, +, 0)$ the *monoid-valued functor* $M^{(-)}$ sends a set X to the set of maps $f\colon X \to M$ that are finitely supported,

i.e. $f(x) = 0$ for almost all $x \in X$. Given a map $f\colon X \to Y$, $M^{(f)}\colon M^{(X)} \to M^{(Y)}$ sends a map $v\colon X \to M$ in $M^{(X)}$ to the map $y \mapsto \sum_{x \in X, f(x)=y} v(x)$, corresponding to the standard image measure construction.

Weighted tree automata are coalgebras for the composite functor $FX = M \times M^{(\Sigma X)}$; indeed, given a coalgebra $c = \langle c_1, c_2 \rangle\colon S \to M \times M^{(\Sigma S)}$, its first component c_1 is the output map, and the second component c_2 is equivalent to the family of transitions maps μ_k described above.

As proven by Wißmann et al. [41, Prop. 6.6], the coalgebraic behavioural equivalence is precisely backward bisimulation of weighted tree automata as introduced by Högberg et al. [20, Def. 16].

(4) The *bag functor* $\mathcal{B}\colon \mathsf{Set} \to \mathsf{Set}$ sends a set X to the set of all finite multisets (or *bags*) over X. This is the special case of the monoid-valued functor for the monoid $(\mathbb{N}, +, 0)$. Accordingly, $\mathcal{B}$-coalgebras are weighted transition systems with positive integers as weights, or they may be regarded as finitely branching transition systems where multiple transitions between a pair of states are allowed. Behavioural equivalence coincides with weighted (or strong) bisimilarity.

(5) Markov chains. The *finite distribution functor* $\mathcal{D}_\omega$ is a subfunctor of the monoid-valued functor $\mathbb{R}^{(-)}$ for the usual monoid of addition on the real numbers. It maps a set X to the set of all finite probability distributions on X. That means that $\mathcal{D}_\omega X$ is the set of all finitely supported maps $d\colon X \to [0, 1]$ such that $\sum_{x \in X} d(x) = 1$. The action of $\mathcal{D}_\omega$ on maps is the same as that of $\mathbb{R}^{(-)}$.

As shown by Rutten and de Vink [35], coalgebras $c\colon S \to (\mathcal{D}_\omega S + 1)^A$ are precisely Larsen and Skou's probabilistic transition systems [28] (aka. labelled Markov chains [14]) with the label alphabet A. In fact, for each state $s \in S$ and action label $a \in A$, that state either cannot perform an a-action (when $c(s)(a) \in 1$) or the distribution $c(s)(a)$ determines for every state $t \in C$ the probability with which s transitions to t with an a-action.

Coalgebraic behavioural equivalence is precisely probabilistic bisimilarity in the sense of Larsen and Skou, see Rutten and de Vink [35, Cor. 4.7].

(6) Markov decision processes are systems which feature both non-deterministic and probabilistic branching. They are coalgebras for composite functors such as $\mathcal{P}_\omega(A \times \mathcal{D}_\omega(-))$ or $\mathcal{P}_\omega(\mathcal{D}_\omega(A \times (-))$ (simple/general Segala systems); Bartels et al. [2] list further functors for various species of probabilistic systems.

Encodings. To supply coalgebras as inputs to CoPaR and in order to speak about the size of a coalgebra in terms of states and transitions, we need

Definition 2.3 [12, Def. 3.1]. An *encoding* of a set functor F consists of a set A of *labels* and a family of maps $\flat_X\colon FX \to \mathcal{B}(A \times X)$, one for every set X, such that the map $\langle F!, \flat_X \rangle\colon FX \to F1 \times \mathcal{B}(A \times X)$ is injective.

The *encoding* of a coalgebra $c\colon S \to FS$ is $\langle F!, \flat_S \rangle \cdot c\colon S \to F1 \times \mathcal{B}(A \times S)$. For $s \in S$ we write $s \xrightarrow{a} t$ whenever (a, t) is contained in the bag $\flat_S(c(s))$. The *number of states* and *edges* of a given encoded input coalgebra are $n = |S|$ and $m = \sum_{s \in S} |\flat_S(c(s))|$, respectively, where $|b| = \sum_{x \in X} b(x)$ for a bag $b\colon X \to \mathbb{N}$.

An encoding of a set functor F specifies how F-coalgebras are represented as directed graphs, and the required injectivity ensures that different coalgebras have different encodings.

Example 2.4. We recall a few key examples of encodings used by CoPaR [42]; for the required injectivity, see [12, Prop. 3.3].
(1) For the finite powerset functor $\mathcal{P}_\omega$ one takes a singleton label set $A = 1$ and $\flat_X \colon \mathcal{P}_\omega X \to \mathcal{B}(1 \times X)$ is the obvious inclusion: $\flat_X(U)(*, x) = 1$ iff $x \in U \subseteq X$.
(2) For the monoid-valued functor $M^{(-)}$ we take labels $A = M$, and the map $\flat_X \colon M^{(X)} \to \mathcal{B}(M \times X)$ is given by $\flat_X(t)(m, x) = 1$ if $t(x) = m \neq 0$ and 0 else.
(3) As a special case, the bag functor $\mathcal{B}$ has labels $A = \mathbb{N}$, and the map $\flat_X \colon \mathcal{B}X \to \mathcal{B}(\mathbb{N} \times X)$ is given by $\flat_X(t)(n, x) = 1$ if $t(x) = n$ and 0 else.

Remark 2.5. (1) Readers familiar with category theory may wonder about the *naturality* of encodings $\flat_X$. It turns out [12] that in almost all instances, our encodings are not natural transformations, except for polynomial functors. As shown in *op. cit.*, all our encodings satisfy a property called *uniformity*, which implies that they are subnatural transformations [12, Prop. 3.15].
(2) Having an encoding of a set functor F does not imply a reduction of the problem of minimizing F-coalgebras to that of coalgebras for $\mathcal{B}(A \times -)$. In fact, the behavioural equivalence of F-coalgebras and coalgebras for $\mathcal{B}(A \times -)$ may be very different unless $\flat_X$ is natural, which is not the case for most encodings.

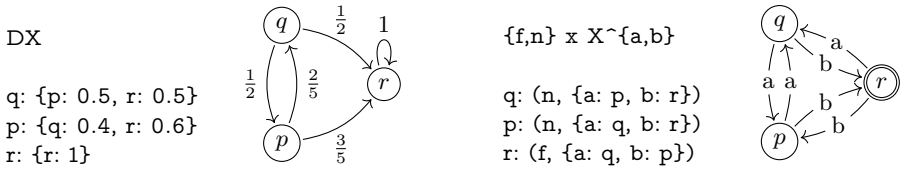
Functors in CoPaR can be combined by product, coproduct or composition, leading to modularity. But in order to automatically handle combined functors, our tool crucially depends on the ability to form products and coproducts of encodings [41, 42]. We refrain from going into technical details, but note for further use that given a pair of functors F_1, F_2 with encodings $A_i, \flat_{X,i}$ one obtains encodings for the functors $F_1 \times F_2$ (cartesian product) and $F_1 + F_2$ (disjoint union) with the label set $A = A_1 + A_2$.

Input syntax and processing. We briefly recall the input format of CoPaR and how inputs are processed; for more details see [41, Sec. 3.1]. CoPaR accepts input files representing a finite F-coalgebra. The first line of an input file specifies the functor F which is written as a term according to the following grammar:

$$\begin{aligned} &T ::= \mathtt{X} \mid \mathcal{P}_\omega\, T \mid \mathcal{B}\, T \mid \mathcal{D}_\omega\, T \mid M^{(T)} \mid \Sigma \\ &\Sigma ::= C \mid T + T \mid T \times T \mid T^A \qquad C ::= \mathbb{N} \mid A \qquad A ::= \{s_1, \ldots, s_n\} \mid n, \end{aligned} \tag{1}$$

where $n \in \mathbb{N}$ denotes the set $\{0, \ldots, n-1\}$, the s_k are strings subject to the usual conventions for variable names (a letter or an underscore character followed by alphanumeric characters or underscore), exponents F^A are written `F^A`, and M is one of the monoids $(\mathbb{Z}, +, 0)$, $(\mathbb{R}, +, 0)$, $(\mathbb{C}, +, 0)$, $(\mathcal{P}_\omega(64), \cup, \emptyset)$ (the monoid of 64-bit words with bitwise or), and $(\mathbb{N}, \max, 0)$ (the additive monoid of the tropical semiring). Note that C effectively ranges over at most countable sets, and A over finite sets. A term T determines a functor $F \colon \mathsf{Set} \to \mathsf{Set}$ in the evident way, with X interpreted as the argument.

The remaining lines of an input file specify a finite coalgebra $c \colon S \to FS$. Each line has the form $s{:}\,t$ for a state $s \in S$, and t represents the element $c(s) \in FS$. The syntax for t depends on the specified functor F and follows the structure of



(a) Markov chain (b) Deterministic finite automaton

Fig. 1: Examples of input files with encoded coalgebras [41]

the term T defining F; the details are explained in [41, Sec. 3.1.2]. Fig. 1 from *op. cit.* shows two coalgebras and the corresponding input files.

After reading the functor term T, CoPaR builds a parser for the functor-specific input format and then parses the input coalgebra given in that format into an intermediate format which internally represents the encoding of the input coalgebra (Definition 2.3). For composite functors the parsed coalgebra then undergoes a substantial amount of preprocessing, which also affects how transitions are counted; see [41, Sec. 3.5] for more details.

3 Coalgebraic Partition Refinement

As mentioned in the introduction, the sequential partition refinement algorithm previously implemented in CoPaR is based on ideas used in the Paige-Tarjan algorithm [31] for transition systems. However, as has been mentioned by Blom and Orzan [8], the Paige-Tarjan algorithm carefully selects the block of states to split in each iteration, and the data structures used for this selection take a lot of memory and require modification to allow a distributed implementation. Hence, Blom and Orzan have built their distributed algorithm from a rather simple sequential partition refinement algorithm based on what Kanellakis and Smolka refer to as the *naive method* [23]. We now recall this algorithm and subsequently show how it can be adapted to the coalgebraic level of generality.

Signature Refinement. Given a finite labelled transition system with the state set S, a partition on S may be presented by a function $\pi\colon S \to \mathbb{N}$, i.e. two states $s, t \in S$ lie in the same block of the partition iff $\pi(s) = \pi(t)$. The *signature* of a state $s \in S$ is the set of outgoing transitions to blocks of π:

$$\mathsf{sig}_\pi(s) = \{(a, \pi(t)) \mid s \xrightarrow{a} t\} \subseteq \mathcal{P}_\omega(A \times \mathbb{N}). \tag{2}$$

A *signature refinement step* then refines π by putting $s, t \in S$ into different blocks iff $\mathsf{sig}_\pi(s) \neq \mathsf{sig}_\pi(t)$. Concretely, we put $\pi_{\mathrm{new}}(s) = \mathsf{hash}(\mathsf{sig}_\pi(s))$ using a perfect, deterministic hash function hash. The signature refinement algorithm (Fig. 2) starts with a trivial initial partition on S and repeats the refinement step until the partition stabilizes, i.e. until two subsequent partitions have the same size.

Coalgebraic Signature Refinement. Regarding a labelled transition system as a coalgebra $c\colon S \to \mathcal{P}_\omega(A \times S)$ (Example 2.2(1)), signatures are obtained by postcomposing the transition structure with the partition under the functor:

$$\mathsf{sig}_\pi = S \xrightarrow{c} \mathcal{P}_\omega(A \times S) \xrightarrow{\mathcal{P}_\omega(A \times \pi)} \mathcal{P}_\omega(A \times \mathbb{N}). \tag{3}$$

Variables : old and new partitions represented by $\pi, \pi_{\text{new}} \colon S \to \mathbb{N}$ with sizes l, l_{new}, resp.; set H for counting block numbers;

foreach $s \in S$ **do**
  $\pi_{\text{new}}(s) \leftarrow 0$;
end
$l_{new} \leftarrow 1$;
while $l \neq l_{new}$ **do**
  $\pi \leftarrow \pi_{\text{new}}, H \leftarrow \emptyset$;
  foreach $s \in S$ **do**
    $\pi_{\text{new}}(s) \leftarrow \mathsf{hash}(\mathsf{sig}_\pi(s))$;
    $H \leftarrow H \cup \{\pi_{\text{new}}(s)\}$;
  end
  $l \leftarrow l_{\text{new}}$;
  $l_{\text{new}} \leftarrow |H|$;
end

Fig. 2: Signature refinement for labelled transition systems

The generalisation to coalgebras for arbitrary F is immediate: the *signature of a state* of an F-coalgebra $c \colon S \to FS$ w.r.t. a partition π is given by the function $\mathsf{sig}_\pi = F\pi \cdot c$. In the refinement step of the above algorithm two states are identified by the next partition if they have the same signatures currently:

$$\pi_{\text{new}}(s) = \pi_{\text{new}}(t) \iff \mathsf{sig}_\pi(s) = \mathsf{sig}_\pi(t) \iff (F\pi)(c(s)) = (F\pi)(c(t)). \quad (4)$$

Hence, the algorithm in fact simply applies $F(-) \cdot c$ to the initial partition corresponding to the trivial quotient $! \colon S \to 1$ until stability is reached. Note that this is precisely the *Final Chain Algorithm* by König and Küpper [25, Alg. 3.2] computing behavioural equivalence of a given F-coalgebra. Its correctness thus proves correctness of the *coalgebraic signature refinement* which is the algorithm in Fig. 2 with $\mathsf{sig}_\pi = F\pi \cdot c$. Since we represent functors and their coalgebras by encodings we use an interface to F to compute signatures based on encodings.

Definition 3.1. Given a functor F with encoding $A, \flat_X$, a *signature interface* consists of a function $\mathsf{sig} \colon F1 \times \mathcal{B}(A \times \mathbb{N}) \to F\mathbb{N}$ such that for every finite set S and every partition $\pi \colon S \to \mathbb{N}$ we have

$$F\pi = \big(FS \xrightarrow{\langle F!, \flat_S \rangle} F1 \times \mathcal{B}(A \times S) \xrightarrow{F1 \times \mathcal{B}(A \times \pi)} F1 \times \mathcal{B}(A \times \mathbb{N}) \xrightarrow{\mathsf{sig}} F\mathbb{N}\big). \quad (5)$$

Given a coalgebra $c \colon S \to FS$, a state $s \in S$ and a partition $\pi \colon S \to \mathbb{N}$, the two arguments of sig should be understood as follows. The first argument is the value $F!(c(s)) \in F1$, which intuitively provides an observable output of the state s. The second argument is the bag $\mathcal{B}(A \times \pi)(\flat_S(c(s)))$ formed by those pairs (a, n) of labels a and numbers n of blocks of the partition π to which s has an edge; that is, that bag contains one pair (a, n) for each edge $s \xrightarrow{a} s'$ where $\pi(s') = n$. Thus, when supplied with these inputs, sig correctly computes the signature of s; indeed, to see this, precompose equation (5) with the coalgebra structure c.

Example 3.2. (1) The constant functor $!C$ has the label set $A = \emptyset$, so we have $\mathcal{B}(\emptyset \times \mathbb{N}) \cong 1$, and we define the function $\mathsf{sig} \colon C \times \mathcal{B}(\emptyset \times \mathbb{N}) \to C$ by $\mathsf{sig}(c, *) = c$.

(2) The powerset functor $\mathcal{P}_\omega$ has the label set $A = 1$, and we define the function $\mathsf{sig}\colon \mathcal{P}_\omega 1 \times \mathcal{B}(1 \times \mathbb{N}) \to \mathcal{P}_\omega \mathbb{N}$ by $\mathsf{sig}(z, b) = \{n : b(*, n) \neq 0\}$.
(3) The monoid-valued functor $\mathbb{R}^{(-)}$ has the label set $A = \mathbb{R}$, and we define the function $\mathsf{sig}\colon \mathbb{R} \times \mathcal{B}(\mathbb{R} \times \mathbb{N}) \to \mathbb{R}^{(\mathbb{N})}$ by $\mathsf{sig}(z, b)(n) = \Sigma\{r \mid b(r, n) \neq 0\}$.

Next we show how signature interfaces can be combined by products ($\times$) and coproducts ($+$). This is the key to the modularity of the implementation (be it distributed or sequential) of the coalgebraic signature refinement in CoPaR.

Construction 3.3. Given a pair of functors F_1, F_2 with encodings $A_i, \flat_{X,i}$ and signature interfaces sig_i, we put $A = A_1 + A_2$ and define the following functions:
(1) for the product functor $F = F_1 \times F_2$ we take $\mathsf{sig}\colon F1 \times \mathcal{B}(A \times \mathbb{N}) \to F_1\mathbb{N} \times F_2\mathbb{N}$,

$$\mathsf{sig}(t, b) = \big(\mathsf{sig}_1(\mathsf{pr}_1(t), \mathsf{filter}_1(b)), \mathsf{sig}_2(\mathsf{pr}_2(t), \mathsf{filter}_2(b))\big).$$

Here, $\mathsf{pr}_i\colon F1 \to F_i 1$ is the projection map and $\mathsf{filter}_i\colon \mathcal{B}(A \times \mathbb{N}) \to \mathcal{B}(A_i \times \mathbb{N})$ is given by $\mathsf{filter}_i(b)(a, n) = b(\mathsf{in}_i\, a, n)$, where $\mathsf{in}_i\colon F_i\mathbb{N} \to F\mathbb{N}$ is the injection map.
(2) for the coproduct functor $F = F_1 + F_2$ we take

$$\mathsf{sig}\colon F1 \times \mathcal{B}(A \times \mathbb{N}) \to F_1\mathbb{N} + F_2\mathbb{N}, \qquad \mathsf{sig}(\mathsf{in}_i\, t, b) = \mathsf{in}_i(\mathsf{sig}_i(t, \mathsf{filter}_i(b))).$$

Proposition 3.4. *The functions* sig *defined in Construction 3.3 yield signature interfaces for the functors $F_1 \times F_2$ and $F_1 + F_2$, respectively.*

As a consequence of this result, it suffices to implement signature interfaces only for *basic* functors according to the grammar in (1), i.e. the trivial identity and constant functors as well as the functors $\mathcal{P}_\omega$, $\mathcal{B}$, $\mathcal{D}_\omega$ and the supported monoid-valued functors $M^{(-)}$. Signature interfaces of products, coproducts and exponents, being a special form of product, are derived using Construction 3.3.

Functor composition can be reduced to these constructions by a technique called *desorting* [42, Sec. 8.2], which transforms a coalgebra of a composite functor into a coalgebra for a coproduct of basic functors whose signature interfaces can then be combined by $+$ (see also [41, Sec. 3.5]). As for the previous Paige-Tarjan style algorithm, this leads to the modularity in the functor of the coalgebraic signature refinement algorithm: signature interfaces for composed functors are automatically derived in CoPaR. Moreover, a new basic functor F may be added by implementing a signature interface for F, effectively extending the grammar of supported functors in (1) by a clause FT.

4 The Distributed Algorithm

Our distributed algorithm for coalgebraic signature refinement is a generalization of Blom and Orzan's original algorithm [8] to coalgebras. We highlight differences to *op. cit.* at the end of this section.

We assume a distributed high-bandwidth cluster of W workers $w_1, \ldots, w_W$ that is failure-free, i.e. nodes do not crash, messages do not get lost and between two nodes the order of messages is preserved. The communication is based on non-blocking *send* operations and blocking *receive* operations. Messages are triples of the form $(\mathit{from}, \mathit{to}, \mathit{data})$, where the *data* field may be structured and will often contain a tag to simplify interpretation.

Description. The distributed algorithm is based on the sequential algorithm presented in Fig. 2, using a distributed hashtable to keep track of the partition. As for the sequential algorithm, the input consists of an F-coalgebra (S, c) with $|S| = n$ states. We split the state space evenly among the workers as a preprocessing step. We write S_i with $|S_i| = n/W$ for the set of states of worker w_i. The input for worker w_i is the encoding of that part of the transition structure of the input coalgebra which is needed to compute the signatures of the states in S_i. This information is presented to w_i as the list of all outgoing edges of states of S_i in the encoding of the coalgebra (S, c), i.e. the list of all $s \xrightarrow{a} t$ with $s \in S_i$ (cf. Definition 2.3). We refer to the block number $\pi(s)$ of a state $s \in S$ as its ID.

After processing the input, the algorithm runs in two phases. In the *Initialization Phase* (Fig. 3) the workers exchange update demands about the IDs stored in the distributed hashtable. If w_i has an edge $s \xrightarrow{a} s'$ into some state s' of w_j, then during refinement w_i needs to be kept up to date about the ID of s' and thus instructs w_j to do so. Worker w_j remembers this information by storing w_i in the set $\mathsf{In}_{s'} = \{w_i \mid \exists s \in S_i, a \in A.\, s \xrightarrow{a} s'\}$ of incoming edges of s' (lines 14–16). Hence, for each edge $s \xrightarrow{a} s'$ with $s \in S_i$ and $s' \in S_j$, worker w_i sends a message to w_j, informing w_j to add w_i to $\mathsf{In}_{s'}$ (lines 5–8).

Variables : Set V of visited states; process count d;
for each $s \in S_i$ a list In_s of workers with an edge into s

```
1  V ← ∅, d ← 0;
2  foreach s ∈ S_i do
3  |  In_s ← [];
4  end
5  foreach edge s → s' of w_i with
     s' ∉ V do
6  |  V ← V ∪ {s'};
7  |  send(w_i, w_j, s');
8  end
9  foreach 1 ≤ j ≤ W do
10 |  send(w_i, w_j, DONE);
11 end
12 waitFor(d = W);
13 return([In_s | s ∈ S_i]);

14 on receive (w_k, w_i, s) do
15 |  In_s ← (w_k :: In_s);
16 end

17 on receive (_, _, DONE) do
18 |  d ← d + 1;
19 end
```

Fig. 3: Initialization Phase of worker w_i

The main phase is the *Refinement Phase* (Fig. 4), mimicking the refinement loop of the undistributed algorithm. In each iteration all workers compute their part of the new partition, i.e. the IDs $h_s = \mathsf{hash}(\mathsf{sig}_\pi(s))$ for each of their states $s \in S_i$ (line 5). In addition, every worker w_i is responsible for sending the computed ID of $s \in S_i$ to workers in In_s that need it for computation of their own signatures in the next iteration (lines 6–9). The IDs are also sent to a designated worker $\mathsf{counterOf}(h_s)$ (lines 10–12). This ensures that IDs are counted precisely once at the end of the round when the partition size is computed after all messages have been received (lines 14–17). The actual counting (line 19) is a

Variables : Old, respectively new partitions π, π_{new} with sizes l, l_{new}; finished workers d; ID-counting set H;

```
1  π_new ← 0!, l ← −1, l_new ← 0, H ← ∅;
2  while l ≠ l_new do
3  |  l ← l_new, π ← π_new;
4  |  foreach s ∈ S_i do
5  |  |  π_new(s) ← hash(sig_π(s));
6  |  |  foreach w_j ∈ In_s do
7  |  |  |  send(w_i, w_j,
8  |  |  |       ⟨UPD, s, π_new(s)⟩);
9  |  |  end
10 |  |  send(w_i,
11 |  |       counterOf(π_new(s)),
12 |  |       ⟨COUNT, π_new(s)⟩);
13 |  end
14 |  foreach 1 ≤ j ≤ W do
15 |  |  send(w_i, w_j, DONE);
16 |  end
17 |  waitFor(d = W);
18 |  l ← l_new;
19 |  l_new ← distribSum(sizeOf(H));
20 |  synchronize;
21 end

22 on receive
       (w_k, w_i, (UPD, s, h_s)) do
23 |  π_new(s) ← h_s;
24 end

25 on receive
       (w_k, w_i, (COUNT, h_s)) do
26 |  H ← H ∪ {h_s};
27 end

28 on receive (_, w_i, DONE) do
29 |  d ← d + 1;
30 end
```

Fig. 4: Refinement Phase of worker w_i

primitive operation in the MPI library, for an explicit $\mathcal{O}(\log W)$ algorithm using messages see e.g. Blom and Orzan [8, Fig. 6]. Finally, the workers synchronize before starting the next iteration (line 20). The refinement phase stops if two consecutive partitions have the same size (line 2).

Correctness. The Initialization Phase (Fig. 3) terminates since every worker reaches line 10, sends DONE to all workers and thus also receives it (lines 17–19) a total of W times, allowing it to progress past line 12. An analogous argument proves termination of every iteration of the Refinement Phase (Fig. 4). The sequential algorithm is correct, hence we know the loop of the refinement phase terminates when all IDs are computed and counted correctly, since then the distributed and the sequential algorithm compute precisely the same partitions.

To show that the signatures are computed correctly, we note that if all DONE messages have been received in a round, then, by order-preservation of messages, all messages sent previously in this round have also been received. This ensures that no workers are missing from the lists In_s computed in the Initialization Phase and that during the Refinement Phase new IDs are sent to all concerned workers (Fig. 4, lines 6–8). This establishes correctness of the signature computation, and the signatures coincide on all workers since we assume that the hash function is deterministic. Finally, the use of the counterOf function (line 11) ensures that each ID is included in the counting set of exactly one worker. Thus, the distributed sum of the sizes of all counting sets is equal to the size of the partition.

Complexity. Let us assume that not only states, but also outgoing transitions are distributed evenly among the workers, i.e. every worker has about m/W outgoing transitions. In the Initialization Phase, the loop sending messages runs in $\mathcal{O}(\frac{m}{W})$ and receiving takes $\mathcal{O}(W \cdot \frac{n}{W}) = \mathcal{O}(n)$, since for worker w_i every other worker w_j might have an edge into every state in S_i. Both are executed in parallel so in total the phase runs in $\mathcal{O}(\max(\frac{m}{W}, n)) = \mathcal{O}(\frac{m}{W} + n)$. In the Refinement Phase, we assume the run time of computing signatures and their hashes is linear in the number of edges. Then the loop for computing and hashing ($\mathcal{O}(\frac{m}{W})$) and counting ($\mathcal{O}(\frac{n}{W})$) signatures runs in total in $\mathcal{O}(\frac{m+n}{W})$, since it is performed by all workers independently. Each worker receives at most m/W ID-updates each round and the partition size is computable in $\mathcal{O}(W)$ giving the complexity of one refinement step in $\mathcal{O}(\frac{m+n}{W})$. As many as n iterations might be needed for a total complexity of $\mathcal{O}(\frac{m}{W} + n) + n \cdot \mathcal{O}(\frac{n+m}{W}) = \mathcal{O}\big(\frac{mn+n^2}{W} + n\big)$.

Remark 4.1. The above analysis assumes that signature interfaces are implemented with a linear run time in their input bag. This could in fact be theoretically realized for all basic functors (whence also for their combinations) currently implemented in CoPaR, which would involve using bucket sort for the grouping of bag elements by the target block (second component), e.g. for monoid-valued functors. However, since the table used in bucket sort would be very large (the size of the last partition) and memory conscience is our main motivation, we opted for an implementation using a standard $n \log n$ sorting algorithm instead.

Implementation details. CoPaR is implemented in Haskell. We were able to reuse, with only minor adjustments, major parts of the code base of CoPaR dedicated to the representation and processing of coalgebras. This includes the implemented functors and their encodings together with the corresponding parser and preprocessing algorithms (see Section 2). As explained in Section 3 the sequential Paige-Tarjan-style algorithm of CoPaR was not used; we implemented an additional "algorithmic frontend" to our "coalgebraic backend". To compute signatures during the Refinement Phase, each functor implements the signature interface (Definition 3.1), which is written in Haskell as follows:

```
class Hashable (Signature f) => SignatureInterface f where
  type Signature f :: Type
  sig  :: F1 f -> [(Label f, Int)] -> Signature f
```

We require in the second line a type `Signature f`, that serves as an implementation-specific datatype representation of $F\mathbb{N}$. In the type of `sig`, the types `f`, `Label f` and `F1 f` correspond to the name of F, its label type and the set $F1$, respectively.

Example 4.2. The Haskell-implementation of the signature interface for the finite power set functor $\mathcal{P}_\omega$ from Example 3.2(2) is as follows:

```
data P x = P x  -- already defined in CoPaR
type instance Label P = () -- also already defined
instance SignatureInterface P where
  type Signature P = Set Int
```

```
sig :: F1 f -> [((), Int)] -> Set Int
sig _ = setFromList . map snd
```

Signature interfaces for the other basic functors according to the grammar in (1) are implemented similarly. For combined functors CoPaR automatically derives their signature interface based on Construction 3.3.

In the algorithm itself, each worker runs three threads in parallel: The first thread is for computing, the second one is for sending and the third one is for receiving signatures. This allows us to keep calls to the MPI interface separated from (pure) signature computation, simplifying logic and allowing the workers to scatter the ID of one state while simultaneously computing the signature of the next one to ensure that neither signature computation nor network traffic become bottlenecks. For inter-thread communication and synchronization we rely on Haskell's *software transactional memory* [19] to ease concurrent programming, e.g. to avoid race conditions.

Comparison to Blom and Orzan's algorithm. We now discuss a few differences of our algorithm to Blom and Orzan's original one [8].

In Blom and Orzan's algorithm for LTSs the sets In_s of $s \in S_i$ are in fact *lists* and contain worker w_k a total of r times if there exist r edges from states in S_k to s. This induces a redundancy in messages of ID updates, since w_i sends r (instead of one) messages with the ID of s to w_k. If the LTS has an average fanout of f then each worker has $t = n/W \cdot f$ outgoing transitions; this is the number of ID updates received every round. Since there are only n states, at most $n/t = W/f$ of those messages are necessary. In our scenario, we have $W \ll f$ for large coalgebras, hence the overhead becomes massive; e.g. for $W = 10, f = 100$ already 90% of all ID messages are redundant. We use sets instead of lists for In_s to avoid this redundancy.

Signature computation and communication do not proceed simultaneously in Blom and Orzan's original algorithm. However, in their optimized version [9] and in Blom et al.'s algorithm for state labelled continuous-time Markov chains [4] they do.

Another difference of our implementation is that we decided to hash the signatures directly on the workers of the respective states while Blom and Orzan decided to first send the signatures to some dedicated hashing worker who is then (uniquely) responsible for hashing, i.e. computing a new ID. This method allows to compute new IDs in constant time. However, for more complex functors supported by CoPaR, sending signatures could result in very large messages, so we opted for minimizing network traffic at the cost of slower signature computation.

5 Evaluation

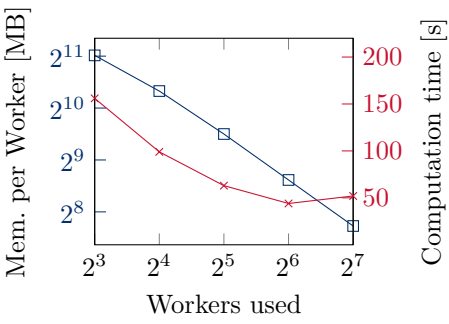
To illustrate the practical utility and scalability of the algorithm and its implementation in CoPaR, we report on a number of benchmarks performed on a selection of randomly generated and real world data. In previous evaluations of sequential CoPaR [41], we were limited by the 16GB RAM of a standard workstation. Here we demonstrate that our distributed implementation fulfills its

main objective of handling larger systems without lifting the memory restriction per process. All benchmarks were run on a high performance computing cluster consisting of nodes with two Xeon 2660v2 "Ivy Bridge" chips (10 cores per chip + SMT) with 2.2GHz clock rate and 64GB RAM. The nodes are connected by a fat-tree InfiniBand interconnect fabric with 40 GBit/s bandwidth. Most execution runs were performed using 32 workers on 8 nodes, resulting in 4 worker processes per node. No process used more than 16GB RAM. Execution times of the sequential algorithm were taken using one node of the cluster. No times are given for executions that ran out of 16GB memory previously [41]; those were not run on the cluster.

Weighted Tree Automata. In previous work [41], we have determined the size of the largest weighted tree automata for different parameters that the sequential version of CoPaR could handle in 16GB of RAM. Here, we demonstrate that the distributed version can indeed overcome these memory constraints and process much larger inputs.

Recall from Example 2.2 that weighted tree automata are coalgebras for the functor $FX = M \times M^{(\Sigma X)}$. For these benchmarks, we use $\Sigma X = 4 \times X^r$ with rank $r \in \{1, \dots, 5\}$ and the monoids $(2, \vee, 0)$ (available as the finite powerset functor in CoPaR), $(\mathbb{N}, \max, 0)$ and $(\mathcal{P}_\omega(64), \cup, \emptyset)$. To generate a random automaton with n states, we uniformly chose $k = 50 \cdot n$ transitions from the set of all possible transitions (using an efficient sampling algorithm by Vitter [39]) resulting in a coalgebra encoding with $n' = 51 \cdot n$ states and $m = (r + 1) \cdot k$ edges. We took care to restrict the state and transition weights to at most 50 different monoid elements in each example, to avoid the situation where all states are already distinguished in the first iteration of the algorithm.

Table 1 lists results for both the sequential and distributed implementation when run on the same input. These are the largest WTAs for their respective rank and monoid that sequential CoPaR could handle using at most 16GB of RAM [41]. In contrast, the distributed implementation uses less than 1GB per worker for those examples and is thus able to handle much larger inputs. Incidentally, the distributed implementation is also faster despite the overhead incurred by network communication. This can partly be attributed to the input-parsing stage, which does not need inter-worker synchronization and is thus perfectly parallelizable.



To test the scaling properties of the distributed algorithm, we ran CoPaR with the same input WTA but a varying number of worker processes. For this we chose the WTA for the monoid $(2, \vee, 0)$ with $\Sigma X = 4 \times X^5$ having 86852 states with 4342600 transitions and file size 186MB. The figure on the right above depicts the maximum memory usage per worker and the overall running time. The results show that both data points scale nicely with up to 32 workers, but while the

running time even increases when using up to 128 workers, the memory usage per worker (the main motivation for this work) continues to decrease significantly.

Monoid	r	k	n	Mem. (MB)	Time (s)	Seq. Time (s)
$(\mathcal{P}_\omega(64), \cup, \emptyset)$	5	4630750	92615	849	61	511
	4	4171550	83431	663	52	642
	3	4721250	94425	639	59	528
	2	6704100	134082	675	76	471
	1	7605350	152107	642	79	566
	3	47212500	944250	6786	675	–
$(\mathbb{N}, \max, 0)$	5	4722550	94451	871	61	445
	4	4643950	92879	754	56	463
	3	5039950	100799	628	64	391
	2	5904200	118084	633	74	403
	1	7845650	156913	677	82	438
	3	50399500	1007990	5644	645	–
$(2, \vee, 0)$	5	4342600	86852	701	71	537
	4	4624550	92491	728	67	723
	3	6710350	134207	825	113	689
	2	6900000	138000	715	129	467
	1	7743150	154863	621	160	449
	3	65000000	1300000	7092	1377	–

Table 1: Maximally manageable WTAs for sequential CoPaR; "Mem." and "Time" are the memory and time required for the distributed algorithm and are the maximum over all workers. "Seq. Time" is the time needed by sequential CoPaR.

PRISM Models. Finally, we show how our distributed partition refinement implementation performs on models from the benchmark suite [27] of the PRISM model checker [26]. These model (aspects of) real-world protocols and are thus a good fit to evaluate how CoPaR performs on inputs that arise in practice. Specifically, we use the *fms* and *wlan_time_bounded* families of systems. These are continuous time Markov chains, regarded as coalgebras for $FX = \mathbb{R}^{(X)}$, and Markov decision processes regarded as coalgebras for $FX = \mathbb{N} \times \mathcal{P}_\omega(\mathbb{N} \times (\mathcal{D}_\omega X))$, respectively. Again, our translation to coalgebras took care to force a coarse initial partition in the algorithm.

The results in Table 2 show that the distributed implementation is again able to handle larger systems than sequential CoPaR in 16GB of RAM per process. For the *fms* benchmarks, the distributed implementation is again faster than the sequential one. However, this is not the case for the *wlan* examples. The larger run times might be explained by the much higher number of iterations of the refinement phase (i-column of the table). This means that only few states are distinguished in each phase, and thus signatures are re-computed more often and more network traffic is incurred.

Model	n	m	Mem. (MB)	Time (s)	i	Seq. Time (s)
fms (n=4)	35910	237120	13	2	4	4
fms (n=5)	152712	1111482	62	8	5	17
fms (n=6)	537768	4205670	163	26	5	68
fms (n=7)	1639440	13552968	514	84	5	232
fms (n=8)	4459455	38533968	1690	406	7	–
wlan_tb (K=0)	582327	771088	90	297	306	39
wlan_tb (K=1)	1408676	1963522	147	855	314	105
wlan_tb (K=2)	1632799	5456481	379	2960	374	–

Table 2: Benchmarks on PRISM models: n and m are the numbers of states and edges of the input coalgebra; i is the number of refinement steps (iterations). The other columns are analogous to Table 1.

6 Conclusions and Future Work

We have presented a new and simple partition refinement algorithm in coalgebraic genericity which easily lends itself to a distributed implementation. Our algorithm is based on König and Küpper's final chain algorithm [25] and Blom and Orzan's signature refinement algorithm for labelled transition systems [8]. We have provided a distributed implementation in the tool CoPaR. Like the previous sequential Paige-Tarjan style partition refinement algorithm, our new algorithm is modular in the system type. This is made possible by combining signature interfaces by product and coproduct, which is used by CoPaR for handling combined type functors. Experimentation has shown that with the distributed algorithm CoPaR can handle larger state spaces in general. Run times stay low for weighted tree automata, whereas we observed severe penalties on some models from the PRISM benchmark suite.

An additional optimization of the coalgebraic signature refinement algorithm should be possible using Blom and Orzan's idea [9] to mark in each iteration those states whose signatures can change in the next iteration and only recompute signatures for those states in the next round. This might mitigate the run time penalties we have seen in some of the PRISM benchmarks.

Further work on CoPaR concerns symbolic techniques: we have a prototype sequential implementation of the coalgebraic signature refinement algorithm where state spaces are represented using BDDs. In a subsequent step it could be investigated whether this can be distributed. In another direction the distributed algorithm might be extended to compute distinguishing formulas, as recently achieved for the sequential algorithm [43], for which there is also an implemented prototype. Finally, there is still work required to integrate all these new features, i.e. distribution, distinguishing formulas, reachability and computation of minimized systems, into one version of CoPaR.

Data Availability Statement The software CoPaR and the input files that were used to produce the results in this paper are available for download [3]. The latest version of CoPaR can be obtained at https://git8.cs.fau.de/software/copar.

References

1. Balcazar, J., Gabarro, J., Santha, M.: Deciding bisimilarity is P-complete. Form. Asp. Comput. **4**(6A), 638–648 (1992)
2. Bartels, F., Sokolova, A., de Vink, E.: A hierarchy of probabilistic system types. In: Coalgebraic Methods in Computer Science, CMCS 2003. Electron. Notes Theor. Comput. Sci., vol. 82, pp. 57–75. Elsevier (2003)
3. Birkmann, F., Deifel, H.P., Milius, S.: Software and Benchmarks for Distributed Coalgebraic Partition Refinement (Jan 2022). https://doi.org/10.5281/zenodo.5907084
4. Blom, S., Haverkort, B.R., Kuntz, M., van de Pol, J.: Distributed Markovian bisimulation reduction aimed at CSL model checking. In: Proceedings of the 7th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2008). Electron. Notes Theor. Comput. Sci., vol. 220, pp. 35–50. Elsevier (2008)
5. Blom, S., Orzan, S.: A distributed algorithm for strong bisimulation reduction of state spaces. In: Brim, L., Grumberg, O. (eds.) Proc. Parallel and Distributed Model Checking (PDMC). Electron. Notes Theor. Comput. Sci., vol. 68, pp. 523–538. Elsevier (2002)
6. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. In: Sokolsky, O., Viswanathan, M. (eds.) Proc. Parallel and Distributed Model Checking (PDMC). Electron. Notes Theor. Comput. Sci., vol. 89, pp. 99–113. Elsevier (2003)
7. Blom, S., Orzan, S.: Distributed state space minimization. In: Arts, T., Fokkink, W. (eds.) Proc. Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS). Electron. Notes Theor. Comput. Sci., vol. 80, pp. 109–123. Elsevier (2003)
8. Blom, S., Orzan, S.: A distributed algorithm for strong bisimulation reduction of state spaces. International Journal on Software Tools for Technology Transfer **7**(1), 74–86 (2005). https://doi.org/10.1007/s10009-004-0159-4
9. Blom, S., Orzan, S.: Distributed state space minimization. International Journal on Software Tools for Technology Transfer **7**(3), 280–291 (Jun 2005). https://doi.org/10.1007/s10009-004-0185-2
10. Buchholz, P.: Bisimulation relations for weighted automata. Theoret. Comput. Sci. **393**, 109–123 (2008)
11. Deifel, H.P., Milius, S., Schröder, L., Wißmann, T.: Generic partition refinement and weighted tree automata. In: ter Beek et al., M. (ed.) Proc. International Symposium on Formal Methods (FM). Lecture Notes Comput. Sci., vol. 11800, pp. 280–297. Springer (2019)
12. Deifel, H.P., Milius, S., Wißmann, T.: Coalgebra encoding for efficient minimization. In: Kobayashi, N. (ed.) Proc. 6th International Conference on Formal Structures for Computation and Deduction (FSCD). LIPIcs, vol. 195, pp. 28:1–28:19. Schloss Dagstuhl (2021)
13. Derisavi, S., Hermanns, H., Sanders, W.: Optimal state-space lumping in Markov chains. Inf. Process. Lett. **87**(6), 309–315 (2003)
14. Desharnais, J., Edalat, A., Panangaden, P.: Bisimulation for labelled markov processes. Inform. Comput. **179**(2), 163–193 (2002)
15. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. International Journal on Software Tools for Technology Transfer **20**(2), 157–177 (Apr 2018). https://doi.org/10.1007/s10009-017-0468-z, http://link.springer.com/10.1007/s10009-017-0468-z
16. Dorsch, U., Milius, S., Schröder, L., Wißmann, T.: Efficient coalgebraic partition refinement. In: Meyer, R., Nestmann, U. (eds.) Proc. 28th International Conference

on Concurrency Theory (CONCUR). LIPIcs, vol. 85, pp. 28:1–28:16. Schloss Dagstuhl (2017)
17. van Glabbeek, R.: The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 3–99. Elsevier (2001)
18. Gries, D.: Describing an algorithm by Hopcroft. Acta Informatica **2**, 97–109 (1973)
19. Harris, T., Marlow, S., Peyton Jones, S.: Composable memory transactions. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 48–60. ACM Press (January 2005), https://www.microsoft.com/en-us/research/publication/composable-memory-transactions/
20. Högberg (Björklund), J., Maletti, A., May, J.: Bisimulation minimisation for weighted tree automata. In: Developments in Language Theory, 11th International Conference, DLT 2007, Turku, Finland, July 3-6, 2007, Proceedings. Lecture Notes Comput. Sci., vol. 4588, pp. 229–241. Springer (2007). https://doi.org/10.1007/978-3-540-73208-2
21. Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Theory of Machines and Computations. pp. 189–196. Academic Press (1971)
22. Huynh, D., Tian, L.: On some equivalence relations for probabilistic processes. Fund. Inform. **17**, 211–234 (1992)
23. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Inform. Comput. **86**(1), 43–68 (1990). https://doi.org/10.1016/0890-5401(90)90025-D
24. Knuutila, T.: Re-describing an algorithm by Hopcroft. Theoret. Comput. Sci. **250**, 333–363 (2001)
25. König, B., Küpper, S.: Generic partition refinement algorithms for coalgebras and an instantiation to weighted automata. In: Theoretical Computer Science, IFIP TCS 2014. Lecture Notes Comput. Sci., vol. 8705, pp. 311–325. Springer (2014)
26. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification, CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer (2011)
27. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012. pp. 203–204. IEEE Computer Society (2012). https://doi.org/10.1109/QEST.2012.14
28. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inform. Comput. **94**(1), 1–28 (1991)
29. Milner, R.: A Calculus of Communicating Systems, Lecture Notes Comput. Sci., vol. 92. Springer (1980)
30. Milner, R.: Communication and Concurrency. International Series in Computer Science, Prentice Hall (1989)
31. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
32. Park, D.: Concurrency on automata and infinite sequences. In: Deussen, P. (ed.) Proc. Conf. on Theoretical Computer Science. Lecture Notes Comput. Sci., vol. 104, pp. 167–183 (1981)
33. Rajasekaran, S., Lee, I.: Parallel algorithms for relational coarsest partition problems. IEEE Trans. Parallel Distributed Syst. **9**(7), 687–699 (1998). https://doi.org/10.1109/71.707548
34. Rutten, J.: Universal coalgebra: a theory of systems. Theoret. Comput. Sci. **249**, 3–80 (2000)

35. Rutten, J., de Vink, E.: Bisimulation for probabilistic transition systems: a coalgebraic approach. Theoret. Comput. Sci. **221**, 271–293 (1999)
36. Valmari, A.: Bisimilarity minimization in $\mathcal{O}(m \log n)$ time. In: Applications and Theory of Petri Nets, PETRI NETS 2009. Lecture Notes Comput. Sci., vol. 5606, pp. 123–142. Springer (2009)
37. Valmari, A.: Simple bisimilarity minimization in o(m log n) time. Fundam. Inform. **105**(3), 319–339 (2010). https://doi.org/10.3233/FI-2010-369
38. Valmari, A., Franceschinis, G.: Simple $\mathcal{O}(m \log n)$ time Markov chain lumping. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010. Lecture Notes Comput. Sci., vol. 6015, pp. 38–52. Springer (2010)
39. Vitter, J.S.: An efficient algorithm for sequential random sampling. ACM Trans. Math. Softw. **13**(1), 58–67 (1987). https://doi.org/10.1145/23002.23003
40. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref – A Symbolic Bisimulation Tool Box. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Graf, S., Zhang, W. (eds.) Automated Technology for Verification and Analysis, vol. 4218, pp. 477–492. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11901914_35
41. Wißmann, T., Deifel, H.P., Milius, S., Schröder, L.: From generic partition refinement to weighted tree automata minimization. Form. Asp. Comput. **33**, 695–727 (2021)
42. Wißmann, T., Dorsch, U., Milius, S., Schröder, L.: Efficient and modular coalgebraic partition refinement. Log. Methods Comput. Sci. **16**(1), 8:1–8:63 (2020)
43. Wißmann, T., Milius, S., Schröder, L.: Explaining behavioural inequivalence generically in quasilinear time. In: Haddad, S., Varacca, D. (eds.) Proc. 32nd International Conference on Concurrency Theory (CONCUR). LIPIcs, vol. 203, pp. 31:1–32:18. Schloss Dagstuhl (2021)

From Bounded Checking to Verification of Equivalence via Symbolic Up-to Techniques *

Vasileios Koutavas[1], Yu-Yang Lin[1](✉), and Nikos Tzevelekos[2]

[1] Trinity College Dublin, Dublin, Ireland {Vasileios.Koutavas,linhouy}@tcd.ie
[2] Queen Mary University of London, London, UK nikos.tzevelekos@qmul.ac.uk

Abstract. We present a bounded equivalence verification technique for higher-order programs with local state. This technique combines fully abstract *symbolic environmental bisimulations* similar to symbolic game semantics, novel *up-to techniques*, and lightweight *state invariant annotations*. This yields an equivalence verification technique with no false positives or negatives. The technique is bounded-complete, in that all inequivalences are automatically detected given large enough bounds. Moreover, several hard equivalences are proved automatically or after being annotated with state invariants. We realise the technique in a tool prototype called Hobbit and benchmark it with an extensive set of new and existing examples. Hobbit can prove many classical equivalences including all Meyer and Sieber examples.

Keywords: Contextual equivalence · bounded model checking · symbolic bisimulation · up-to techniques · operational game semantics.

1 Introduction

Contextual equivalence is a relation over program expressions which guarantees that related expressions are interchangeable in any program context. It encompasses verification properties like safety and termination. It has attracted considerable attention from the semantics community (cf. the 2017 Alonzo Church Award), and has found its main applications in the verification of cryptographic protocols [4], compiler correctness [26] and regression verification [10,11,9,17].

In its full generality, contextual equivalence is hard as it requires reasoning about the behaviour of all program contexts, and becomes even more difficult in languages with higher-order features (e.g. callbacks) and local state. Advances in bisimulations [16,29,3], logical relations [1,13,15] and game semantics [18,25,8,20] have offered powerful theoretical techniques for hand-written proofs of contextual equivalence in higher-order languages with state. However, these advancements have yet to be fully integrated in verification tools for contextual equivalence in programming languages, especially in the case of bisimulation techniques. Existing tools [12,24,14] only tackle carefully delineated language fragments.

* This publication has emanated from research supported in part by a grant from Science Foundation Ireland under Grant number 13/RC/2094_2.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 178–195, 2022.
https://doi.org/10.1007/978-3-030-99527-0_10

In this paper we aim to push the frontier further by proposing a bounded model checking technique for contextual equivalence for the entirety of a higher-order language with local state (Sec. 3). This technique, realised in a tool called Hobbit,[3] automatically detects inequivalent program expressions given sufficient bounds, and proves hard equivalences automatically or semi-automatically.

Our technique uses a labelled transition system (LTS) for open expressions in order to express equivalence as a bisimulation. The LTS is symbolic both for higher-order arguments (Sec. 4), similarly to symbolic game models [8,20] and derived proof techniques [3,15], and first-order ones (Sec. 6), adopting established techniques (e.g. [6]) and tools such as Z3 [23]. This enables the definition of a fully abstract *symbolic environmental bisimulation*, the bounded exploration of which is the task of the Hobbit tool. Full abstraction guarantees that our tool finds all inequivalences given sufficient bounds, and only reports true inequivalences. As is corroborated by our experiments, this makes Hobbit a practical inequivalence detector, similar to traditional bounded model checking [2] which has been proved an effective bug detection technique in industrial-scale C code [6,7,30].

However, while proficient in bug finding, bounded model checking can rarely prove the absence of errors, and in our setting prove an equivalence: a bound is usually reached before all—potentially infinite—program runs are explored. Inspired by hand-written equivalence proofs, we address this challenge by proposing two key technologies: new *bisimulation up-to techniques*, and lightweight user guidance in the form of *state invariant annotations*. Hence we increase significantly the number of equivalences proven by Hobbit, including for example all classical equivalences due to Meyer and Sieber [21].

Up-to techniques [28] are specific to bisimulation and concern the reduction of the size of bisimulation relations, oftentimes turning infinite transition systems into finite ones by focusing on a core part of the relation. Although extensively studied in the theory of bisimulation, up-to techniques have not been used in practice in an equivalence checker. We specifically propose three novel up-to techniques: *up to separation* and *up to re-entry* (Sec. 5), dealing with infinity in the LTS due to the higher-order nature of the language, and *up to state invariants* (Sec. 7), dealing with infinity due to state updates. Up to separation allows us to reduce the knowledge of the context the examined program expressions are running in, similar to a frame rule in separation logic. Up to re-entry removes the need of exploring unbounded nestings of higher-order function calls under specific conditions. Up to state invariants allows us to abstract parts of the state and make finite the number of explored configurations by introducing state invariant predicates in configurations.

State invariants are common in equivalence proofs of stateful programs, both in handwritten (e.g. [16]) and tool-based proofs. In the latter they are expressed manually in annotations (e.g. [9]) or automatically inferred (e.g. [14]). In Hobbit we follow the manual approach, leaving heuristics for automatic invariant inference for future work. An important feature of our annotations is the ability to express relations between the states of the two compared terms, enabled by the up to

[3] Higher Order Bounded BIsimulation Tool (Hobbit), https://github.com/LaifsV1/Hobbit.

state invariants technique. This leads to finite bisimulation transition systems in examples where concrete value semantics are infinite state.

The above technologies, combined with standard up-to techniques, transform HOBBIT from a bounded checker into an equivalence prover able to reason about infinite behaviour in a finite manner in a range of examples, including classical example equivalences (e.g. all in [21]) and some that previous work on up-to techniques cannot algorithmically decide [3] (cf. Ex. 22). We have benchmarked HOBBIT on examples from the literature and newly designed ones (Sec. 8). Due to the undecidable nature of contextual equivalence, up-to techniques are not exhaustive: no set of up-to techniques is guaranteed to finitise all examples. Indeed there are a number of examples where the bisimulation transition system is still infinite and HOBBIT reaches the exploration bound. For instance, HOBBIT is not able to prove examples with inner recursion and well-bracketing properties, which we leave to future work. Nevertheless, our approach provides a contextual equivalence tool for a higher-order language with state that can prove many equivalences and inequivalences which previous work could not handle due to syntactic restrictions and other limitations (Sec. 9).

Related work Our paper marries techniques from environmental bisimulations up-to [16,29,28,3] with the work on fully abstract game models for higher-order languages with state [18,8,20]. The closest to our technique is that of Biernacki et al. [3], which introduces up-to techniques for a similar symbolic LTS to ours, albeit with symbolic values restricted to higher-order types, resulting in infinite LTSs in examples such as Ex. 21, and with inequivalence decided outside the bisimulation by (non-)termination, precluding the use up-to techniques in examples such as Ex. 22. Close in spirit is the line of research on logical relations [1,13,15] which provides a powerful tool for hand-written proofs of contextual equivalence. Also related are the tools HECTOR [12] and CONEQCT [24], and SYTECI [14], based on game semantics and step-indexed logical relations respectively (cf. Sec. 9).

2 High-Level Intuitions

Contextual equivalence requires that two program expressions lead to the same observable result *in any program context* these may be fed in. Instead of working directly with this definition, we can translate programs into a semantic model that is *fully abstract*, reducing contextual equivalence to semantic equality.

The semantic model we use is that of Game Semantics [18]. We model programs as formal interactions between two *players*: a *Proponent* (corresponding to the program) and an *Opponent* (standing for any program context). Concretely, these interactions are sets of traces produced from a Labelled Transition System (LTS), the nodes and labels of which are called *configurations* and *moves* respectively. The LTS captures the interaction of the program with its environment, which is realised via function applications and returns: moves can be *questions* (i.e. function applications) or *answers* (returns), and belong to proponent or opponent. E.g. a program calling an external function will issue a proponent question, while the return of the external function will be an opponent answer. In the examples that follow, moves that correspond to the opponent shall be underlined.

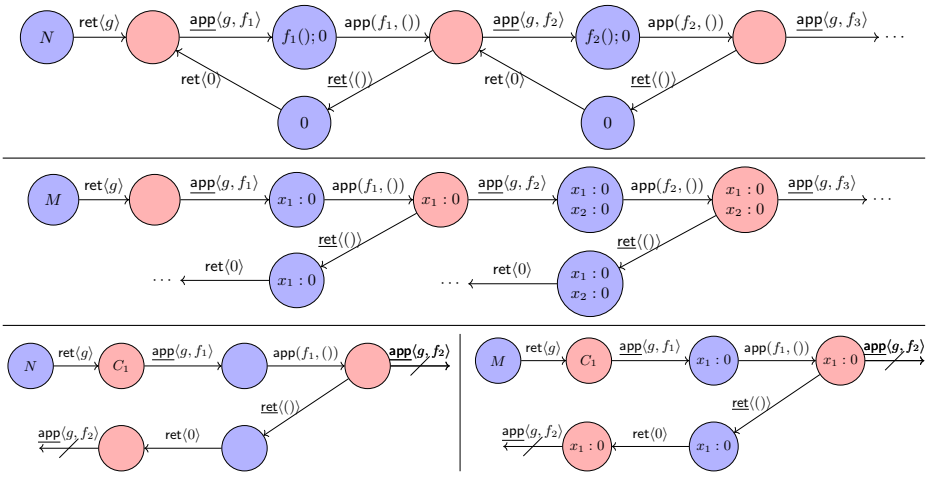


Fig. 1. Sample LTS's modelling expressions in Section 2.

Example 1. Consider the expression $N =$ `(fun f -> f (); 0)` of type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int}$. Evaluating N leads to a function g being returned (i.e. g is $\lambda f.f();0$). When g is called with some input f_1, it will always return 0 but in the process it may call the external function f_1. The call to f_1 may immediately return or it may call g again (i.e. reenter), and so on. The LTS for N is as in Fig. 1 (top).

Given two expressions M, N, checking their equivalence will amount to checking bisimulation equivalence of their (generally infinite) LTS's. Our checking routine performs a bounded analysis that aims to either find a finite counterexample and thus prove inequivalence, or build a bisimulation relation that shows the equivalence of the expressions. The former case is easier as it is relatively rapid to explore a bisimulation graph up to a given depth. The latter one is harder, as the target bisimulation can be infinite. To tackle part of this infinity, we use three novel *up-to techniques* for environmental bisimulation.

Up-to techniques roughly assert that if a core set of configurations in the bisimulation graph explored can be proven to be part of a relation satisfying a definition that is more permissive than standard bisimulation, then a superset of configurations forms a proper bisimulation relation. This has the implication that a bounded analysis can be used to explore a finite part of the bisimulation graph to verify potentially infinitely many configurations. As there can be no complete set of up-to techniques, the pertaining question is how useful they are in practice. In the remainder of this section we present the first of our up-to techniques, called *up to separation*, via an example equivalence. The intuition behind this technique comes from Separation Logic and amounts to saying that functions that access separate regions of the state can be explored independently. As a corollary, a function that manipulates only its own local references may be explored independently of itself, i.e. it suffices to call it once.

$$
\begin{array}{rl}
\text{Loc:} & l, k \qquad \text{Var:} x, y, z \qquad \text{Const:} c \\
\text{Type:} & T ::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{unit} \mid T \to T \mid T_1 * \ldots * T_n \\
\text{Exp:} & e, M, N ::= v \mid (\vec{e}) \mid op(\vec{e}) \mid e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{ref}\, l = v\, \mathsf{in}\, e \mid !l \mid l := e \mid \mathsf{let}(\vec{x}) = e\, \mathsf{in}\, e \\
\text{Val:} & u, v ::= c \mid x \mid \mathsf{fix} f(x).e \mid (\vec{v}) \\
\text{ECxt:} & E ::= [\cdot]_T \mid (\vec{v}, E, \vec{e}) \mid op(\vec{v}, E, \vec{e}) \mid E\, e \mid v\, E \mid l := E \mid \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{let}(\vec{x}) = E\, \mathsf{in}\, e \\
\text{Cxt:} & D ::= [\cdot]_{i,T} \mid e \mid (\vec{D}) \mid op(\vec{D}) \mid D\, D \mid l := D \mid \mathsf{if}\, D\, \mathsf{then}\, D\, \mathsf{else}\, D \mid \mathsf{fix} f(x).D \\
& \quad \mid \mathsf{ref}\, l = D\, \mathsf{in}\, D \mid \mathsf{let}(\vec{x}) = D\, \mathsf{in}\, D \\
\text{St:} & s, t \in \mathsf{Loc} \xrightarrow{fin} \mathsf{Val}
\end{array}
$$

$$
\begin{array}{lll}
\langle s\,;\, op(\vec{c})\rangle & \hookrightarrow \langle s\,;\, w\rangle & \text{if } op^{\mathsf{arith}}(\vec{c}) = w \\
\langle s\,;(\mathsf{fix} f(x).e)\, v\rangle & \hookrightarrow \langle s\,;\, e[v/x][\mathsf{fix} f(x).e/f]\rangle & \\
\langle s\,;\, \mathsf{let}(\vec{x}) = (\vec{v})\ \mathsf{in}\ e\rangle & \hookrightarrow \langle s\,;\, e[\vec{v}/\vec{x}]\rangle & \\
\langle s\,;\, \mathsf{ref}\, l = v\, \mathsf{in}\, e\rangle & \hookrightarrow \langle s[l \mapsto v]\,;\, e\rangle & \text{if } l \notin \mathsf{dom}(s) \\
\langle s\,;\, !l\rangle & \hookrightarrow \langle s\,;\, v\rangle & \text{if } s(l) = v \\
\langle s\,;\, l := v\rangle & \hookrightarrow \langle s[l \mapsto v]\,;()\rangle & \\
\langle s\,;\, \mathsf{if}\, c\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\rangle & \hookrightarrow \langle s\,;\, e_i\rangle & \text{if } (c, i) \in \{(\mathsf{tt}, 1), (\mathsf{ff}, 2)\} \\
\langle s\,;\, E[e]\rangle & \to \langle s'\,;\, E[e']\rangle & \text{if } \langle s\,;\, e\rangle \hookrightarrow \langle s'\,;\, e'\rangle
\end{array}
$$

Fig. 2. Syntax and reduction semantics of the language λ^{imp}.

Example 2. Consider $M =$ `(fun f -> ref x = 0 in f (); !x)` and N from Ex. 1. The LTS corresponding to M and N are shown in Fig. 1 (middle and top). Regarding M, we can see that opponent is always allowed to reenter the proponent function g, which creates a new reference x_n each time. This makes each configuration unique, which prevents us from finding cycles and thus finitise the bisimulation graph. Moreover, both the LTS for M and N are infinite because of the stack discipline they need to adhere to when O issues reentrant calls.

With separation, however, we could prune the two LTS's as in Fig. 1 (bottom). We denote the configurations after the first opponent call as C_1. Any opponent call after C_1 leads to a configuration which differs from C_1 either by a state component that is not accessible anymore and can thus be separated, or by a stack component that can be similarly separated. Hence, the LTS's that we need to consider are finite and thus the expressions are proven equivalent.

3 Language and Semantics

We develop our technique for the language λ^{imp}, a simply typed lambda calculus with local state whose syntax and reduction semantics are shown in Fig. 2. Expressions (Exp) include the standard lambda expressions with recursive functions ($\mathsf{fix} f(x).e$), together with location creation ($\mathsf{ref}\, l = v\, \mathsf{in}\, e$), dereferencing ($!l$), and assignment ($l := e$), as well as standard base type constants (c) and operations ($op(\vec{e})$). Locations are mapped to values, including function values, in a store (St). We write $\cdot$ for the empty store and let $\mathsf{fl}(\chi)$ denote the set of free locations in χ.

The language λ^{imp} is simply-typed with typing judgements of the form $\Delta; \Sigma \vdash e : T$, where Δ is a type environment (omitted when empty), Σ a store typing and T a value type (Type); Σ_s is the typing of store s. The rules of the type system are standard and omitted here. Values consist of boolean, integer, and unit constants,

functions and arbitrary length tuples of values. To keep the presentation of our technique simple we do not include reference types as value types, effectively keeping all locations local. Exchange of locations between expressions can be encoded using get and set functions. In Ex. 22 we show the encoding of a classic equivalence with location exchange between expressions and their context. Future work extensions to our technique to handle location types can be informed from previous work [18,14].

The reduction semantics is by small-step transitions between configurations containing a store and an expression, $\langle s\,;e\rangle \to \langle s'\,;e'\rangle$, defined using single-hole evaluation contexts (ECxt) over a base relation $\hookrightarrow$. Holes $[\cdot]_T$ are annotated with the type T of closed values they accept, which we may omit to lighten notation. Beta substitution of x with v in e is written as $e[v/x]$. We write $\langle s\,;e\rangle \Downarrow$ to denote $\langle s\,;e\rangle \to^* \langle t\,;v\rangle$ for some t, v. We write $\vec{\chi}$ to mean a syntactic sequence, and assume standard syntactic sugar from the lambda calculus. In our examples we assume an ML-like syntax and implementation of the type system, which is also the concrete syntax of Hobbit.

We consider environments $\Gamma \in \mathbb{N} \xrightarrow{\mathsf{fin}} \mathsf{Val}$ which map natural numbers to closed values. The concatenation of two such environments Γ_1 and Γ_2, written Γ_1, Γ_2 is defined when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. We write $({}^{i_1}v_1, \ldots, {}^{i_n}v_n)$ for a concrete environment mapping $i_1, \ldots, i_n$ to $v_1, \ldots, v_n$, respectively. When indices are unimportant we omit them and treat Γ environments as lists.

General contexts D contain multiple, non-uniquely indexed holes $[\cdot]_{i,T}$, where T is the type of value that can replace the hole. Notation $D[\Gamma]$ denotes the context D with each hole $[\cdot]_{i,T}$ replaced with $\Gamma(i)$, provided that $i \in \mathsf{dom}(\Gamma)$ and $\Sigma \vdash \Gamma(i) : T$, for some Σ. We omit hole types where possible and indices when all holes in D are annotated with the same i. In the latter case we write $D[v]$ instead of $D[({}^{i}v)]$ and allow to replace all holes of D with a closed expression e, written $D[e]$. We assume the Barendregt convention for locations, thus replacing context holes avoids location capture. Standard contextual equivalence [22] follows.

Definition 3 (Contextual Equivalence). *Expressions* $\vdash e_1 : T$ *and* $\vdash e_2 : T$ *are* contextually equivalent*, written as* $e_1 \equiv e_2$*, when for all contexts* D *such that* $\vdash D[e_1] : \mathsf{unit}$ *and* $\vdash D[e_2] : \mathsf{unit}$ *we have* $\langle \cdot\,; D[e_1]\rangle \Downarrow$ *iff* $\langle \cdot\,; D[e_2]\rangle \Downarrow$.

4 LTS with Symbolic Higher-Order Transitions

Our Labelled Transition System (LTS) has symbolic transitions for both higher-order and first-order transitions. For simplicity we first present our LTS with symbolic higher-order and concrete first-order transitions. We develop our theory and most up-to techniques on this simpler LTS. We then show its extension with symbolic first-order transitions and develop up to state invariants which relies on this extension. We extend the syntax with abstract function names α:

$$\mathsf{Val}: \quad u, v, w ::= c \mid \mathsf{fix} f(x).e \mid (\vec{v}) \mid \alpha_T$$

Abstract function names α_T are annotated with the type T of function they represent, omitted where possible; $\mathsf{an}(\chi)$ is the set of abstract names in χ.

$$\textsc{PropApp} : \langle A\,;\Gamma\,;K\,;s\,;E[\alpha\,v]\rangle \xrightarrow{\mathsf{app}(\alpha,D)} \langle A\,;\Gamma,\Gamma'\,;E[\cdot],K\,;s\,;\cdot\rangle \quad \text{if } (D,\Gamma') \in \mathsf{ulpatt}(v)$$

$$\textsc{PropRet} : \langle A\,;\Gamma\,;K\,;s\,;v\rangle \xrightarrow{\mathsf{ret}(D)} \langle A\,;\Gamma,\Gamma'\,;K\,;s\,;\cdot\rangle \quad \text{if } (D,\Gamma') \in \mathsf{ulpatt}(v)$$

$$\textsc{OpApp} : \langle A\,;\Gamma\,;K\,;s\,;\cdot\rangle \xrightarrow{\mathsf{app}(i,D[\vec{\alpha}])} \langle A \uplus \vec{\alpha}\,;\Gamma\,;K\,;s\,;e\rangle \quad \text{if } \Sigma_s \vdash \Gamma(i) : T \to T' \text{ and } (D,\vec{\alpha}) \in \mathsf{ulpatt}(T) \text{ and } \Gamma(i)\,D[\vec{\alpha}] \succ e$$

$$\textsc{OpRet} : \langle A\,;\Gamma\,;E[\cdot]_T,K\,;s\,;\cdot\rangle \xrightarrow{\mathsf{ret}(D[\vec{\alpha}])} \langle A \uplus \vec{\alpha}\,;\Gamma\,;K\,;s'\,;E[D[\vec{\alpha}]]\rangle \quad \text{if } (D,\vec{\alpha}) \in \mathsf{ulpatt}(T)$$

$$\textsc{Tau} : \langle A\,;\Gamma\,;K\,;s\,;e\rangle \xrightarrow{\tau} \langle A\,;\Gamma\,;K\,;s\,;e'\rangle \quad \text{if } \langle s\,;e\rangle \to \langle s'\,;e'\rangle$$

$$\textsc{Response} : C \xrightarrow{\eta} \langle\bot\rangle \quad \text{if } \eta \neq \downarrow$$

$$\textsc{Term} : \langle A\,;\Gamma\,;\cdot\,;s\,;\cdot\rangle \xrightarrow{\downarrow} \langle\bot\rangle$$

Fig. 3. The Labelled Transition System.

We define our LTS (shown in Fig. 3) by opponent and proponent call and return transitions, based on Game Semantics [18]. Proponent transitions are the moves of an expression interacting with its context. Opponent transitions are the moves of the context surrounding this expression. These transitions are over proponent and opponent configurations $\langle A\,;\Gamma\,;K\,;s\,;e\rangle$ and $\langle A\,;\Gamma\,;K\,;s\,;\cdot\rangle$, respectively. In these configurations:

- A is a set of abstract function names been used so far in the interaction;
- Γ is an environment indexing proponent functions known to opponent;[4]
- K is a stack of proponent continuations, created by nested proponent calls;
- s is the store containing proponent locations;
- e is the expression reduced in proponent configurations; $\hat{e}$ denotes e or $\cdot$.

In addition, we introduce a special configuration $\langle\bot\rangle$ which is used in order to represent expressions that cannot perform given transitions (cf. Remark 6). We let a *trace* be a sequence of app and ret moves (i.e. labels), as defined in Fig. 3.

For the LTS to provide a fully abstract model of the language, it is necessary that functions which are passed as arguments or return values from proponent to opponent be abstracted away, as the actual syntax of functions is not directly observable in λ^{imp}. This is achieved by deconstructing such values v to:

- an *ultimate pattern* D (cf. [19]), which is a context obtained from v by replacing each function in v with a distinct numbered hole; together with
- an environment Γ mapping indices of these holes to values, and $D[\Gamma] = v$.

We let $\mathsf{ulpatt}(v)$ contain all such pairs (D,Γ) for v; e.g.: $\mathsf{ulpatt}((\lambda x.e_1, 5)) = \{(([\cdot]_i, 5), [^i\lambda x.e_1]) \mid \text{for any } i\}$. We extend ulpatt to types through the use of symbolic function names: $\mathsf{ulpatt}(T)$ is the largest set of pairs (D,Γ) such that $\vdash D[\Gamma] : T$, where $rng(\Gamma) = \vec{\alpha}_{\vec{T}}$, and D does not contain functions.

[4] thus, Γ is encoding the environment of Environmental Bisimulations (e.g. [16])

In Fig. 3, proponent application and return transitions (PropApp, PropRet) use ultimate pattern matching for values and accumulate the functions generated by the proponent in the Γ environment of the configuration, leaving only their indices on the label of the transition itself. Opponent application and return transitions (OpApp, OpRet) use ultimate pattern matching for types to generate opponent-generated values which can only contain abstract functions. This eliminates the need for quantifying over all functions in opponent transitions but still includes infinite quantification over all base values. Symbolic first-order values in Sec. 6 will obviate the latter.

At opponent application the following preorder performs a beta reduction when opponent applies a concrete function. This technicality is needed for soundness.

Definition 4 ($\succ$). *For application $v\,u$ we write $v\,u \succ e$ to mean $e = \alpha\,u$, when $v = \alpha$; and $e = e'[u/x][\mathsf{fix} f(x).e'/f]$, when $v = \mathsf{fix} f(x).e'$.*

In our LTS, C ranges over configurations and η over transition labels; $\overset{\eta}{\Longrightarrow}$ means $\overset{\tau}{\longrightarrow}{}^*$, when $\eta = \tau$, and $\overset{\tau}{\Longrightarrow}\overset{\eta}{\longrightarrow}\overset{\tau}{\Longrightarrow}$ otherwise. Standard weak (bi-)simulation follows.

Definition 5 (Weak Bisimulation). *Binary relation $\mathcal{R}$ is a* weak simulation *when for all $C_1\ \mathcal{R}\ C_2$ and $C_1 \overset{\eta}{\longrightarrow} C_1'$, there exists C_2' such that $C_2 \overset{\eta}{\Longrightarrow} C_2'$ and $C_1'\ \mathcal{R}\ C_2'$. If $\mathcal{R}$, $\mathcal{R}^{-1}$ are weak simulations then $\mathcal{R}$ is a* weak bisimulation. *Similarity ($\underset{\approx}{\sqsubset}$) and bisimilarity ($\approx$) are the largest weak simulation and bisimulation, respectively.*

Remark 6. Any proponent configuration that cannot match a standard bisimulation transition challenge can trivially respond to the challenge by transitioning into $\langle\bot\rangle$ by the Response rule in Fig. 3. By the same rule, this configuration can trivially perform all transitions except a special termination transition, labelled with $\downarrow$. However, regular configurations that have no pending proponent calls ($K = \cdot$), can perform the special termination transition (Term rule), signalling the end of a *complete trace*, i.e. a completed computation. This mechanism allows us to encode complete trace equivalence, which coincides with contextual equivalence [18], as bisimulation equivalence. In a bisimulation proof, if a proponent configuration is unable to match a bisimulation transition with a regular transition, it can still transition to $\langle\bot\rangle$ where it can simulate every transition of the other expression, apart from $\overset{\downarrow}{\longrightarrow}$ leading to a complete trace.

Our mechanism for treating unmatched transitions has the benefit of enabling us to use the standard definition of bisimulation over our LTS. This is in contrast to previous work [3,15], where termination/non-termination needed to be proven independently or baked in the simulation conditions. More importantly, our approach allows us to use bisimulation up-to techniques even when one of the related configurations diverges, which is not possible in previous symbolic LTSs [18,15,3], and is necessary in examples such as Ex. 22.

Definition 7 (Bisimilar Expressions). *Expressions $\vdash e_1 : T$ and $\vdash e_2 : T$ are bisimilar, written $e_1 \approx e_2$, when $\langle \cdot ; \cdot ; \cdot ; \cdot ; e_1 \rangle \approx \langle \cdot ; \cdot ; \cdot ; \cdot ; e_2 \rangle$.*

Theorem 8 (Soundness and Completeness). *$e_1 \approx e_2$ iff $e_1 \equiv e_2$.*

As a final remark, the LTS presented in this section is finite state only for a small number of trivial equivalence examples. The following section addresses sources of infinity in the transition systems through bisimulation up-to techniques.

5 Up-to Techniques

We start by the definition of a sound up-to technique.

Definition 9 (Weak Bisimulation up to f). *$\mathcal{R}$ is a* weak simulation up to f *when for all $C_1\ \mathcal{R}\ C_2$ and $C_1 \xrightarrow{\eta} C_1'$, there is C_2' with $C_2 \xRightarrow{\eta} C_2'$ and $C_1'\ f(\mathcal{R})\ C_2'$. If $\mathcal{R}$, $\mathcal{R}^{-1}$ are weak simulations up to f then $\mathcal{R}$ is a* weak bisimulation up to f.

Definition 10 (Sound up-to technique). *A function f is a* sound up-to technique *when for any $\mathcal{R}$ which is a simulation up to f we have $R \subseteq (\lessapprox)$.*

Hobbit employs the standard techniques: up to identity, up to garbage collection, up to beta reductions and up to name permutations. Here we present two novel up-to techniques: up to separation and up to reentry.

Up to Separation Our experience with Hobbit has shown that one of the most effective up-to techniques for finitising bisimulation transition systems is the novel *up to separation* which we propose here. The intuition of this technique is that if different functions operate on disjoint parts of the store, they can be explored in disjoint parts of the bisimulation transition system. Taken to the extreme, a function that does not contain free locations can be applied only once in a bisimulation test as two copies of the function will not interfere with each other, even if they allocate new locations after application. To define up to separation we need to define a separating conjunction for configurations.

Definition 11 (Stack Interleaving). *Let K_1, K_2 be lists of evaluation contexts from* ECxt *(Fig. 2); we define the interleaving operation $K_1 \#_{\vec{k}} K_2$ inductively, and write $K_1 \# K_2$ to mean $K_1 \#_{\vec{k}} K_2$ for unspecified $\vec{k}$. We let $\cdot \# \cdot = \cdot$ and:*

$$E_1, K_1 \#_{(1,\vec{k})} K_2 = E_1, (K_1 \#_{\vec{k}} K_2) \qquad K_1 \#_{(2,\vec{k})} E_2, K_2 = E_2, (K_1 \#_{\vec{k}} K_2)\,.$$

Definition 12 (Separating Conjuction). *Let $C_1 = \langle A_1 ; \Gamma_1 ; K_1 ; s_1 ; \hat{e}_1 \rangle$ and $C_2 = \langle A_2 ; \Gamma_2 ; K_2 ; s_2 ; \hat{e}_2 \rangle$ be well-formed configurations. We define:*

- $C_1 \oplus^1_{\vec{k}} C_2 \stackrel{def}{=} \langle A_1 \cup A_2 ; \Gamma_1, \Gamma_2 ; K_1 \#_{\vec{k}} K_2 ; s_1, s_2 ; \hat{e}_1 \rangle$ *when* $\hat{e}_2 = \cdot$
- $C_1 \oplus^2_{\vec{k}} C_2 \stackrel{def}{=} \langle A_1 \cup A_2 ; \Gamma_1, \Gamma_2 ; K_1 \#_{\vec{k}} K_2 ; s_1, s_2 ; \hat{e}_2 \rangle$ *when* $\hat{e}_1 = \cdot$

provided $\mathsf{dom}(s_1) \cap \mathsf{dom}(s_2) = \emptyset$. *We let $C_1 \oplus C_2$ denote $\exists i, \vec{k}.\ C_1 \oplus^i_{\vec{k}} C_2$.*

The function sep provides the up to separation technique; it is defined as:

$$\textsc{UpTo}\oplus\ \frac{C_1\ \mathcal{R}\ C_2 \qquad C_3\ \mathcal{R}\ C_4}{C_1 \oplus^i_{\vec{k}} C_3\ \mathsf{sep}(\mathcal{R})\ C_2 \oplus^i_{\vec{k}} C_4} \qquad \textsc{UpTo}\oplus\bot_L\ \frac{C_1\ \mathcal{R}\ \langle\bot\rangle \qquad C_3\ \mathcal{R}\ C_4}{C_1 \oplus C_3\ \mathsf{sep}(\mathcal{R})\ \langle\bot\rangle} \qquad \textsc{UpTo}\oplus\bot_R\ \frac{C_1\ \mathcal{R}\ C_2 \qquad C_3\ \mathcal{R}\ \langle\bot\rangle}{C_1 \oplus C_3\ \mathsf{sep}(\mathcal{R})\ \langle\bot\rangle}$$

Soundness follows by extending [28,27] with a weaker, sufficient proof obligation.

Lemma 13. *Function* sep *is a sound up-to technique.*

Many example equivalences have a finite transition system when using up to separation in conjunction with the simple techniques of the preceding section.

Example 14. The following is a classic example equivalence from Meyer and Sieber [21]. The following expressions are equivalent at type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$.

$M =$ `fun f -> ref x = 0 in f ()` $\qquad$ $N =$ `fun f -> f ()`

For both functions, after initial application of the function by the opponent, the proponent calls `f`, growing the stack K in the two configurations. At that point the opponent can apply the same functions again. The LTS of both M and N is thus infinite because K can grow indefinitely, and so is a bisimulation proving this equivalence. It is additionally infinite because the opponent can keep applying the initial function applications even after these return. However, if we apply the up-to separation technique immediately after the first opponent application, the Γ environments become empty, and thus no second application of the same functions can happen. The LTS thus becomes trivially small. Note that no other up to technique is needed here. Hobbit applies up-to separation after every opponent application transition and explores the configuration containing the application expression and the smallest possible Γ; this does not lead to false-negative (or false-positive) results.

Example 15. This example is due to Bohr and Birkedal [5] and includes a non-synchronised divergence.

$M =$
```
fun f ->
  ref l1 = false in ref l2 = false in
  f (fun () -> if !l1 then _bot_ else l2 := true);
  if !l2 then _bot_ else l1 := true
```

$N =$ `fun f -> f (fun () -> _bot_)`

Note that `_bot_` is a diverging computation. This is a hard example to prove using environmental bisimulation even with up to techniques, requiring quantification over contexts within the proof. However, with up-to separation after the opponent applies the initial functions, the Γ environments are emptied, thus leaving only one application of M and N that needs to be explored by the bisimulation. Applications of the inner function provided as argument to `f` only leads to a small number of reachable configurations. Hobbit can indeed prove this equivalence.

Up to Proponent Function Re-entry The higher-order nature of λ^{imp} and its LTS allows infinite nesting of opponent and proponent calls. Although up to separation avoids those in a number of examples, here we present a second novel up-to technique, which we call *up to proponent function re-entry* (or simply, up to re-entry). This technique has connections to the induction hypothesis in the definition of environmental bisimulations in [16]. However up to re-entry is specifically aimed at avoiding nested calls to proponent functions, and it is designed to work with our symbolic LTS. In combination with other techniques this eliminates the need to consider configurations with unbounded stacks K in many classical equivalences, including those in [21].

$$\textsc{UpToReentry}\quad \frac{\begin{array}{l} C_1 = \langle A\,;\Gamma_1\,;K_1\,;s_1\,;\cdot\rangle\ \mathcal{R}\ \langle A\,;\Gamma_2\,;K_2\,;s_2\,;\cdot\rangle = C_2 \\ \forall \vec{\eta}, C, A', \Gamma_1', \Gamma_2', s_1', s_2'.\ \big[(\underline{\mathsf{app}}(i,\_) \notin \{\vec{\eta}\} \text{ and} \\ \qquad \langle A\,;\Gamma_1\,;\cdot\,;s_1\,;\cdot\rangle \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}} \asymp \langle A'\,;\Gamma_1'\,;\cdot\,;s_1'\,;\cdot\rangle \text{ and} \\ \qquad \langle A\,;\Gamma_2\,;\cdot\,;s_2\,;\cdot\rangle \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}} \asymp \langle A'\,;\Gamma_2'\,;\cdot\,;s_2'\,;\cdot\rangle \\ \qquad \text{implies } \Gamma_1' = \Gamma_1 \text{ and } \Gamma_2' = \Gamma_2 \text{ and } s_1 = s_1' \text{ and } s_2 = s_2'\big] \\ C_1 \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}'} \xrightarrow{\underline{\mathsf{app}}(i,C')} \langle A'\,;\Gamma_1\,;K_1'\,,K_1\,;s_1\,;e_1'\rangle \\ C_2 \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}'} \xrightarrow{\underline{\mathsf{app}}(i,C')} \langle A'\,;\Gamma_2\,;K_2'\,,K_2\,;s_2\,;e_2'\rangle \end{array}}{\langle A'\,;\Gamma_1\,;K_1'\,,K_1\,;s_1\,;e_1'\rangle\ \mathsf{reent}(\mathcal{R})\ \langle A'\,;\Gamma_2\,;K_2'\,,K_2\,;s_2\,;e_2'\rangle}$$

Fig. 4. Up to Proponent Function Re-entry (omitting rules for $\perp$-configurations).

Up to re-entry is realised by function reent in Fig. 4. The intuition of this up-to technique is that if the application of related functions at i in the Γ environments has no potential to change the local stores (up to garbage collection, encoded by $(\asymp)$) or increase the Γ environments, then there are no additional observations to be made by nested calls to the i-functions, thus configurations reached by such nested calls are added to the relation by this up-to technique. Soundness follows similarly to up-to separation.

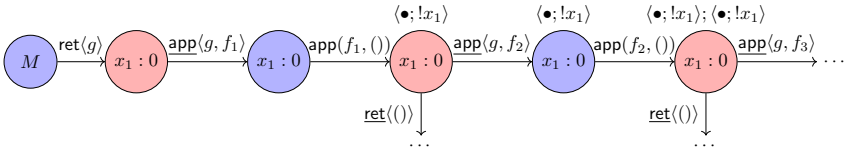
In Hobbit we require the user to flag the functions to be considered for the up to re-entry technique. This annotation is later combined with state invariant annotations, as they are often used together. Inequivalences found while using the up to re-entry and state invariant annotations could be false-negatives due to incorrect user annotations. Hobbit ensures that no such false-negatives are reported by re-running discovered inequivalences with these two techniques off.

Below is an example where the state invariant needed is trivial and up to separation together with up to re-entry are sufficient to prove the equivalence.

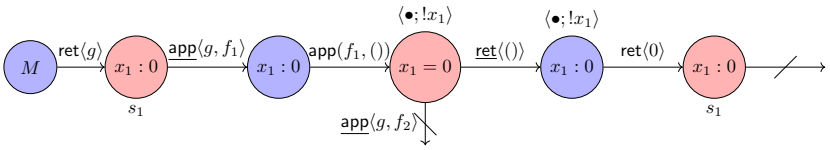
Example 16.

$M =$ `ref x = 0 in fun f -> f (); !x` $\qquad N =$ `fun f -> f (); 0`

This is like Ex. 2 except the reference in M is created outside of the function body. The LTS for this is as follows. Labels $\langle \bullet; !x_1\rangle$ are continuations.



Again, the opponent is allowed to reenter g as before. With up-to reentry, however, the opponent skips nested calls to g as these do not modify the state.



N mirrors the above LTS without the x_1 reference and with continuation $\langle \bullet; 0 \rangle$.

6 Symbolic First-Order Transitions

We extend λ^{imp} constants (Const) with a countable set of symbolic constants ranged over by κ. We define symbolic environments $\sigma ::= \cdot \mid (\kappa \frown e), \sigma$, where $\frown$ is either $=$ or $\neq$, and e is an arithmetic expression over constants, and interpret them as conjunctions of (in-)equalities, with the empty set interpreted as $\top$.

Definition 17 (Satisfiability). *Symbolic environment σ is* satisfiable *if there exists an* assignment *δ, mapping the symbolic constants of σ to actual constants, such that $\delta\sigma$ is a tautology; we then write $\delta \vDash \sigma$.*

We extend reduction configurations with a symbolic environment σ, written as $\sigma \vdash \langle s \,;\, e \rangle$. These constants are implicitly annotated with their type. We modify the reduction semantics from Fig. 2 to consider symbolic constants:

$$\begin{array}{lll}
\sigma \vdash \langle s \,;\, op(\vec{c}) \rangle & \hookrightarrow \sigma \wedge (\kappa = op(\vec{c})) \vdash \langle s \,;\, \kappa \rangle & \text{if } \kappa \text{ fresh} \\
\sigma \vdash \langle s \,;\, \mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rangle & \hookrightarrow \sigma \wedge (\kappa = \mathsf{tt}) \vdash \langle s \,;\, e_1 \rangle & \text{if } \sigma \wedge (\kappa = \mathsf{tt}) \text{ is sat.} \\
\sigma \vdash \langle s \,;\, \mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rangle & \hookrightarrow \sigma \wedge (\kappa = \mathsf{ff}) \vdash \langle s \,;\, e_2 \rangle & \text{if } \sigma \wedge (\kappa = \mathsf{ff}) \text{ is sat.}
\end{array}$$

All other reduction semantics rules carry the σ. The LTS from Sec. 4 is modified to operate over configurations of the form $\sigma \vdash C$ or $\cdot \vdash \langle \bot \rangle$. We let $\widetilde{C}$ range over both forms of configurations. All LTS rules for proponent transitions simply carry the σ; rule Tau may increase σ due to the inner reduction. Opponent transitions generate fresh symbolic constants, instead of actual constants: labels $\underline{\mathsf{app}}(i, D[\vec{\alpha}])$ and $\underline{\mathsf{ret}}(D[\vec{\alpha}])$ in rules OpApp and OpRet of Fig. 3, respectively, contain D with symbolic, instead of concrete constants. We adapt (bi-)simulation as follows.

Definition 18. *Binary relation $\mathcal{R}$ on symbolic configurations is a* weak simulation *when for all $\widetilde{C}_1\ \mathcal{R}\ \widetilde{C}_2$ and $\widetilde{C}_1 \xrightarrow{\eta_1} \widetilde{C}_1'$, $\exists \widetilde{C}_2'$ such that $\widetilde{C}_2 \overset{\eta_2}{\Longrightarrow} \widetilde{C}_2'$ and*

$$\widetilde{C}_1'\ \mathcal{R}\ \widetilde{C}_2' \qquad (\widetilde{C}_1'.\sigma, \widetilde{C}_2'.\sigma) \text{ is sat.} \qquad \forall \delta.\ \delta \vDash (\widetilde{C}_1'.\sigma, \widetilde{C}_2'.\sigma) \implies \delta\eta_1 = \delta\eta_2$$

Lemma 19. $(\sigma_1 \vdash C_1) \sqsubseteq\!\!\!\approx (\sigma_2 \vdash C_2)$ *iff for all $\delta \vDash \sigma_1, \sigma_2$ we have $\delta C_1 \sqsubseteq\!\!\!\approx \delta C_2$.*

Corollary 20 (Soundness, Completeness). $(\cdot \vdash C_1) \sqsubseteq\!\!\!\approx (\cdot \vdash C_2)$ *iff* $C_1 \sqsubseteq\!\!\!\approx C_2$.

The up-to techniques we have developed in previous sections apply unmodified to the extended LTS as the techniques do not involve symbolic constants, with the exception of up to beta which requires adapting the definition of a beta move to consider all possible δ. The introduction of symbolic first-order transitions allows us to prove many interesting first-order examples, such as the equivalence of

bubble sort and insertion sort, an example borrowed from Hector [12] (omitted here, see the Hobbit distribution). Below is a simpler example showing the equivalence of two integer swap functions which, by leveraging Z3 [23], Hobbit is able to prove.

Example 21.

$M =$
```
let swap xy =
  let (x,y) = xy
  in (y, x)
in swap
```

$N =$
```
fun xy -> let (x,y) = xy in
  ref x = x in ref y = y in
  x := !x - !y; y := !x + !y;
  x := !y - !x; (!x, !y)
```

7 Up to State Invariants

The addition of symbolic constants into λ^{imp} and the LTS not only allows us to consider all possible opponent-generated constants simultaneously in a symbolic execution of proponent expressions, but also allows us to define an additional powerful up-to technique: *up to state invariants*. We define this technique in two parts: *up to abstraction* and *up to tautology* realised by abs and taut.[5]

$$\textsc{UpTo}\mathsf{abs}\ \frac{(\sigma_1 \vdash C_1)\ \mathcal{R}\ (\sigma_2 \vdash C_2)}{(\sigma_1 \vdash C_1)[\vec{c}/\vec{\kappa}]\ \mathsf{abs}(\mathcal{R})\ (\sigma_2 \vdash C_2)[\vec{c}/\vec{\kappa}]} \qquad \textsc{UpTo}\mathsf{taut}\ \frac{(\sigma_1, \sigma_1' \vdash C_1)\ \mathcal{R}\ (\sigma_2, \sigma_2' \vdash C_2) \quad \sigma_1, \sigma_2, \sigma_1', \sigma_2' \text{ is sat.} \quad \sigma_1, \sigma_2 \wedge \neg(\sigma_1', \sigma_2') \text{ is not sat.}}{(\sigma_1 \vdash C_1)\ \mathsf{taut}(\mathcal{R})\ (\sigma_2 \vdash C_2)}$$

The first function abs allows us to derive the equivalence of configurations by abstracting constants with fresh symbolic constants (of the same type) and instead prove equivalent the more abstract configurations. The second function taut allows us to introduce tautologies into the symbolic environments. These are predicates which are valid; i.e., they hold for all instantiations of the abstract variables. Combining the two functions we can introduce a tautology $I(\vec{c})$ into the symbolic environments, and then abstract constants $\vec{c}$ from the predicate but also from the configurations with symbolic ones, obtaining $I(\vec{\kappa})$, which encodes an invariant that always holds.

Currently in Hobbit, up to abstraction and tautology are combined and applied in a principled way. Functions can be annotated with the following syntax:

F `= fun x {`$\vec{\kappa}$ `|` l_1 `as` $C_1[\vec{\kappa}]$`, ..,` l_n `as` $C_n[\vec{\kappa}]$ `|` ϕ`} -> e`

The annotation instructs Hobbit to use the two techniques when opponent applies related functions where at least one of them has such an annotation. If both functions contain annotations, then they are combined and the same $\vec{\kappa}$ are used in both annotations. The techniques are used again when proponent returns from the functions, and proponent calls opponent from within the functions.[6] As discussed in Sec. 5, the same annotation enables up to reentry in Hobbit.

When Hobbit uses the above two up-to techniques it *1)* pattern-matches the values currently in each location l_i with the value context C_i where fresh

[5] Hobbit also implements an *up to σ-normalisation and garbage collection* technique.
[6] Finer-grain control of application of these up-to techniques is left to future work.

symbolic constants $\vec{\kappa}$ are in its holes, obtaining a substitution $[\vec{c}/\vec{\kappa}]$; *2)* the up to tautology technique is applied for the formula $\phi[\vec{c}/\vec{\kappa}]$; and *3)* the up to abstraction technique is applied by replacing $\phi[\vec{c}/\vec{\kappa}]$ in the symbolic environment with ϕ, and the contents of locations l_i with $C_i[\vec{\kappa}]$.

Example 22. Following is an example by Meyer and Sieber [21] featuring location passing, adapted to λ^{imp} where locations are local.

$M =$
```
let loc_eq loc1loc2 = [...] in
fun q -> ref x = 0 in
         let locx = (fun () -> !x) , (fun v -> x := v) in
         let almostadd_2 locz {w | x as w | w mod 2 == 0} =
           if loc_eq (locx,locz) then x := 1 else x := !x + 2
         in q almostadd_2; if !x mod 2 = 0 then _bot_ else ()
```

$N =$
```
fun q -> _bot_
```

In this example we simulate general references as a pair of read-write functions. Function `loc_eq` implements a standard location equality test. The two higher-order expressions are equivalent because the opponent can only increase the contents of `x` through the function `almostadd_2`. As the number of times the opponent can call this function is unbounded, the LTS is infinite. However, the annotation of function `almostadd_2` applies the up to state invariants technique when the function is called (and, less crucially, when it returns), replacing the concrete value of `x` with a symbolic integer constant w satisfying the invariant `w mod 2 == 0`. This makes the LTS finite, up to permutations of symbolic constants. Moreover, up to separation removes the outer functions from the Γ environments, thus preventing re-entrant calls to these functions. Note the up to techniques are applied even though one of the configurations is diverging (`_bot_`). This would not be possible with the LTS and bisimulation of [3].

8 Implementation and Evaluation

We implemented the LTS and up-to techniques for λ^{imp} in a tool prototype called Hobbit, which we ran on a test-suite of 105 equivalences and 68 inequivalences—3338 and 2263 lines of code for equivalences and inequivalences respectively.

Hobbit is bounded in the total number of function calls it explores per path. We ran Hobbit with a default bound of 6 calls except where a larger bound was found to prove or disprove equivalence—46 examples required a larger bound, and the largest bound used was 348. To illustrate the impact of up-to techniques, we checked all files (pairs of expressions to be checked for equivalence) in five configurations: default (all up-to techniques on), up to separation off, annotations (up to state invariants and re-entry) off, up to re-entry off, and everything off. The tool stops at the first trace that disproves equivalence, after enumerating all traces up to the bound, or after timing out at 150 seconds. Time taken and exit status (equivalent, inequivalent, inconclusive) were recorded for each file; an overview of the experiment can be seen in the following table. All experiments ran on an Ubuntu 18.04 machine with 32GB RAM, Intel Core i7 1.90GHz CPU, with intermediate calls to Z3 4.8.10 to prune invalid internal symbolic branching

and decide symbolic bisimulation conditions. All constraints passed to Z3 are of propositional satisfiability in conjunctive normal form (CNF).

	default	**sep. off**	**annot. off**	**ree. off**	**all off**
eq.	72 | 0 [5.6s]	32 | 0 [1622.9s]	47 | 0 [178.3s]	57 | 0 [177.6s]	3 | 0 [2098.5s]
ineq.	0 | 68 [20.0s]	0 | 66 [312.8s]	0 | 68 [19.6s]	0 | 68 [20.1s]	0 | 65 [515.7s]
$a \mid b$ [c] for a (out of 105) equivalences and b (out of 68) inequivalences reported taking c seconds in total.					

We can observe that Hobbit was sound and bounded-complete for our examples; no false reports and all inequivalences were identified. Up-to techniques also had a significant impact on proving equivalence. With all techniques on, it proved 68.6% of our equivalences; a dramatic improvement over 2.9% proven with none on. The most significant technique was up-to separation—necessary for 55.6% of equivalences proven and reducing time taken by 99.99%—which was useful when functions could be independently explored by the context. Following was annotations—necessary for 34.7% of equivalences and decreasing time by 96.9%—and up-to re-entry—20.8% of files and decreased time by 96.8%. Although the latter two required manual annotation, they enabled equivalences where our language was able to capture the proof conditions. Note that, since turning off invariant annotations also turns off re-entry, only 10 files needed up-to re-entry on top of invariant annotations. In contrast, inequivalences did not benefit as much. This was expected as without up-to techniques Hobbit is still based on bounded model checking, which is theoretically sound and complete for inequivalences, and finds the shortest counterexample traces using breadth-first search. Nonetheless, with up-to techniques turned off, inequivalences were discovered in 515.7s (vs. 20s with techniques on) and three files timed out, due to the techniques reducing the size and branching factor of configurations. This suggests that the reduction in state space is still relevant when searching for counterexamples.

9 Comparison with Existing Tools

There are two main classes of tools for contextual equivalence checking. The first one includes semantics-driven tools that tackle higher-order languages with state like ours. In this class belong game-based tools Hector [12] and Coneqct [24], which can only address carefully crafted fragments of the language, delineated by type restrictions and bounded data types. The most advanced tool in this class is SyTeCi [14], which is based on logical relations and removes a good part of the language restrictions needed in the previous tools. The second class concerns tools that focus on first-order languages, typically variants of C, with main tools including Rêve [9], SymDiff [17] and RVT [11]. These are highly optimised for handling *internal loops*, a problem orthogonal to handling the interactions between higher-order functions and their environment, addressed by Hobbit and related tools. We believe the techniques used in these tools may be useful when adapted to Hobbit, which we leave for future work.

In the higher-order contextual equivalence setting, the most relevant tool to compare with Hobbit is SyTeCi. This is because SyTeCi supersedes previous tools by proving examples with fewer syntactical limitations. We ran the tools on

examples from both SyTeCi's and our own benchmarks—7 and 15 equivalences, and 2 and 7 inequivalences from SyTeCi and Hobbit respectively—with a timeout of 150s and using Z3. Unfortunately, due to differences in parsing and SyTeCi's syntactical restrictions, the input languages were not entirely compatible and only few manually translated programs were chosen.

	SyTeCi	**Hobbit**
SyTeCi eq. examples	3 | 0 | 4 (0.03s)	1 | 0 | 6 (<0.01s)
Hobbit eq. examples	8 | 0 | 7 (0.4s)	15 | 0 | 0 (<0.01s)
SyTeCi ineq. examples	0 | 2 | 0 (0.06s)	0 | 2 | 0 (0.02s)
Hobbit ineq. examples	2 | 3 | 2 (0.52s)	0 | 7 | 0 (0.45s)
$a \mid b \mid c$ (d) for a eq's, b ineq's and c inconclusive's reported taking d sec in total		

We were unable to translate many of our examples because of restrictions in the input syntax supported by SyTeCi. Some of these restrictions were inessential (e.g. absence of tuples) while others were substantial: the tool does not support programs where references are allocated both inside and outside functions (e.g. Ex. 15), or with non-synchroniseable recursive calls. Moreover, SyTeCi relies on Constrained Horn Clause satisfiability which is undecidable. In our testing SyTeCi sometimes timed out on examples; in private correspondence with its creator this was attributed to Z3's ability to solve Constrained Horn Clauses. Finally, SyTeCi was sound for equivalences, but not always for inequivalences as can be seen in the table above; the reason is unclear and may be due to bugs. On the other hand, SyTeCi was able to solve equivalences we are not able to handle; e.g. synchronisable recursive calls and examples with well-bracketing properties.

10 Conclusion

Our experience with Hobbit suggests that our technique provides a significant contribution to verification of contextual equivalence. In the higher-order case, Hobbit does not impose language restrictions as present in other tools. Our tool is able to solve several examples that can not be solved by SyTeCi, which is the most advanced tool in this family. In the first-order case, the problem of contextual equivalence differs significantly as the interactions that a first-order expression can have with its context are limited; e.g. equivalence analyses do not need to consider callbacks or re-entrant calls. Moreover, the distinction between global and local state is only meaningful in higher-order languages where a program phrase can invoke different calls of the same function, each with its own state. Therefore, tools for first-order languages focus on what in our setting are internal transitions and the complexities arising from e.g. unbounded datatypes and recursion, whereas we focus on external interactions with the context.

As for limitations, Hobbit does not handle synchronised internal recursion and well-bracketed state, which SyTeCi can often solve. More generally, Hobbit is not optimised for internal recursion as first-order tools are. In this work we have also disallowed reference types in λ^{imp} to simplify the technical development; location exchange is encoded via function exchange (cf. Ex. 22). We intend to address these limitations in future work and explore applications of Hobbit to real-world examples.

References

1. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: POPL. Association for Computing Machinery (2009)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. Springer Berlin Heidelberg (1999)
3. Biernacki, D., Lenglet, S., Polesiuk, P.: A complete normal-form bisimilarity for state. In: FOSSACS 2019, ETAPS 2019, Prague, Czech Republic. Springer (2019)
4. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: IEEE Symposium on Security and Privacy (2006)
5. Bohr, N., Birkedal, L.: Relational reasoning for recursive types and references. In: Kobayashi, N. (ed.) APLAS. LNCS, vol. 4279, pp. 79–96. Springer (2006)
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Springer Berlin Heidelberg (2004)
7. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded model checking for Java Bytecode. In: TACAS. Springer (2019)
8. Dimovski, A.: Program verification using symbolic game semantics. TCS **560** (2014)
9. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ACM/IEEE ASE '14. ACM (2014)
10. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Informatica **45**(6) (2008)
11. Godlin, B., Strichman, O.: Regression verification. In: DAC. ACM (2009)
12. Hopkins, D., Murawski, A.S., Ong, C.L.: Hector: An equivalence checker for a higher-order fragment of ML. In: CAV. LNCS, Springer (2012)
13. Hur, C.K., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and Kripke logical relations. SIGPLAN Not. (2012)
14. Jaber, G.: SyTeCi: Automating contextual equivalence for higher-order programs with references. Proc. ACM Program. Lang. **4**(POPL) (2020)
15. Jaber, G., Tabareau, N.: Kripke open bisimulation - A marriage of game semantics and operational techniques. In: APLAS. Springer (2015)
16. Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: POPL. ACM (2006)
17. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: CAV. Springer (2012)
18. Laird, J.: A fully abstract trace semantics for general references. In: ICALP, Wroclaw, Poland. LNCS, Springer (2007)
19. Lassen, S.B., Levy, P.B.: Typed normal form bisimulation. In: Computer Science Logic. Springer Berlin Heidelberg (2007)
20. Lin, Y., Tzevelekos, N.: Symbolic execution game semantics. In: FSCD. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
21. Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables. In: POPL. Association for Computing Machinery (1988)
22. Morris, Jr., J.H.: Lambda Calculus Models of Programming Languages. Ph.D. thesis, MIT, Cambridge, MA (1968)
23. 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)
24. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: A contextual equivalence checker for IMJ*. In: ATVA. Springer (2015)
25. Murawski, A.S., Tzevelekos, N.: Nominal game semantics. FTPL **2**(4) (2016)

26. Patterson, D., Ahmed, A.: The next 700 compiler correctness theorems (functional pearl). Proc. ACM Program. Lang. **3**(ICFP) (2019)
27. Pous, D.: Coinduction all the way up. In: ACM/IEEE LICS. ACM (2016)
28. Pous, D., Sangiorgi, D.: Enhancements of the bisimulation proof method. In: Advanced Topics in Bisimulation and Coinduction. CUP (2012)
29. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: LICS. IEEE Computer Society (2007)
30. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: FMICS (2015)

Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time*

Simon Guilloud(✉) and Viktor Kunčak

EPFL IC LARA, Station 14, CH-1015 Lausanne, Switzerland
{Simon.Guilloud,Viktor.Kuncak}@epfl.ch

Abstract. Motivated by proof checking, we consider the problem of efficiently establishing equivalence of propositional formulas by relaxing the completeness requirements while still providing certain guarantees. We present a quasilinear time algorithm to decide the word problem on a natural algebraic structures we call orthocomplemented bisemilattices, a subtheory of Boolean algebra. The starting point for our procedure is a variation of Aho, Hopcroft, Ullman algorithm for isomorphism of trees, which we generalize to directed acyclic graphs. We combine this algorithm with a term rewriting system we introduce to decide equivalence of terms. We prove that our rewriting system is terminating and confluent, implying the existence of a normal form. We then show that our algorithm computes this normal form in log linear (and thus sub-quadratic) time. We provide pseudocode and a minimal working implementation in Scala.

1 Introduction

Reasoning about propositional logic and its extensions is a basis of many verification algorithms [19]. Propositional variables may correspond to, for example, sub-formulas in first-order logic theories of SMT solvers [2,5,26], hypotheses and lemmas inside proof assistants [13,27,32], or abstractions of sets of states. In particular, it is often of interest to establish that *two propositional formulas are equivalent*. The equivalence problem for propositional logic is coNP-complete as a negation of propositional satisfiability [8]. From proof complexity point of view [18] many known proof systems, including (non-extended) resolution [31] and cutting planes [29] have exponential-sized shortest proofs for certain propositional formulas. SAT and SMT solvers rely on DPLL-style algorithms [9,10] and do not have polynomial run-time guarantees on equivalence checking, even if formulas are syntactically close. Proof assistants implement such algorithms as tactics, so they have similar difficulties. A consequence of this is that implemented systems may take a very long time (or fail to acknowledge) that a large formula is equivalent to its minor variant differing in, for example, reordering of internal conjuncts or disjuncts. Similar situations also arise in program verifiers [12,21,30,34,35], where assertions act as lemmas in a proof.

* We acknowledge the financial support of the Swiss National Science Foundation project 200021_197288 "A Foundational Verifier".

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 196–214, 2022.
https://doi.org/10.1007/978-3-030-99527-0_11

It is thus natural to ask for an approximation of the propositional equivalence problem: *can we find an expressive theory supporting many of the algebraic laws of Boolean algebra but for which we can still have a complete and efficient algorithm for formula equivalence?* By efficient, we mean about as fast, up to logarithmic factors, as the simple linear-time syntactic comparison of formula trees.

We can use such an efficient equivalence algorithm to construct more flexible proof systems. Consider any sound proof system for propositional logic and replace the notion of *identical* sub-formulas with our notion of fast equivalence. For example, the axiom schema $p \to (q \to p)$ becomes $p \to (q \to p')$ for all equivalent p and p'. The new system remains sound. It accepts all the previously admissible inference steps, but also some new ones, which makes it more flexible.

L1:	$x \sqcup y = y \sqcup x$	L1':	$x \wedge y = y \wedge x$
L2:	$x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$	L2':	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
L3:	$x \sqcup x = x$	L3':	$x \wedge x = x$
L4:	$x \sqcup 1 = 1$	L4':	$x \wedge 0 = 0$
L5:	$x \sqcup 0 = x$	L5':	$x \wedge 1 = x$
L6:	$\neg\neg x = x$	L6':	same as L6
L7:	$x \sqcup \neg x = 1$	L7':	$x \wedge \neg x = 0$
L8:	$\neg(x \sqcup y) = \neg x \wedge \neg y$	L8':	$\neg(x \wedge y) = \neg x \sqcup \neg y$

Table 1. Laws of an algebraic structures $(S, \wedge, \sqcup, 0, 1, \neg)$. Our algorithm is complete (and log-linear time) for structures that satisfy laws L1-L8 and L1'-L8'. We call these structures orthocomplemented bisemilattices (OCBSL).

L9:	$x \sqcup (x \wedge y) = x$	L9':	$x \wedge (x \sqcup y) = x$
L10:	$x \sqcup (y \wedge z) = (x \sqcup y) \wedge (x \sqcup z)$	L10':	$x \wedge (y \sqcup z) = (x \wedge y) \sqcup (x \wedge z)$

Table 2. Neither the absorption law L9,L9' nor distributivity L10,L10' hold in OCBSL. Without L9,L9', the operations $\wedge$ and $\sqcup$ induce different partial orders. If an OCBSL satisfies L10,L10' then it also satisfies L9,L9' and is precisely a Boolean algebra.

1.1 Problem Statement

This paper proposes to approximate propositional formula equivalence using a new algorithm that solves exactly the word problem for structures we call orthocomplemented bisemilattices (axiomatized in Table 1), in only log-linear time. In general, the word problem for an algebraic theory with signature S and axioms A is the problem of determining, given two terms t_1 and t_2 in the language of S with free variables, whether $t_1 = t_2$ is a consequence of the axioms. Our main interest in the problem is that orthocomplemented bisemilattices (OCBSL) are a generalisation of Boolean algebra. This structure satisfies a weaker set of axioms that omits the distributivity law as well as its weaker variant, the absorption law (Table 2). Hence, this problem is a relaxation "up to distributivity" of the propositional formula equivalence. A positive answer implies formulas are equivalent in all Boolean algebras, hence also in propositional logic.

Definition 1 (Word Problem for Orthocomplemented Bisemilattices). *Consider the signature with two binary operations* $\wedge, \sqcup$*, unary operation* $\neg$ *and constants,* $0, 1$*. The OCBSL-word problem is the problem of determining, given two terms* t_1 *and* t_2 *in this signature, containing free variables, whether* $t_1 = t_2$ *is a consequence (in the sense of first-order logic with equality) of the universally quantified axioms L1-L8,L1'-L8' in Table 1.*

Contribution. We present an $\mathcal{O}(n \log^2(n))$ algorithm for the word problem of orthocomplemented lattices. In the process, we introduce a confluent and terminating rewriting system for OCBSL on terms modulo commutativity. We analyze the algorithm to show its correctness and complexity. We present its executable description and a Scala implementation at https://github.com/epfl-lara/OCBSL.

1.2 Related Work

The word problem on *lattices* has been studied in the past. The structure we consider is, in general, *not* a lattice. Whitman [33] showed decidability of the word problem on free lattices, essentially by showing that the natural order relation on lattices between two words can be decided by an exhaustive search. The word problem on *orthocomplemented lattices* has been solved typically by defining a suitable sequent calculus for the order relation with a cut rule for transitivity [4,17]. Because a cut elimination theorem can be proved similarly to the original from Gentzen [11], the proof space is finite and a proof search procedure can decide validity of the implication in the logic, which translates to the original word problem.

The word problem for free lattices was shown to be in PTIME by Hunt et al. [15] and the word problem for orthocomplemented lattices was shown to be in PTIME by Meinander [25]. Those algorithms essentially rely on similar proof-search methods as the previous ones, but bound the search space. These results make no mention of a specific degree of the polynomial; our analysis suggest that, as described, these algorithms run in $\mathcal{O}(n^4)$. Related techniques of locality have been applied more broadly and also yield polynomial bounds, with the specific exponents depending on local Horn clauses that axiomatize the theory [3,24].

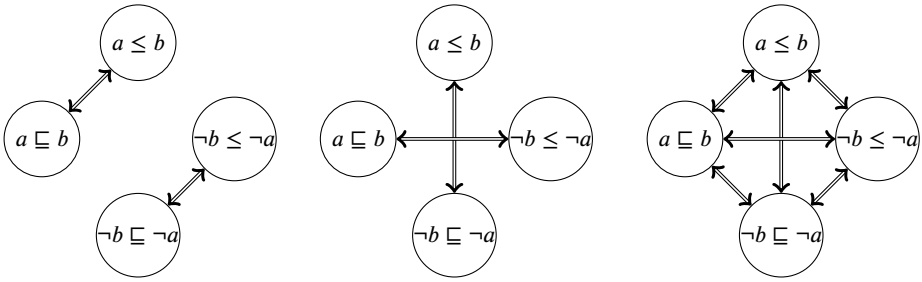
Aside from the use in equivalence checking, the problem is additionally of independent interest because OCBSL are a natural weakening of Boolean Algebra and orthocomplemented lattices. They are dual to complemented lattices in the sense illustrated by Figure 1. A slight weakening of OCBSL, called de Morgan bisemilattice, has been used to simulate electronic circuits [6,22]. OCBSL may be applicable in this scenario as well. Moreover, our algorithm can also be adapted to decide, in log-linear time, the word problem for this weaker theory.

To the best of our knowledge, no solution was presented in the past for the word problem for orthocomplemented bisemilattices (OCBSL). Moreover, we are not aware of previous log-linear algorithms for the related previously studied theories either.

1.3 Overview of the Algorithm

It is common to represent a term, like a Boolean formula, as an abstract syntax tree. In such a tree, a node corresponds to either a function symbol, a constant symbol or a

variable, and the children of a function node represent the arguments of the function. In general, for a symbol function f, trees $f(x, y)$ and $f(y, x)$ are distinct; the children of a node are stored in a specific order. Commutativity of a function symbol f corresponds to the fact that children of a node labelled by f are instead unordered. Our algorithm thus uses as its starting point a variation of the algorithm of Aho, Hopcroft, and Ullman [14] for tree isomorphism, as it corresponds to deciding equality of two terms modulo commutativity. However, the theory we consider contains many more axioms than merely commutativity. Our approach is to find an equivalent set of reduction rules, themselves understood modulo commutativity, that is suitable to compute a normal form of a given formula with respect to those axioms using the ideas of term rewriting [1]. The interest of tree isomorphism in our approach is two-fold: first, it helps to find application cases of our reduction rules, and second, it compares the two terms of our word problem. In the final algorithm, both aspects are realized simultaneously.



(a) Complemented lattice (b) Orthocomplemented bisemilattice(c) Orthocomplemented lattice

Fig. 1. Bisemilattices satisfying absorption or de Morgan laws.

2 Preliminaries

2.1 Lattices and Bisemilattices

To define and situate our problem, we present a collection of algebraic structures satisfying certain subsets of the laws in tables 1 and 2.

A structure $(S, \wedge)$ that is associative (L1), commutative (L2) and idempotent (L3) is a **semilattice**. A semilattice induces a partial order relation on S defined by $a \leq b \iff (a \wedge b) = a$. Indeed, one can verify that $\exists c.(b \wedge c) = a \iff (b \wedge a) = a$, from which transitivity follows. Antisymetry is immediate. In such partially ordered set (poset) S, two elements a and b always have a *greatest lower bound*, or *glb*, $a \wedge b$. Conversely, a poset such that any two elements have a *glb* is always a semilattice. A structure $(S, \wedge, 0, 1)$ that satisfies L1, L2, L3, L4, and L5 is a bounded **upper-semilattice**. Equivalently, 1 is the maximum element and 0 the minimum element in the corresponding poset. Similarly, a structure $(S, \sqcup, 0, 1)$ that satisfies L1' to L5' is a bounded **lower-semilattice**. In that

case, we write the corresponding ordering relation $\sqsupseteq$. Note that it points in the direction opposite to $\leq$, so that 1 is always the "maximum" element and 0 the "minimum" element. A structure $(S, \wedge, \sqcup)$ is a **bisemilattice** if $(S, \wedge)$ is an upper semilattice and $(S, \sqcup)$ a lower semilattice. There are in general no specific laws relating the two semilattices of a bisemilattice. They can be the same semilattice or completely different. If the bisemilattice satisfies the absorption law (L9), then the two semilattices are related in such a way that $a \leq b \iff a \sqsupseteq b$, i.e. the two orders $\leq$ and $\sqsupseteq$ are equal and the structure is called a lattice. A bisemilattice is **consistently bounded** if both semilattices are bounded and if $0_\wedge = 0_\sqcup = 0$ and $1_\wedge = 1_\sqcup = 1$, which will be the case in this paper. A structure $(S, \wedge, \sqcup, \neg, 0, 1)$ that satisfies L1 to L7 and L1' to L7' is called a **complemented bisemilattice**, with complement operation $\neg$. A complemented bisemilattice satisfying de Morgan's Law (L8 and L8') is an **orthocomplemented bisemilattice** and implies $\neg 0 = \neg(\neg 1 \wedge 0) = \neg\neg 1 \sqcup \neg 0 = 1$. A structure satisfying L1-L9 and L1'-L9' is an **orthocomplemented lattice**. Both de Morgan laws (L8, L8') and absorption laws (L9 and L9') relate the two semilattices, in a way summarised in Figure 1. In bisemilattices, orthocomplementation is (merely) equivalent to $a \leq b \iff \neg b \sqsupseteq \neg a$. Indeed, we have:

$$a \leq b \overset{def}{\iff} a \wedge b = a \overset{L8'}{\iff} \neg a \sqcup \neg b = \neg a \overset{def}{\iff} \neg b \sqsupseteq \neg a$$

In the presence of L1-L8,L1'-L8', the law of absorption (L9 and L9') is implied by distributivity. In fact, an orthocomplemented bisemilattice with distributivity is a lattice and even a Boolean algebra. In this sense, we can consider orthocomplemented bisemilattices as "Boolean algebra without distributivity".

2.2 Term Rewriting Systems

We next review basics of term rewriting systems. For a more complete treatment, see [1].

Definition 2. *A **term rewriting system** is a list of rewriting rules of the form $e_l = e_r$ with the meaning that the occurence of e_l in a term t can be replaced by e_r. e_l and e_r can contain free variables. To apply the rule, e_l is unified with a subterm of t, and that subterm is replaced by e_r with the same unifier. If applying a rewriting rule to t_1 yields t_2, we say that t_1 reduces to t_2 and write $t_1 \to t_2$. We denote by $\overset{*}{\to}$ the transitive closure of $\to$ and by $\overset{*}{\leftrightarrow}$ its transitive symmetric closure.*

An axiomatic system such as L1-L9, L1'-L9' induces a term rewriting system, interpreting equalities from left to right. In that case $t_1 \overset{*}{\leftrightarrow} t_2$ coincides with the validity of the equality $t_1 = t_2$ in the theory given by the axioms [1, Theorem 3.1.12].

Definition 3. *A term rewriting system is terminating if there exists no infinite chain of reducing terms $t_1 \to t_2 \to t_3 \to ....$*

Fact 1 *If there is a well-founded order $<$ (or, in particular, a measure m) on terms such that $t_1 \to t_2 \implies t_2 < t_1$ (or, in particular $m(t_2) < m(t_1)$) then the term rewriting system is terminating.*

Definition 4. *A term rewriting system is* ***confluent*** *iff: for all* t_1, t_2, t_3, $t_1 \overset{*}{\rightarrow} t_2 \wedge t_1 \rightarrow^* t_3$ *implies* $\exists t_4 . t_2 \overset{*}{\rightarrow} t_4 \wedge t_3 \overset{*}{\rightarrow} t_4$.

Theorem 1 (Church-Rosser Property). *[1, Chapter 2] A term rewriting system is confluent if and only if* $\forall t_1, t_2 . (t_1 \overset{*}{\leftrightarrow} t_2) \implies (\exists t_3 . t_1 \overset{*}{\rightarrow} t_3 \wedge t_2 \overset{*}{\rightarrow} t_3)$.

A terminating and confluent term rewriting system directly implies decidability of the word problem for the underlying structure, as it makes it possible to compute the normal form of two terms to check if they are equivalent. Note that commutativity is not a terminating rewriting rule, but similar results holds if we consider the set of all terms, as well as rewrite rules, modulo commutativity [1, Chapter 11], [28]. To efficiently manipulate terms modulo commutativity and achieve log-linear time, we will employ an algorithm for comparing trees with unordered children.

3 Directed Acyclic Graph Equivalence

The structure of formulas with commutative nodes correspond to the usual mathematical definition of a labelled rooted tree, i.e. an acyclic graph with one distinguished vertex (root) where there is no order on the children of a node. For this reason, we use as our starting point the algorithm of Hopcroft, Ullman and Aho for tree isomorphism [14, Page 84, Example 3.2], which has also been studied subsequently [7, 23].

To account for structure sharing, we further generalize this representation to singly-rooted, labeled, Directed Acyclic Graphs, which we simply call DAGs. Our DAGs generalize rooted directed trees. Any DAG can be transformed into a rooted tree by duplicating subgraphs corresponding to nodes with multiple parent, as in Figure 2. This transformation in general results in an exponential blowup in the number of nodes. Dually, using DAGs instead of trees can exponentially shrink space needed to represent certain terms.

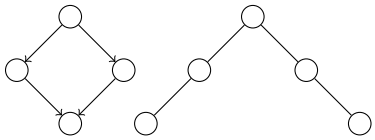


Fig. 2. A DAG and the corresponding Tree

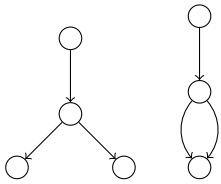


Fig. 3. Two equivalent DAGs with different number of nodes.

Checking for equality between *ordered* trees or DAGs is easy in linear time: we simply recursively check equality between the children of two nodes.

Definition 5. *Two ordered nodes* τ *and* π *with children* $\tau_0, ..., \tau_m$ *and* $\pi_0, ..., \pi_n$ *are equivalent (noted* $\tau \sim \pi$*) iff*

$$label(\tau) = label(\pi),\ m = n \text{ and } \forall i < n, \tau_i \sim \pi_i$$

For unordered trees or DAG, the equivalence checking is less trivial, as the naive algorithm has exponential complexity due to the need to find the adequate permutation.

Definition 6. *Two unordered nodes τ and π with children $\tau_0, ..., \tau_m$ and $\pi_0, ..., \pi_n$ are equivalent (noted $\tau \sim \pi$) iff*

$$label(\tau) = label(\pi),\ m = n \text{ and there exists a permutation } p \text{ s.t. } \forall i < n, \tau_{p(i)} \sim \pi_i$$

For trees, note that this definition of equivalence corresponds exactly to isomorphism. It is known that DAG-isomorphism is GI-complete, so it is conjectured to have complexity greater than PTIME. Fortunately, this does not prevent our solution because our notion of equivalence on DAGs is not the same as isomorphism on DAGs. In particular, two DAGs can be equivalent without having the same number of nodes, i.e. without being isomorphic, as Figure 3 illustrates.

Algorithm 1: Unordered DAG equivalence. The operator ++ is concatenation.

```
   input  : two unordered DAGs τ and π
   output: True if τ and π are equivalent, False else.
 1 codes ←HashMap[(String, List[Int]), Int];
 2 map ←HashMap[Node, Int];
 3 s_τ : List ← ReverseTopologicalOrder(τ);
 4 s_π : List ← ReverseTopologicalOrder(π);
 5 for (n:Node in s_τ++s_π) do
 6     l_n ← [map(c) for c in children(n)];
 7     r_n ← (label(n), sort(l_n));
 8     if codes contains r then
 9         map(n) ← codes(r_n);
10     else
11         codes(r_n) ← codes.size;
12         map(n) ← codes(r_n);
13     end
14 end
15 return map(τ) == map(π)
```

Algorithm 1 is the generalization of Hopcroft, Ullman and Aho's algorithm. It decides in log-linear time if two labelled (unordered) DAGs are equivalent according to definition 5. The algorithm generalizes straightforwardly to DAGs with a mix of ordered and unordered nodes: if a node is ordered, we skip the sorting operation in line 7.

The algorithm works bottom to top. We first sort the DAG in reverse topological order using, for example, Kahn's algorithm [16]. This way, we explore the DAG starting from a leaf and finishing with the root. It is guaranteed that when we treat a node, all its children have already been treated.

The algorithm recursively assigns codes to the nodes of both DAGs recursively. In the unlabelled case:

- The first node, necessarily a leaf, is assigned the integer 0
- The second node gets assigned 0 if it is a leaf or 1 if it is a parent of the first node
- For any node, the algorithm makes a list of the integer assigned to that node's children and sort it (if the node is commutative). We call this the signature of the node. Then it checks if that list has already been seen. If yes, it assigns to the node the number that has been given to other nodes with the same signature. Otherwise, it assigns a new integer to that node and its signature.

Lemma 1 (Algorithm 1 Correctness). *The codes assigned to any two nodes n and m of $s_\tau{+}{+}s_\pi$ are equal if and only if $n \sim m$.*

Proof. Let n and m denote any two DAG nodes. By induction on the height of n:

- In the case where n is a leaf, we have $r_n = (label(n), Nil)$. Note that for any node n, $map(n) = codes(r_n)$. Since every time the map $codes$ is updated, it is with a completely new number, $codes(r_n) = codes(r_m)$ if and only if $r_n = r_m$, i.e. iff $label(m) = label(n)$ and m has no children (like n).
- In the case where n has children n_i, again $codes(r_n) = codes(r_m)$ if and only if $r_m = r_n$, which is equivalent to ($label(m) = label(n)$ and $sort(l_m) = sort(l_n)$. This means this means there is a permutation of children of n such that $\forall i, codes(n_{p(i)}) = codes(m_i)$. By induction hypothesis, this is equivalent to $\forall i, n_{p(i)} \sim m_i$. Hence we find that $map(n) = map(m)$ if and only if both:
  1. Their labels are equal
  2. There exist a permutation p s.t. $n_{p(i)} \sim m_i$

  i.e n and m have the same code if and only if $n \sim m$.

Corollary 1. *The algorithm returns True if and only if $\tau \sim \pi$.*

Time Complexity. Using Kahn's algorithm, sorting τ and π is done in linear time. Then the loop touches every node a single time. Inside the loop, the first line takes linear time with respect to the number of children of the node and the second line takes log-linear time with respect to the number of children. Since we use HashMaps, the last instructions take effectively constant time (because hash code is computed from the address of the node and not its content).

So for general DAG, the algorithm runs in time at most log-quadratic in the number of nodes. Note however that for DAGs with bounded number of children per node as well as for DAGs with bounded number of parents per nodes, the algorithm is log-linear. In fact, the algorithm is log-linear with respect to the total number of edges in the graph. For this reason, the algorithm is still only log-linear in input size. It also follows that the algorithm is always at most log-linear with respect to the tree or formula underlying the DAG, which may be much larger than the DAG itself. Moreover, there exists cases where the algorithm is log-linear in the number of nodes, but the underlying tree is exponentially larger. The full binary symmetric graph is such an example.

4 Word Problem on Orthocomplemented Bisemilattices

We will use the previous algorithm for DAG equivalence applied to a formula in the language of bisemilattices $(S, \wedge, \sqcup)$ to account for commutativity (axioms L1, L1'), but we need to combine it with the remaining axioms. From now on we work with axioms L1-L8, L1'-L8' in Table 1. The plan is to express those axioms as reduction rules. Of rules L2-L8 and L2'-L8', all but L8 and L8' reduce the size of the term when applied from left to right, and hence seem suitable as rewrite rules.

It may seem that the simplest way to deal with de Morgan law is to use it (along with double negation elimination) to transform all terms into negation normal form. It happens, however, that doing this causes troubles when trying to detect application cases of rule L7 (complementation). Indeed, consider the following term:

$$f = (a \wedge b) \sqcup \neg(a \wedge b)$$

Using complementation it clearly reduces to 1, but pushing into negation-normal form, it would first be transformed to $(a \wedge b) \sqcup (\neg a \vee \neg b)$. To detect that these two disjuncts are actually opposite requires to recursivly verify that $\neg(a \wedge b) = (\neg a \vee \neg b)$.

It is actually simpler to apply de Morgan law the following way:

$$x \wedge y = \neg(\neg x \sqcup \neg y)$$

Instead of removing negations from the formula, we remove one of the binary semilattice operators. (Which one we keep is arbitrary; we chose to keep $\sqcup$.) Now, when we look if rule L7 can be applied to a disjunction node (i.e. two children y and z such that $y = \neg z$), there are two cases: if x is not itself a negation, i.e. it starts with $\sqcup$, we compute $\neg x$ code from the code of x in constant time. If $x = \neg x'$ then $\neg x \sim x'$ so the code of $\neg x$ is simply the code of x', in constant time as well. Hence we obtain the code of all children and their negation and we can sort those codes to look for collisions, all of it in time linear in the number of children.

We now restate the axioms L1-L8 ,L1'-L8' in this updated language in Table 3.

$A1 : \bigsqcup(..., x_i, x_j, ...) = \bigsqcup(..., x_j, x_i, ...)$	$A1' : \neg\bigsqcup(\neg x, \neg y) = \neg\bigsqcup(\neg y, \neg x)$
$A2 : \bigsqcup(\vec{x}, \bigsqcup(\vec{y})) = \bigsqcup(\vec{x}, \vec{y})$ $\bigsqcup(x) = x$	$A2' : \neg\bigsqcup(\neg\vec{x}, \neg\neg\bigsqcup(\neg\vec{y})) = \neg\bigsqcup(\neg\vec{x}, \neg\vec{y})$
$A3 : \bigsqcup(x, x, \vec{y}) = \bigsqcup(x, \vec{y})$	$A3' : \neg\bigsqcup(\neg x, \neg x, \neg\vec{y}) = \neg\bigsqcup(\neg x, \neg\vec{y})$
$A4 : \bigsqcup(1, \vec{x}) = 1$	$A4' : \neg\bigsqcup(\neg 0, \neg\vec{y}) = 0$
$A5 : \bigsqcup(0, \vec{x}) = \bigsqcup(\vec{x})$	$A5' : \neg\bigsqcup(\neg 1, \neg\vec{x}) = \neg\bigsqcup(\neg\vec{x})$
$A6 : \neg\neg x = x$	
$A7 : \bigsqcup(x, \neg x, \vec{y}) = 1$	$A7' : \neg\bigsqcup(\neg x, \neg\neg x, \neg\vec{y}) = 0$
$A8 : \neg\bigsqcup(x_1, ...x_i) = \neg\bigsqcup(\neg\neg x_1, ...\neg\neg x_i)$	$A8' : \neg\neg\bigsqcup(\neg x_1, ...\neg x_i) = \bigsqcup(\neg x_1, ...\neg x_i)$

Table 3. Laws of algebraic structures $(S, \sqcup, 0, 1, \neg)$, equivalent to L1-L8, L1-L8' under de Morgan transformation.

It is straightforward and not surprising that axiom A8 as well as A1'-A8' all follow from axioms A1-A7, so A1-A7 are actually complete for our theory.

4.1 Confluence of the Rewriting System

In our equivalence algorithm, A1 is taken care of by the arbitrary but consistent ordering of the nodes. Axioms A2-A7 form a term rewriting system. Since all those rules reduce the size of the term, the system is terminating in a number of steps linear in the size of the term. We will next show that it is confluent. We will thus obtain the existence of a normal form for every term, and will finally show how our algorithm computes that normal form.

Definition 7. *Consider a pair of reduction rules $l_0 \to r_0$ and $l_1 \to r_1$ with disjoint sets of free variables such that $l_0 = D[s]$, s is not a variable and σ is the most general unifier of $\sigma s = \sigma l_1$. Then $(\sigma r_0, (\sigma D)[\sigma r_2])$ is called a* critical pair.

Informally, a critical pair is a most general pair of term (with respect to unification) (t_1, t_2) such that for some t_0, $t_0 \to t_1$ and $t_0 \to t_2$ via two "overlapping" rules. They are found by matching the left-hand side of a rule with a non-variable subterm of the same or another rule.

Example 1 (Critical Pairs).

1. Matching left-hand side of A6 with the subterm $\neg x$ of rule A7, we obtain the pair

$$(1, \bigsqcup(\neg x, x, \vec{y}))$$

   which arises from reducing the term $t_0 = \bigsqcup(\neg x, \neg\neg x, \vec{y})$ in two different ways.
2. Matching left-hand sides of A2 and A7 gives

$$(\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})), 1)$$

   which arise from reducing $\bigsqcup(\vec{x}, \bigsqcup(\vec{y}), \neg \bigsqcup(\vec{y}))$ using A2 or A7.
3. Matching left-hand sides of A5 and A7 gives

$$(\neg 0, 1)$$

   which arise from reducing $0 \sqcup \neg 0$ in two different ways.

Proposition 1 ([1, Chapter 6]). *A terminating term rewriting system is confluent if and only if all critical pairs (t_1, t_2) are joinable i.e. $\exists t_3.\ t_1 \overset{*}{\to} t_3 \wedge t_2 \overset{*}{\to} t_3$.*

In the first of the previous examples, the pair is clearly joinable by commutativity and a single application of rule A7 itself. The second example is more interesting. Observe that $\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = 1$ is a consequence of our axiom, but the left part cannot be reduced to 1 in general in our system. To solve this problem we need to add the rule A9: $\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = 1$. Similarly, the third example forces us to add A10: $\neg 0 = 1$ to our set of rules. From A10 and A6 we then find the expected critical pair A11: $\neg 1 = 0$.

$$
\begin{aligned}
A1 &: \bigsqcup(..., x_i, x_j, ...) = \bigsqcup(..., x_j, x_i, ...) \\
A2 &: \bigsqcup(\vec{x}, \bigsqcup(\vec{y})) = \bigsqcup(\vec{x}, \vec{y}) \\
&\quad \bigsqcup(x) = x \\
A3 &: \bigsqcup(x, x, \vec{y}) = \bigsqcup(x, \vec{y}) \\
A4 &: \bigsqcup(1, \vec{x}) = 1 \\
A5 &: \bigsqcup(0, \vec{x}) = \bigsqcup(\vec{x}) \\
A6 &: \neg\neg x = x \\
A7 &: \bigsqcup(x, \neg x, \vec{y}) = 1 \\
A9 &: \bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = 1 \\
A10 &: \neg 0 = 1 \\
A11 &: \neg 1 = 0
\end{aligned}
$$

Table 4. Terminating and confluent set of rewrite rules equivalent to L1-L8, L1'-L8'

4.2 Complete Terminating Confluent Rewrite System

The analysis of all possible pairs of rules to find all critical pairs is straightforward. It turns out that the A9, A10 and A11 are the only rules we need to add to our system to obtain confluence. We have checked the complete list of critical pairs for rules A2-A11 (we omit the details due to lack of space). All those pairs are joinable, i.e. reduce to the same term, which implies, by Proposition 1, that the system is confluent. Table 4 shows the complete set of reduction rules (as well as commutativity).

Since the system A2-A11 considered over the language $(S, \bigsqcup, \neg, 0, 1)$ modulo commutativity of $\bigsqcup$ is terminating and confluent, it implies the existence of a normal form reduction. For any term t, we note its normal form $t\downarrow$. In particular, for any two terms t_1 and t_2, we have $t_1 = t_2$ in our theory iff $t_1 \overset{*}{\leftrightarrow} t_2$ iff $t_1\downarrow$ and $t_2\downarrow$ are equivalent terms modulo commutativity. We finally reach our conclusion: an algorithm that computes the normal form (modulo commutativity) of any term gives a decision procedure for the word problem for orthocomplemented bisemilattices.

5 Algorithm and Complexity

The rewriting system readily gives us a quadratic algorithm. Indeed, using our base algorithm for DAG equivalence, we can check, in linear time, for application cases of any one of rewriting rules A2-A11 of Table 4 modulo commutativity. Since a term can only be reduced up to n times, the total time spent before finding the normal form of a term is at most quadratic. It is however possible to find the normal form of a term in a single pass of our equivalence algorithm, resulting in a more efficient algorithm.

5.1 Combining Rewrite Rules and Tree Isomorphism

We give an overview on how to combine rules A2-A7, A9, A10, A11 within the tree isomorphism algorithm, which we present using Scala-like [1] pseudo code in Figure 7.

[1] https://www.scala-lang.org/

For conciseness, we omit the dynamic programming optimizations allowed by structure sharing in DAGs (which would store the normal form and additionally check if a node was already processed.) For each rule, we indicate the most relevant lines of the algorithm in Figure 7.

A2 (Associativity, Lines 10, 20, 32, 42) When analysing a $\bigsqcup$ node, after the recursive call, find all children that are $\bigsqcup$ themselves and replace them by their own children. This is simple enough to implement but there is actually a caveat with this in term of complexity. We will come back to it in section 5.

A3 (Idempotence, Lines 8, 31, 35) This corresponds to the fact that we eliminate duplicate children in disjunctions. When reaching a $\bigsqcup$ node, after having sorted the code of its children, remove all duplicates before computing its own code.

A4, A5 (Bounds, Lines 8, 31, 35, 11, 36) To account for those axioms, we reserve a special code for the nodes 1 and 0. For A4, when we reach some $\bigsqcup$ node, if it has 1 as one of its children, we accordingly replace the whole node by 1. For A5, we just remove nodes with the same codes as 0 from the parent node before computing its own code.

A6 (Involution, Lines 17, 22) When reaching a negation node, if its child is itself a negation node, replace the parent node by its grandchildren before assigning it a code.

A7 (Complement, Lines 11, 36) As explained earlier, our representation of nodes let us do the following to detect cases of A7: First remember that we already applied double negation elimination, so that two "opposite" nodes cannot both start with a negation. Then we can simply separate the children between negated and non-negated (after the recursive call), sort them using their assigned code and look for collisions.

A9 (Also Complement, Lines 11, 36) This rule is slightly more tricky to apply. When analysing a $\bigsqcup$ node x, after computing the code of all children of x, find all children of the form $\neg \bigsqcup$. For every such node, take the set of its own children and verify if it is a subset of the set of all children of x. If yes, then rule A9 applies. Said otherwise, we look for collisions between grandchildren (through a negation) and children of every $\bigsqcup$ node.

A10, A11 (Identities, Lines 17, 26) These rules are simple. In a $\neg$ node, if its child has the same code as 0 (resp 1), assign code 1 (resp 0) to the negated node.

5.2 Case of Quadratic Runtime for the Basic Algorithm

All the rules we introduced in the previous section into Algorithm 1 take time (log)linear in the number of children of a node to apply, which is not more than the time we spent in the DAG/tree isomorphism algorithm. For A3, checking for duplicates is done in linear time in an ordered data structure. A4 and A5 (Bounds) consist in searching for specific values, which take logarithmic time in the size of the list. A6 (Involution) takes constant time. A7 (Complement) is detected by finding a collision between two separate ordered

lists, also easily done in (log) linear time. A9 (Also complement) consists in verifying if grandchildren of a node are also children, and since children are sorted this takes log-linear time in the number of grandchildren. Since a node is the grandchild of only one other node, the same computation as in the original algorithm holds. A10 and A11 take constant time. Hence, the total time complexity is $\mathcal{O}(n \log(n))$, as in the algorithm for tree isomorphism.

As stated in Section 3 regarding the algorithm for DAG equivalence whose complexity we aim to preserve, the time complexity analysis crucially relies on the fact that in a tree, a node is never the child (or grandchild) of more than one node during the execution. However, this is generally not true in the presence of associativity. Indeed consider the term represented in Figure 4. The 5th $\bigsqcup$ has 2 children, but after applying

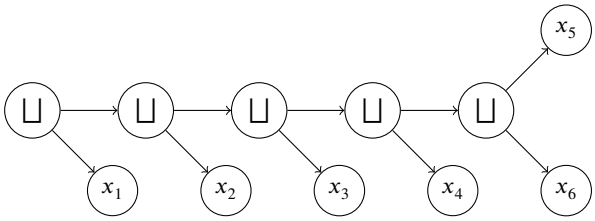


Fig. 4. A term with quadratic runtime

A2, the 4th has 3 children, the 3rd has 4 children and so on. On the generalization of such an example, since an x_i is the child of all higher $\bigsqcup$, our key property does not hold and the algorithm runtime would be quadratic. Of course, such a simple counterexample is easily solved by applying a leading pass of associativity reduction before actually running the whole algorithm. It turns out however that it is not sufficient, since cases of associativity can appear after the application of the other A-rules.

In fact, there is only one rule that can creates case of rule A2, and this rule is A6 (Involution). The remaining rules whose right-hand side can start with a $\bigsqcup$ have their left-hand side already starting with $\bigsqcup$. It may seem simple enough to also apply double negation elimination in a leading pass, but unfortunately, cases of A6 can also be created from other rules. It is easy to see, for similar reasons, that only the application of A2b ($\bigsqcup(x) = x$) can create such cases. And unfortunately, such cases of A2b can arise from rules A3 and A5 which can only be detected using the full algorithm. To summarize, the typical problematic case is depicted in Figure 5. This term is clearly equivalent to $\bigsqcup(x_1, x_2, x_3, x_4)$, but to detect it we must first find that z_1 and z_2 are equivalent to 0, so we cannot simply solve it with an early pass.

5.3 Final Log-Linear Time Algorithm

Fortunately, we can solve this problem at a logarithmic-only price. Observe that if we are able to detect early nodes which would cancel to 0, the problem would not exist: When analysing a node, we would first call the algorithm on all subnodes equivalent to

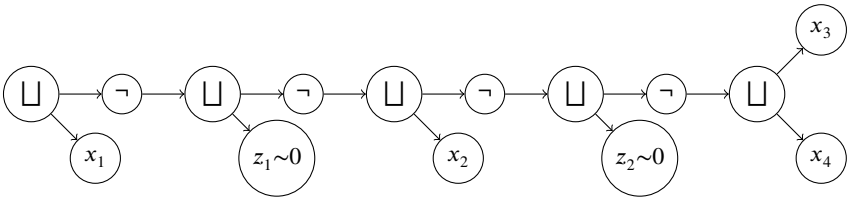


Fig. 5. A non-trivial term with quadratic runtime

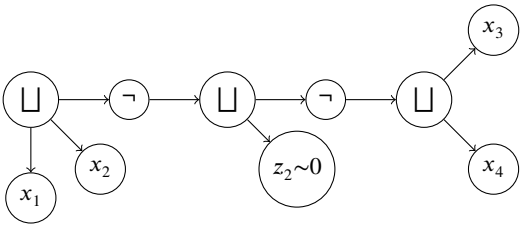


Fig. 6. the term of Figure 5 during the algorithm's execution

0, remove them and then when there is a single children left, remove the trivial disjunct, the double negation and the successive disjunction (as in Figure 5) before doing the recursive call on the unique nontrivial child. However, we of course cannot know in advance which child will be equivalent to 0.

Moreover note (still using Figure 5) that if the z-child is as large as the non-trivial node, then even if we do the "useless" work, we at least obtain that the size a tree is divided by two, and hence the potential depth of the tree as well. By standard complexity analysis, the time penalty would only be a logarithmic factor.

The previous analysis suggests the following solution, reflected in Figure 7 lines 28-29. When analysing a node, make recursive calls on children in order of their size, starting with the smallest up to the second biggest. If any of those children are non-zero, proceed as normal. If all (but possibly the last) children are equivalent to zero, then replace the current node by its biggest (and at this point non-analyzed) child, i.e. apply second half of rule A2 (associativity). If applicable, apply double negation elimination and associativity as well before continuing the recursive call.

We illustrate this on the example of Figure 5. Consider the algorithm when reaching the second $\bigsqcup$ node. There are two cases:

1. Suppose z_1 is a smaller tree than the non-trivial child. In this case the algorithm will compute a code for z_1, find that it is 0 and delete it. Then the non trivial node is a single child so the whole disjunction is removed. Hence, the double negation can be removed and the two consecutive disjunction of x_1 and x_2 merged, obtaining the term illustrated in Figure 6. In particular we did not compute a code for the two deleted $\bigsqcup$ nodes, which is exactly what we wanted for our initial analysis.
2. Suppose z_1 is larger tree than the non-trivial child. In this case, we would first recursively compute the code of the non-trivial child and then detect that $z_1 \sim 0$. We

indeed computed the code of the disjunction that contains x_2 when it was unnecessary since we apply associativity anyway. This "useless" work consists in sorting and applying axioms to the true children of the node (in this case x_2, x_3 and x_4) and takes time quasilinear in the number of such children. In particular, it is bounded by the size of the subtree itself and we know it is the smallest of the two.

Analogous situation can arise from the use of rule A3 (idempotence), but here trivially the two subtrees must have the same number of (real) subnodes, so that the same reasoning holds.

Denote by $|n|$ the size of a node, i.e. the number of descendants of n. We compute the penalty of useless work we incur by computing children of a node n in the wrong order, i.e. by computing a non-0 child n_w when all other are 0. n_w cannot be the largest child of n for otherwise we would have found that all other children are 0 before needing to compute n_w. Hence $|n_w| \leq |n|/2$. It follows that the total amount of useless work is bounded by $\log(|n|) \cdot W(n)$, where

$$W(n) \leq |n|/2 + \sum_i W(n_i) \quad \text{for} \quad \sum_i |n_i| < |n|.$$

It is clear that $W(n)$ is maximized when n has exactly two children of equal size:

$$W(n) \leq |n|/2 + 2 \cdot W(n/2)$$

By observing that we can divide n by 2 only $\log(n)$ times,

$$W(n) \leq \sum_{m=1}^{\log(n)} 2^m \cdot |n|/2^m$$

so we obtain $W(n) = \mathcal{O}(|n| \log(|n|))$ and hence the total runtime is $\mathcal{O}(n(\log n)^2)$.

6 Conclusion

We have described a decision procedure with log-linear time complexity for the word problem on orthocomplemented bisemilattices. This algorithm can also be simplified to apply to weaker theories. Dually, we believe it can be generalized to decide some stronger theories (still weaker than Boolean algebras) efficiently. While the word problem for orthocomplemented *lattices* was known to be in PTIME [15] and as such the membership of orthocomplemented *bisemilattices* in PTIME may not come as a surprise, this is, to the best of our knowledge, the first time that this result has been explicitly stated, and the first time that an algorithm with such low log-linear complexity was proposed for this or a related problem. The algorithm has not only low complexity but, according to our experience, is easy to implement. It can be used as an approximation for Boolean algebra equivalence, and we plan to use it as the basis of a kernel for a proof assistant. We also envision possible uses of the algorithm in SMT and SAT solvers. The algorithm is able to detect many natural and non-trivial cases of equivalence even on formulas that may be too large for existing solvers to deal with, so it may also complement an existing repertoire of subroutines used in more complex reasoning tasks. For a minimal working implementation in Scala closely following Figure 7, see https://github.com/epfl-lara/OCBSL.

```
1  def equivalentTrees(tau: Term, pi: Term): Boolean =
2      val codesSig: HashMap[(String, List[Int]), Int] = Empty
3      codesSig.update(("zero", Nil), 0); codesSig.update(("one", Nil), 1)
4      val codesNodes: HashMap[Term, Int] = Empty
5      def updateCodes(sig: (String, List[Int]), n: Node): Unit = ... // codesSig, codesNodes
6      def bool2const(b:Boolean): String = if b then "one" else "zero"
7      def rootCode(n: Term): Int =
8          val L = pDisj(n, Nil).map(codesNodes).sorted.filter(_≠ 0).distinct
9          if L.isEmpty then ("zero", Nil), n)
10         else if L.length == 1 then codesNodes.update(n, L.head)
11         else if L.contains(1) or checkForContradiction(L) then updateCodes(("one", Nil), n)
12         else updateCodes(("or", L), n)
13         codesNodes(n)
14     def pDisj(n:Node, acc:List[Node]): List[Node] = n match
15         case Variable(id) ⇒ updateCodes((id.toString, Nil), n); return n :: acc
16         case Literal(b) ⇒ updateCodes((bool2const(b), Nil), n); return n :: acc
17         case Negation(child) ⇒ pNeg(child, n, acc)
18         case Disjunction(children) ⇒ children.foldleft(acc)(pDisj)
19     def pNeg(n:Node, parent:Node, acc:List[Node]): List[Node] = n match // under negation
20         case Negation(child) ⇒ pDisj(child, acc)
21         case Variable(id) ⇒ updateCodes((id.toString, Nil), n)
22                               updateCodes(("neg", List(codesNodes(n))), parent)
23                               List(parent)::acc
24         case Literal(b) ⇒ updateCodes((bool2const(b), Nil), n)
25                           updateCodes((bool2const(!b), Nil), parent)
26                           List(parent)::acc
27         case Disjunction(children) ⇒
28             val r0 = orderBySize(children)
29             val r1 = r0.tail.foldLeft(Nil)(pDisj)
30             val r2 = r1.map(codesNodes).sorted.filter(_≠ 0).distinct
31             if isEmpty(r2) then pNeg(r0.head, parent, acc)
32             else val s1 = pDisj(r0.head, r1)
33                 val s2 = s1 zip (s1 map codesNodes)
34                 val s3 = s2.sorted.filter(_≠ 0).distinct // all wrt. 2nd element
35                 if s3.contains(1) or checkForContradiction(s3)
36                 then updateCodes(("one", Nil), n); updateCodes(("zero", Nil), parent)
37                       List(parent)::acc
38                 else if isEmpty(s3) then updateCodes(("zero", Nil), n)
39                                          updateCodes(("one", Nil), parent)
40                                          List(parent)::acc
41                 else if s3.length == 1 then pNeg(s3.head._1, parent, acc)
42                 else updateCodes(("or", s3 map (_._2)), n)
43                     updateCodes(("neg", List(n)), parent)
44                     List(parent)::acc
45     return rootCode(tau) == rootCode(pi)
```

Fig. 7. Final algorithm. distinctBy runs in log-linear time. checkForContradiction detects application cases of A7 and A9 (Complement). Maintenance of size field used by orderBySize elided.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998). https://doi.org/10.1017/CBO9781139172752
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 171–177. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Basin, D.A., Ganzinger, H.: Automated complexity analysis based on ordered resolution. J. ACM **48**(1), 70–109 (2001). https://doi.org/10.1145/363647.363681
4. Bruns, G.: Free Ortholattices. Canadian Journal of Mathematics **28**(5), 977–985 (Oct 1976). https://doi.org/10.4153/CJM-1976-095-6
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, vol. 6015, pp. 150–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_12
6. Brzozowski, J.: De Morgan bisemilattices. In: Proceedings 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2000). pp. 173–178 (May 2000). https://doi.org/10.1109/ISMVL.2000.848616
7. Buss, S.R.: Alogtime algorithms for tree isomorphism, comparison, and canonization. In: Gottlob, G., Leitsch, A., Mundici, D. (eds.) Computational Logic and Proof Theory. pp. 18–33. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
8. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. p. 151–158. STOC '71, Association for Computing Machinery, New York, NY, USA (1971). https://doi.org/10.1145/800157.805047
9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (Jul 1962). https://doi.org/10.1145/368273.368557
10. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Alur, R., Peled, D.A. (eds.) Computer Aided Verification, vol. 3114, pp. 175–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_14
11. Gentzen, G.: Untersuchungen über das logische schließen. I. Mathematische Zeitschrift **39**, 176–210 (1935)
12. Hamza, J., Voirol, N., Kunčak, V.: System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang **3** (November 2019). https://doi.org/https://doi.org/10.1145/3360592
13. Harrison, J.: HOL Light: An Overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, vol. 5674, pp. 60–66. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_4
14. Hopcroft, J., UIIman, J., Aho, A.: The Design And Analysis Of Computer Algorithms. Addison-Wesley (1974)
15. Hunt, H. B., I., Rosenkrantz, D.J., Bloniarz, P.A.: On the Computational Complexity of Algebra on Lattices. SIAM Journal on Computing **16**(1), 129–148 (Feb 1987). https://doi.org/10.1137/0216011
16. Kahn, A.B.: Topological sorting of large networks. Communications of the ACM **5**(11), 558–562 (Nov 1962). https://doi.org/10.1145/368996.369025

17. Kalmbach, G.: Orthomodular Lattices. Academic Press Inc, London ; New York (Mar 1983)
18. Krajíček, J.: Proof Complexity. Encyclopedia of Mathematics and Its Appplications, Vol.170, Cambridge University Press (2019)
19. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Springer (2016)
20. Kuncak, V.: Modular Data Structure Verification. Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (February 2007), http://hdl.handle.net/1721.1/38533
21. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments. pp. 170–190. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54108-7_9
22. Lewis, D.W.: Hazard detection by a quinary simulation of logic devices with bounded propagation delays. In: Proceedings of the 9th Design Automation Workshop. pp. 157–164. DAC '72, Association for Computing Machinery, New York, NY, USA (Jun 1972). https://doi.org/10.1145/800153.804941
23. Lindell, S.: A logspace algorithm for tree canonization (extended abstract). In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing. p. 400–404. STOC '92, Association for Computing Machinery, New York, NY, USA (1992). https://doi.org/10.1145/129712.129750
24. McAllester, D.A.: Automatic recognition of tractability in inference relations. Journal of the ACM **40**(2), 284–303 (1993). https://doi.org/10.1145/151261.151265
25. Meinander, A.: A solution of the uniform word problem for ortholattices. Mathematical Structures in Computer Science **20**(4), 625–638 (Aug 2010). https://doi.org/10.1017/S0960129510000125
26. Merz, S., Vanzetto, H.: Automatic Verification of TLA+ Proof Obligations with SMT Solvers. In: Bjørner, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 289–303. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_23
27. Naumowicz, A., Korniłowicz, A.: A brief overview of mizar. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. pp. 67–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_5
28. Peterson, G.E., Stickel, M.E.: Complete sets of reductions for some equational theories. J. ACM **28**(2), 233–264 (Apr 1981). https://doi.org/10.1145/322248.322251
29. Pudlák, P.: The Lengths of Proofs. In: Studies in Logic and the Foundations of Mathematics, vol. 137, pp. 547–637. Elsevier (1998). https://doi.org/10.1016/S0049-237X(98)80023-2
30. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: Autoproof: Auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 566–580. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_53
31. Urquhart, A.: Hard examples for resolution. J. ACM **34**(1), 209–219 (Jan 1987). https://doi.org/10.1145/7531.8928
32. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: Theorem Proving in Higher Order Logics. pp. 33–38. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_7
33. Whitman, P.M.: Free Lattices. Annals of Mathematics **42**(1), 325–330 (1941). https://doi.org/10.2307/1969001
34. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI) (2008). https://doi.org/10.1145/1375581.1375624, see also [20]
35. Zee, K., Kuncak, V., Rinard, M.: An integrated proof language for imperative programs. In: ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI) (2009). https://doi.org/10.1145/1543135.1542514

Monitoring and Analysis

A Theoretical Analysis of Random Regression Test Prioritization

Pu Yi[1], Hao Wang[1], Tao Xie[1](✉), Darko Marinov[2], and Wing Lam[3]

[1] Peking University, Beijing, China
lukeyi@pku.edu.cn, tony.wanghao@stu.pku.edu.cn, taoxie@pku.edu.cn
[2] University of Illinois Urbana-Champaign, Urbana, IL, USA
marinov@illinois.edu
[3] George Mason University, Fairfax, VA, USA
winglam@gmu.edu

Abstract. Regression testing is an important activity to check software changes by running the tests in a test suite to inform the developers whether the changes lead to test failures. Regression test prioritization (RTP) aims to inform the developers faster by ordering the test suite so that tests likely to fail are run earlier. Many RTP techniques have been proposed and are often compared with the random RTP baseline by sampling some of the $n!$ different test-suite orders for a test suite with n tests. However, there is no theoretical analysis of random RTP. We present such an analysis, deriving probability mass functions and expected values for metrics and scenarios commonly used in RTP research. Using our analysis, we revisit some of the most highly cited RTP papers and find that some presented results may be due to insufficient sampling. Future RTP research can leverage our analysis and need not use random sampling but can use our simple formulas or algorithms to more precisely compare with random RTP.

Keywords: Regression Test Prioritization · Random · Analysis

1 Introduction

Software developers commonly check their code by running tests. *Regression testing* [48] runs tests after code changes, to check whether the changes break the existing functionality. A test that passes before the changes but fails after indicates that the changes should be debugged (unless the test is flaky [25]). Finding test failures faster enables the developers to start debugging earlier.

A popular regression testing approach is *regression test prioritization (RTP)* [12, 19, 21, 23, 38, 39, 48], which runs the tests from a test suite in an order that aims to find test failures sooner. For example, Google [14] and Microsoft [42] report on using RTP in industry. More formally, a test suite T is a set (unordered) of tests, and RTP techniques produce a test-suite *order*—a permutation of the tests in the test suite—in which to run the tests. Various RTP techniques have been proposed in the literature since the seminal papers from 20+ years ago [12, 36, 38, 47] that have garnered thousands of citations.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 217–235, 2022.
https://doi.org/10.1007/978-3-030-99527-0_12

RTP techniques are often compared with random RTP. Our inspection [44] of the 100 most cited papers on RTP shows that 56 papers use random RTP as a comparison baseline. Although random RTP often performs worse than advanced techniques, recent papers still use random RTP, because it has a small overhead and may perform well in certain scenarios. We additionally check papers published in the latest testing conferences (ICST and ISSTA 2020/2021) and find that 50% (2/4) of the RTP papers [6,15,30,34] use random RTP. While random RTP has been used as a baseline for 20+ years, *all* evaluations have been empirical, performed by randomly sampling some of the $n!$ orders for a test suite with n tests. The selected sample size varies (20, 50, 100, 200, 1000), with no clear correlation with n; some papers do not even report the sample size [44]. However, no prior work has presented a theoretical analysis of random RTP.

Before we summarize our analysis, we describe some metrics and scenarios most commonly used in RTP research. We first introduce some terms: *failure* is simply a failing test, *fault* is the root cause (bug in the code) for the failure, and we say that a failure *detects* a fault if the failure is caused by the fault [36]. In general, many failures may detect the same fault, and one failure may detect many faults. We capture the relationship between failures and faults by a *failure-to-fault matrix*. To compare RTP techniques, researchers quantify how fast (test-suite) orders find all *faults* (not failures because having many failures that detect the same fault is not as valuable as having a few failures that detect many faults).

RTP evaluations involve three aspects: RTP metric, failure-to-fault matrix, and allowed orders. The most widely used metric is Average Percentage of Faults Detected (APFD) [38], denoted as α for short. Another popular metric is Cost-Cognizant APFD (APFD_c) [11], denoted as γ for short. Section 2 formally defines these metrics based on the failure-to-fault matrix; each metric assigns to an order a value between 0 and 1, with higher values indicating better orders. Traditional RTP research used seeded faults, which allow fairly precisely deriving the failure-to-fault matrix [10,22,37] that can arbitrarily map failures and faults. Recent RTP research mostly uses real failures, e.g., analyzing real regression testing runs from continuous integration systems [14,15,23,24,27,34], making it rather difficult to precisely derive the failure-to-fault matrix. As a result, the increasingly popular failure-to-fault matrices are *all-to-one*, where all failures map to the same one fault, and *one-to-one*, where each failure maps to a distinct fault.

To describe allowed orders, we note that real test suites often partition tests, e.g., in JUnit [20], each test method belongs to a test class. Traditional research ignores this partitioning and allows all $n!$ orders ($\Omega_a(T)$ for short) of n tests. We introduced *compatible*[4] orders [46] ($\Omega_c(T)$ for short) that consider the partitioning and allow only orders that do not interleave tests from different classes.

We present the first theoretical analysis for the cases most commonly used in RTP research. We introduce an algorithm for efficiently computing the exact probability mass functions (PMFs) of α for all failure-to-fault matrices and $\Omega_a(T)$. We demonstrate the efficiency of our algorithm on the benchmarks from

[4] Our original term was *class-compatible* [46] because we considered as tests only test methods in test classes, but the concept easily generalizes to other kinds of tests.

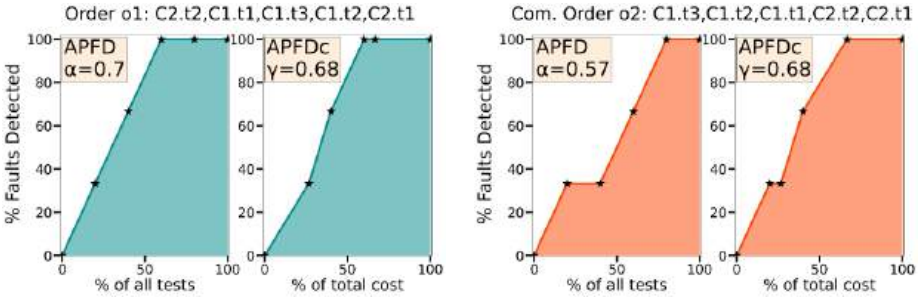


Fig. 1: Example metrics for two orders (Com. is compatible) for $n = 5, m = 3$; class C1 has 3 tests with costs $\langle 40, 20, 60 \rangle$, class C2 has 2 with $\langle 100, 80 \rangle$; C1.t1 detects fault F1; C1.t3 detects F2; C2.t1 detects F2 and F3; C2.t2 detects F3.

the largest RTP dataset for Java projects [34]. For the common all-to-one and one-to-one cases, we further derive a closed-form formula and a good approximation, respectively. We also derive closed-form formulas for the expected values for both α and γ for the general failure-to-fault matrix, for both $\Omega_a(T)$ and $\Omega_c(T)$, and we compare these values in various scenarios. Interestingly, on average, $\Omega_a(T)$ can perform much better (up to 1/2) than $\Omega_c(T)$ for certain scenarios, but cannot perform much worse (only up to 1/6) for any scenario; Section 5.1 presents this comparison, including two scenarios near the limits (1/2 and 1/6).

We finally derive two interesting properties for the α and γ metrics. Using these properties, we revisit some of the highly cited papers on RTP and find that some presented results may be biased due to insufficient sampling. Overall, our theoretical analysis provides new insights into the random RTP widely used in prior work but only via empirical sampling. Our results show that in many cases researchers need not run sampling but can use simple formulas or algorithms to obtain more precise statistics for the random RTP metrics.

2 Preliminaries

Our notation largely follows the prior work that introduced APFD (α) [38] and APFD$_c$ (γ) [11], but we make explicit the failure-to-fault matrix. Let n be the number of tests and m be the number of faults detected by (some of) these tests. Let M be a failure-to-fault matrix, i.e., a $n \times m$ Boolean matrix such that $M_{j,i} =$ true iff (failure of) test j detects fault i, and each fault has at least one failure (i.e., $\forall i.\exists j.M_{j,i}$). Let T be the set of tests in the test suite. We denote the set of tests that detect the fault i as $T_i = \{j | M_{j,i}\}$. In general, T_i and $T_{i'}$ for $i \neq i'$ need not be disjoint because one failing test can detect multiple faults. The total number of failures is $k = |\{j | \exists i.M_{j,i}\}|$, and we use $k_i = |T_i|$.

For an order o (a permutation of T), we use $<_o$ to compare the positions of two tests t and t' in the order: $t <_o t'$ denotes that t precedes t' in o, and $t \leq_o t'$ denotes that $t = t'$ or $t <_o t'$. We denote the j^{th} test in an order o as $t_j(o)$. Let $\tau_i(o) = \min_j M_{t_j(o),i}$ be the position of the first test to detect the fault i in o. Prior work [11,38] defined metrics α and γ (using the notation TF instead of τ).

We use $\alpha(o)$ and $\gamma(o)$ to indicate α and γ, respectively, for a given order o. We drop o from $<_o$, $\leq_o$, $t_j(o)$, $\tau_i(o)$, $\alpha(o)$, and $\gamma(o)$ when clear from the context.

The most popular RTP metric is α [38], defined for an order o as follows.

Definition 1 (α)**.** APFD *is defined as*

$$\alpha = 1 - \frac{\sum_{i=1}^{m} \tau_i}{nm} + \frac{1}{2n} \tag{1}$$

Plotting the percentage of faults detected against the percentage of executed tests, α represents the area under the curve, as shown in two examples in Fig. 1. The diagonal lines interpolate the percentage of faults detected and lead to nice properties of mean/median α values and symmetry (Section 6). α ranges between 0 and 1, more precisely between $1/(2n)$ and $1-1/(2n)$. A larger α indicates that an order detects faults earlier, on average.

While α effectively considers the number of tests, the "cost cognizant" metric γ considers the cost of tests [11]. The cost can be measured in various ways, but most work uses the test runtime. We use $\sigma(t)$ to denote the cost (runtime) of a test t; the total cost of a set of tests T is $\sigma(T) = \sum_{t \in T} \sigma(t)$.

Definition 2 (γ)**.** APFD_c *is defined as*

$$\gamma = \frac{\sum_{i=1}^{m} \left(\sum_{j=\tau_i}^{n} \sigma(t_j) - \frac{1}{2}\sigma(t_{\tau_i}) \right)}{m \cdot \sigma(T)} \tag{2}$$

Plotting the percentage of faults detected against the percentage of total test-suite cost, γ represents the area under the curve, as shown in Fig. 1. Note that α can be viewed as a special case of γ where $\forall t, t' \in T.\sigma(t) = \sigma(t')$.

In practice, tests often belong to *classes*[5]—e.g., JUnit [20] test methods belong to test classes, Maven [28] test classes belong to modules, and pytest [35] test functions belong to test files—and tests from each class run together. Our prior work [46] defined *compatible* orders as those where all tests from each class are consecutive. We use T_C to denote the set of tests in a class C. An order o is compatible iff $\forall C, j \leq j' \leq j''.t_j(o) \in T_C \wedge t_{j''}(o) \in T_C \Rightarrow t_{j'}(o) \in T_C$. For example, $o2$ in Fig. 1 is compatible, while $o1$ is not. To distinguish the cases for all orders from the cases for only compatible orders, we use the subscripts $_a$ and $_c$, respectively, e.g., $\text{E}_\text{a}[x]$ and $\text{E}_\text{c}[x]$ represent the expected value of x for the uniform selection of all orders and compatible orders, respectively, and $P_a(A)$ and $P_c(A)$ represent the probability of event A for the uniform selection of all orders and compatible orders, respectively. We denote the set of all orders and all compatible orders for T as $\Omega_a(T)$ and $\Omega_c(T)$, respectively [46].

We analyze RTP techniques in *scenarios*, each of which consists of a test suite with n tests, m faults, the failure-to-fault matrix, the cost of each test, and for $\Omega_c(T)$ the class of each test. To analyze compatible orders, we introduce some new notation to indicate the class of tests. We use $T_{i,C} = T_i \cap T_C$ to denote the

[5] The term *class* for a set of tests that run together need not represent a test class.

set of tests in class C that detect the fault i. Let $\mathcal{C}$ be the set of all classes, and $\mathcal{C}_i$ be the set of classes that contain at least one test that detects the fault i, i.e., $\mathcal{C}_i = \{C \in \mathcal{C} | T_{i,C} \neq \emptyset\}$. Let $C(t)$ be the class that t belongs to, i.e., $t \in T_{C(t)}$. The number of compatible orders is $|\Omega_c(T)| = |\mathcal{C}|! \prod_{C \in \mathcal{C}} |T_C|!$.

For a set of orders S, be it $\Omega_a(T)$ or $\Omega_c(T)$, the probability mass function (PMF) of a metric, α or γ, is a function p from the metric value to its probability: $p(x) = P(\text{metric} = x) = |\{o \in S | \text{metric}(o) = x\}|/|S|$. We next derive some PMFs as all prior RTP work shows only sampled distributions of random RTP.

3 PMF of α

To analyze the PMF of the metric α, we first propose an algorithm to calculate the PMF of α for the general case of M. We then discuss two special cases, i.e., all-to-one and one-to-one, which are the most common in recent RTP research.

3.1 Algorithm to Calculate PMF of α for the General Case

To calculate the PMF of α, a naïve algorithm would enumerate all $n!$ orders and compute α for each order. In theory, α can take $O(n!)$ different values, e.g., when $m = \sum_{i=1}^{n} n^i$ and all n tests fail and detect $n, n^2, \ldots, n^n$ different faults, then each of the $n!$ orders has a different α. In practice, however, the number of faults m and the number of failing tests k are usually small, e.g., in our evaluation dataset [34], 2906 out of 2980 (98%) scenarios have $k \leq 10$. We present an algorithm that computes the exact PMF with $O(n^2 mk \cdot k!)$ time complexity. Despite the $k!$ factor, the algorithm runs in reasonable time in practice, under 30sec for any of the 2906 scenarios. When $k > 10$, one can resort to sampling.

We next describe the intuition for our algorithm. $\sum_{i=1}^{m} \tau_i$ is the only part of α that depends on the (test-suite) order, so we first calculate the PMF of this sum and then convert it to the PMF of α. Iterating over the faults does *not* lead to a nice recursive formulation. Our key insight is to instead *iterate over the positions of all k failing tests.* We view $\sum_{i=1}^{m} \tau_i$ as a *weighted sum*

$$\sum_{i=1}^{m} \tau_i = \sum_{j=1}^{k} w_j \phi_j \tag{3}$$

where ϕ_j is the position of the j^{th} failing test in the order, and $w_j \geq 0$ is the weight, calculated as the number of faults detected *first* by the j^{th} failing test (Line 11 of Algorithm 1). For example, consider the order $o1$ in Fig. 1. The *relative order* of the $k = 4$ failing tests is $\rho = \langle \text{C2.t2}, \text{C1.t1}, \text{C1.t3}, \text{C2.t1} \rangle$; we use metavariable ρ to distinguish the notation from o for the order of all n tests. For this relative order, $w = \langle 1, 1, 1, 0 \rangle$ because the $m = 3$ faults are detected first by C2.t2, C1.t1, and C1.t3. The positions for this relative order ρ are $\phi = \langle 1, 2, 3, 5 \rangle$ because the 4 failing tests in ρ appear in these positions in the order $o1$.

We call a $\phi = \langle \phi_1 \ldots \phi_k \rangle$ *valid* if $1 \leq \phi_1 < \ldots < \phi_k \leq n$. Both sequences ϕ and $w = \langle w_1 \ldots w_k \rangle$ can vary for different orders. While ϕ has $\binom{n}{k}$ valid

Algorithm 1: Calculate the PMF of α

```
1  Input: n,m,M          // the number of tests and faults, and the failure-to-fault matrix
2  Output: p             // the PMF of α: p(x) = P(α = x)
3  Function PMF()        // main function; return the PMF of α for all orders
4      k = |{j|∃i.M_{j,i}}|          // number of failing tests in M, in practice k≪n
5      q = PMF_sum()                 // compute the PMF of Σ_{i}^{m} τ_i
6      return λx.q(mn − mnx + m/2)   // con ert that PMF to the PMF of α
7  Function PMF_sum()    // return the PMF of Σ_{i}^{m} τ_i for all orders
8      P = ⟨PMF_rorder(ρ), ∀ρ ∈ perms({j|∃i.M_{j,i}})⟩  // enumerate all relati e orders
9      return λx. Σ_{p∈P} p(x)/|P|   // a erage PMFs of Σ_{i}^{m} τ_i for each relati e order
10 Function PMF_rorder(ρ)    // return the PMF of Σ_{i}^{m} τ_i for a relati e order ρ
11     w = ⟨|{i|M_{ρ_j,i} ∧ ∄j′ < j.M_{ρ_{j′},i}}|, ∀j ∈ 1..k⟩   // w are the weights in formula (3)
12     return λs.f(w, k, n)(s)/C(n,k)   // the total number of φ is C(n,k)
13         // the function should be memoized to reuse the results for the repeated w,g,h
   Function f(w, g, h)   // return f_{g,h} gi en weights w, calculated with formula (4)
14     if g > h then
15         return λs.0
16     if g = 0 then
17         return λs.1_{s=0}
18     return λs.f(w, g, h − 1)(s)+f(w, g − 1, h − 1)(s − w_g h)
```

possibilities, we note that w has at most $k!$ possibilities (with $k! \ll \binom{n}{k}$ as $k \ll n$ in practice) because w depends only on ρ. Therefore, we first fix w by enumerating the $k!$ relative orders of the k failing tests. Then for each relative order, the problem of calculating the PMF of $\sum_{i=1}^{m} \tau_i = \sum_{j=1}^{k} w_j \phi_j$ becomes *"given w, count the number of valid ϕ such that $\sum_{j=1}^{k} w_j \phi_j = s$ for each s"*, which can be solved recursively as follows.

Let $f_{g,h}(s)$ be the number of assignments for the values of $\phi_1, \ldots, \phi_g$ such that $1 \le \phi_1 < \ldots < \phi_g \le h$ and $\sum_{j=1}^{g} w_j \phi_j = s$. The problem is to find $f_{k,n}(s)$. As the base case, (1) $f_{g,h}(s) = 0$ for $g > h$ because $\phi_g < g$ cannot hold; (2) $f_{0,h}(s) = \mathbf{1}_{s=0}$, where $\mathbf{1}$ is the indicator function, because only the empty sequence $\langle\rangle$ is valid and $\sum_{j=1}^{0} w_j \phi_j = 0$. For all $h \ge g > 0$, the number of assignments for $f_{g,h}(s)$ has two cases: (1) if $\phi_g \le h - 1$, the number is equal to $f_{g,h-1}(s)$ by definition; (2) if $\phi_g = h$, the number for s is equal to the number of assignments for $\phi_1, \ldots, \phi_{g-1}$ such that $\phi_{g-1} \le \phi_g - 1 = h - 1$ and $\sum_{j=1}^{g-1} w_j \phi_j = (\sum_{j=1}^{g} w_j \phi_j) - w_g \phi_g = s - w_g h$, which is $f_{g-1,h-1}(s - w_g h)$. In total,

$$f_{g,h}(s) = \begin{cases} 0 & g > h \\ \mathbf{1}_{s=0} & g = 0 \\ f_{g,h-1}(s) + f_{g-1,h-1}(s - w_g h) & \text{otherwise} \end{cases} \tag{4}$$

After solving $f_{k,n}$, we get the PMF of $\sum_{i=1}^{m} \tau_i$ for each relative order of the k failing tests. Because each of $k!$ relative orders has the same probability by symmetry, we simply take the average of their PMFs to get the PMF of $\sum_{i=1}^{m} \tau_i$ for all orders. Finally, we convert the PMF of $\sum_{i=1}^{m} \tau_i$ to the PMF of α.

Table 1: Number of tests, failures, runtime (in ms), and Jensen-Shannon (JS) distance for 10 largest scenarios [34] and one synthetic scenario (TSmax)

Test suite	**#Tests** (n)	**#Failures** (k)	**Runtime [ms]** **all-to-one**	**one-to-one**	**Jensen-Shannon distance** (§3.2.2)
TS1	2118	1	513	505	0.0000
TS2	1986	2	563	629	0.0005
TS3	2080	3	617	871	0.0003
TS4	1929	4	680	1147	0.0004
TS5	1795	5	731	1408	0.0006
TS6	339	6	627	732	0.0040
TS7	465	7	678	756	0.0034
TS8	813	8	829	2009	0.0023
TS9	52	9	1496	1846	0.0442
TS10	161	10	10989	27095	0.0150
TSmax	2118	10	32801	242400	0.0011

We next describe Algorithm 1 in more detail. The input is the number of tests n, the number of faults m, and the failure-to-fault matrix M. The main function `PMF` invokes `PMF_sum` to get the PMF of $\sum_{i=1}^{m} \tau_i$ and converts it to the PMF of α. The function `PMF_sum` enumerates all relative orders ρ of the k failing tests, invokes `PMF_rorder(`ρ`)` to get the PMF of $\sum_{i=1}^{m} \tau_i$ for each relative order, and averages these PMFs to get the PMF of $\sum_{i=1}^{m} \tau_i$ for all (relative) orders. Function `PMF_rorder(`ρ`)` computes the weights w from formula (3), invokes `f(w,k,n)` to get $f_{k,n}$ for w, and converts it to the PMF of $\sum_{i=1}^{m} \tau_i$.

We finally discuss the time complexity and the empirical performance of Algorithm 1. The major cost comes from computing the function `f`. Because there are $O(k!)$ different w and $0 \le g \le k, g \le h \le n$, we have $O(nk \cdot k!)$ different inputs for which to compute `f`. With memoization, `f` is computed only once for each input. Each computation takes $O(nm)$ because $|\text{support}(f_{g,h})| = O(nm)$ as $1 \le \tau_i \le n$ for $1 \le i \le m$. Therefore, the cost of computing `f` for all inputs is $O(n^2 mk \cdot k!)$. The other costs in the algorithm are lower than the cost of `f`; hence, the overall time complexity of Algorithm 1 is $O(n^2 mk \cdot k!)$.

Implementation: While top-down recursion makes it easier to present the algorithm, for better performance our implementation uses bottom-up dynamic programming to compute `f`. Our implementation fits in only 117 lines of C++.

Dataset: We use the RTP dataset with the most Java projects [34] for our evaluation. In this dataset, each test is a test class and each class is a Maven module [28]. The dataset has 2980 scenarios, and 2906 (98%) have $k \le 10$. We select, for each $k \le 10$, the scenario with the maximum number of tests (n) from the dataset. We also make a synthetic scenario with 2118 tests, being the largest number of tests in the dataset, and 10 failures. We use both all-to-one and one-to-one failure-to-fault matrices on the selected scenarios.

Evaluation: As Table 1 shows, the code finishes in under 30sec (on a common laptop) for all real scenarios; it takes more time on the synthetic one for all-to-one and one-to-one, but the runtime is still 33sec and 4min, respectively.

3.2 PMFs of α for Special Cases

As mentioned in Section 1, recent RTP research uses real failures and faults, with two kinds of failure-to-fault matrices: all-to-one and one-to-one. We discuss the PMFs of α for these two commonly used cases.

3.2.1 All-to-One: We first derive the PMF of α for all-to-one. In this case, $m = 1$, $k \geq 1$, and $w_1 = 1, \forall j > 1.w_j = 0$ in formula (3). Therefore, the recursive formula (4) becomes $f_{g,h}(s) = f_{g,h-1}(s) + f_{g-1,h-1}(s)$ for $g > 1$, which is similar to Pascal's triangle. This observation hints that the PMF of α for all-to-one may have a closed formula with binomial coefficients.

Theorem 3 (The PMF of α for all-to-one failure-to-fault matrix)**.**

$$P(\alpha = 1 - \frac{s}{n} + \frac{1}{2n}) = \frac{\binom{n-s}{k-1}}{\binom{n}{k}}, s \in \{1, 2, \ldots, n-k+1\} \tag{5}$$

Proof. For all-to-one, the α value depends solely on τ_1, which is essentially ϕ_1 in formula (3). For $1 \leq s \leq n-k+1$, $\tau_1 = s$ holds as long as $s = \phi_1 < \ldots < \phi_k \leq n$. To satisfy the condition, we just need to choose the $k-1$ positions after position s. Therefore, $\binom{n-s}{k-1}$ out of $\binom{n}{k}$ ways to choose k positions in n satisfy the condition, so $P(\tau_1 = s) = \binom{n-s}{k-1}/\binom{n}{k}$, and formula (5) directly follows. □

With (5), we can use $O(n)$ time to compute the PMF of α for all-to-one. We can compute the needed binomial coefficients iteratively, starting from $\binom{k-1}{k-1} = 1$, with the recurrence $\binom{n'+1}{k-1} = \frac{n'+1}{n'-k+2}\binom{n'}{k-1}, n' \geq k-1$, and get $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$.

3.2.2 One-to-One: We next consider the PMF of α for one-to-one. In this case, $m = k$ and each failing test finds a distinct fault, so for every relative order of the k failing tests, $\forall j.w_j = 1$ in formula (3). Therefore, running Algorithm 1 and memoizing on w, the complexity becomes $O(n^2k^2 + k!)$. $k!$ is because we need to iterate through all the relative orders. We can avoid $k!$ if we check in advance that the failure-to-fault matrix is one-to-one, so the complexity is $O(n^2k^2)$.

Moreover, considering formula (4) when $\forall j.w_j = 1$, $f_{k,n}$ essentially models the problem *"counting the number of partitions of s into k distinct summands from $\{1, 2, \ldots, n\}$"*. Specifically, $f_{g,h}(s)$ can be viewed as the number of partitions of s into g distinct summands in $\{1, 2, \ldots, h\}$, and $f_{g,h}(s) = f_{g,h-1}(s) + f_{g-1,h-1}(s-h)$ holds because the summand g can be either less than h or exactly h, corresponding to $f_{g,h-1}(s)$ and $f_{g-1,h-1}(s-h)$, respectively. To the best of our knowledge, no closed formula is known for this problem. Considering that in our evaluation dataset, 99.8% (2975/2980) of scenarios have $n^2k^2 < 10^9$, the $O(n^2k^2)$ algorithm is efficient enough for practical use for almost all cases.

Approximation: Furthermore, we can approximate the PMF by ignoring the distinct-number constraint, i.e., *"counting the number of partitions of s into k summands from $\{1, 2, \ldots, n\}$"*. This problem has a nice generating function $(x + x^2 + \ldots + x^n)^k$, where the coefficient of x^s is the number of partitions [43]:

$$\sum_{i=0}^{\lfloor \frac{s-k}{n} \rfloor} \binom{k}{i} (-1)^i \binom{s - ni - 1}{k - 1} \tag{6}$$

We can calculate these coefficients using two algorithms with different tradeoffs. The first algorithm first pre-calculates the binomial coefficients with Pascal's triangle and then calculates all the coefficients with formula (6). The first step takes $O(nk^2)$ because $s - ni - 1 \leq nk$ and $i \leq k$. The second step takes $O(nk^2)$ because each of $O(nk)$ coefficients takes $O(k)$ to compute as $\lfloor \frac{s-k}{n} \rfloor \leq k$. Thus, the overall time complexity of the first algorithm is $O(nk^2)$. The second algorithm calculates the generating function directly with the fast Fourier transform [4] by first converting $x + x^2 + \ldots + x^n$ to the point-value representation, calculating each point value to the k^{th} power, and interpolating to get the coefficients. The second algorithm takes $O(nk \log(nk))$ because the length of the polynomial is $O(nk)$. Comparing the complexity, the first algorithm is better when k is small compared to n (i.e., $k - \log k < \log n$), and the second is better otherwise.

To evaluate the approximation, we use Jensen–Shannon (JS) distance [16] between the exact and the approximated PMFs. We check our approximation on the same real scenarios as in Section 3.1. As Table 1 shows, the approximation yields PMFs with a small JS distance, the largest only 0.0442 for $n = 52, k = 9$.

3.3 PMF of γ

The PMF of γ is more complex than that of α because even for the simplest all-to-one failure-to-fault matrix, the number of possible values of γ can be $\Omega(2^n)$. For example, consider n tests with costs $1, 2, 4, \ldots, 2^{n-1}$, and only one test fails and detects the only fault. The γ value depends on the sum of the costs of the tests that precede the failure. 2^{n-1} different sets of the tests can precede the failure, and every set has a distinct sum of the costs. Even for the example in Fig. 1, the support of PMF for γ (33) is much bigger than that for α (8).

4 Expected Values for All Orders $\Omega_a(T)$

While some comparisons of RTP techniques use full samples of PMFs, many use just the arithmetic mean of the samples. We next derive formulas for expected values to obtain the mean faster and without the imprecision from sampling.

In this section, we consider the case where order o is uniformly selected from $\Omega_a(T)$, allowing $n!$ orders of n tests. Because α is a special case of γ where $\forall t, t' \in T.\sigma(t) = \sigma(t')$, we first derive γ.

To start with a simple example, consider a test suite with only one failing test ($k = 1$). For a random order, the test can be at any position with equal probability. Intuitively, the expected position across all of the orders is at the middle of the sequence, hence α and γ should be about 1/2. In fact, we will show that they are exactly 1/2. Moreover, the expected values of both α and γ are 1/2 as long as each fault is detected by only one failing test ($\forall i.k_i = |T_i| = 1$, which includes one-to-one). In general, the failure-to-fault matrix can be more complex: many tests could detect the same fault, and a test could detect many faults. To compute the expected values of α and γ, we first prove a useful lemma.

Lemma 4. *For every fault i,*

$$\forall t \notin T_i . P_a(t < t_{\tau_i}) = P_a(\forall t' \in T_i . t < t') = \frac{1}{k_i + 1} \tag{7}$$

Proof. Since τ_i is the position of the first test from T_i in the order, t precedes t_{τ_i} iff t precedes every $t' \in T_i$. Consider the relative position of each $t \notin T_i$ with respect to all the tests from T_i in a random order. By symmetry, it is equally likely that t is in any of the $k_i + 1$ relative positions created by the relative order of the k_i tests from T_i. Therefore, the probability that t is in the relative position preceding all the k_i tests from T_i is $\frac{1}{k_i+1}$. □

We first use this lemma to compute $\mathrm{E_a}[\gamma]$.

Theorem 5 (The expected value of γ for $\Omega_a(T)$)**.**

$$\mathrm{E_a}[\gamma] = 1 - \frac{\sum_{i=1}^{m} \left(\frac{\sigma(T \setminus T_i)}{k_i+1} + \frac{\sigma(T_i)}{2k_i} \right)}{m \cdot \sigma(T)} \tag{8}$$

Proof. From (2), the two key terms in γ are $\sigma(t_{\tau_i})$ and $\sum_{j=\tau_i}^{n} \sigma(t_j)$. By symmetry, any test $t \in T_i$ can be the first in the order, or equivalently $t = t_{\tau_i}$, with probability $\frac{1}{k_i}$. Thus

$$\mathrm{E_a}[\sigma(t_{\tau_i})] = \sum_{t \in T_i} P(t = t_{\tau_i}) \sigma(t) = \frac{\sigma(T_i)}{k_i} \tag{9}$$

Next, consider that $\sum_{j=\tau_i}^{n} \sigma(t_j) = \sum_{t \in T} \sigma(t) \mathbf{1}_{t_{\tau_i} \le t}$ can be also calculated as $\sum_{t \in T_i} \sigma(t) \mathbf{1}_{t_{\tau_i} \le t} + \sum_{t \notin T_i} \sigma(t) \mathbf{1}_{t_{\tau_i} \le t}$. For every test $t \in T_i$, $t_{\tau_i} \le t$ by definition, so $\forall t \in T_i . \mathrm{E_a}[\mathbf{1}_{t_{\tau_i} \le t}] = 1$. For every test $t \notin T_i$, $\mathrm{E_a}[\mathbf{1}_{t_{\tau_i} \le t}] = P_a(t_{\tau_i} \le t) = 1 - P_a(t < t_{\tau_i}) = \frac{k_i}{k_i+1}$. The last equality stems from Lemma 4. Therefore, by the linearity of expectation, we get

$$\mathrm{E_a}[\sum_{j=\tau_i}^{n} \sigma(t_j)] = \sigma(T_i) + \frac{k_i}{k_i + 1} \sigma(T \setminus T_i) \tag{10}$$

From (2), (9), and (10), we get (8). □

Corollary 5.1 (The expected value of α for $\Omega_a(T)$)**.**

$$\mathrm{E_a}[\alpha] = 1 - \frac{(n+1) \sum_{i=1}^{m} \frac{1}{k_i+1}}{nm} + \frac{1}{2n} \tag{11}$$

Revisiting the case where each fault can be detected by only one failing test, setting $\forall i . k_i = 1$ in (8) or (11), gives exactly $1/2 = \mathrm{E_a}[\alpha] = \mathrm{E_a}[\gamma]$. In fact, even in the general case of any failure-to-fault matrix, we find that the two expected values are similar if not the same, inspiring us to derive the following bound:

Theorem 6 (The expected difference of α and γ for $\Omega_a(T)$)**.**

$$-\frac{1}{12} < \mathrm{E_a}[\alpha] - \mathrm{E_a}[\gamma] < \frac{1}{2n} \tag{12}$$

Proof. From formulas (8) and (11), we have $\mathrm{E_a}[\alpha] - \mathrm{E_a}[\gamma] = \Delta_\gamma - \Delta_\alpha + \frac{1}{2n}$, where $\Delta_\gamma = \frac{\sum_i^m{}_1 (\frac{1}{2k_i} - \frac{1}{k_i+1})\sigma(T_i)}{m \cdot \sigma(T)}$ and $\Delta_\alpha = \frac{\sum_i^m{}_1 \frac{1}{k_i+1}}{nm}$. Since $k_i \geq 1$, we have $-\frac{1}{12} \leq \frac{1}{2k_i} - \frac{1}{k_i+1} \leq 0$ (with basic calculus, minimum is for $k_i = 2$ or $k_i = 3$), which, combined with $\sigma(T_i) \leq \sigma(T)$, gives $-\frac{1}{12} \leq \Delta_\gamma \leq 0$. Since $k_i \geq 1$, we also have $0 < \frac{1}{k_i+1} \leq \frac{1}{2}$, which gives $0 < \Delta_\alpha \leq \frac{1}{2n}$. Thus, we have $-\frac{1}{12} \leq \Delta_\gamma - \Delta_\alpha + \frac{1}{2n} < \frac{1}{2n}$. However, $\Delta_\gamma - \Delta_\alpha + \frac{1}{2n} = -\frac{1}{12}$ would require $\Delta_\alpha = \frac{1}{2n}$ and thus $\forall i.k_i = 1$, in which case $\Delta_\gamma = 0$ and $\Delta_\gamma - \Delta_\alpha + \frac{1}{2n} = 0 \neq -\frac{1}{12}$. Therefore, the equality cannot hold and $-\frac{1}{12} < \mathrm{E_a}[\alpha] - \mathrm{E_a}[\gamma] < \frac{1}{2n}$. □

5 Expected Values for Compatible Orders $\Omega_c(T)$

In this section, we consider the expected values of α and γ for $\Omega_c(T)$. Compatible orders do not interleave tests from different classes, as defined in Section 2. Similar to $\Omega_a(T)$, we first prove a useful lemma for $\Omega_c(T)$.

Lemma 7. *For every fault i, (note that if $t \notin T_i$, $C(t)$ may have another $t' \in T_i$)*

$$\forall t \notin T_i . P_c(t < t_{\tau_i}) = P_c(\forall t' \in T_i . t < t') = \begin{cases} \frac{1}{|\mathcal{C}_i|(|T_{i,C(t)}|+1)} & C(t) \in \mathcal{C}_i \\ \frac{1}{|\mathcal{C}_i|+1} & C(t) \notin \mathcal{C}_i \end{cases} \tag{13}$$

Proof. For $C(t) \in \mathcal{C}_i$ case, two conditions must hold for $t \notin T_{i,C(t)}$ to precede all tests that detect the fault i. First, among all classes in $\mathcal{C}_i$, $C(t)$ must be the first in the order, and by symmetry, each class in $\mathcal{C}_i$ can be the first with the same probability $\frac{1}{|\mathcal{C}_i|}$. Second, t must precede all tests from $T_{i,C(t)}$, which (similar to Lemma 4) holds with the probability $\frac{1}{|T_{i,C(t)}|+1}$. The two conditions are independent because they are about the class order and the test order inside the class, respectively, and these orders are independent of each other. Therefore, the probability that t precedes the first test that detects the fault i is $\frac{1}{|\mathcal{C}_i|(|T_{i,C(t)}|+1)}$.

For $C(t) \notin \mathcal{C}_i$ case, only one condition—$C(t)$ precedes all classes in $\mathcal{C}_i$—must hold for t to precede the first test that detects the fault i, which (similar to Lemma 4) happens with probability $\frac{1}{|\mathcal{C}_i|+1}$. □

Theorem 8 (The expected value of γ for $\Omega_c(T)$)**.**

$$\mathrm{E_c}[\gamma] = 1 - \frac{1}{m \cdot \sigma(T)} \sum_{i=1}^{m} \left(\frac{\sum_{C \notin \mathcal{C}_i} \sigma(T_C)}{|\mathcal{C}_i|+1} + \frac{1}{|\mathcal{C}_i|} \sum_{C \in \mathcal{C}_i} \left(\frac{\sigma(T_C \setminus T_{i,C})}{|T_{i,C}|+1} + \frac{\sigma(T_{i,C})}{2|T_{i,C}|} \right) \right) \tag{14}$$

Proof. We first compute the two key terms $\sigma(t_{\tau_i})$ and $\sum_{j=\tau_i}^{n}\sigma(t_j)$ in γ. For each test $t \in T_i$ to be the first, its class $C(t) \in \mathcal{C}_i$ should be the first among all classes in $\mathcal{C}_i$ with probability $\frac{1}{|\mathcal{C}_i|}$, and t must be the first among all tests in $T_{i,C(t)}$ with probability $\frac{1}{|T_{i,C(t)}|}$. These two events are independent, so the joint probability is $\frac{1}{|\mathcal{C}_i||T_{i,C(t)}|}$. By $\sigma(t_{\tau_i}) = \sum_{t\in T_i}\sigma(t)\cdot\mathbf{1}_{t=t_{\tau_i}}$, we have

$$\mathrm{E_c}[\sigma(t_{\tau_i})] = \sum_{t\in T_i}\frac{\sigma(t)}{|\mathcal{C}_i||T_{i,C(t)}|} = \frac{1}{|\mathcal{C}_i|}\sum_{C\in\mathcal{C}_i}\frac{\sigma(T_{i,C})}{|T_{i,C}|} \tag{15}$$

Next, consider $\sum_{j=\tau_i}^{n}\sigma(t_j) = \sum_{t\in T}\sigma(t)\cdot\mathbf{1}_{t_{\tau_i}\le t}$. Each t is either (1) $t \in T_i$, where $\mathbf{1}_{t_{\tau_i}\le t} = 1$ by definition of τ_i; or (2) $t \notin T_i$, where $\mathrm{E_c}[\mathbf{1}_{t_{\tau_i}\le t}] = \mathrm{E_c}[\mathbf{1}_{t_{\tau_i}<t}] = P_c(t_{\tau_i} < t) = 1 - P_c(t < t_{\tau_i})$ can be obtained from Lemma 7. Combining these cases, we have

$$\begin{aligned}\mathrm{E_c}[\textstyle\sum_{j=\tau_i}^{n}\sigma(t_j)] = \sigma(T_i) + \tfrac{|\mathcal{C}_i|}{|\mathcal{C}_i|+1}\textstyle\sum_{C\notin\mathcal{C}_i}\sigma(T_C) + \\ \textstyle\sum_{C\in\mathcal{C}_i}\left(1 - \frac{1}{|\mathcal{C}_i|(|T_{i,C}|+1)}\right)\sigma(T_C \setminus T_{i,C})\end{aligned} \tag{16}$$

From (2), (15), and (16), we get (14). □

Corollary 8.1 (The expected value of α for $\Omega_c(T)$)**.**

$$\mathrm{E_c}[\alpha] = 1 - \frac{1}{nm}\sum_{i=1}^{m}\left(\frac{\sum_{C\notin\mathcal{C}_i}|T_C|}{|\mathcal{C}_i|+1} + \frac{1}{|\mathcal{C}_i|}\sum_{C\in\mathcal{C}_i}\frac{|T_C|+1}{|T_{i,C}|+1}\right) + \frac{1}{2n} \tag{17}$$

We next discuss the expected difference of $\mathrm{E_c}[\alpha]$ and $\mathrm{E_c}[\gamma]$. Unlike the case with $\Omega_a(T)$, where the difference has a rather small bound, we find that the difference can be rather large for $\Omega_c(T)$.

Theorem 9 (The expected difference of α and γ for $\Omega_c(T)$)**.**

$$-\frac{1}{2} < \mathrm{E_c}[\alpha] - \mathrm{E_c}[\gamma] \le \frac{1}{2} - \frac{1}{2n} \tag{18}$$

Proof. From (14) and (17), we get $\mathrm{E_c}[\alpha] - \mathrm{E_c}[\gamma] = \Delta_\gamma - \Delta_\alpha + \frac{1}{2n}$, where $\Delta_\gamma = \frac{1}{m\cdot\sigma(T)}\sum_{i=1}^{m}\left(\frac{\sum_{C\notin\mathcal{C}_i}\sigma(T_C)}{|\mathcal{C}_i|+1} + \frac{1}{|\mathcal{C}_i|}\sum_{C\in\mathcal{C}_i}\left(\frac{\sigma(T_C\setminus T_{i,C})}{|T_{i,C}|+1} + \frac{\sigma(T_{i,C})}{2|T_{i,C}|}\right)\right)$ and $\Delta_\alpha = \frac{1}{nm}\sum_{i=1}^{m}(\frac{\sum_{C\notin\mathcal{C}_i}|T_C|}{|\mathcal{C}_i|+1} + \frac{1}{|\mathcal{C}_i|}\sum_{C\in\mathcal{C}_i}\frac{|T_C|+1}{|T_{i,C}|+1})$. $\Delta_\gamma > 0$ because all the terms in Δ_γ are positive. From $\forall i, C \in \mathcal{C}_i.|\mathcal{C}_i| \ge 1, |T_{i,C}| \ge 1$, we have

$$\begin{aligned}\Delta_\gamma &\le \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}\left(\frac{\sum_{C\notin\mathcal{C}_i}\sigma(T_C)}{1+1} + \frac{1}{1}\sum_{C\in\mathcal{C}_i}\left(\frac{\sigma(T_C\setminus T_{i,C})}{1+1} + \frac{\sigma(T_{i,C})}{2\cdot 1}\right)\right) \\ &= \tfrac{1}{m\cdot\sigma(T)}\cdot\tfrac{1}{2}\textstyle\sum_{i=1}^{m}\sigma(T) = \frac{1}{2}\end{aligned}$$

Similarly,

$$\begin{aligned}\Delta_\alpha &\le \tfrac{1}{nm}\textstyle\sum_{i=1}^{m}(\frac{\sum_{C\notin\mathcal{C}_i}|T_C|}{|\mathcal{C}_i|+1}+\frac{1}{|\mathcal{C}_i|}\sum_{C\in\mathcal{C}_i}\frac{|T_C|+1}{1+1})\\ &\le \tfrac{1}{nm}\textstyle\sum_{i=1}^{m}(\frac{\sum_{C\notin\mathcal{C}_i}|T_C|}{2|\mathcal{C}_i|}+\frac{\sum_{C\in\mathcal{C}_i}(|T_C|+1)}{2|\mathcal{C}_i|})=\frac{1}{nm}\sum_{i=1}^{m}(\frac{n}{2|\mathcal{C}_i|}+\frac{1}{2})\le\frac{n+1}{2n}\end{aligned}$$

From $0 \le |T_{i,C}| \le |T_C|$, we also have $\Delta_\alpha \ge \frac{1}{n}$. Combining $0 < \Delta_\gamma \le \frac{1}{2}$ and $\frac{1}{n} \le \Delta_\alpha \le \frac{n+1}{2n}$, we get $-\frac{1}{2} < \Delta_\gamma - \Delta_\alpha + \frac{1}{2n} \le \frac{1}{2} - \frac{1}{2n}$ □

Considering many inequalities in the preceding proof, one may expect the bounds to be loose, but we show two scenarios where bounds are close to tight. Both scenarios have only one fault. Scenario one has two classes: C_1 has only one passing test t with cost qN ($q > 0$ is arbitrary), and C_2 has N failing tests each with cost $\frac{q}{N}$. We assume $N \gg 1$. t must be the first or last in any compatible order, each with probability 1/2 (when C_1 is first or second). $\mathrm{E_c}[\alpha]$ is close to 1, and $\mathrm{E_c}[\gamma]$ is only about 1/2. Precisely, $\mathrm{E_c}[\alpha] - \mathrm{E_c}[\gamma] = \frac{N^2-2N+2}{2N^2+2N} \approx \frac{1}{2}$ when $N \gg 1$. Scenario two has two classes: C_2 has N failing tests with cost $\frac{q}{N}$, and C_3 has N^2 passing tests each with cost $\frac{q}{N^3}$. The two classes have only two orders, each with probability 1/2. $\mathrm{E_c}[\gamma]$ is close to 1, and $\mathrm{E_c}[\alpha]$ is only about 1/2. Precisely, $\mathrm{E_c}[\alpha] - \mathrm{E_c}[\gamma] = \frac{1}{N+1} - \frac{N^2+2}{2N^2+2N} + \frac{1}{2N} \approx -\frac{1}{2}$ when $N \gg 1$.

5.1 Comparison of $\Omega_a(T)$ and $\Omega_c(T)$

Orders that are compatible have more constraints on the PMF, which could increase or decrease average α or γ values. To compare how orders in $\Omega_a(T)$ and $\Omega_c(T)$ perform on average, we compare $\mathrm{E_a}[\alpha]$ with $\mathrm{E_c}[\alpha]$ and $\mathrm{E_a}[\gamma]$ with $\mathrm{E_c}[\gamma]$.

Theorem 10 (Difference of $\mathrm{E_c}[\gamma]$ and $\mathrm{E_a}[\gamma]$)**.**

$$\frac{1}{2n} - \frac{1}{2} \le \mathrm{E_c}[\gamma] - \mathrm{E_a}[\gamma] \le \frac{1}{6} \tag{19}$$

Proof. From (8) and (14), we have

$$\begin{aligned}\mathrm{E_c}[\gamma] - \mathrm{E_a}[\gamma] = \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}\Bigg(&\frac{\sigma(T_i)}{2k_i}+\frac{\sigma(T\backslash T_i)}{k_i+1}-\frac{\sum_{C\notin\mathcal{C}_i}\sigma(T_C)}{|\mathcal{C}_i|+1}-\\ &\frac{1}{|\mathcal{C}_i|}\sum_{C\in\mathcal{C}_i}\left(\frac{\sigma(T_C\backslash T_{i,C})}{|T_{i,C}|+1}+\frac{\sigma(T_{i,C})}{2|T_{i,C}|}\right)\Bigg)\end{aligned} \tag{20}$$

Because $\forall i.1 \le k_i \le n, |C_i| \ge 1, |T_{i,c}| \ge 1$, we have

$$\mathrm{E_c}[\gamma] - \mathrm{E_a}[\gamma] \ge \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}(\frac{1}{2n}-\frac{1}{2})\sigma(T) = \frac{1}{2n}-\frac{1}{2}$$

For the other side, because $\forall i.|C_i| \le k_i, |T_{i,c}| \le k_i$, we have

$$\begin{aligned}\mathrm{E_c}[\gamma] - \mathrm{E_a}[\gamma] &\le \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}\Bigg(\frac{\sigma(T_i)}{2k_i}+\frac{\sigma(T\backslash T_i)}{k_i+1}-\frac{\sum_{C\notin\mathcal{C}_i}\sigma(T_C)}{k_i+1}-\\ &\qquad\frac{(\sum_{C\in\mathcal{C}_i}\sigma(T_C))-\sigma(T_i)}{|\mathcal{C}_i|(k_i+1)}-\frac{\sigma(T_i)}{2|\mathcal{C}_i|k_i}\Bigg)\\ &= \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}\left(1-\frac{1}{|\mathcal{C}_i|}\right)\left(\frac{\sum_{C\in\mathcal{C}_i}\sigma(T_C)}{k_i+1}-\sigma(T_i)\left(\frac{1}{k_i+1}-\frac{1}{2k_i}\right)\right)\\ &\le \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}\left(1-\frac{1}{|\mathcal{C}_i|}\right)\frac{\sum_{C\in\mathcal{C}_i}\sigma(T_C)}{k_i+1}\\ &\le \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}\frac{|\mathcal{C}_i|-1}{|\mathcal{C}_i|(|\mathcal{C}_i|+1)}\sum_{C\in\mathcal{C}_i}\sigma(T_C)\\ &\le \tfrac{1}{m\cdot\sigma(T)}\textstyle\sum_{i=1}^{m}\frac{\sigma(T)}{6}=\frac{1}{6}\end{aligned}$$

The third last inequality holds because $\forall k_i \geq 1. \frac{1}{k_i+1} - \frac{1}{2k_i} \geq 0$. The last inequality holds because $\forall |\mathcal{C}_i| \geq 1. \frac{|\mathcal{C}_i|-1}{|\mathcal{C}_i|(|\mathcal{C}_i|+1)} \leq \frac{1}{6}$, which can be shown with simple calculus, and $\sum_{C \in \mathcal{C}_i} \sigma(T_C) \leq \sigma(T)$. □

Corollary 10.1 (Difference of $\mathrm{E_c}[\alpha]$ and $\mathrm{E_a}[\alpha]$)**.**

$$\frac{1}{2n} - \frac{1}{2} \leq \mathrm{E_c}[\alpha] - \mathrm{E_a}[\alpha] \leq \frac{1}{6} \tag{21}$$

We give two scenarios where the preceding bounds are close to tight. In both scenarios, we set $\forall t, t' \in T. \sigma(t) = \sigma(t')$, so that $\alpha = \gamma$ and $\mathrm{E_c}[\alpha] - \mathrm{E_a}[\alpha] = \mathrm{E_c}[\gamma] - \mathrm{E_a}[\gamma]$. The first scenario has one fault F, each of the $|\mathcal{C}|$ classes contains $\frac{n}{|\mathcal{C}|}$ tests, and tests from only one class detect F but all tests in that class detect F. In this scenario, $\mathrm{E_a}[\alpha] = 1 - \frac{|\mathcal{C}|(n+1)}{n(n+|\mathcal{C}|)} + \frac{1}{2n}$, and $\mathrm{E_c}[\alpha] = 1 - \frac{|\mathcal{C}|-1}{2|\mathcal{C}|} - \frac{1}{2n}$. If we consider $|\mathcal{C}| = \sqrt{n}$, when $n \gg 1$, $\mathrm{E_a}[\alpha] \approx 1$ but $\mathrm{E_c}[\alpha] \approx 1/2$, hence $\mathrm{E_c}[\alpha] - \mathrm{E_a}[\alpha] \approx -1/2$. The second scenario has one fault F and two classes with 1 and $n-1$ tests, and each class contains only one test that detects F. In this scenario, $\mathrm{E_a}[\alpha] = \frac{2}{3} - \frac{1}{2n}$ and $\mathrm{E_c}[\alpha] = \frac{3}{4}$. When $n \gg 1$, $\mathrm{E_c}[\alpha] - \mathrm{E_a}[\alpha] \approx \frac{1}{12}$, close to the upper bound of 1/6.

In brief, measured by α or γ, compatible orders can be much worse on average than all orders (up to 1/2) but cannot be much better (up to 1/6).

6 Properties of Metrics and Checking Prior RTP Work

Prior work on random RTP uses sampling and often visualizes α and γ values as *boxplots* that may show the median, mean, quartiles (25% and 75%), and "whiskers" (1.5 times the interquartile range) of the sampled distribution. For papers that show these boxplots, we identify two properties for the boxplots, focusing on $\Omega_a(T)$ because it is used in almost all prior work instead of $\Omega_c(T)$ [46]:

- **Mean/Median at Least Half:** $\mathrm{E_a}[\alpha], \mathrm{Med_a}(\alpha), \mathrm{E_a}[\gamma], \mathrm{Med_a}(\gamma) \geq 1/2$.
- **Symmetric PMF:** $\mathrm{E_a}[\alpha] = 1/2 \Leftrightarrow \mathrm{Med_a}(\alpha) = 1/2 \Leftrightarrow \mathrm{E_a}[\gamma] = 1/2 \Leftrightarrow \mathrm{Med_a}(\gamma) = 1/2 \Leftrightarrow \forall i. k_i = 1 \Rightarrow$ PMFs of α and γ are symmetric around 1/2.

To check the boxplots from prior work, we search on Google Scholar for papers related to "test prioritization" and keep only the papers that contain both "test" and "prioriti" in the titles. We sort these papers based on their citation count and check the top 100 papers with the highest citation count [44].

6.1 Mean/Median at Least Half

Lemma 11. *$\forall o \in \Omega_a(T)$ and its reverse order $\overline{o} \in \Omega_a(T)$,*

$$\gamma(o) + \gamma(\overline{o}) \geq 1 \tag{22}$$

The equality holds iff $\forall i. k_i = 1$.

Proof sketch. To give some intuition, when $\forall i.k_i = 1$, the test that detects the fault i first does not change by reversing the order, so the "prefixes" of the test in o and $\overline{o}$ complement each other and form the entire test suite. In this case, $\gamma(o) + \gamma(\overline{o}) = 1$. If $\exists i.k_i \geq 2$, the test that detects the fault i first in o is not the same test in $\overline{o}$, and the "prefixes" of these two tests in o and $\overline{o}$ do not form the entire test suite, so $\gamma(o) + \gamma(\overline{o}) > 1$. We omit the details due to space limit. □

Theorem 12 (Measures of central tendency are at least half)**.**

$$\min\{\mathrm{E_a}[\alpha], \mathrm{Med_a}(\alpha), \mathrm{E_a}[\gamma], \mathrm{Med_a}(\gamma)\} \geq 1/2 \tag{23}$$

The equality holds iff $\forall i.k_i = 1$.

Proof sketch. From (22), we get $\mathrm{E_a}[\gamma] = \frac{1}{2} \cdot \frac{\sum_{o \in \Omega_a(T)} (\gamma(o) + \gamma(\overline{o}))}{n!} \geq \frac{1}{2}$ and the equality holds iff $\forall i.k_i = 1$. Because α can be viewed as a special case of γ, we also have the same result for $\mathrm{E_a}[\alpha]$. The same result for $\mathrm{Med_a}(\alpha)$ and $\mathrm{Med_a}(\gamma)$ can also be derived from (22). We omit the details due to space limit. □

When we inspect the top 100 most cited RTP papers, we find at least five papers with boxplots clearly showing a mean or median below 1/2. These papers range from seminal papers [12, Figs. 2b, 2c, 2e] (year 2000) and [13, Fig. 3: schedule, tcas] (2002) to more recent [29, Fig. 4] (2007), [5, Fig. 2] (2016 – a co-author of this prior paper is also in this paper), and [41, Fig. 5] (2017). Instead of sampling random orders for an arbitrary number of times, future RTP research could use our formulas or algorithm to obtain correct mean and median values.

6.2 Symmetric PMF

We also prove that α and γ PMFs are symmetric when (23)'s equality holds.

Theorem 13 (Symmetry of the α and γ PMFs)**.** *If* $\mathrm{E_a}[\alpha] = 1/2 \vee \mathrm{Med_a}(\alpha) = 1/2 \vee \mathrm{E_a}[\gamma] = 1/2 \vee \mathrm{Med_a}(\gamma) = 1/2 \vee \forall i.k_i = 1$*, then*

$$\forall \delta. P(\alpha = 1/2 - \delta) = P(\alpha = 1/2 + \delta) \ \wedge \ P(\gamma = 1/2 - \delta) = P(\gamma = 1/2 + \delta) \tag{24}$$

Proof. From Theorem 12, $\min\{\mathrm{E_a}[\alpha], \mathrm{Med_a}(\alpha), \mathrm{E_a}[\gamma], \mathrm{Med_a}(\gamma)\} = 1/2 \vee \forall i.k_i = 1 \Leftrightarrow \forall i.k_i = 1 \Rightarrow \forall o.\alpha(o) + \alpha(\overline{o}) = 1 \wedge \gamma(o) + \gamma(\overline{o}) = 1$. Each order has exactly one reverse order, so the PMFs of α and γ are symmetric around 1/2. □

When we inspect the top 100 most cited RTP papers again, we find at least three papers relevant to this property. Based on the information in these papers, we believe that $\forall i.k_i = 1$ is true. Ideally, we would confirm each paper's failure-to-fault matrix, but papers often omit such details. On a positive note, the authors of one paper [38] released their dataset, which we analyze and confirm that $\forall i.k_i = 1$. The papers that violate this property include the most widely cited paper on RTP [38, Fig. 5: schedule, schedule2, tcas] (year 2001; 1563 citations per Google Scholar) and others, both older [36, Fig. 4: schedule, schedule2, tcas] (1999) and newer [40, Fig. 2] (2015) papers.

Instead of randomly sampling orders to approximate PMFs, future RTP papers could use our algorithm to compute exact PMFs. While we find only five and three papers that definitely violate Mean/Median at Least Half and Symmetric PMF, respectively, we suspect that many others may violate these or similar properties. However, due to the lack of data in many papers (e.g., no boxplot for random RTP), we cannot easily identify all violations.

7 Related Work

Some prior work [45, 49] considers expected values of α and γ but in different contexts from ours. Random testing (but *not* random RTP) has been studied for a while [7–9,17,18,31–33,50]. The most related are theoretical analyses of random test generation. Böhme and Paul [2, 3] analyze how random sampling of test inputs compares to systematic generation: random can be more efficient when the cost to systematically generate a test input exceeds the cost to randomly sample an input by some factor. Böhme et al. [1] analyze the connection between Shannon's entropy and the discovery rate of a fuzzer that randomly generates inputs. They provide the foundation for identifying random seeds for the fuzzer to improve the overall efficiency. Their analysis also enables future systematic approaches for test generation to be more efficiently compared with random. Similarly, our analysis can help future RTP work more efficiently compare against random RTP and avoid insufficient sampling. Beyond random test generation, Majumdar and Niksic [26] present a theoretical analysis on the effectiveness of randomly inserted partition faults to find bugs in distributed systems. In contrast, our analysis is on test-suite orders for random RTP.

8 Conclusion

Regression test prioritization (RTP) is a popular regression testing approach. Majority of highly cited RTP papers have compared RTP techniques with random RTP. However, all evaluations have been empirical, with no prior theoretical analysis of random RTP. This paper has presented such analysis, by introducing an algorithm for efficiently computing the exact probability mass function of APFD, deriving closed-form formulas and approximations for various metrics and scenarios, and deriving two interesting properties forAPFD and APFD_c. Overall, our analysis provides new insights into the random RTP, and our results show that future RTP work often need not use random sampling but can use our simple formulas or algorithms to more precisely evaluate random RTP.

Acknowledgments. We thank Anjiang Wei, Dezhi Ran, and Sasa Misailovic for their help. This work was partially supported by US NSF grants CCF-1763788. CCF-1956374, NSFC grant No. 62161146003, Tencent Foundation, and XPLORER PRIZE. We acknowledge support for research on regression testing from Dragon Testing, Microsoft, and Qualcomm. Tao Xie is the corresponding author, and also affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China.

References

1. Böhme, M., Manès, V.J.M., Cha, S.K.: Boosting fuzzer efficiency: An information theoretic perspective. In: ESEC/FSE (2020)
2. Böhme, M., Paul, S.: On the efficiency of automated testing. In: FSE (2014)
3. Böhme, M., Paul, S.: A probabilistic analysis of the efficiency of automated software testing. TSE (2016)
4. Brigham, E.O.: The fast Fourier transform and its applications. Prentice-Hall, Inc. (1988)
5. Busjaeger, B., Xie, T.: Learning for test prioritization: An industrial case study. In: FSE (2016)
6. Cheng, R., Zhang, L., Marinov, D., Xu, T.: Test-case prioritization for configuration testing. In: ISSTA (2021)
7. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP (2000)
8. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. TOSEM (2008)
9. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. TSE (1984)
10. Elbaum, S., Kallakuri, P., Malishevsky, A., Rothermel, G., Kanduri, S.: Understanding the effects of changes on the cost-effectiveness of regression testing techniques. STVR (2003)
11. Elbaum, S., Malishevsky, A., Rothermel, G.: Incorporating varying test costs and fault severities into test case prioritization. In: ICSE (2001)
12. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. In: ISSTA (2000)
13. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: A family of empirical studies. TSE (2002)
14. Elbaum, S., Rothermel, G., Penix, J.: Techniques for improving regression testing in continuous integration development environments. In: FSE (2014)
15. Elsner, D., Hauer, F., Pretschner, A., Reimer, S.: Empirically evaluating readily available information for regression test optimization in continuous integration. In: ISSTA (2021)
16. Endres, D.M., Schindelin, J.E.: A new metric for probability distributions. Transactions on Information Theory (2003)
17. Fraser, G., Zeller, A.: Generating parameterized unit tests. In: ISSTA (2011)
18. Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering (1994)
19. Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H.: Adaptive random test case prioritization. In: ASE (2009)
20. JUnit (2022), https://junit.org
21. Kim, J.M., Porter, A.: A history-based test prioritization technique for regression testing in resource constrained environments. In: ICSE (2002)
22. Kim, J.M., Porter, A., Rothermel, G.: An empirical study of regression test application frequency. STVR (2005)
23. Liang, J., Elbaum, S., Rothermel, G.: Redefining prioritization: Continuous prioritization for continuous integration. In: ICSE (2018)
24. Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., Zhang, L.: How does regression test prioritization perform in real-world software evolution? In: ICSE (2016)
25. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: FSE (2014)

26. Majumdar, R., Niksic, F.: Why is random testing effective for partition tolerance bugs? In: POPL (2017)
27. Mattis, T., Rein, P., Dürsch, F., Hirschfeld, R.: RTPTorrent: An open-source dataset for evaluating regression test prioritization. In: MSR (2020)
28. Maven (2022), https://maven.apache.org
29. Mirarab, S., Tahvildari, L.: A prioritization approach for software test cases based on Bayesian networks. In: FASE (2007)
30. Mondal, S., Nasre, R.: Summary of Hansie: Hybrid and consensus regression test prioritization. In: ICST (2021)
31. Ntafos, S.: On random and partition testing. In: ISSTA (1998)
32. Ozkan, B.K., Majumdar, R., Oraee, S.: Trace aware random testing for distributed systems. OOPSLA (2019)
33. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE (2007)
34. Peng, Q., Shi, A., Zhang, L.: Empirically revisiting and enhancing IR-based test-case prioritization. In: ISSTA (2020)
35. pytest (2022), https://docs.pytest.org
36. Rothermel, G., Untch, R., Chu, C., Harrold, M.: Test case prioritization: An empirical study. In: ICSM (1999)
37. Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., Davia, B.: The impact of test suite granularity on the cost-effectiveness of regression testing. In: ICSE (2002)
38. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. TSE (2001)
39. Rummel, M.J., Kapfhammer, G.M., Thall, A.: Towards the prioritization of regression test suites with data flow information. In: SAC (2005)
40. Saha, R.K., Zhang, L., Khurshid, S., Perry, D.E.: An information retrieval approach for regression test prioritization based on program changes. In: ICSE (2015)
41. Spieker, H., Gotlieb, A., Marijan, D., Mossige, M.: Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: ISSTA (2017)
42. Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: ISSTA (2002)
43. Stanley, R.P.: Enumerative Combinatorics, Volume 1. Cambridge University Press (2011)
44. A Theoretical Analysis of Regression Test Prioritization website (2022), https://sites.google.com/view/theoretical-analysis-of-rtp
45. Wang, Z., Chen, L.: Improved metrics for non-classic test prioritization problems. In: SEKE (2015)
46. Wei, A., Yi, P., Xie, T., Marinov, D., Lam, W.: Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In: TACAS (2021)
47. Wong, W., Horgan, J., London, S., Agrawal, H.: A study of effective regression testing in practice. In: ISSRE (1997)
48. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: A survey. STVR (2012)
49. Zhai, K., Jiang, B., Chan, W.: Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. IEEE TSC (2012)
50. Zhang, S., Saff, D., Bu, Y., Ernst, M.D.: Combined static and dynamic automated test generation. In: ISSTA (2011)

Verified First-Order Monitoring with Recursive Rules

Sheila Zingg[1], Srđan Krstić[1](✉), Martin Raszyk[1](✉), Joshua Schneider[1](✉), and Dmitriy Traytel[2](✉)

[1] Institute of Information Security, Department of Computer Science, ETH Zürich, Zurich, Switzerland, {srdan.krstic,martin.raszyk,joshua.schneider}@inf.ethz.ch
[2] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, traytel@di.ku.dk

Abstract. First-order temporal logics and rule-based formalisms are two popular families of specification languages for monitoring. Each family has its advantages and only few monitoring tools support their combination. We extend metric first-order temporal logic (MFOTL) with a recursive let construct, which enables interleaving rules with temporal logic formulas. We also extend VeriMon, an MFOTL monitor whose correctness has been formally verified using the Isabelle proof assistant, to support the new construct. The extended correctness proof covers the interaction of the new construct with the existing verified algorithm, which is subtle due to the presence of the bounded future temporal operators. We demonstrate the recursive let's usefulness on several example specifications and evaluate our verified algorithm's performance against the DejaVu monitoring tool.

Keywords: Rule-based specifications · Monitoring · Formal verification.

1 Introduction

In runtime verification, a monitor observes events generated by a running system and analyzes the event streams for compliance with a given specification. Temporal specification languages for monitoring are often classified as operational or declarative [10]. Operational languages explicitly describe how the monitor's input should be transformed to obtain an output. Two important subclasses of operational languages are rule-based formalisms [2,13] and stream runtime verification (SRV) languages [6,8,11,20]. Both formulate the transformations as recursive equations. In contrast, declarative languages, such as first-order temporal logics [4,15], describe the output by composing high-level operators.

Operational and declarative languages have complementary advantages: declarative languages let specification authors focus on the "what" and not the "how", whereas operational languages offer the authors more control over the evaluation. Most runtime verification tools do not support mixing the paradigms, especially when it comes to parametric, i.e., first-order, specification languages. A notable exception is the recent addition of recursive rules to past-time first-order temporal logic (PFLTL), implemented in the DejaVu monitoring tool [14]. As another important benefit, recursive rules can express operations like transitive closure that are not expressible in first-order logics.

In this paper, we introduce recursion in metric first-order temporal logic (MFOTL) [4] in the form of a recursive let construct. We develop and implement an evaluation algorithm for MFOTL with recursion in VeriMon [3, 21], an MFOTL monitor whose correctness has been formally verified in the Isabelle proof assistant. To this end, we extend the formal correctness proof to cover the recursive let construct.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 236–253, 2022.
https://doi.org/10.1007/978-3-030-99527-0_13

Unlike PFLTL, MFOTL supports bounded future temporal operators and aggregations (Section 2). The interaction of recursion with bounded future operators is subtle. To avoid non-termination, DejaVu requires all recursive occurrences to be guarded by a previous operator. We similarly require the recursive occurrences to be guarded in our monitor, but we relax the requirement on the guard to other past-time operators which ensure that their subformulas are evaluated strictly in the past. Moreover, we allow future operators in the recursive let construct, as long as no recursion takes place in the future operator's arguments. These restrictions ensure that the fixpoint given by the recursive let operator is well-defined. At the same time, they are permissive and allow us to formulate interesting examples, several of which are beyond what PFLTL with recursion can express.

Consider a specification that aims to secure hosts in a network that communicate with each other and with the outside world. A host is *tainted* by an address range iff there is a chain of communication from the address to the host and all hosts on the chain trigger an intrusion detection alert within one hour after communicating with the previous host. This specification can be expressed directly using our recursive let construct (to model chains of communication) and future temporal operators (to specify "within one hour after").

We start by extending MFOTL with a non-recursive let operator (Section 3). This special case is mainly of pedagogical value: aspects common to both let operators are easier to explain on the simpler non-recursive variant. Yet, this construct is useful in practice to structure complex formulas and improve monitoring performance by sharing common subformulas. Thus we extend VeriMon's algorithms and proofs with the non-recursive let.

We then introduce the recursive let operator (Section 4.1), exemplify its semantics with several specifications (Section 4.2), and develop the monitoring algorithm and sketch its correctness (Section 4.3). VeriMon's repository [24] contains complete formal proofs.

This work is part of the long-term effort to develop a trustworthy monitor that surpasses in expressiveness and efficiency other non-verified tools. In this work, our focus is on expressiveness (and trustworthiness). Nonetheless, we evaluate our algorithmic additions to VeriMon on a micro-benchmark and observe that even without further optimizations it exhibits an incomparable performance to DejaVu (Section 5). Moreover, we detected a problem in DejaVu's handling of variable names in recursive subformulas.

In summary, our main contribution is the extension of MFOTL with a recursive let operator and the design of an evaluation algorithm for it. Along the way, we introduce a non-recursive let operator, which proved essential when writing complex specifications. Our contributions are implemented as part of VeriMon and proved correct using Isabelle.

Related Work. Our work adds rule-based specification features [13] to a first-order specification language [16]. Above we describe our contribution's relationship to DejaVu and VeriMon, two monitors for first-order temporal specifications. VeriMon's algorithm [21], which we extend, is based on the algorithm used in the MonPoly monitor [5], although VeriMon has optimizations that are not present in MonPoly and vice versa [3]. VeriMon supports a more expressive specification language than MonPoly, and our introduction of the recursive let has increased the gap between the two. VeriMon's and MonPoly's algorithms work with finite relations. These tools are thus restricted to MFOTL's *monitorable fragment* [4], which ensures that all subformulas evaluate to finite results. In contrast, DejaVu finitely represents infinite relations using BDDs and thus supports the full PFLTL (but only closed formulas). Both DejaVu and our work restrict the recursive let syntactically.

$$\textbf{datatype}\ \mathit{data} = \mathsf{Int}\ \mathit{int} \mid \mathsf{Flt}\ \mathit{double} \mid \mathsf{Str}\ \mathit{string} \qquad \textbf{type\_synonym}\ \mathit{ts} = \mathit{nat}$$
$$\textbf{type\_synonym}\ \mathit{db} = \mathit{string} \Rightarrow \mathit{data\ list\ set} \qquad \textbf{typedef}\ \mathit{trace} = \{s :: (\mathit{db} \times \mathit{ts})\ \mathit{stream}.\ \mathsf{trace}\ s\}$$
$$\textbf{datatype}\ \mathit{trm} = \mathsf{V}\ \mathit{nat} \mid \mathsf{C}\ \mathit{data} \mid \mathit{trm} + \mathit{trm} \mid \ldots \qquad \textbf{typedef}\ \mathcal{I} = \{(a :: \mathit{nat}, b :: \mathit{enat}).\ a \leq b\}$$
$$\textbf{datatype}\ \mathit{frm} = \mathit{string}(\mathit{trm\ list}) \mid \mathit{trm} \circ \mathit{trm} \mid \neg\,\mathit{frm} \mid \exists\,\mathit{frm} \mid \mathit{frm} \vee \mathit{frm} \mid \mathit{frm} \wedge \mathit{frm}$$
$$\mid \bullet_{\mathcal{I}}\ \mathit{frm} \mid \bigcirc_{\mathcal{I}}\ \mathit{frm} \mid \mathit{frm}\ \mathsf{S}_{\mathcal{I}}\ \mathit{frm} \mid \mathit{frm}\ \mathsf{U}_{\mathcal{I}}\ \mathit{frm} \mid \mathit{nat} \leftarrow \mathit{agg\_op}(\mathit{trm}; \mathit{nat})\ \mathit{frm}$$

$$\textbf{fun}\ \mathsf{etrm} :: \mathit{data\ list} \Rightarrow \mathit{trm} \Rightarrow \mathit{data}\ \textbf{where}$$
$$\mathsf{etrm}\ v\ (\mathsf{V}\ x) = v\,!\,x \mid \mathsf{etrm}\ v\ (\mathsf{C}\ x) = x \mid \mathsf{etrm}\ v\ (t_1 + t_2) = \mathsf{etrm}\ v\ t_1 + \mathsf{etrm}\ v\ t_2 \mid \ldots$$

$$\textbf{fun}\ \mathsf{sat} :: \mathit{trace} \Rightarrow \mathit{data\ list} \Rightarrow \mathit{nat} \Rightarrow \mathit{frm} \Rightarrow \mathit{bool}\ \textbf{where}$$
$$\mathsf{sat}\ \sigma\ v\ i\ (p(as)) = (\mathsf{map}\ (\mathsf{etrm}\ v)\ as \in \Gamma\ \sigma\ i\ p) \mid \mathsf{sat}\ \sigma\ v\ i\ (t_1 \circ t_2) = (\mathsf{etrm}\ v\ t_1 \circ \mathsf{etrm}\ v\ t_2)$$
$$\mid \mathsf{sat}\ \sigma\ v\ i\ (\neg\varphi) = (\neg \mathsf{sat}\ \sigma\ v\ i\ \varphi) \mid \mathsf{sat}\ \sigma\ v\ i\ (\exists\varphi) = (\exists z.\ \mathsf{sat}\ \sigma\ (z\#v)\ i\ \varphi)$$
$$\mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha \vee \beta) = (\mathsf{sat}\ \sigma\ v\ i\ \alpha \vee \mathsf{sat}\ \sigma\ v\ i\ \beta) \mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha \wedge \beta) = (\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge \mathsf{sat}\ \sigma\ v\ i\ \beta)$$
$$\mid \mathsf{sat}\ \sigma\ v\ i\ (\bullet_I \varphi) = (\underline{\mathsf{case}}\ i\ \underline{\mathsf{of}}\ 0 \Rightarrow \mathsf{False} \mid j+1 \Rightarrow \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ j\ \varphi)$$
$$\mid \mathsf{sat}\ \sigma\ v\ i\ (\bigcirc_I \varphi) = (\mathsf{T}\ \sigma\ (i+1) - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ (i+1)\ \varphi)$$
$$\mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha\ \mathsf{S}_I\ \beta) = (\exists j \leq i.\ \mathsf{T}\ \sigma\ i - \mathsf{T}\ \sigma\ j \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ j\ \beta \wedge (\forall k \in \{j <.. i\}.\ \mathsf{sat}\ \sigma\ v\ k\ \alpha))$$
$$\mid \mathsf{sat}\ \sigma\ v\ i\ (\alpha\ \mathsf{U}_I\ \beta) = (\exists j \geq i.\ \mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} I \wedge \mathsf{sat}\ \sigma\ v\ j\ \beta \wedge (\forall k \in \{i ..< j\}.\ \mathsf{sat}\ \sigma\ v\ k\ \alpha))$$
$$\mid \mathsf{sat}\ \sigma\ v\ i\ (y \leftarrow \Omega(t; b)\ \varphi) = \big(\underline{\mathsf{let}}\ M = \{(x, \mathsf{card}^{\infty}\ Z) \mid x\ Z.$$
$$Z = \{z.\ \mathsf{length}\ z = b \wedge \mathsf{sat}\ \sigma\ (z\ @\ v)\ i\ \varphi \wedge \mathsf{etrm}\ (z\ @\ v)\ t = x\} \wedge Z \neq \{\}\}$$
$$\underline{\mathsf{in}}\ (M = \{\} \longrightarrow \mathsf{fv}\ \varphi \subseteq \{0 ..< b\}) \wedge v\,!\,y = \mathsf{eval\_agg\_op}\ \Omega\ M)$$

Fig. 1. Formal syntax and semantics of MFOTL with aggregations, where $\circ \in \{=, <, \leq\}$

Other rule-based [2,13] and SRV-based monitors [6,8,11,20] can express the temporal operators present in LTL, but struggle with extensions that introduce parameters. Even for the operators they can express, specialized algorithms that are carefully tuned for the operators tend to exhibit a better performance. Instead of encoding temporal operators, we take the opposite approach and enrich a monitor that uses specialized algorithms for temporal operators with general-purpose recursion.

Datalog [1] adds recursion to first-order logic, similarly to our addition of recursion to temporal logic. However, Datalog has no built-in notion of time and hence other measures must be taken to ensure that the fixpoints are well-defined, e.g., by restricting negation. Restricting the recursive occurrences to be strictly in the past is a natural and expressive alternative for monitoring, as we do not restrict negation beyond of what the monitorable fragment requires. Works on Datalog extensions with metric temporal operators [7,19,22] mostly study the decidability and complexity of computational problems related to these extensions, whereas we design, implement, and formally verify an executable algorithm.

2 Metric First-Order Temporal Logic

MFOTL extends linear temporal logic with first-order quantification, past-time operators, and interval bounds on the temporal operators [4]. The VeriMon monitor [3] supports a fragment of this logic. It also adds new features, specifically regular matching operators as in linear dynamic logic [9], which results in metric first-order dynamic logic (MFODL), as well as aggregations. Our extension of VeriMon with recursive rules retains the additional features of MFODL. However, the additional features are orthogonal to our extension and hence we base our presentation in this paper on MFOTL with aggregations.

We summarize MFOTL's syntax and semantics, as well as the monitorable fragment. The presentation generally follows the Isabelle formalization; however, we sometimes

deviate from Isabelle's concrete syntax for simplicity. We begin by defining some auxiliary types (top of Fig. 1). The logic's universe (type *data*) is fixed and infinite: it is a disjoint sum of integers, 64-bit IEEE floats, and strings of 8-bit characters. Databases (type *db*) encode first-order structures as functions from predicate names to relations over *data*. Relations are represented as sets of lists. A *trace* is a *stream* (an infinite sequence) of time-stamped databases. Time-stamps (type *ts*) are modeled as natural numbers (type *nat*). We write $\Gamma\ \sigma\ i$ for the ith database in σ, and $\mathsf{T}\ \sigma\ i$ for its time-stamp. The predicate trace enforces monotone and eventually increasing time-stamps, i.e., $\forall i \leq j.\ \mathsf{T}\ \sigma\ i \leq \mathsf{T}\ \sigma\ j$ and $\forall x.\ \exists i.\ x < \mathsf{T}\ \sigma\ i$. Non-empty intervals (type $\mathcal{I}$) are represented by their end-points. We write $[a,b]$ for the unique interval satisfying $n \in_{\mathcal{I}} [a,b]$ iff $a \leq n \leq b$, where $n \in_{\mathcal{I}} I$ denotes that I contains the natural number n. The interval is unbounded from above if $b = \infty$, which the type *enat* adds to the natural numbers.

Terms (type *trm*) are constructed recursively from variables (represented by De Bruijn indices), constants, and arithmetic operators. We use named variables in examples and omit the V and C constructors. There are two kinds of atomic formulas (type *frm*): flexible predicates of the form $p(as)$, where as is a list of terms, and rigid predicates $t_1 \circ t_2$ for $\circ \in \{=, <, \leq\}$, which have a fixed interpretation. Formally, the existential quantifier $\exists$ does not carry a variable name because of the De Bruijn encoding. We use $\mathsf{fv}\ \alpha$ to denote the set of De Bruijn indices of α's free variables.

The semantics is given by the functions etrm and sat (Fig. 1). Both depend on a valuation, which is a *data list* assigning a value to each variable. The satisfaction function sat for formulas additionally depends on a trace σ and a time-point i, which is an index into the trace. Indexing into lists is denoted by $v\,!\,x$, the operation $z\#v$ prepends the value z to the list v, and @ concatenates two lists. The notation $\{x\ ..< y\}$ and $\{x <.. y\}$ is shorthand for the sets $\{x, x+1, \ldots, y-1\}$ and $\{x+1, x+2, \ldots, y\}$ of natural numbers, respectively.

An aggregation formula $y \leftarrow \Omega(t;b)\ \varphi$ binds b variables in the subformula φ; the remaining free variables of φ are used for grouping. Each group is assigned an aggregate value y, which is computed by first evaluating the term t on each valuation that matches the group and that satisfies φ, then aggregating the results using the operator Ω (e.g., MIN for minimum). To this end, $\mathsf{eval\_agg\_op}\ \Omega\ M$ (not shown) applies Ω to a set M of value–multiplicity pairs [3]; $\mathsf{card}^{\infty}\ Z$ is the cardinality of Z, or ∞ if Z is infinite. The conjunct $M = \{\} \longrightarrow \mathsf{fv}\ \varphi \subseteq \{0\ ..< b\}$ ensures that the formula is satisfied by the aggregate value of an empty M only if there are no grouping variables. Otherwise, infinitely many groups would be labeled with that value, rendering such aggregations non-monitorable.

The decidable predicate mon :: *frm* ⇒ *bool* specifies the monitorable fragment. We omit its formal definition and refer to the earlier descriptions of VeriMon [3,21] for details. Intuitively, mon places restrictions on the formula's structure to ensure that all subformulas have finitely many satisfying valuations. Also, the interval I of every U_I operator must be bounded. A monitor for a monitorable formula can thus compute a finite set of satisfying valuations for every time-point after observing a sufficiently long trace prefix.

3 Non-Recursive Let Operator

We first introduce a non-recursive let operator Let *string* ≔ *frm* in *frm* to the *frm* datatype. The formula Let $p \coloneqq \alpha$ in β associates the formula α with the predicate named p, which may be used in the formula β. We call such a predicate *let-bound*. The operator is

non-recursive: p has the same meaning within α as in the surrounding context (unless it is bound by a nested let in α). Although the non-recursive let operator does not enhance MFOTL's expressiveness, it improves readability (by using descriptive let-bound predicate names), as well as modularity and evaluation efficiency (by sharing subformulas).

Intuitively, the meaning of $\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta$ is the same as that of β after replacing all its predicates of the form $p(as)$ with the formula α, whose free variables have been replaced with the terms as in a capture-avoiding way. The formal syntax does not specify explicitly how α's free variables map to p's arguments. The mapping is induced by the De Bruijn indices: the variable with index 0 becomes the first argument, and so forth. We list the arguments explicitly in examples that use named variables. For instance, the formula $\mathsf{Let}\ p(x) \coloneqq p(x) \wedge \exists y.\ q(x,y)\ \mathsf{in}\ \bullet_{[0,2]}\, p(y)$ should be equivalent to $\bullet_{[0,2]}(p(y) \wedge \exists z.\ q(y,z))$. We achieve this by defining Let's semantics as follows.

$$\mathsf{sat}\ \sigma\ v\ i\ (\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta) = \mathsf{sat}\ (\sigma[p \Rrightarrow \lambda j.\ \mathsf{satrel}\ \sigma\ j\ \alpha])\ v\ i\ \beta$$

We write $\mathsf{satrel}\ \sigma\ j\ \alpha$ as an abbreviation for $\{v.\ \mathsf{sat}\ \sigma\ v\ j\ \alpha \wedge \mathsf{length}\ v = \mathsf{nfv}\ \alpha\}$, i.e., the relation containing the valuations that satisfy α. The function $\mathsf{nfv}\ \alpha$ returns the minimum length of v needed to cover all of α's free variables, i.e., 0 if α is closed and $\mathsf{Max}\ (\mathsf{fv}\ \alpha) + 1$ otherwise. The trace $\sigma[p \Rrightarrow R]$ is the same as the trace σ except that for every time-point i, the database at i maps the predicate name p to $R\ i$, where R has type $nat \Rightarrow data\ list\ set$ and is called a *temporal relation*. Note that the subformula α is not necessarily evaluated at time-point i. Instead, the choice of the time-point is deferred until the predicate p is used within β, which we achieve by updating the entire trace. This supports the intuition behind *unfolding* the let operator $\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta$ described above, especially as subformulas $p(as)$ may occur under temporal operators in β.

Implementation. To evaluate an MFOTL formula on a trace, VeriMon computes a finite set of satisfying valuations (represented by the type *table*) recursively for each subformula. It applies standard table operations such as the natural join ($\bowtie$) and union. Tables are sets of tuples, which are lists of optional *data* values (with missing values denoted by $\bot$) and thus refine valuations. This representation allows us to use lists of the same length for subformulas with different free variables. As with valuations, the variables' De Bruijn indices are used to look up their value in a tuple.

VeriMon processes an unbounded trace incrementally. Its interface consists of two functions $\mathsf{init} :: frm \Rightarrow state$ and $\mathsf{step} :: dbs \times ts\ list \Rightarrow state \Rightarrow (nat \times table)\ list \times state$. The function init initializes the monitor's state (type *state*), and step updates it with a batch of new time-stamped databases to produce a list of new satisfactions. Instead of *db list*, step uses the type $dbs = (string \rightharpoonup table\ list)$ (a partial mapping from *string* to *table list*) to efficiently retrieve all relations (encoded as tables) associated with a predicate name at once. Besides some auxiliary data, *state* stores an *inductive state* of type *sfrm* that mirrors the inductive representation of formulas, augmented with data structures for evaluating temporal operators and buffering intermediate results. Internally $\mathsf{step}\ (dbs, tss)\ st$ calls $\mathsf{eval}\ j\ n\ tss\ dbs\ s_\varphi$, where j is the combined length of the trace prefix including the new batch, $n = \mathsf{nfv}\ \varphi$ for the monitored formula φ, and s_φ is the inductive state, all stored in st. The function eval returns a list of tables with new satisfactions, as well as the updated inductive state. Satisfactions are reported for every time-point in order. They may be delayed if the formula contains future operators.

To evaluate $\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta$, we use the tables with α's satisfactions to evaluate p within β, which requires that the tuples in these tables do not have missing values. Therefore, we require that let operators satisfy $\mathsf{mon}\ (\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta) = (\{0\ ..{<}\ \mathsf{nfv}\ \alpha\} \subseteq \mathsf{fv}\ \alpha \wedge \mathsf{mon}\ \alpha \wedge \mathsf{mon}\ \beta)$. Specifically, the (indices of) α's free variables must not have gaps. We add the constructor $\mathsf{SLet}\ p\ m\ s_\alpha\ s_\beta$ to the inductive state, which stores p, the number $m = \mathsf{nfv}\ \alpha$ of free variables in α, and the states for subformulas α and β. It is initialized by initializing s_α and s_β recursively. The function eval evaluates it as follows.

$$\begin{aligned}&\mathsf{eval}\ j\ n\ tss\ dbs\ (\mathsf{SLet}\ p\ m\ s_\alpha\ s_\beta) =\\ &\quad\big(\underline{\mathsf{let}}\ (xs,\ s'_\alpha) = \mathsf{eval}\ j\ m\ tss\ dbs\ s_\alpha;\ (ys,\ s'_\beta) = \mathsf{eval}\ j\ n\ tss\ (dbs[p \mapsto xs])\ s_\beta\\ &\quad\ \underline{\mathsf{in}}\ (ys,\ \mathsf{SLet}\ p\ m\ s'_\alpha\ s'_\beta)\big)\end{aligned}$$

We write $dbs[p \mapsto xs]$ for the partial mapping dbs updated at p with xs. The recursive call of eval on s_α may return multiple tables in the list xs. Note that step generalizes the original VeriMon interface [3] as it consumes multiple time-stamped databases at once. The generalized interface of eval allows us to pass all tables at once to the recursive call for s_β.

Correctness. We relate the outputs of step and sat to prove our monitor correct. As mentioned earlier, the monitor may delay its output. We precisely characterize its *progress* for a given formula and trace prefix. Intuitively, the progress is the number of time-points that the monitor is able to evaluate given a trace prefix. Progress is a useful tool in the correctness proof as it helps us describe the output *at every time-point*. Moreover, we show below that progress can be made arbitrarily large, which is important for completeness.

Formally, $\mathsf{prog}\ \sigma\ P\ \varphi\ j$ is φ's progress i_φ after reading the first j databases of trace σ. We added the partial mapping P that assigns to every let-bound predicate its own progress, i.e., the progress of the formula defining the predicate. For example, the progress of a predicate p that is not let-bound is j. Otherwise, it is equal to the progress of the formula it is bound to (stored in $P\ p$). The progress of $\alpha\,\mathsf{U}_{[a,b]}\,\beta$ is the smallest i such that $\tau\ \sigma\ i \geq \tau\ \sigma\ (\mathsf{Min}\ \{i_\alpha, i_\beta, j-1\}) - b$. The progress of both $\alpha \wedge \beta$ and $\alpha \vee \beta$ is $\mathsf{Min}\ \{i_\alpha, i_\beta\}$.

The invariant $\mathsf{invar}\ \sigma\ j\ P\ n\ s_\varphi\ \varphi$ relates an inductive state s_φ to the formula φ. The inductive state must reflect the monitor's state after processing the first j databases in the trace σ, assuming that P specifies the let-bound predicates' progress. The parameter n is the length of the tuples stored within s_φ. The invariant is defined inductively over s_φ; we reuse VeriMon's definition for the MFOTL operators and add a case for Let:

$$\frac{\mathsf{invar}\ \sigma\ j\ P\ m\ s_\alpha\ \alpha \quad \mathsf{invar}\ (\sigma[p \Rrightarrow \lambda i.\ \mathsf{satrel}\ \sigma\ i\ \alpha])\ j\ (P[p \mapsto \mathsf{prog}\ \sigma\ P\ \alpha\ j])\ n\ s_\beta\ \beta \quad m = \mathsf{nfv}\ \alpha \quad \{0\ ..{<}\ m\} \subseteq \mathsf{fv}\ \alpha}{\mathsf{invar}\ \sigma\ j\ P\ n\ (\mathsf{SLet}\ p\ m\ s_\alpha\ s_\beta)\ (\mathsf{Let}\ p \coloneqq \alpha\ \mathsf{in}\ \beta)}$$

The first two premises restrict the subformula states s_α and s_β, where s_β reflects the evaluation of β on the modified trace, and p's progress is that of α. The premise $m = \mathsf{nfv}\ \alpha$ enforces that m is equal to p's arity, and $\{0\ ..{<}\ m\} \subseteq \mathsf{fv}\ \alpha$ is the constraint from mon.

Our extensions preserve the monitor's correctness: we formally proved the theorem below, which characterizes the monitor's eval function. The theorem is stated here for the empty progress mapping $\varnothing$, which must be generalized in the proof (as P changes in the above rule). Let δ be a natural number and φ be a monitorable formula with $n = \mathsf{nfv}\ \varphi$. The function the maps the optional value $\langle x \rangle$ to x and $\bot$ to some unspecified value.

Theorem 1. *(a)* $\mathsf{invar}\ \sigma\ 0\ \varnothing\ n\ s_\varphi^0\ \varphi$ *for the initial state* s_φ^0*. (b) Suppose that* s_φ *satisfies* $\mathsf{invar}\ \sigma\ j\ \varnothing\ n\ s_\varphi\ \varphi$ *and that dbs contains all relations from* σ *for the indices in the list* $js = [j\,..{<}\,j+\delta]$*. Then* $(xs, s'_\varphi) = \mathsf{eval}\ (j+\delta)\ n\ (\mathsf{map}\ (\tau\ \sigma)\ js)\ dbs\ s_\varphi$ *satisfies* $\mathsf{invar}\ \sigma\ (j+\delta)\ \varnothing\ n\ s'_\varphi\ \varphi$*, and the i-th table in the list xs, for* $\mathsf{prog}\ \sigma\ \varnothing\ \varphi\ j \leq i < \mathsf{prog}\ \sigma\ \varnothing\ \varphi\ (j+\delta)$*, contains (only) all tuples v of length n satisfying* $\mathsf{sat}\ \sigma\ (\mathsf{map}\ \mathsf{the}\ v)\ \sigma\ i\ \varphi$*.*

Soundness follows immediately from Thm. 1, whereas completeness additionally requires the aforementioned property that any progress can be reached by making the trace prefix long enough, which we also proved for our modified progress function:

Theorem 2. *If* $\mathsf{mon}\ \varphi$*, then for all i there exists a j such that* $\mathsf{prog}\ \sigma\ \varnothing\ \varphi\ j \geq i$*.*

4 Past-Recursive Let Operator

It is well-known that first-order logic (FOL) cannot express certain queries, notably the transitive closure of a binary relation. This remains true when restricted to finite structures [18]. Although MFOTL is rather different from ordinary FOL, we conjecture that it cannot express transitive closure either. This hampers its ability to model hierarchies of unbounded depth. Moreover, recursive patterns are sometimes the most natural way to express certain specifications. We describe an extension of MFOTL that can encode a "temporally directed" form of transitive closure and other recursive patterns.

Specifically, we introduce another let operator in which the predicate may refer to itself recursively. The intended semantics is that of a fixpoint, i.e., the predicate p defined by a formula α should be interpreted by a temporal relation that is equal to the evaluation of α under that interpretation of p. The fixpoint might not always exist or it might not be unique. Therefore, different fixpoint operators have been studied in the context of nontemporal logics and query languages [1]. For instance, it is common to require that all recursive occurrences of p in its defining formula are positive, i.e., under an even number of negations. This ensures monotonicity and hence the existence of a least fixpoint.

MFOTL's future operators are interpreted over infinite traces. This poses a new challenge for monitoring recursively defined predicates, even if we restrict our attention to positive formulas. Consider the recursive definition of p by $q \vee \bigcirc_{[0,\infty]} p$, where q is a predicate from the trace. Although $q \vee \bigcirc_{[0,\infty]} p$ is monitorable (at most one additional time-point must be known to evaluate it), the recursive definition of p is equivalent to $\Diamond_{[0,\infty]} q$ under the least fixpoint semantics. However, $\Diamond_{[0,\infty]} q$ is not monitorable, as one might need the entire, infinite trace to evaluate it. Therefore, we focus on a fragment where every recursive occurence of p must be *strictly in the past*. This guarantees a unique fixpoint even if the defining formula is not monotone, so the predicate may occur negatively as well.

The syntax of our past-recursive let operator is similar to the one of Let: we add the constructor LetPast *string* $:=$ *frm* in *frm* to the *frm* datatype. However, the semantics is different (Section 4.1). The restriction to strictly past recursion is enforced by a syntactic monitorability condition that is checked by mon. Consider the formula LetPast $p := \alpha$ in β. Intuitively, every recursive occurrence of p in α must be *guarded* by at least one strictly past operator, and there must be no future operator on the path from the occurrence to α's root. We *do* allow future operators in the other parts of α, though.

We give examples of LetPast in Section 4.2. The evaluation of LetPast requires an extension of VeriMon's algorithm (Section 4.3), which we also formally prove correct.

```
datatype recSafety = U | P | NF | A

fun (∗) :: recSafety ⇒ recSafety ⇒ recSafety
where
  U ∗ _ = U
| _ ∗ U = U
| A ∗ _ = A
| _ ∗ A = A
| P ∗ _ = P
| _ ∗ P = P
| NF ∗ NF = NF

fun slp :: string ⇒ frm ⇒ recSafety where
  slp p (q(as)) = (if p = q then NF else U)
| slp p (Let q := α in β) =
    (slp q β ∗ slp p α) ⊔ (if p = q then U else slp p β)
| slp p (LetPast q := α in β) =
    (if p = q then U else (slp q β ∗ slp p α) ⊔ slp p β)
| slp p (t1 ∘ t2) = U                    | slp p (y ← Ω(t;b) φ) = slp p φ
| slp p (¬φ) = slp p φ                   | slp p (∃φ) = slp p φ
| slp p (α ∨ β) = slp p α ⊔ slp p β
| slp p (α ∧ β) = slp p α ⊔ slp p β
| slp p (●I φ) = P ∗ slp p φ             | slp p (○I φ) = A ∗ slp p φ
| slp p (α SI β) = slp p α ⊔ ((if 0 ∈I I then NF else P) ∗ slp p β)
| slp p (α UI β) = A ∗ (slp p α ⊔ slp p β)
```

Fig. 2. Auxiliary definitions for the syntactic restriction on LetPast

4.1 Semantics

The semantics of the past-recursive let operator is defined by the equation

$$\mathsf{sat}\,\sigma\,v\,i\,(\mathsf{LetPast}\,p := \alpha\,\mathsf{in}\,\beta) = \mathsf{sat}\,(\sigma[p \Rrightarrow \mathsf{recp}\,(\lambda R\,j.\,\mathsf{satrel}\,(\sigma[p \Rrightarrow R])\,j\,\alpha)])\,v\,i\,\beta$$

We evaluate β at the same time-point i as the recursive let operator using an appropriately updated trace. The temporal relation assigned to p is computed by the combinator recp:

fun recp :: $((nat \Rightarrow data\ list\ set) \Rightarrow nat \Rightarrow data\ list\ set) \Rightarrow nat \Rightarrow data\ list\ set$ **where**
$\mathsf{recp}\,f\,v\,i = f\,(\lambda j.\ \underline{\text{if}}\ j < i\ \underline{\text{then}}\ \mathsf{recp}\,f\,j\ \underline{\text{else}}\ \{\})\,i$

The argument f is a function that transforms temporal relations, and recp f returns again a temporal relation. Intuitively, recp f evaluates to the fixpoint $f\,(\mathsf{recp}\,f)$, except that $f\,R\,i$ can only access time-points of R before i. For all other time-points $j \geq i$, the relation $R\,j$ is empty. The combinator recp is well-defined because i is a natural number; the recursive call recp $f\ j$ affects the result only if $j < i$ and hence we can prove termination using i as a variant. For the semantics of LetPast, we choose $f\,R\,i = \mathsf{satrel}\,(\sigma[p \Rrightarrow R])\,i\,\alpha$, i.e., the satisfactions of α with p mapped to f's argument R, to which recp supplies the result of the recursive evaluation (up to but excluding i).

Our definition of sat is total: it gives meaning to every formula. This includes formulas LetPast $p := \alpha$ in β where p occurs in α without a past guard or under a future operator. However, the semantics behaves unexpectedly in such cases. For example, LetPast $p := (q \vee \bigcirc_{[0,\infty]}\, p)$ in p is equivalent to q. Our monitor therefore requires properly guarded formulas. Not only does this avoid confusion about the semantics, it also simplifies the implementation because the monitor need not eliminate unguarded occurrences.

Next, we describe the formalization of the syntactic restriction. The idea is to determine for every predicate whether it is used strictly in the past by analyzing the formula recursively. The datatype *recSafety* (Fig. 2) represents the possible outcomes. U(nused) means that a predicate does not occur in the formula. P(ast) means that it is evaluated at strictly earlier time-points, whereas NF (Non-Future) additionally allows the current time-point. A(ny) covers all remaining cases. The linear order $<$ on *recSafety* is induced by U $<$ P $<$ NF $<$ A. Its reflexive closure $\leq$ corresponds to implication. For example, if the predicate p is unused (U), it is clearly evaluated at earlier time-points only (P). The least upper bound $x \sqcup y$ with respect to $\leq$ corresponds to logical disjunction.

The function $\mathsf{slp}\ p\ \varphi$ (Fig. 2) analyzes the past-guardedness of a predicate p in a formula φ. It uses a composition operator $y * x$ on *recSafety*. The patterns in the definition of $*$ should be matched sequentially from top to bottom; e.g., $\mathsf{A} * \mathsf{U}$ is equal to U. Intuitively, $y * x$ describes the guardedness of a predicate that is x-used in some subformula, which is then y-used. For example, $\mathsf{slp}\ p\ (\bullet_I \varphi) = \mathsf{P} * \mathsf{slp}\ p\ \varphi$ because φ and all occurences of p therein are evaluated at time-points that are strictly in the past relative to $\bullet_I \varphi$. Note that we make a case distinction for $\alpha\, \mathsf{S}_I \beta$: if the interval I excludes zero, β is always evaluated strictly in the past. Future operators always result in A if p is used in an operand.

Finally, we define the mon predicate for the recursive let operator:

$$\mathsf{mon}\ (\mathsf{LetPast}\ p \coloneqq \alpha\ \mathsf{in}\ \beta) = (\mathsf{slp}\ p\ \alpha \leq \mathsf{P} \wedge \{0\ ..{<}\ \mathsf{nfv}\ \alpha\} \subseteq \mathsf{fv}\ \alpha \wedge \mathsf{mon}\ \alpha \wedge \mathsf{mon}\ \beta)$$

The only difference to Let is the restriction of p's occurrences in α via slp, which is generally an over-approximation. For example, $\mathsf{slp}\ p\ (\bullet_I \bullet_I \bigcirc_I p) = \mathsf{A}$ even though p is evaluated at strictly earlier time-points. Therefore, some instances of LetPast that our algorithm could evaluate correctly are not considered to satisfy mon. We plan to replace *recSafety* with a more precise lattice in future work.

4.2 Examples

Temporal Operators. We first show that the non-metric S operator can be reduced to LetPast and $\bullet$. (We omit the interval subscripts if the interval is $[0,\infty]$.) Using the special $\mathsf{ts}(t)$ predicate, which is true iff t is the current time-stamp, we can also express the metric version. This example serves to gently illustrate the semantics of LetPast. In general, formulas are more readable if they are directly expressed in terms of S, and monitoring can be more efficient. Below we give further examples in which LetPast adds expressiveness.

Let α and β be two monitorable MFOTL formulas with free variables $\mathsf{fv}\ \alpha$ and $\mathsf{fv}\ \beta$, respectively. The formula $\alpha\, \mathsf{S}\, \beta$ is monitorable only if $\mathsf{fv}\ \alpha \subseteq \mathsf{fv}\ \beta$, so let us assume that, too. The following unfolding of S's semantics is well-known:

$$\mathsf{sat}\ \sigma\ v\ i\ (\alpha\, \mathsf{S}\, \beta) \iff \mathsf{sat}\ \sigma\ v\ i\ \beta \vee \big(\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge i > 0 \wedge \mathsf{sat}\ \sigma\ v\ (i-1)\ (\alpha\, \mathsf{S}\, \beta)\big) \quad (1)$$

As the unfolding recursively evaluates the formula at the previous time-point, we can directly translate it into a recursive let operator: $\varphi_{\mathsf{S}} \equiv \mathsf{LetPast}\ s(\overline{x}) \coloneqq \psi\ \mathsf{in}\ s(\overline{x})$, where $\psi \equiv \beta \vee (\alpha \wedge \bullet\, s(\overline{x}))$. The predicate name s must be fresh, i.e., it must not occur in α nor β. The variable list $\overline{x}$ enumerates $\mathsf{fv}\ \beta$. The formula φ_{S} is monitorable because $\bullet\, s(\overline{x})$ is clearly past-guarded, and hence $\mathsf{slp}\ s\ \psi = \mathsf{P}$. (We also need $\mathsf{fv}\ \beta = \{0\ ..{<}\ \mathsf{nfv}\ \beta\}$, which can be achieved by renaming variables in α and β.) Let us analyze the semantics of φ_{S}:

$$\begin{aligned}
\mathsf{sat}\ \sigma\ v\ i\ \varphi_{\mathsf{S}} &\iff \mathsf{sat}\ (\sigma[s \Rrightarrow \mathsf{recp}\ \underbrace{(\lambda R\ j.\ \mathsf{satrel}\ (\sigma[s \Rrightarrow R])\ j\ \psi)}_{=f_\psi}])\ v\ i\ (s(\overline{x})) \\
&\iff v \in \mathsf{recp}\ f_\psi\ i \\
&\iff \mathsf{sat}\ (\sigma[s \Rrightarrow \lambda j.\ \underline{\mathsf{if}}\ j < i\ \underline{\mathsf{then}}\ \mathsf{recp}\ f_\psi\ j\ \underline{\mathsf{else}}\ \{\}])\ v\ i\ \psi \\
&\overset{(*)}{\iff} \mathsf{sat}\ \sigma\ v\ i\ \beta \vee \big(\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge i > 0 \\
&\qquad \wedge v \in (\underline{\mathsf{if}}\ i-1 < i\ \underline{\mathsf{then}}\ \mathsf{recp}\ f_\psi\ (i-1)\ \underline{\mathsf{else}}\ \{\})\big) \\
&\iff \mathsf{sat}\ \sigma\ v\ i\ \beta \vee \big(\mathsf{sat}\ \sigma\ v\ i\ \alpha \wedge i > 0 \wedge \mathsf{sat}\ \sigma\ v\ (i-1)\ \varphi_{\mathsf{S}}\big)
\end{aligned}$$

These equations hold for all valuations v of length $\mathsf{nfv}\ \beta$ and if the variables $\overline{x}$ are ordered by their De Bruijn indices. Step $(*)$ exploits the freshness of s with respect to α and β,

which allows us to replace $\sigma[s \Rrightarrow \ldots]$ by σ. The equations result in the same unfolding as (1). Hence, we can prove the semantic equivalence of φ_{S} and $\alpha\,\mathsf{S}\,\beta$ by induction on i.

The following *SinceLet* formula encodes $\alpha\,\mathsf{S}_{[a,b]}\,\beta$. Other encodings exist, however.

$$\mathsf{LetPast}\ s(\overline{x},t) \coloneqq (\beta \wedge \mathsf{ts}(t)) \vee (\alpha \wedge \bullet\, s(\overline{x},t))\ \mathsf{in}\ \exists t,u.\ s(\overline{x},t) \wedge \mathsf{ts}(u) \wedge a \leq u-t \wedge u-t \leq b$$

Here, t and u are fresh variables, where t records the time-stamp of the past satisfaction of β, whereas u is the time-stamp at which we evaluate *SinceLet*. The subformula $a \leq u-t \wedge u-t \leq b$ corresponds to $\mathsf{T}\ \sigma\ j - \mathsf{T}\ \sigma\ i \in_{\mathcal{I}} [a,b]$, which is part of $\mathsf{S}_{[a,b]}$'s semantics (Fig. 1).

Temporally-Directed Transitive Closure. We proceed by showing that LetPast can compute a temporally-directed transitive closure over events observed at a sequence of distinct time-points. Hence, we assume that the trace contains a single event at every time-point. The closure is directed in the sense that the transitive chains can only be extended by *newer* events. We consider the following two types of events from [14]: $r(y,x,d)$ denotes that process y reports some data d to another process x, and $s(x,y)$ denotes that process x spawns process y. The *Spawn* formula

$$\mathsf{LetPast}\ p(u,v) \coloneqq s(u,v) \vee (\bullet\, p(u,v)) \vee (\exists t.\ (\bullet\, p(u,t)) \wedge s(t,v))\ \mathsf{in}\ r(y,x,d) \wedge \neg p(x,y)$$

encodes violations of the property that whenever process y sends some data d to a process x, denoted as $r(y,x,d)$, then there was a chain of process spawns: $s(x,x_1), s(x_1,x_2), \ldots, s(x_k,y)$, occurring in this order in the trace. In other words, a process may only send data to its "ancestors". To check this property, a monitor needs to compute the (temporally-directed) transitive closure $p(u,v)$ of the relation s. The definition of the closure has two recursive predicate instances with different arguments. The *Spawn* formula is inspired by a similar one used to evaluate the DejaVu monitor [14]. Unlike DejaVu, we do not require the formula to be closed and thus leave the variables x, y, and d free.

The *Trans* formula

$$\begin{aligned}\mathsf{LetPast}\ p(u,v) \coloneqq\ & s(u,v) \vee (\bullet\, p(u,v)) \vee \\ & (\exists t.\ (\bullet\, p(u,t)) \wedge s(t,v)) \vee (\exists t.\ s(u,t) \wedge (\bullet\, p(t,v))) \vee \\ & (\exists t,t'.\ (\bullet\, p(u,t)) \wedge s(t,t') \wedge (\bullet\, p(t',v)))\quad \mathsf{in}\quad r(y,x,d) \wedge \neg p(x,y)\end{aligned}$$

encodes violations of the same property as *Spawn* even if $s(x,x_1), s(x_1,x_2), \ldots, s(x_k,y)$ are received by the monitor out-of-order, i.e., they do not occur in this order in the trace.

We can interpret the events $s(x,y)$ as edges in a directed graph and the predicate $p(x,y)$ in *Trans* as computing the reachability of vertices in the directed graph. We also extend the directed edges $s(x,y)$ with a weight w to $s^+(x,y,w)$. Then the *Trans*$^+$ formula

$$\begin{aligned}\mathsf{LetPast}\ p(u,v,w) \coloneqq\ & s^+(u,v,w) \vee (\bullet\, p(u,v,w)) \vee \\ & (\exists t,w_1,w_2.\ (\bullet\, p(u,t,w_1)) \wedge s^+(t,v,w_2) \wedge w = w_1 + w_2) \vee \\ & (\exists t,w_1,w_2.\ s^+(u,t,w_1) \wedge (\bullet\, p(t,v,w_2)) \wedge w = w_1 + w_2) \vee \\ & (\exists t,t',w_1,w_2,w_3.\ (\bullet\, p(u,t,w_1)) \wedge s^+(t,t',w_2) \wedge (\bullet\, p(t',v,w_3)) \wedge \\ & \qquad w = w_1 + w_2 + w_3)\ \mathsf{in} \\ \mathsf{Let}\ m(u,v,w) \coloneqq\ & w \leftarrow \mathsf{MIN}(w;u,v).\ p(u,v,w)\quad \mathsf{in}\quad m(x,y,w) \wedge \neg(\bullet\, m(x,y,w))\end{aligned}$$

yields all pairs of vertices x, y and the length w of the shortest path from x to y whenever y becomes reachable from x or the length of the shortest path changes. The relation

$s^+(x,y,w)$ can itself be obtained by evaluating a more complex temporal formula, e.g., $s^+(x,y,w) \equiv e(x,y,w) \wedge \neg \Diamond_{[0,10]}\, d(x,y)$ with the following two types of events: $e(x,y,w)$ denotes an edge from x to y with weight w; $d(x,y)$ denotes deletion of the edge from x to y. The *eventually* operator $\Diamond_I\, \varphi$ abbreviates $(\exists x.\ x = x)\, \mathsf{U}_I\, \varphi$. Such a relation $s^+(x,y,w)$ contains all edges that are not revoked within 10 time units after receiving $e(x,y,w)$. We could use the non-recursive let operator $\mathsf{Let}\ s^+(x,y,w) \coloneqq e(x,y,w) \wedge \neg \Diamond_{[0,10]}\, d(x,y)$ to precompute the relation and use it when evaluating the recursive let operator in Trans^+.

As another application of future operators under LetPast, recall our introductory example. Suppose that hosts in a network communicate with each other and with the outside world: $\mathit{comm}(\mathit{src},\mathit{dest})$ indicates that host *src* sends a message to host *dest*; $\mathit{in}(r,h)$ and $\mathit{out}(h,r)$ indicate that the host h receives or sends traffic from or to an IP address in the range r, respectively. The hosts are equipped with an intrusion detection system (IDS), whose alerts are denoted by $\mathit{ids}(h)$. We say that a host h is *tainted* by an address range r iff there is a chain of communication from r to h and all hosts on the chain (including h) trigger an IDS alert within one hour after communicating with the previous host. The formula

$$\mathsf{LetPast}\ \mathit{taint}(r,h) \coloneqq \big(\big(\mathit{in}(r,h) \vee \exists h'.\ (\bullet\, \mathit{taint}(r,h')) \wedge \mathit{comm}(h',h)\big) \wedge \Diamond_{[0,1\mathrm{h}]}\, \mathit{ids}(h)\big) \vee (\bullet\, \mathit{taint}(r,h))\quad \mathsf{in}\quad \mathit{taint}(r,h) \wedge \mathit{out}(h,r)$$

is true whenever a host communicates back to the IP range by which it was tainted.

Periodic Behavior. Suppose that we monitor a boolean signal $b(x)$, parametrized by an integer parameter x, between the user's $\mathit{start}(x)$ and $\mathit{stop}(x)$ commands. An arbitrary amount of time may pass between these two commands. Our task is to detect periodic activations of $b(x)$, with a fixed period $t > 0$ and error tolerance $0 \le \varepsilon < t$. We shall ignore positive noise in $b(x)$, i.e., additional activations besides the periodic ones.

Let us make the task more precise. An alarm must be raised at time-point i_n iff there exist time-points $i_0 < i_1 < \cdots < i_n$ such that $\mathit{start}(x)$ holds at i_0, $\mathit{stop}(x)$ holds at i_n, and $b(x)$ holds at all i_k for $1 \le k \le n-1$. Moreover, the difference of time-stamps for adjacent time-points i_k and i_{k+1}, where $1 \le k \le n-2$, must be in the interval $[t-\varepsilon, t+\varepsilon]$; the differences for the pairs i_0, i_1 and i_{n-1}, i_n must each be at most $t+\varepsilon$.

Our first attempt *PB* to formalize the alarm condition without recursion is

$$\mathit{stop}(x) \wedge \big(\blacklozenge_I(\mathit{start}(x) \vee b(x))\big) \wedge \big((b(x) \longrightarrow (\blacklozenge_I\, \mathit{start}(x)) \vee (\blacklozenge_J\, b(x)))\, \mathsf{S}\, \mathit{start}(x)\big)$$

where $I = [0, t+\varepsilon]$, $J = [t-\varepsilon, t+\varepsilon]$, and $\blacklozenge_K\, \varphi$ abbreviates $(\exists x.\ x = x)\, \mathsf{S}_K\, \varphi$. This formula follows an inductive approach: every $b(x)$ between $\mathit{start}(x)$ and $\mathit{stop}(x)$ must be preceded by $b(x)$ or $\mathit{start}(x)$, with the appropriate time difference. However, *PB* does not ignore noise, as adding $b(x)$ events to the trace may silence an alarm. For example, let $t = 10$, $\varepsilon = 0$, and σ be a trace starting with $(\{\mathit{start}(1)\}, 0)$, $(\{b(1)\}, 10)$, $(\{\mathit{stop}(1)\}, 20)$. We write $\{p(1), p(2)\}$ for the database where the predicate p holds for 1 and 2. On σ, *PB* is true at the third time-point. Inserting a database $\{b(1)\}$ with time-stamp 15 falsifies *PB* at the now fourth time-point, although the trace still satisfies the natural language description.

The following *PBLet* formula expresses the intended condition using LetPast:

$$\mathsf{LetPast}\ \mathit{periodic}(x) \coloneqq \mathit{start}(x) \vee \big(b(x) \wedge \big((\blacklozenge_I\, \mathit{start}(x)) \vee (\blacklozenge_J\, \mathit{periodic}(x))\big)\big)\ \mathsf{in}\ \mathit{stop}(x) \wedge \blacklozenge_I\, \mathit{periodic}(x)$$

This example depends crucially on the flexible past guards we support: here, the recursion goes through $\blacklozenge$ with an interval constraint. Note that $0 \notin J$ because we assumed $\varepsilon < t$.

As another example of periodic behavior, we analyze an integer-valued $signal(y)$ between the (now non-parametric) commands *start* and *stop*. We aim to discover whether $signal(y)$ is piecewise constant, with the constant segments being exactly t time units long. Moreover, the signal's values for subsequent segments must differ by at most δ. The next formula uses the general S operator as the recursion guard to capture this property.

$$\begin{aligned}&\mathsf{LetPast}\ segment(y) \coloneqq \exists z.\ signal(y) \wedge \Big(\big((\bullet\, signal(z))\ \mathsf{S}_{[0,t]}\ (signal(z) \wedge \bullet\, start)\big) \vee \\ &\qquad \big((\bullet\, signal(z))\ \mathsf{S}_{[t,t]}\ segment(z)\big)\Big) \wedge -\delta \le y - z \wedge y - z \le \delta\ \mathsf{in} \\ &stop \wedge \exists y.\ ((\bullet\, signal(y))\ \mathsf{S}_{[0,t]}\ segment(y))\end{aligned}$$

Turing Machines. Every MFOTL formula can be viewed as a function on traces, where the function's output is the set of satisfying valuations, either at a fixed or at all time-points. VeriMon's monitorable fragment guarantees that one can compute the valuation at every time-point. Thus, monitorable formulas correspond to computable functions. If we give up on the requirement that the function's output must be available at a fixed time-point, the past-recursive let operator is expressive enough to simulate arbitrary Turing machines (TM). This is not a contradiction: we simulate a single TM step at every time-point, and there is an infinite supply of time-points. Running the monitor on a configuration that does not halt will never produce an output, i.e., a nonempty set of satisfying valuations.

Let $M = \langle \Sigma, b, Q, q_0, q_f, \delta \rangle$ be a deterministic TM with tape alphabet Σ, blank symbol $b \in \Sigma$, control states Q, initial state $q_0 \in Q$, final state $q_f \in Q$, and transition function $\delta \in (Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\})$. Whenever the machine is in state q_1 and reads the symbol s_1, it enters state q_2, writes the symbol s_2, and moves the head by m tape cells to the right, where $\delta(q_1, s_1) = \langle q_2, s_2, m \rangle$. Without loss of generality, we assume that Σ and Q are finite subsets of the integers. We simulate M using the formula φ_M shown below.

$$\begin{aligned}&\mathsf{LetPast}\ cfg(q,i,s) \coloneqq \\ &\quad \mathsf{Let}\ cfg(q,i,s) \coloneqq \bullet\, cfg(q,i,s)\ \mathsf{in} \\ &\quad \mathsf{Let}\ head(q,s) \coloneqq cfg(q,0,s) \vee \big(\neg(\exists x, z.\ cfg(x,0,z)) \wedge (\exists y, z.\ cfg(q,y,z)) \wedge s = b\big)\ \mathsf{in} \\ &\quad \big(input(i,s) \wedge q = q_0\big) \vee \\ &\qquad \bigvee_{\substack{q_1, s_1 \\ \delta(q_1,s_2) = \langle q_2, s_2, m \rangle}} \Big(head(q_1,s_1) \wedge q = q_2 \wedge \big((i = -m \wedge s = s_2) \vee \\ &\qquad\qquad (\exists j.\ cfg(q_1,j,s) \wedge j \neq 0 \wedge i = j - m)\big)\Big)\ \mathsf{in}\ cfg(q_f,i,s)\end{aligned}$$

The idea is that *cfg* represents the current configuration of the TM. Specifically, $cfg(q,i,s)$ holds if the machine is in control state q and the tape contains the symbol s in the ith cell to the right of the head (i may be negative). Note that we use nested, non-recursive let operators to abbreviate repeated subformulas. In the body of $\mathsf{Let}\ cfg(q,i,s) \coloneqq \bullet\, cfg(q,i,s)\ \mathsf{in}\ \ldots$, the predicate *cfg* refers to the previous configuration. The predicate *head* provides the current state and the symbol under the head. Its definition extends the tape by a blank symbol if necessary. The simulation is started at time-point 0 by providing the tape's initial content in the predicate *input*, which must include the cell $input(0, s_0)$ with the symbol s_0 under the head's initial position. If and only if M halts on this input, there exists a time-point i at which φ_M is satisfied by at least one valuation (i, s). Moreover, the satisfying valuations at i represent the final state of the tape.

4.3 Algorithm

The restriction to past-guarded recursion allows for an efficient evaluation algorithm for LetPast formulas. It is efficient because no fixpoint iteration is required at individual time-points. To evaluate LetPast $p \coloneqq \alpha$ in β, we first try to evaluate α for as many time-points as possible and then use the results to interpret p in β. This part is the same as for the non-recursive Let, but the evaluation of α itself differs. The syntactic monitorability condition guarantees that α at time-point i depends on the predicate p only for time-points strictly less than i. Specifically, we have defined mon (LetPast $p \coloneqq \alpha$ in β) such that the progress of α's evaluation does not depend on p's progress beyond time-point $i-1$. Therefore, we can evaluate α at time-point 0 without providing any table for p, then use the result to evaluate α at time-point 1, and so forth.

There are two cases that require care. First, if α contains future operators, multiple time-points may be evaluated at once. The above process must then be repeated within a single monitor step. Second, if α contains no future operators, α is evaluated at all time-points $i < j$, where j is the current trace prefix length. We could then attempt to evaluate α once more at time-point j using the table computed at $j-1$ for p. However, this would not yield any further tables because all occurrences of p are below at least one past operator that tries to access the time-stamp at time-point j, which is not yet known. Therefore, this last evaluation attempt would needlessly traverse the formula state. We optimize this case and buffer α's result at time-point $j-1$ until the next input database arrives.

It is crucial that the evaluation of a recursive let does not get stuck waiting for tables that it needs to produce itself. Therefore, all operators that are strictly past-guarding as defined by slp (Fig. 2) must be well-behaved: the evaluation algorithm must compute a result at time-point $i < j$ even if the operands' results are available only for time-points $i' < i$. In particular, S_I without 0 in the interval is considered strictly past-guarding. We have modified VeriMon's evaluation algorithm for $\alpha\, \mathsf{S}_I\, \beta$ to achieve this behavior.

The inductive state SLetPast $p\ m\ s_\alpha\ s_\beta\ i\ buf$ for a recursive let operator extends SLet with a counter $i :: nat$, which tracks the progress of p as observed by s_α, and an optional buffer $buf :: table\ option$. The meaning of the other arguments is the same as for SLet. In the initial state, i is zero and buf is $\bot$. Let the function list_opt map $\bot$ to $[]$ and $\langle x \rangle$ to $[x]$, where $\langle x \rangle$ is the embedding of x into the *option* type. A single monitor step updates the state as follows (see Section 3 for a description of eval's interface):

$$\begin{aligned}
&\mathsf{eval}\ j\ n\ tss\ dbs\ (\mathsf{SLetPast}\ p\ m\ s_\alpha\ s_\beta\ i\ buf) = \\
&\quad \big(\underline{\mathsf{let}}\ (xs, s'_\alpha, i', buf') = \mathsf{eval}_{\mathsf{LP}}\ j\ m\ tss\ dbs\ p\ []\ s_\alpha\ i\ (\mathsf{list_opt}\ buf); \\
&\qquad (ys, s'_\beta) = \mathsf{eval}\ j\ n\ tss\ (dbs[p \mapsto xs])\ s_\beta \\
&\quad \underline{\mathsf{in}}\ (ys, \mathsf{SLetPast}\ p\ m\ s'_\alpha\ s'_\beta\ i'\ buf')\big)
\end{aligned}$$

The heavy lifting is performed by $\mathsf{eval}_{\mathsf{LP}}$, which is mutually recursive with eval. We forward relevant variables from eval. The accumulator $xs :: table\ list$ collects s_α's results.

$$\begin{aligned}
&\mathsf{eval}_{\mathsf{LP}}\ j\ m\ tss\ dbs\ p\ xs\ s_\alpha\ i\ buf = \\
&\quad \big(\underline{\mathsf{let}}\ (xs', s'_\alpha) = \mathsf{eval}\ j\ m\ tss\ (dbs[p \mapsto buf])\ s_\alpha;\ \ i' = i + \mathsf{length}\ buf \\
&\quad \underline{\mathsf{in}}\ \big(\underline{\mathsf{case}}\ xs'\ \underline{\mathsf{of}}\ [] \Rightarrow (xs, s'_\alpha, i', \bot) \\
&\qquad\qquad \mid x \# _ \Rightarrow (\underline{\mathsf{if}}\ i' + 1 \geq j\ \underline{\mathsf{then}}\ (xs\ @\ xs', s'_\alpha, i', \langle x \rangle) \\
&\qquad\qquad\qquad\qquad \underline{\mathsf{else}}\ \mathsf{eval}_{\mathsf{LP}}\ m\ j\ []\ (\mathsf{clear_dbs}\ dbs)\ p\ (xs\ @\ xs')\ s'_\alpha\ i'\ xs')\big)\big)
\end{aligned}$$

First, $\mathsf{eval}_{\mathsf{LP}}$ evaluates s_α with *dbs* updated at p using the current buffer, which may be empty. Since i tracks p's progress, we then increase its new value i' by the length of *buf*. The evaluation results in a list xs' of tables and a new state s'_α. We continue to iterate $\mathsf{eval}_{\mathsf{LP}}$ only if two conditions are met: xs' must be nonempty, as otherwise there is no new data to evaluate s'_α on, and $i'+1$ must be less than the current input prefix length. The latter condition serves as an obvious termination criterion, although it is stricter than necessary. We could perform an additional iteration in the case that $i'+1=j$. However, such an iteration would never produce new results because the past operators guarding p can only be evaluated further if there are new time-stamps. Therefore, we optimize this case by choosing the stricter condition. If we continue the iteration, we append xs' to the accumulator *xs*. Moreover, we clear *tss* and *dbs* because all tables from the new input database have already been processed by the first call to eval. Specifically, the function clear_dbs *dbs* updates *dbs* at all points at which it is defined to an empty list.

We illustrate our algorithm with an example, tracing the computations of eval and $\mathsf{eval}_{\mathsf{LP}}$. We evaluate $\mathsf{LetPast}\, p(x) \coloneqq q(x) \vee \bullet p(x)\ \mathsf{in}\ p(x)$, which has the same semantics as $\blacklozenge_{[0,\infty]} q(x)$, on a prefix with two time-points at time-stamps 0 and 3. We omit details about the subformulas' states, as well as brackets around singleton lists, i.e., $[1]$ is displayed as 1. Let $dbs_0 = \{q \mapsto [\{1\},\{2\}]\}$ be the content of the trace prefix.

$$
\begin{aligned}
&\mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[0,3]\ dbs{:}dbs_0\ s_\varphi{:}(\mathsf{SLetPast}\ p\ 1\ \alpha_0\ \beta_0\ 0\ \bot)\\
&\mid \mathsf{eval}_{\mathsf{LP}}\ j{:}2\ m{:}1\ tss{:}[0,3]\ dbs{:}dbs_0\ p{:}p\ xs{:}[]\ s_\alpha{:}\alpha_0\ i{:}0\ buf{:}[]\\
&\mid\ \mid \mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[0,3]\ dbs{:}(dbs_0[p \mapsto []])\ s_\varphi{:}\alpha_0 && = ([\{1\}],\alpha_1)\\
&\mid\ \mid \mathsf{eval}_{\mathsf{LP}}\ j{:}2\ m{:}1\ tss{:}[]\ dbs{:}\{q \mapsto []\}\ p{:}p\ xs{:}[\{1\}]\ s_\alpha{:}\alpha_1\ i{:}0\ buf{:}[\{1\}]\\
&\mid\ \mid\ \mid \mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[]\ dbs{:}\{p \mapsto [\{1\}], q \mapsto []\}\ s_\varphi{:}\alpha_1 && = ([\{1,2\}],\alpha_2)\\
&\mid\ \mid\ \mid \text{iteration stops because } i'=1 \text{ and hence } i'+1=2 \geq j=2\\
&\mid\ \mid = ([\{1\},\{1,2\}],\alpha_2,1,\langle\{1,2\}\rangle)\\
&\mid = ([\{1\},\{1,2\}],\alpha_2,1,\langle\{1,2\}\rangle)\\
&\mid \mathsf{eval}\ j{:}2\ n{:}1\ tss{:}[0,3]\ dbs{:}(dbs_0[p \mapsto [\{1\},\{1,2\}]])\ s_\varphi{:}\beta_0 && = ([\{1\},\{1,2\}],\beta_2)\\
&= ([\{1\},\{1,2\}],\mathsf{SLetPast}\ p\ 1\ \alpha_2\ \beta_2\ 1\ \langle\{1,2\}\rangle)
\end{aligned}
$$

Correctness. We extended the correctness proof of eval (Thm. 1) to cover the new state constructor SLetPast. The added case differs from the one for the non-recursive let in that $\mathsf{eval}_{\mathsf{LP}}$ is used to evaluate the first subformula. The proof also required additional invariants for the i and *buf* arguments of SLetPast, as well as a characterization of LetPast's progress. Recall that progress describes the number of time-points that the monitor is able to evaluate given a trace prefix of length j. We express the progress of the let-bound predicate p, which is defined in terms of α, as a least fixpoint:

$$\mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j = \bigsqcap\{i.\ i = \mathsf{prog}\ \sigma\ (P[p \mapsto i])\ \alpha\ j\}$$

$$\mathsf{prog}\ \sigma\ P\ (\mathsf{LetPast}\ p \coloneqq \alpha\ \mathsf{in}\ \beta)\ j = \mathsf{prog}\ \sigma\ (P[p \mapsto \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j])\ \beta\ j$$

(We do not update σ in these definitions as progress depends only on the time-stamp sequence but not on the databases in σ.) The above characterization follows the iteration in $\mathsf{eval}_{\mathsf{LP}}$: Since prog is pointwise monotone in P and at most j (both facts we prove in the formalization), the fixpoint can be reached by iteratively computing $\mathsf{prog}\ \sigma\ (P[p \mapsto i])\ \alpha\ j$ starting with $i=0$. Similarly, $\mathsf{eval}_{\mathsf{LP}}$ starts by evaluating α with no data for p and it feeds the results back into the evaluation until no further results can be obtained. Theorem 2 remains true after adding the above equation to prog.

The state invariant for SLetPast is given by the rule

$$\frac{\begin{array}{c}\mathsf{invar}\ \big(\sigma[p \Rrightarrow \mathsf{recp}\ (\lambda R\ k.\ \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ k\ \alpha)]\big)\ j\ (P[p \mapsto i])\ m\ s_\alpha\ \alpha \\ \mathsf{invar}\ \big(\sigma[p \Rrightarrow \mathsf{recp}\ (\lambda R\ k.\ \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ k\ \alpha)]\big)\ j\ (P[p \mapsto \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j])\ n\ s_\beta\ \beta \\ buf = \bot \longrightarrow i = \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j \\ \big(\forall Z.\ buf = \langle Z \rangle \longrightarrow i + 1 = \mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j \\ \qquad \wedge\, \mathsf{table}\ m\ (\mathsf{fv}\ \alpha)\ (\mathsf{recp}\ (\lambda R\ k.\ \mathsf{satrel}\ (\sigma[p \Rrightarrow R])\ k\ \alpha))\ Z\big) \\ m = \mathsf{nfv}\ \alpha \quad \mathsf{slp}\ p\ \alpha \leq \mathsf{P} \quad \{0\ ..{<}\ m\} \subseteq \mathsf{fv}\ \alpha\end{array}}{\mathsf{invar}\ \sigma\ j\ P\ n\ (\mathsf{SLetPast}\ p\ m\ s_\alpha\ s_\beta\ i\ buf)\ (\mathsf{LetPast}\ p := \alpha\ \mathsf{in}\ \beta)}$$

The first two premises use the same updated trace as in the semantics of LetPast (Section 4.1). The updated progress for p differs slightly between the premise for s_α and that for s_β. For the latter it is given by $\mathsf{prog}_{\mathsf{LP}}$, as expected. The predicate p's progress within s_α is equal to the state variable i, which is one less than $\mathsf{prog}_{\mathsf{LP}}\ \sigma\ P\ p\ \alpha\ j$ if the buffer buf is nonempty. This reflects to the optimization discussed in Section 4.3. The predicate $\mathsf{table}\ A\ n\ R\ Z$ is true iff the table Z contains tuples of length n that assign values to variables A and they are exactly the tuples of this kind satisfying $\mathsf{map}\ \mathsf{the}\ v \in R$.

5 Evaluation

We have used Isabelle/HOL's code generator [12] to export a certified implementation of VeriMon's core init and step functions and every function those depend on (e.g., operations on red-black trees), which amounts to about 10 000 lines of OCaml code. VeriMon augments this generated code with unverified parsers and pretty-printers. We evaluate this implementation to answer the following research questions: (1) How does VeriMon perform when monitoring formulas with the recursive let operator?; and (2) How does it compare to existing monitors for temporal first-order specifications with recursive rules?

To answer these questions, we run VeriMon and DejaVu and benchmark some of the example formulas introduced in Section 4.2. Instead of *SinceLet*, we opt for the simpler $\mathit{OnceLet} = \mathsf{LetPast}\ o(u,v) := s(u,v) \vee \bullet\, o(u,v)\ \mathsf{in}\ \mathit{filter}(x,y) \wedge o(x,y)$ encoding the non-metric $\blacklozenge$ operator. We also include $\mathit{Once} = \mathit{filter}(x,y) \wedge \blacklozenge\, s(x,y)$ for comparison. The predicate $\mathit{filter}(x,y)$ keeps the output size small. The *OnceLet* formula uses only one recursive predicate instance, whose variable order matches the one in the predicate's definition. Other formulas have more than one instance with different variable orders.

For the *PBLet* formula, we use an existing random trace generator [17] configured to pick parameters from a small integer domain, which increases the probability of producing satisfactions. For the other formulas, we generate traces using a similar strategy to the one used in DejaVu's benchmarks on the *Spawn* formula [14]. Namely, edges of a tree of spawned processes with a configurable branching factor are linearized into a trace, level by level. In the final level all edges converge to a single node for the formulas *Trans* and Trans^+. We define the edges by $\mathsf{Let}\ s^+(x,y,w) := e(x,y,w) \wedge \neg \Diamond_{[0,10]}\, d(x,y)$ in the Trans^+ formula and revoke one half of the edges on the second level of the branching.

We have executed our experiments on an Intel Core i5-4200U CPU using 8 GB RAM. Initially, DejaVu crashed on the *OnceLet* and *Spawn* formulas. We investigated the issue and found that its formula's abstract syntax tree was disconnected in these cases. We assume that this is caused by naming variables in the recursive rules' definitions

Trace length	*Once*		*OnceLet*		*Spawn*		*Trans*		$Trans^+$	*PBLet*
	VeriMon	DejaVu	VeriMon	DejaVu	VeriMon	DejaVu	VeriMon	DejaVu	VeriMon	VeriMon
100	0.0	1.1	0.0	1.1	0.6	1.5	1.3	3.7	5.6	0.0
200	0.0	1.2	0.0	1.2	3.1	2.1	6.1	8.1	25.9	0.0
400	0.0	1.3	0.0	1.3	14.0	3.4	28.3	23.6	117.4	0.0
800	0.0	1.5	0.0	1.4	64.8	8.2	TO	83.4	TO	0.0
4000	0.2	41.3	0.1	40.5	TO	TO	TO	TO	TO	0.1
8000	0.4	TO	0.2	TO	TO	TO	TO	TO	TO	0.1
10000	0.5	TO	0.3	TO	TO	TO	TO	TO	TO	0.2

Fig. 3. Execution times of the monitors in seconds (TO = timeout of 120 seconds)

differently from those in the rules' usages. After renaming the variables in the let-bound predicates of these two formulas, the issue was fixed and we restarted the experiments.

The evaluation results (Figure 3) show that DejaVu's performance is incomparable to VeriMon's. VeriMon outperforms DejaVu on the formulas *Once* and *OnceLet* and scales well on *PBLet*, which, together with the $Trans^+$ formula, we could not express in PFLTL with recursion. DejaVu outperforms VeriMon on the *Spawn* and *Trans* formulas for which VeriMon's time complexity of processing one event is linear in the trace length because the number N of valuations satisfying the recursive predicates grows linearly in the trace length and the time complexity of updating the recursive predicate is linear in N. We conjecture based on some preliminary experiments that VeriMon's performance can be significantly improved by optimizing the representation of sets of tuples in two ways: (a) using tuples of a fixed length with a fixed assignment of variables to positions in a tuple (i.e., no De Bruijn indices); (b) using a collection of indices to optimize the computation of joins on various sets of shared columns. Nevertheless, processing one event can unlikely be made trace-length independent: *Trans* encodes the *incremental dynamic transitive closure* graph problem, with the best known algorithm processing every new edge in the input in amortized linear time (in the graph's maximum out-degree) [23].

6 Conclusion

We have presented the extension of a monitor for MFOTL with non-recursive and past-recursive let operators. The presence of bounded future temporal operators complicates both the semantics and the evaluation algorithms for the new constructs, compared to earlier unverified extensions of past-only monitors [14]. Yet, the formal correctness proofs that we have carried out ensure the trustworthiness of our development.

As future work we plan to improve the performance of evaluating expensive joins by introducing indices, as used in database management systems. Expressiveness-wise we will consider further relaxing the requirements on the recursive let. We can omit the past guard if we define a Datalog-style fragment for which the fixpoint is well-defined. Beyond relaxing guards, we may want to allow recursion through future operators in certain situations. The main challenge is that this would make the progress notion data-dependent (unlike currently, where it only depends on the time-stamps).

Acknowledgments We thank David Basin for supporting this work and the anonymous TACAS reviewers for their helpful comments. Dmitriy Traytel is supported by a Novo Nordisk Fonden Start Package Grant (NNF20OC0063462).

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_5
3. Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 432–453. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_25
4. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15:1–15:45 (2015). https://doi.org/10.1145/2699444
5. Basin, D., Klaedtke, F., Zălinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. Kalpa Publications in Computing, vol. 3, pp. 19–28. EasyChair (2017). https://doi.org/10.29007/89hs
6. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal stream-based specification language. In: Massoni, T., Mousavi, M.R. (eds.) SBMF 2018. LNCS, vol. 11254, pp. 144–162. Springer (2018). https://doi.org/10.1007/978-3-030-03044-5_10
7. Cucala, D.J.T., Walega, P.A., Grau, B.C., Kostylev, E.V.: Stratified negation in Datalog with metric temporal operators. In: AAAI 2021. pp. 6488–6495. AAAI Press (2021)
8. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: TIME 2005. pp. 166–174. IEEE Computer Society (2005). https://doi.org/10.1109/TIME.2005.26
9. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Rossi, F. (ed.) IJCAI 2013. pp. 854–860. IJCAI/AAAI (2013)
10. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. **23**(2), 255–284 (2021). https://doi.org/10.1007/s10009-021-00609-z
11. Gorostiaga, F., Sánchez, C.: Stream runtime verification of real-time event streams with the striver language. Int. J. Softw. Tools Technol. Transf. **23**(2), 157–183 (2021). https://doi.org/10.1007/s10009-021-00605-3
12. Haftmann, F.: Code generation from specifications in higher-order logic. Ph.D. thesis, Technical University Munich (2009)
13. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Transf. **17**(2), 143–170 (2015). https://doi.org/10.1007/s10009-014-0309-2
14. Havelund, K., Peled, D.: An extension of LTL with rules and its application to runtime verification. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 239–255. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_14
15. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. Formal Methods Syst. Des. **56**(1), 1–21 (2020). https://doi.org/10.1007/s10703-018-00327-4
16. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification – Introductory and Advanced Topics, LNCS, vol. 10457, pp. 61–102. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_3
17. Krstić, S., Schneider, J.: A benchmark generator for online first-order monitoring. In: Deshmukh, J., Ničković, D. (eds.) RV 2020. LNCS, vol. 12399, pp. 482–494. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_27
18. Libkin, L.: Elements of Finite Model Theory. Springer (2004)

19. Ronca, A., Kaminski, M., Grau, B.C., Motik, B., Horrocks, I.: Stream reasoning in temporal Datalog. In: McIlraith, S.A., Weinberger, K.Q. (eds.) AAAI 2018. pp. 1941–1948. AAAI Press (2018)
20. Sánchez, C.: Online and offline stream runtime verification of synchronous systems. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 138–163. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_9
21. Schneider, J., Basin, D., Krstić, S., Traytel, D.: A formally verified monitor for metric first-order temporal logic. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 310–328. Springer (2019). https://doi.org/10.1007/978-3-030-32079-9_18
22. Walega, P.A., Kaminski, M., Grau, B.C.: Reasoning over streaming data in metric temporal Datalog. In: AAAI 2019. pp. 3092–3099. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33013092
23. Yellin, D.M.: Speeding up dynamic transitive closure for bounded degree graphs. Acta Informatica **30**(4), 369–384 (1993). https://doi.org/10.1007/BF01209711
24. Zingg, S., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: VeriMon's development repository. https://bitbucket.org/jshs/monpoly/src/887b996966/thys/ (2021)

Maximizing Branch Coverage with Constrained Horn Clauses

Ilia Zlatkin, Grigory Fedyukovich(✉)

Florida State University, Tallahassee, FL, USA,
iz20e@fsu.edu, grigory@cs.fsu.edu

Abstract. State-of-the-art solvers for constrained Horn clauses (CHC) are successfully used to generate reachability facts from symbolic encodings of programs. In this paper, we present a new application to test-case generation: if a block of code is provably unreachable, no test case can be generated allowing to explore other blocks of code. Our new approach uses CHC to incrementally construct different program unrollings and extract test cases from models of satisfiable formulas. At the same time, a CHC solver keeps track of CHCs that represent unreachable blocks of code which makes the unrolling process more efficient. In practice, this lets our approach to terminate early while guaranteeing maximal coverage. Our implementation called HORNTINUUM exhibits promising performance: it generates high coverage in the majority of cases and spends less time on average than state-of-the-art.

1 Introduction

Branch coverage is a method for testing that aims to maximize the number of program branches to be collectively visited by a set of test cases. Branches in the code are commonly attributed to the conditional statements or loops. For testing a loop-free program, possible test cases for all the branches can be identified by *symbolic execution*, powered by efficient solvers for Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT). If a conditional is placed inside or after a loop, test-case generation immediately becomes challenging because the cost of exploration of every next iteration grows exponentially in the worst case.

Many verification problems can be reduced to synthesizing interpretations of predicates in systems of SMT formulas, also known as constrained Horn clauses (CHC), that provide a modular encoding for programs with arbitrary control flow. In this paper, we propose to use CHC also for test-case generation. Solutions to CHC, also called *inductive invariants*, carry reachability information and are useful in pruning the search space explored by test-case generators. If an invariant shows that a branch can never be taken, then it is guaranteed that no test can ever reach the branch, and thus a test-case generator can safely proceed to discovery of the next test case.

We contribute a new approach to test-case generation that aims at maximizing branch coverage using inductive invariants. In essence, our approach gradually enumerates different unrollings and uses an off-the-shelf SMT solver to get

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 254–272, 2022.
https://doi.org/10.1007/978-3-030-99527-0_14

values for program variables that represent test cases. Unrollings are constructed on-the-fly by exploring the CHC encoding of programs. Concurrently, an incremental CHC solver determines a subset of unreachable CHCs which allows the algorithm to explore fewer unrollings in the next iterations. The algorithm terminates when the test cases are generated for all reachable branches, but all the remaining branches are *provably unreachable*.

These features distinguish our approach from other white-box test generators [1,8,9] that consider reachability information only in the bounded context. That is, in the presence of unreachable branches and loops, they may continue iterating forever, even if all possible test cases have already been generated. Reliance on invariants lets our tool to terminate early while still guaranteeing maximal possible coverage.

The approach has been implemented on top of the FreqHorn CHC solver [14] and the Z3 SMT solver [27]. It enables test-case generation for C programs, converted to CHCs by the SeaHorn [21] tool. Experiments conducted on a range of public benchmarks demonstrate the strengths of our approach compare to state-of-the-art: SMT-based incremental test-case generation is able to detect high-quality solutions in the majority of cases and is on average less expensive.

2 Related Work

Automated test generation has two main approaches: fuzzing (e.g., [7,20,25,26,29,31,33,34]) and symbolic/concolic execution (e.g., [3,8,11,22,23,28,32]). The former group uses user-given seed inputs and further mutates them based on various heuristics (sometimes using the source code as well). The latter group, which also includes our one, proceeds by enumerating paths and generating test cases, often using constraint solvers. Recent algorithms, including FuSeBMC [1] and Verifuzz [9], follow both approaches: begin with symbolic execution (namely, some bounded model checking [10,19]) and then proceed to fuzzing.

The closest related work [22] suggests to accelerate testing using interpolation. Aiming the same goal as us, i.e., prune unreachable paths, they however do not generate inductive invariants, which limits the generality of their method.

Earlier attempts to combine static analysis techniques and testing [11] were tailored to particular frameworks and languages. With the rise of SMT solvers, approaches became more scalable, goal-oriented [3], and at the same time more agnostic to programming languages. Recent works, e.g. [33], offer a great flexibility of applications of static analyzers to test-case generation, e.g., to direct fuzzers to specific blocks of code. Following this trend, our approach continues bridging the gap between state-of-the-art in automated reasoning and testing.

While we are not aware of any specific applications of CHC solvers to test-case generation, we are largely inspired by the work in model checking, e.g., [6,21] that can both discover invariants and find counterexamples (from which a test case can be extracted). The main difference is in the application: model checkers often focus on a single property/bug, while our goal is to cover the maximal number of branches. Furthermore, many practical approaches including [1,9]

```
1  int x = 0;
2  int y = nondet();
3  int z = nondet();
4  while (1) {
5    if (x >= 5)
6      y++;      // needs at least 6 iterations to reach
7    else
8      x++;      // x ∈ [0,5] always holds
9    if (y <= 5)
10     z++;
11   else
12     if (x > y)
13       y++;    // this is unreachable
14     else
15       x = 0;
16   if (z == 0)
17     break;
18 }
```

Fig. 1: Loopy program with control-flow divergence and unreachable branches.

are based on existing model checkers (that typically use constraint solvers as blackbox), CHC formulation allows to build tools *modularly* and directly on top of an SMT solver, thus allowing to use it incrementally for both counterexample finding and invariant generation.

3 Motivating Example

Fig. 1 gives a program with a single loop. It has three variables: `x` is assigned zero before the loop, which we cannot change, and the remaining `y` and could be taken from the user. The loop has four if-then-elses (including one nested), and it terminates when the value of at the end of an iteration equals zero. To completely cover all the branches, we need to consider seven cases, in particular:

line 6: In order to reach the first `then`-branch, the loop needs to iterate at least six times and do not reach lines 15 and 17 at the first five iterations. Thus, line 8 should be visited at the first five times. A possible scenario for that would be if initially `y` $= 0$ and $= 0$.

line 8: The loop always reaches the `else`-branch of the first conditional because initially `x` is zero, and the guard trivially does not hold.

line 10: The guard of the second conditional might hold even at the first iteration if `y` is sufficiently small. Since we know that the increment at line 6 does not happen at the first iteration, `y` might initially be 5 (and any). Then the branch is reachable.

line 13: The branch is never reachable because $0 \leq x \leq 5$ is a loop invariant (and thus, holds at each iteration) and the path condition $x > y \wedge y > 5$ is unsatisfiable.

line 15: Because line 13 is unreachable, we know that line 15 is reached always if the guard of the second conditional does not hold, e.g., when `y` is initially greater than 5.

line 17: If initially = 0 and `y` is greater than 5, the loop executes the `break` statement at the end of its only iteration.
line 19: We have already seen a test case (for line 6) that gives a possible condition for the loop to continue iterating. In fact, for any values of greater than zero (and any values of `y`), the loop does not terminate at all.

All these make the program quite interesting and its analysis challenging.

4 Background

This paper approaches the problem of automated test-case generation by reduction to the *Satisfiability Modulo Theories* (SMT) problem. Automated SMT solvers determine the existence of a satisfying assignment to variables (also called a *model*) of a first-order logic formula. Formula φ is logically stronger than formula ψ (denoted $\varphi \implies \psi$), if every model of φ also satisfies ψ. The unsatisfiability of formula φ is denoted $\varphi \implies \bot$, and we also write $\mathcal{M} \in \varnothing$ to indicate that no model $\mathcal{M}$ of the formula (which is clear from the context) exists. By writing $\psi(x)$, we denote a predicate over free variables x.

Constrained Horn clauses (CHC) are used as intermediate verification language used by both verification frontends and backend SMT solves. This allows to split efforts while designing a verification tool for a new language: while focusing on encoding programs to CHCs, researchers rely on advances of CHC solvers that will solve these CHCs. Thus, by demonstrating our algorithms at the level of CHCs, we allow for many their particular instantiations for various programming languages (that support CHC encoding).

Definition 1. *A* linear constrained Horn clause *(CHC) over a set of uninterpreted relation symbols $\mathcal{R}$ is a first-order-logic formula having the form of either:*

$$\varphi(\vec{x_1}) \implies \boldsymbol{inv}_1(\vec{x_1})$$
$$\boldsymbol{inv}_i(\vec{x_i}) \wedge \varphi(\vec{x_i}, \vec{x_j}) \implies \boldsymbol{inv}_j(\vec{x_j})$$
$$\boldsymbol{inv}_n(\vec{x_n}) \wedge \varphi(\vec{x_n}) \implies \bot$$

where all $\boldsymbol{inv}_i \in \mathcal{R}$ are uninterpreted symbols, all $\vec{x_i}$ are vectors of variables, and φ is a fully interpreted formula called constraint*.*

These types of implications are called respectively a *fact*, an *inductive clause*, and a *query*. Note that constraint φ_C of each CHC C does not have applications of any predicates from $\mathcal{R}$. Further, by $body(C)$, we denote the premise of C, by $src(C)$ an application of $\boldsymbol{inv} \in \mathcal{R}$ in $body(C)$ (but if C is a fact, we write $src(C) \stackrel{\text{def}}{=} \top$). Similarly, by $head(C)$, we denote the conclusion of C, and by $dst(C)$ we denote an application of $\boldsymbol{inv} \in \mathcal{R}$ in $head(C)$ (and if C is a query, we write $dst(C) \stackrel{\text{def}}{=} \bot$).

Intuitively, CHCs allow to generate program encodings with "holes" that represent unrollings of unknown lengths. Then, possible instantiations of these holes can be used in the discovery of meaningful information about the program, such as loop invariants, or function summaries.

Definition 2. *Given a set $\mathcal{R}$ of uninterpreted predicates and a set S of CHCs over $\mathcal{R}$, we say that S is satisfiable if there exists an interpretation for every $\boldsymbol{inv} \in \mathcal{R}$ that makes all implications in S valid.*

CHCs are useful also when there is a need to access various pieces of program encoding and pose reachability queries. In particular, it is straightforward to design a Bounded Model Checking (BMC) [5] tool on top of CHCs and use it for test-case generation. Specifically, by traversing the graph structure imposed on the CHCs, we can access all possible program traces and create the corresponding unrollings.

Definition 3. *Given a system S of CHCs over $\mathcal{R}$, an unrolling of S of length k is a conjunction $\pi_{\langle C_0,\ldots,C_k\rangle} \stackrel{\text{def}}{=} \bigwedge_{0\le i\le k} \varphi_{C_i}(\vec{x}_i, \vec{x_{i+1}})$, such that 1) C_0 is a fact, 2) each $C_i \in S$, 3) for each pair C_i and C_{i+1}, $rel(dst(C_i)) = rel(src(C_{i+1}))$, and variables of each $\vec{x_i}$ are shared only between $\varphi_{C_{i-1}}(\vec{x}_{i-1}, \vec{x_i})$ and $\varphi_{C_i}(\vec{x}_i, \vec{x_{i+1}})$.*

For bug finding, it is essential to enumerate various unrollings and check their satisfiability. Once a satisfiable formula $\pi_{\langle C_0,\ldots,C_k\rangle}$ is found for some query C_k, the bug is found (and its counterexample can be obtained from the model), and thus no interpretation for predicates in $\mathcal{R}$ exists.

Lemma 1. *Given a system of CHCs S, let $\pi_{\langle C_0,\ldots,C_k\rangle}$ be one of its unrollings, such that C_0 is a fact, and C_k is the query. Then if $\pi_{\langle C_0,\ldots,C_k\rangle}$ is satisfiable then S is unsatisfiable.*

In the next section, we expand on the notions of CHCs and unrollings, give examples, and present the application to test-case generation.

5 Test-case Generation for Branch Coverage

The concept of constrained Horn clauses is convenient for formulating the problem of constructing a maximal branch coverage (MBC) of a given program. At the highest level, the problem of finding MBC is concerned with finding a set of program executions that visit all reachable program branches. Given the CHC encoding of the program, this can be reduced to a problem of finding a set of satisfiable unrollings that involve the maximal number of CHCs. However, to guarantee maximality, this needs a special property of the CHC encoding: the constraint φ in each CHC should represent a straight-line code sequence with no branches (a.k.a. *basic block*). Technically, this can be formulated as the requirement for each CHC to have a conjunction of literals (a.k.a. *cube*), i.e., have no disjunctions, in its body.

Example 1. Fig. 2 gives a CHC encoding of the program in Fig. 1. There are eight CHCs over four uninterpreted predicates A, B, C, and D. The program entry is encoded in the first CHC (i.e., the only fact with the *dst*-predicate A), and its exit in the last CHC (i.e., with the *dst*-predicate D). All other CHCs encode

$$
\begin{array}{lr}
(1) & x = 0 \implies A(x, \ , z) \\
(2) & A(x, \ , z) \wedge x \geq 5 \wedge x' = x \wedge \ ' = \ + 1 \wedge z' = z \implies B(x', \ ', z') \\
(3) & A(x, \ , z) \wedge x < 5 \wedge x' = x + 1 \wedge \ ' = \ \wedge z' = z \implies B(x', \ ', z') \\
(4) & B(x, \ , z) \wedge \ \leq 5 \wedge x' = x \wedge \ ' = \ \wedge z' = z + 1 \implies C(x', \ ', z') \\
(5) & B(x, \ , z) \wedge \ > 5 \wedge x > \ \wedge x' = x \wedge \ ' = \ + 1 \wedge z' = z \implies C(x', \ ', z') \\
(6) & B(x, \ , z) \wedge \ > 5 \wedge x \leq \ \wedge x' = 0 \wedge \ ' = \ \wedge z' = z \implies C(x', \ ', z') \\
(7) & C(x, \ , z) \wedge z \neq 0 \implies A(x, \ , z) \\
(8) & C(x, \ , z) \wedge z = 0 \implies D(x, \ , z)
\end{array}
$$

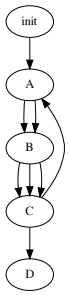


Fig. 2: CHCs of the motivating example (left) and *src*/*dst*-dependency graph (right).

the loop with the total of six symbolic paths, following $A \to B \to C \to A$ (as can be seen from the graphic representation) but involving different CHCs. Each CHCs has no disjunctions in the body: it has the conjunction of the (negation of) guard and the encoding of program instructions following the corresponding branch until either a next conditional or the join occurs. Note that there are no queries in this system since there are no assertions in the program.

To formulate the MBC problem at the level of CHCs, it is convenient to introduce the concept of a *src*/*dst*-dependency graph for a system of CHCs.

Definition 4. *Given a system S of CHCs over a set of uninterpreted predicate symbols $\mathcal{R}$, its* src/dst-dependency graph $\langle \mathcal{R}, E \rangle$ *is a directed graph with edges labeled by CHCs from S:*

$$E \stackrel{\text{def}}{=} \{\langle rel(src(C)), C, rel(dst(C))\rangle \mid C \in S\}.$$

Because we are bound in this paper to use only disjunction-free CHCs, the points of control-flow divergence in a program encoded in these CHCs are captured by vertices in the *src*/*dst*-dependency graph that have more than one outgoing edge[1]. To generate a test case visiting some block of code encoded in a CHC C_k, it is enough to find an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ and show that this unrolling is satisfiable. In this case, the CHC is called *reachable*: i.e., the satisfying assignment would naturally correspond to a program trace beginning at

[1] Thus, in this case, the *src*/*dst*-dependency graph can be seen as a control-flow graph (CFG) of the encoded program. In practice, many verification tools that are based on CHC do not generate CHCs in such form but apply some generalization and compression to CFG during preprocessing. This results in CHCs with disjunctive bodies that are unsuitable for our approach. In these cases, we explicitly convert the body of each CHC to a disjunctive normal form (DNF) and clone the CHC for each cube in the DNF. The CHCs after this transformation is still a correct encoding of the original program, and its *src*/*dst*-dependency graph is suitable for our approach, but it may not exactly match the CFG of the original program.

the program entry point and reaching the code in that branch. Furthermore, if the execution depends on some input values, these values can also be extracted from the satisfying assignment.

Example 2. According to Fig. 2, the first point of control-flow divergence is predicate A. To show that CHC 3 is reachable, we create the following unrolling from bodies of CHCs 1 and 3:

$$x = 0 \wedge x < 5 \wedge x' = x + 1 \wedge y' = y \wedge z' = z.$$

This formula is satisfiable, and there exists a model $\mathcal{M} = \{x \mapsto 0, y \mapsto 0, z \mapsto 0, \ldots\}$, thus giving us two values for input variables y and z (both zeroes).

It can also be seen that some CHCs cannot be visited by any trace. To find them, we can pose additional safety verification queries and aim at generating an appropriate invariant.

Lemma 2. *Given a system S of CHCs over some $\mathcal{R}$, and let C be some CHC from S. If the extended CHC system $S \cup \{src(C) \wedge \varphi_C \implies \bot\}$ is satisfiable, then C is unreachable.*

The proof of the lemma follows directly from Lemma 1.

Example 3. In the CHC system in Fig. 2, CHC 5 is never reachable. We introduce a new query CHC Q as follows:

$$B(x, y, z) \wedge y > 5 \wedge x > y \wedge x' = x \wedge y' = y + 1 \wedge z' = z \implies \bot$$

The system $S \cup \{Q\}$ is satisfiable, with the following interpretation $\mathcal{M}$:

$$\mathcal{M}(A) = \mathcal{M}(B) = \mathcal{M}(C) = \lambda x, y, z \,.\, x \leq 5$$

Because $x \leq 5 \wedge y > 5 \wedge x > y$ is unsatisfiable, CHC 5 is unreachable.

These ingredients lets us state the MBC problem formally.

Definition 5 (MBC). *Given a system S of CHCs over some $\mathcal{R}$, the problem of* maximizing branch coverage *of S is concerned with 1) determining a subset $S_u \subseteq S$ of CHCs which are provably unreachable (i.e., Lemma 2 applies), and 2) finding satisfiable unrollings for all CHCs from $S \setminus S_u$.*

The practical significance of the MBC problem consists in allowing the test-generation tools that are based on bounded model checking, e.g., [1], to terminate earlier. The invariants discovered while iteratively applying Lemma 2 can serve as annotations of various nodes of the program CFG, which further enables to prune the search space of the test cases. In particular, for our running example in Fig. 1, line 13 is provably unreachable, thus it makes no sense to search for its test case.

Furthermore, with the invariant that blocks a branch at hand, the tools can now explore fewer unrollings leading to other branches in the next iterations of the loop. Specifically, to reach line 6, five iterations of the loop will provably skip line 13, so instead of $(2 * 3)^5 = 7776$ unrollings, the tool should only explore $(2 * 2)^5 = 1024$ unrollings.

6 Solving the MBC problem

In this section, we introduce our novel approach to constructing the maximal branch coverage using a system of disjunction-free CHCs. We begin with outlining our key ideas that can be implemented on top of existing test-case generators and invariant generators, and then proceed to describing our efficient implementation.

6.1 Key Insights

The approach has a simple high-level structure. Because the number of CHCs in a program encoding is always finite, we can pose a safety verification query for each of them.

Existing CHC solvers are equipped with the functionality to generate both, the counterexamples and safety invariants. However, recent evaluation [17] show that the bounded-model-checking implementations often outperform general-purpose solvers on unsatisfiable CHC instances (likely, because they do not invest efforts in generating invariants). This suggests that for performance reasons, it makes sense to alternate between separate runs of a counterexample generator (via enumerating the unrollings) and an invariant generator. This allows for two main benefits, outlined in the next two paragraphs.

A *counterexample generator*, in the MBC setting, should handle a large number of unrollings. Many of the unrollings are unsatisfiable since some sequentially aligned branches might be incompatible, and some other branches might be waiting for a certain loop iteration. It is thus essential to share the information about conflicting paths' segments (e.g., unsatisfiable prefixes, as in our implementation) to accelerate the search. Dually, satisfiable unrollings can often be extended to unrollings for other reachable CHCs, and this information can be exploited in the enumerative search for the remaining branches.

An *invariant generator*, invoked multiple times throughout the process, deals with many largely similar safety verification instances (since all CHCs are the same, and only queries are different). Thus, a lot of information can be reused between verification runs, opening the opportunities for *incremental verification* [13]. Formally, all invariants that are discovered while proving the unreachability of a CHC will remain valid after switching to another CHC. Even more, the solvers that target conjunctive invariant generation, e.g., [15,24] can output "partial" invariants (i.e., some lemmas) even for unsatisfiable CHC instances, which then can be reused/completed in the next runs of the solver.

These observations let us to conclude that despite using off-the-shelf tools for bounded model checking and invariant generation is possible, an MBC will likely exhibit a more optimized performance through the design of new algorithms that incorporates the aforementioned insights.

6.2 General Driver

The pseudocode of our approach is given in Alg. 1. The algorithm begins with identifying a subset *cur* of CHCs that need to be considered in its iterations.

Algorithm 1: CHC-based test-case generator.

Input: S: a CHC system over $\mathcal{R}$
Output: T: a set of satisfying assignments to variables in S
Data: *invs*: mapping from $\mathcal{R}$ to invariants, $G = \langle \mathcal{R}, E \rangle$: an edge-labeled graph, $cur \subseteq S$: a subset of CHCs to consider, *length*: counter representing the length of the current unrollings, *traces*: a (global) set of traces to consider

```
1  ⟨R, E⟩ ← src/dst-dependency graph of S;
2  cur ← {C | ⟨u, C, v1⟩ ∈ E and ∃⟨u, _, v2⟩ ∈ E where v1 ≠ v2};
3  if cur = ∅ then cur ← {C | src(C) = ⊤};
4  length ← 1;
5  while cur ≠ ∅ do
6      for chc ∈ cur do
7          ⟨res, invs, cex⟩ ← SOLVECHCS(S ∪ {body(chc) ⟹ ⊥}, invs);
8          if res = SAT then
9              cur ← cur \ {chc};
10             E ← {⟨u, C, v⟩ | ⟨u, C, v⟩ ∈ E and C ≠ chc};
11         else if res = UNSAT then
12             cur ← cur \ {chc};
13             T ← T ∪ {cex};
14         else
15             traces ← ∅;
16             GETTRACES(E, ⊤, chc, length, nil, prefixes, traces);
17             for t ∈ traces do
18                 ⟨res, M⟩ ← CHECKSAT(UNROLL(S, t));
19                 if res = SAT then
20                     T ← T ∪ {M};
21                     cur ← cur \ {chc};
22                     break;
23                 else
24                     prefixes ← prefixes ∪ {t};
25     length ← length + 1;
```

We say that a CHC C *opens a branch*, if the outdegree of $rel(src(C))$ in the *src*/*dst*-dependency graph is greater than one (line 2). Thus, to generate a test case visiting a branch, it is enough to find an unrolling $\pi_{\langle C_0,\ldots,C_k\rangle}$ where C_k opens that branch and show that this unrolling is satisfiable. If, however, there are no branches in the given program at all, then *cur* gets all facts of the CHC system (line 3), and the remaining coverage generation is straightforward.

The rest of the algorithm is organized as a big loop that decides if any CHC from *cur* are (un)reachable and terminates when *cur* is empty. At each iteration of the loop, all CHCs from *cur* are enumerated, and the algorithm seeks to apply Lemma 2, i.e., extends S with one query and solves these CHC (line 7). The algorithm can use any CHC solving algorithm that decides the satisfiability

Algorithm 2: GetTraces: trace enumerator.

Input: $E \in 2^{\mathcal{R}\times S\times\mathcal{R}}$: labeled edges, $u \in \mathcal{R}$, $chc \in S$, *length*: length of trace, t: trace prefix, *prefixes*: to avoid
Output: *traces*: global set of traces of *length* beginning with relation u and ending with *chc*

```
1  if ∃p ∈ prefixes . ∀i . i ∈ [0, |p|) ⟹ p_i = t_i then
2      return;
3  if length = 1 then
4      if ⟨u, chc, _⟩ ∈ E then
5          traces ← traces ∪ {t@chc}
6  else
7      for {u, C, v} ∈ E do
8          GetTraces(E, v, chc, length − 1, t@C, prefixes, traces);
```

of CHCs, returns inductive invariants (line 8) or (optionally[2]) a counterexample (line 11). In both cases, the CHC is excluded from *cur*. Additionally, if satisfiable, this CHC cannot be used in any unrolling, and it is excluded also from the auxiliary graph (line 10, to prune the search space of the remaining test cases). If a counterexample is returned, the branch is reachable, and the test case is extracted from this counterexample (line 13).

It is also possible (and in practice, very likely) that the CHC solver returns UNKNOWN (because the problem is undecidable, and invariant generators are often limited to either a fixed shape of invariants, or a certain timeout). In this case (lines 16-22), the algorithm proceeds with an explicit enumeration of unrollings of a predetermined length (line 16). Each trace $t = \langle t_0, t_1, \ldots, t_{length-1}\rangle$ has an associated unrolling $\pi_{\langle C_{t_0},\ldots,C_{t_{length-1}}\rangle}$ which is checked for the satisfiability (line 18) with an off-the-shelf SMT solver. If satisfiable (line 19), the branch opened by the current CHC is reachable, the test case is generated from the model, and the CHC is excluded from *cur*. If unsatisfiable (line 23), the algorithm registers this t as an *unsatisfiable prefix* to be avoided in the trace generation in the next iterations (see Alg. 2).

Theorem 1. *When Alg. 1 terminates, the resulting set T contains all the variable assignments needed for maximal coverage.*

In the next two paragraphs we discuss two important design choices that do not affect the correctness of our implementation, but optimize it.

[2] In fact, the counterexample detection in some CHC solvers, e.g., [24] proceeds in a similar fashion as described in our algorithm, but if invoked multiple times throughout the algorithm, it is likely that the CHC solver will perform many redundant actions. We thus do not use this functionality in our experiments (and our Alg. 3), but leave it in the pseudocode for the sake of completeness of presentation.

Algorithm 3: SOLVECHCS:

Input: S: a CHC system over $\mathcal{R}$, *invs*: mapping from $\mathcal{R}$ to invariants
Output: $res \in \langle \text{SAT}, \text{UNSAT} \rangle$, *invs*: updated mapping, [*cex*: counterexample]

1 $S' \leftarrow \varnothing$;
2 **for** $chc \in S$ **do**
3 $\quad S' \leftarrow S' \cup \{src(chc) \wedge \Big(body(chc)\Big)[\mathcal{R} \mapsto invs] \implies dst(chc)\}$;
4 **if** $\lambda x.\top$ *is a solution for* S' **then**
5 $\quad$ **return** *invs*;
6 **return** $\langle res, invs, \_ \rangle \leftarrow$ FREQHORN(S');

6.3 Incremental Trace Enumeration

Our algorithm allows for sharing the information obtained during its iterations using two global data structures: the set of unsatisfiable *prefixes* discovered during the trace enumeration and the graph-structure $\langle \mathcal{R}, E \rangle$ representing potentially reachable CHCs (line 10 of Alg. 1). Intuitively, the latter is constructed by an iterative removal of edges from the *src*/*dst*-dependency graph, thus allowing for a more focused search of suitable traces. Both data structures are used in Alg. 2 that is called at the next algorithm iteration.

Conceptually, Alg. 2 is a dynamic-programming implementation of a path finder in an arbitrary directed graph. Given a length of path, its starting point and ending points, the algorithm recursively visits the graph edges and stores them in vectors[3]. In our setting, the algorithm is optimized in two ways. First, at line 1, it skips paths with unsatisfiable prefixes (because the corresponding unrollings will be unsatisfiable too). Second, at lines 4 and 7, it excludes all the unreachable CHCs that have been excluded from the graph previously.

Example 4. Recall our running example for program encoded as CHCs in Fig. 2. For *length* $= 2$ and CHC (2), Alg. 2, constructs a single trace $\langle (1), (2) \rangle$, that corresponds to an unsatisfiable unrolling, found by Alg. 1, and thus added to *prefixes*. Consequently, for *length* $= 3$, traces $\langle (1), (2), (4) \rangle$ and $\langle (1), (2), (6) \rangle$ are not generated. Furthermore, because (5) is never reachable, then edge $\langle B, (5), C \rangle$ is excluded from E permanently.

6.4 Incremental Invariant Discovery

Alg. 3 gives the main idea of our CHC solver, which relies on the FREQHORN [15] algorithm to synthesize invariants (any other CHC solver could be used as well). However, in addition, it recycles the invariants *invs* generated in all previous runs. Specifically, it substitutes interpretations for each $r \in \mathcal{R}$ in the body of each CHC (line 3). Because each such formula represents an over-approximation

[3] We use the notation $t@C$ to represent the "push back" operation over a vector t and an element C.

of the set of reachable states at a particular program location, this substitution is sound.

If it appears that after the substitution, all the remaining invariants are simply true-formulas (line 4) then *invs* is already a solution, and the CHCs solver is not needed. On the other hand, invariants could be generated by an external tool.

While the pseudocode of FreqHorn is omitted from Alg. 3 for simplicity, we list its distinguishing features here. The approach is driven by Syntax Guided Synthesis (SyGuS) [2], and it supports (possibly, non-linear) arithmetic and arrays [16]. It automatically constructs formal grammars $G(\boldsymbol{inv})$ for each $\boldsymbol{inv} \in \mathcal{R}$ based on either source code [14], or program behaviors [15,30]. Importantly, these grammars are *conjunction-free*, and they allow for only a finite number of candidates. FreqHorn iteratively attempts to apply production rules of each $G(\boldsymbol{inv})$ to *sample* a candidate and checks it with an SMT solver (successfully checked candidate is then called a *lemma*). The process continues either until a *conjunction of lemmas* is sufficient, or the search space is exhausted. To make the process less dependent on the order in which candidates are considered, FreqHorn uses *batching* [12] (e.g., checks several candidates at the same time) and effectively filters them using the well known Houdini algorithm [18].

These features make FreqHorn especially useful for the application to test-case generation. Behaviors and counterexamples can be obtained from traces as outlined in Sect. 6.3. Each new counterexample potentially contributes to a new data candidate to be considered in the next invocations of the algorithm. Then, following our incremental schema, new candidates are used in conjunction with previously generated invariants, and either added to *invs*, or dropped. Note that even if FreqHorn returns unknown indicating that it is unable to find a strong enough invariant, it almost always finds some lemmas that might be useful for the next iterations of our main algorithm.

7 Evaluation

We have implemented the approach in a tool called Horntinuum[4]. The backend of Horntinuum is developed on top of FreqHorn [14] and uses it for CHC solving. All the symbolic reasoning in our backend is performed by the Z3 [27] SMT solver, v4.8.10. For encoding C benchmarks to CHCs in our frontend, we use the SeaHorn [21] verification framework, v10.0.0-rc0, via its Docker image[5].

Implementation details. The success of our approach largely depends on the preprocessing performed by SeaHorn while producing the CHC encoding. Since our algorithm works on disjunction-free CHCs (recall Sect. 6.1), we configure SeaHorn to perform a small-step encoding, i.e., introducing a CHC per

[4] The source code of the tool is publicly available at https://github.com/izlatkin/HornLauncher with the CHC-based backend at https://github.com/izlatkin/aeval/tree/tg.

[5] https://hub.docker.com/r/seahorn/seahorn-llvm10.

each basic block (via the `--step=small` option). However, the encoder, based on LLVM, additionally performs several LLVM transformations[6] and auxiliary SeaHorn's passes that may introduce disjunctions to CHCs. Since this recipe is not configurable in SeaHorn yet, we additionally get rid of disjunctions, by performing a DNF-ization, over the CHCs received from SeaHorn.

We also had to overcome a relatively minor engineering obstacle to allow recognizing multiple `nondet()` function calls (see an example in Fig. 1). The CHC representation is in some sense *declarative*, i.e., it is not always possible to detect the order of function calls from formulas that represent program unrollings. Thus, we rename each invocation of `nondet()` in each input C file, e.g., to $\texttt{nondet}_i\texttt{()}$ which lets Horntinuum to associate each function invocation with a sequence of *static-single-assignment* (SSA) variables that encode (possibly many, if $\texttt{nondet}_i\texttt{()}$ is called in a loop) outputs of $\texttt{nondet}_i\texttt{()}$ occurring in an unrolling. Further, it gives a sequence of concrete values obtained for each of the SSA variable by the SMT solver. In a generated test case, sequences of SSA values of each $nondet_i$ are stored in a separate array (to capture values in each loop iteration) and accessed by an automatically generated body of the corresponding unique $\texttt{nondet}_i\texttt{()}$ function.

In a sense, the final output of our tool is a set of context-specific implementations of function `nondet()` written in different header files. The initial C file should include a header from this set, be compiled and run in order to reproduce the detected test case.[7]

Experimental setup. To evaluate Horntinuum, we configured the gcov tool, v9.3.0, a code coverage analysis and profiling tool that tracks all statements visited in a single run of the program. Running gcov for each our generated test case and merging the statistics gives the final coverage: we ultimately target to maximize the number of code visited by *at least one* test case.[8]

We compared Horntinuum with state-of-the-art tools FuSeBMC [1], Verifuzz [9], and KLEE [8][9] that exhibited a decent performance in TestComp 2021. Our experiments were run on a "Dell OptiPlex 7090 Tower" desktop computer with 2.5 GHz Intel Core i7 8-Core (11th Gen), 16GB 3200 MHz DDR4 RAM, and Ubuntu 20.04.1 LTS installed on it.

For the experimentation we considered 316 benchmarks from TestComp (from `loop-*` tracks, excluding the programs with floating points that our CHC solver

[6] One transformation, for instance, removes redundant branches from the code, e.g., replaces `if (nondet()) foo(); else foo();` by just `foo`. Technically, the CHC encoding received by our tool does not represent all branches of the original program, whilch thus leads to a smaller coverage detected. We have not seen many such examples in our benchmarks set, however.

[7] Note the difference with the TestComp format [4] that keeps all values in the same XML file. Our proposed format is more general and easily convertible to TestComp.

[8] The full logs and tables are available at https://www.cs.fsu.edu/~grigory/horntinuum.zip.
All the binaries were downloaded from https://test-comp.sosy-lab.org/2021/systems.php.

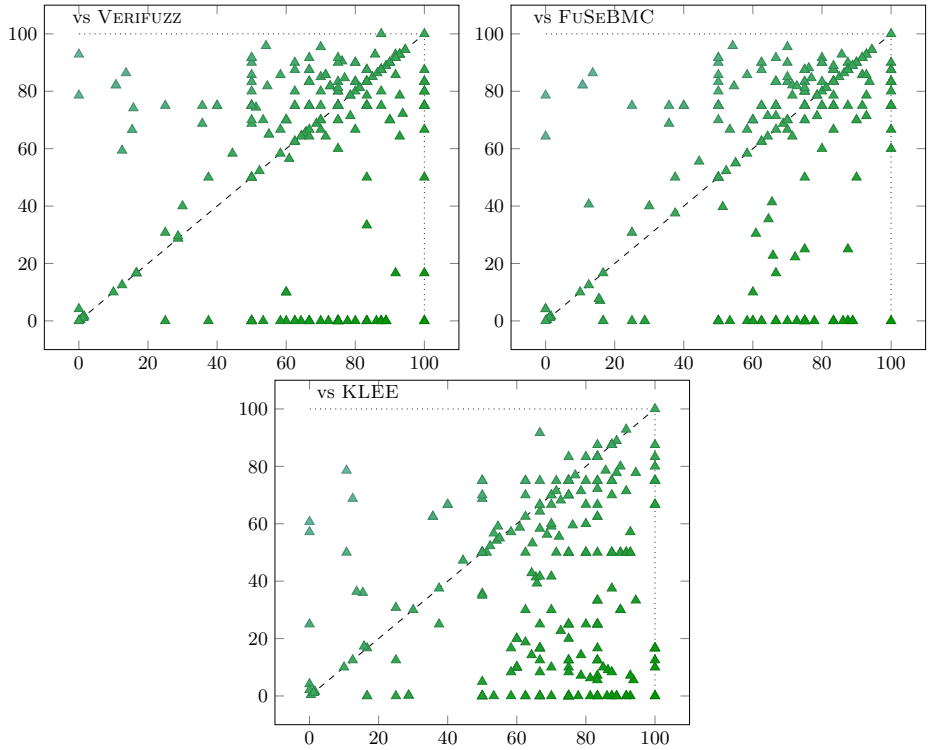


Fig. 3: Coverage comparison: each point in a plot represents a pair of the **coverages** (% × %) of HornInuum (x-axis) and a competitor (y-axis) for the same benchmarks.

does not support yet). The largest considered benchmark has >5K LoC. The performance of all three competitors (using the timeout of 15 minutes) on our machine was consistent to the one exhibited in TestComp 2021: VeriFuzz slightly outperforms FuSeBMC, and both outperform KLEE.

Expectations and results. We aim to answer two main questions:

Q_1 Is it possible to develop a competitive test-case generator based purely on formal verification and SMT solving, i.e., not relying on dynamic analysis and fuzzing?

Q_2 In the cases when a CHC-based test-case generator yields a similar (or better) coverage than a competitor, is it possible to achieve this result faster[10]?

Plots in Fig. 3 and Fig. 4 attempt to answer these questions, respectively.

We first give a pairwise comparison between the coverage %% reported by the tools (Fig. 3). If a tool was unable to analyze a program, the corresponding

[10] We believe the ability to successfully terminate the test-case generation early is of great interest to software engineers. However, unfortunately, it is not the main determining factor in testing competitions.

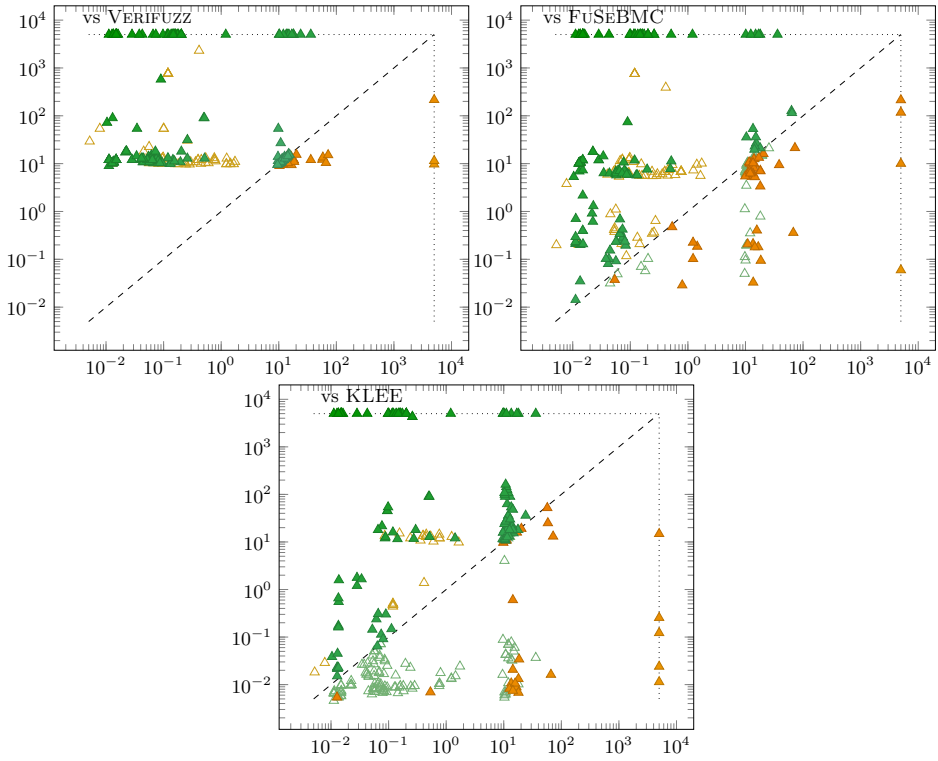


Fig. 4: Runtime needed to get 1% of coverage (sec × sec) of HORNTINUUM (x-axis) and a competitor (y-axis). Solid triangles represent runs (green: HORNTINUUM, orange: the competitor) in which the corresponding tool detected larger coverage and took less time. Blank triangles are the remaining (non-representative) runs. Triangles on the boundaries represent runs in which one of the tool detected zero coverage.

point is placed on the boundary. The experiments revealed that given the same timeout, HORNTINUUM generates test cases with larger or equal coverage than KLEE on 241 programs, FUSEBMC on 178 programs, and VERIFUZZ on 177 programs. These numbers include cases when the competitor crashed or did not return any coverage but exclude cases when HORNTINUUM did so.

A pairwise comparison between the "runtime/coverage" ratio taken by the tools is shown in Fig. 4. For this experiment, for every plot, we only considered benchmarks, on which either of tools generated test cases with larger coverage, and on which it terminated before the competitor. Specifically:

- 177 (resp. 44) on which VERIFUZZ (resp. HORNTINUUM) was outperformed.
- 128 (resp. 44) on which FUSEBMC (resp. HORNTINUUM) was outperformed.
- 124 (resp. 26) on which KLEE (resp. HORNTINUUM) was outperformed.

These numbers lets us conclude that it is much likely that HORNTINUUM could return *a larger coverage in a shorter amount of time*, then a competitor could

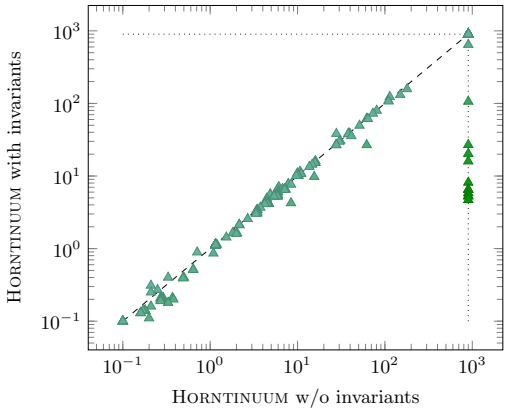


Fig. 5: Impact of invariants: pairs of the runtimes (sec × sec) of HORNTINUUM with and without invariants.

do so. The remaining benchmarks (e.g., on which HORNTINUUM generates more test cases but takes more time than a competitor) are still shown in the plot but are excluded from the statistics: in these cases it is impossible to draw a consistent conclusion on the tools' performance.

Controlled experiment. Lastly we present an interesting statistic on the effect of invariant generation on runtime of test-case generation (Fig. 5). For the sake of experiment, we modified Alg. 1 such that it skips invariant generation but enumerates traces and exploits the unsatisfiable prefixes. It turns out that this negatively affects 184 benchmarks, on which the modified version takes more time. These include 12 benchmarks, on which HORNTINUUM with invariants terminates before the timeout, but HORNTINUUM without invariants does not terminate (represented as points on the right boundary). These benchmarks demonstrate a possible scenario when programs under test have unreachable branches that can be identified by a CHC solver, allowing the test-case generator to terminate earlier.

8 Conclusion

We have shown that CHCs is a promising vehicle that a test-case generators could use in order to improve the quality of solutions and the runtime. Specifically, using CHC encodings of programs, various program unrollings are enumerated, and test cases are extracted from models of satisfiable formulas. Our novel CHC-based approach and its implementation in HORNTINUUM use SMT solvers incrementally. In the future we are going to extend our support for data types and optimize the algorithm for searching *deep counterexamples* a la [6].

Acknowledgments The work is supported in parts by a gift from Amazon Web Services and a grant from FSU's Council on Research & Creativity.

References

1. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs. In: TAP. Lecture Notes in Computer Science, vol. 12740, pp. 85–105. Springer (2021)
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-Guided Synthesis. In: FMCAD. pp. 1–17. IEEE (2013)
3. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 367–381. Springer (2008)
4. Beyer, D., Lemberger, T.: Testcov: Robust test-suite execution and coverage measurement. In: ASE. pp. 1074–1077. IEEE (2019)
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
6. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition Power Abstractions for Deep Counterexample Detection. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg (2022)
7. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. IEEE Trans. Software Eng. **45**(5), 489–506 (2019)
8. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) OSDI. pp. 209–224. USENIX Association (2008)
9. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing - (competition contribution). In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 244–249. Springer (2019)
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
11. Csallner, C., Smaragdakis, Y.: Check 'n' crash: combining static checking and testing. In: Roman, G., Griswold, W.G., Nuseibeh, B. (eds.) ICSE. pp. 422–431. ACM (2005)
12. Fedyukovich, G., Bodík, R.: Accelerating Syntax-Guided Invariant Synthesis. In: TACAS, Part I. LNCS, vol. 10805, pp. 251–269. Springer (2018)
13. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Property directed equivalence via abstract simulation. In: CAV. LNCS, vol. 9780, Part II, pp. 433–453. Springer (2016)
14. Fedyukovich, G., Kaufman, S., Bodík, R.: Sampling Invariants from Frequency Distributions. In: FMCAD. pp. 100–107. IEEE (2017)
15. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving Constrained Horn Clauses Using Syntax and Data. In: FMCAD. pp. 170–178. IEEE (2018)
16. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified Invariants via Syntax-Guided Synthesis. In: CAV, Part I. LNCS, vol. 11561, pp. 259–277. Springer (2019)
17. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) HCVS@ETAPS. EPTCS, vol. 344, pp. 91–108 (2021)
18. Flanagan, C., Leino, K.R.M.: Houdini: an Annotation Assistant for ESC/Java. In: FME. LNCS, vol. 2021, pp. 500–517. Springer (2001)

19. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference - (competition contribution). In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS:, Part III. LNCS, vol. 11429, pp. 209–213. Springer (2019)
20. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI. pp. 206–215. ACM (2008)
21. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: CAV. LNCS, vol. 9206, pp. 343–361. Springer (2015)
22. Jaffar, J., Murali, V., Navas, J.A.: Boosting concolic testing via interpolation. In: Meyer, B., Baresi, L., Mezini, M. (eds.) ESEC/FSE. pp. 48–58. ACM (2013)
23. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
24. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-Based Model Checking for Recursive Programs. In: CAV. LNCS, vol. 8559, pp. 17–34 (2014)
25. Le, H.M.: Llvm-based hybrid fuzzing with libkluzzer (competition contribution). In: Wehrheim, H., Cabot, J. (eds.) FASE. LNCS, vol. 12076, pp. 535–539. Springer (2020)
26. Mathis, B., Gopinath, R., Mera, M., Kampmann, A., Höschele, M., Zeller, A.: Parser-directed fuzzing. In: McKinley, K.S., Fisher, K. (eds.) PLDI. pp. 548–560. ACM (2019)
27. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
28. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) FSE. pp. 263–272. ACM (2005)
29. Serebryany, K.: Continuous fuzzing with libfuzzer and addresssanitizer. In: SecDev. p. 157. IEEE Computer Society (2016)
30. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP. LNCS, vol. 7792, pp. 574–592. Springer (2013)
31. Vikram, V., Padhye, R., Sen, K.: Growing A test corpus with bonsai fuzzing. In: ICSE. pp. 723–735. IEEE (2021)
32. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with java pathfinder. In: Avrunin, G.S., Rothermel, G. (eds.) ISSTA. pp. 97–107. ACM (2004)
33. Wüstholz, V., Christakis, M.: Targeted greybox fuzzing with static lookahead analysis. In: Rothermel, G., Bae, D. (eds.) ICSE. pp. 789–800. ACM (2020)
34. Zalewski, M.: American Fuzzy Lop, https://lcamtuf.coredump.cx/afl/

Efficient Analysis of Cyclic Redundancy Architectures via Boolean Fault Propagation

Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Martin Jonáš(✉)

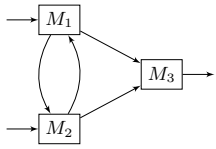
Fondazione Bruno Kessler, Trento, Italy
{cimatti,bozzano,griggio,mjonas}@fbk.eu

Abstract. Many safety critical systems guarantee fault-tolerance by using several redundant copies of their components. When designing such redundancy architectures, it is crucial to analyze their fault trees, which describe combinations of faults of individual components that may cause malfunction of the system. State-of-the-art techniques for fault tree computation use first-order formulas with uninterpreted functions to model the transformations of signals performed by the redundancy system and an AllSMT query for computation of the fault tree from this encoding. Scalability of the analysis can be further improved by techniques such as predicate abstraction, which reduces the problem to Boolean case.
In this paper, we show that as far as fault trees of redundancy architectures are concerned, signal transformation can be equivalently viewed in a purely Boolean way as fault propagation. This alternative view has important practical consequences. First, it applies also to general redundancy architectures with cyclic dependencies among components, to which the current state-of-the-art methods based on AllSMT are not applicable, and which currently require expensive sequential reasoning. Second, it allows for a simpler encoding of the problem and usage of efficient algorithms for analysis of fault propagation, which can significantly improve the runtime of the analyses. A thorough experimental evaluation demonstrates the superiority of the proposed techniques.

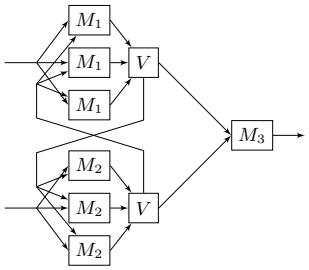
1 Introduction

Fault-tolerance is a fundamental property of safety critical systems that enables their safe operation even in the presence of faults. There are many ways to ensure fault-tolerance, often based on redundancy: spare parts are available for backup and are ready to take over with different degrees of promptness (e.g., hot/warm/cold standby), or with multiple replicas running in parallel. The latter is a common approach to fault-tolerance in computer-based control systems, where the results computed by the independent replicas are combined together by means of voters. The idea dates back to the pioneering space application in Saturn Launch Vehicle [12], and has then been adopted in the Primary Flight Computer [19] of the Boeing 777. The idea is becoming prominent with the advent of modern Integrated Modular Avionics [16], a cost-effective solution for the management of highly intensive software control systems.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 273–291, 2022.
https://doi.org/10.1007/978-3-030-99527-0_15

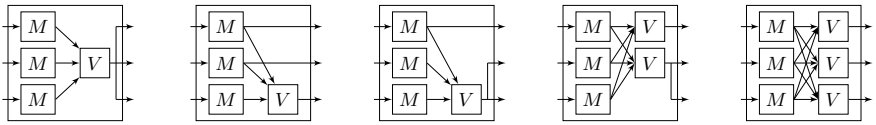


(a) Reference non-redundant system.



(b) TMR redundant system with three replicas of modules M_1, M_2, whose results are combined by a voter.

Fig. 1: Network of computational modules with cyclic dependencies, extended by triple modular redundancy.



(a) V111 (b) V001 (c) V011 (d) V122 (e) V123

Fig. 2: Selected ways of extending a single reference module M with triple modular redundancy (using 1, 2, and 3 voters) [6].

One of the most used instances of the approach to redundancy by using module replicas is the *triple modular redundancy* (TMR) schema, in which the computational modules are replaced by three redundant copies, whose results can be combined by one to three voters. An example of using TMR to add redundancy to a reference non-redundant architecture is shown in Figure 1. Note that there are multiple ways of combining the results of a single triplicated computational module by voters, some of which are shown in Figure 2 [6].

Assessing the actual degree of fault-tolerance of a redundant architecture is directly related to the construction and analysis of the corresponding fault tree [17]. A fault tree describes the combinations of failures of individual components that may cause higher-level malfunction, e.g., bring the system into a dangerous state. Such combinations are traditionally called *cut sets*. Given the set of all cut sets of the system, a fault tree can be reconstructed. Subsequently, from the fault tree expressed as a Binary Decision Diagram, it is possible to compute the reliability of the system from the reliability measures of the components, and to synthesize the analytical form of the reliability function [6].

In this paper, we tackle the problem of automatically analyzing the reliability of redundancy architectures with parallel replicas and voting. We propose a general framework that encompasses also redundancy architectures with cyclic dependencies among components, such as the system from Figure 1, to which

current state-of-the-art approaches [6] are not applicable. The modeling is based on symbolic transition systems over the quantifier-free theory of linear real arithmetic and uninterpreted functions (UFLRA). In particular, real numbers are used to represent the signals of the architecture and multiple instances of the same uninterpreted function symbol are used to represent component replicas. The modeling framework is a strict generalization of the combinational approach proposed in [4,5], that only allows for acyclic architectures.

As the main contribution, we propose an analysis technique based on the reduction to *fault propagation graphs* over Boolean structures [7]. We prove that the reduction is correct: the signal transformation performed by a redundancy architecture can be equivalently viewed in a Boolean way as fault propagation.

We carry out a systematic experimental evaluation on the set of redundancy architectures with cyclic dependencies to evaluate scalability of the proposed solution. Moreover, we perform evaluation on acyclic redundancy architectures to compare the performance against the state-of-the-art approach based on predicate abstraction [5,6], which can be applied only to redundancy architectures without cycles. The proposed approach proves to be very scalable, being able to analyze cyclic architectures with thousands of nodes, and is dramatically more efficient than a direct reduction to model checking of symbolic transition systems over UFLRA. In the restricted set of acyclic benchmarks, the proposed approach provides better performance even over the optimized method proposed in [5] and extended in [6] that adopts a structural form of predicate abstraction to improve over basic AllSMT [14].

The paper is structured as follows. In Section 2, we present logical preliminaries and basic notions of fault propagation graphs. In Section 3, we describe the framework of redundancy architectures with cycles. In Section 4, we present the reduction to fault propagation and prove its correctness. In Section 5, we discuss the related work. The experiments are presented in Section 6. In Section 7, we draw some conclusions and discuss some directions for future work.

2 Preliminaries

2.1 General Background

In this section, we explain the basic mathematical conventions that are used in the paper. We assume that the reader is familiar with standard first-order logic and the basic ideas of Satisfiability Modulo Theories (SMT), as presented e.g. in [1]. A theory in the SMT sense is a pair $(\Sigma, \mathcal{C})$, where Σ is a first-order signature and $\mathcal{C}$ is a class of models over Σ. We use the standard notions of interpretation, assignment, model, satisfiability, validity, and logical consequence. We refer to 0-arity predicates as Boolean variables, and to 0-arity uninterpreted functions as (theory) variables. We denote variables with $x, y, \ldots$, formulas with $\varphi, \psi, \ldots$, and uninterpreted functions with $f, g, \ldots$, possibly with subscripts. We denote vectors with $\bar{\cdot}$ (e.g. $\overline{x}$), and individual components with subscripts (e.g. x_j). We denote the domain of Booleans with $\mathbb{B} = \{\top, \bot\}$. If $x_1, \ldots, x_n$ are variables and

φ is a formula, we write $\varphi(x_1, \ldots, x_n)$ to indicate that all the variables occurring free in φ are in $x_1, \ldots, x_n$. If φ is a formula without uninterpreted functions and μ is a function that maps each free variable of φ to a value of the corresponding sort, $[\![\varphi]\!]_\mu$ denotes the result of the evaluation of φ under this assignment. A Boolean formula is called *positive* if it does not use other logical connectives than conjunctions and disjunctions.

In this paper, we shall use the theory of linear real arithmetic (LRA), in which the numeric constants and the arithmetic and relational operators have their standard meaning, extended with uninterpreted functions (UF), whose interpretation is not fixed in $\mathcal{C}$, and with *voters* (V), which are k-ary functions whose interpretation is the majority function defined as below. For simplicity, we consider only voters with odd arity as even-arity voters are rarely used in practice. However, our approach can be extended to support even-arity voters.

Definition 1. *The k-ary majority function majority*$: \mathbb{R}^k \to \mathbb{R}$ *for an odd $k > 0$ is defined by majority*$(\overline{x}) = y$ *if there is y such that $y = x_j$ for at least $\lceil k/2 \rceil$ distinct j and majority*$(\overline{x}) = x_1$ *otherwise.*

Given a set of variables $\overline{x}$, we denote with $\overline{x}'$ the set $\{x' \mid x \in \overline{x}\}$. A *symbolic transition system* S is a triple $(\overline{x}, I(\overline{x}), T(\overline{x}, \overline{x}'))$, where $\overline{x}$ is a set of variables, and $I(\overline{x})$, $T(\overline{x}, \overline{x}')$ are formulae over some signature. An assignment to the variables in $\overline{x}$ is a *state* of S. A state s is initial iff it is a model of $I(\overline{x})$, i.e., $s \models I(\overline{x})$. The states s, s' denote a transition iff $s \cup s' \models T(\overline{x}, \overline{x}')$, also written $T(s, s')$. A *trace* is a sequence of states $s_0, s_1, \ldots$ such that s_0 is initial and $T(s_i, s'_{i+1})$ for all i. We denote traces with π, and with π_j the j-th element of π. A state s is *reachable* in S iff there exists a trace π such that $\pi_i = s$ for some i.

2.2 Fault Propagation Graphs

In this section we briefly introduce the necessary notions of fault propagation, and in particular the formalism of *symbolic fault propagation graphs*. Intuitively, fault propagation graphs can be used to describe how failures of some components of a given system can cause the failure of other components of a system. In an explicit (hyper)graph representation, components can be represented by nodes, and dependencies by edges among them, with the meaning that an edge from component c_1 to component c_2 states that the failure of c_1 can cause the failure (propagation) of c_2. In the symbolic representation adopted here, we model components as Boolean variables (where $\bot$ means "not failed" and $\top$ means "failed"), and express the dependencies as Boolean formulae encoding the conditions that can lead to the failure of each component. The basic concepts are formalized in the following definitions. For more information, we refer to [7].

Definition 2 (Fault propagation graph). *A symbolic fault propagation graph (*FPG*) is a pair $(C, canFail)$, where C is a finite set of* system components *and canFail is a function that assigns to each component c a Boolean formula canFail(c) over the set of variables C.*

Definition 3 (Trace of FPG). *Let G be a fault propagation graph $(C, canFail)$. A* state *of G is a function from C to $\mathbb{B}$. A trace of G is a sequence of states $\pi = \pi_0\pi_1 \ldots \in (\mathbb{B}^C)^\omega$ such that all $i > 0$ and $c \in C$ satisfy (i) $\pi_i(c) = \pi_{i-1}(c)$ or (ii) $\pi_{i-1}(c) = \bot$ and $\pi_i(c) = [\![canFail(c)]\!]_{\pi_{i-1}}$.*

Example 1 ([7]). Consider a system with components *control on ground* (G), *hydraulic control* (H), and *electric control* (E) such that G can fail if both H and E have failed, H can fail if E has failed, and E can fail if H has failed. This system can be modeled by a fault propagation graph $(\{\text{G}, \text{E}, \text{H}\}, canFail)$, where $canFail(\text{G}) = \text{H} \wedge \text{E}$, $canFail(\text{H}) = \text{E}$, and $canFail(\text{E}) = \text{H}$.

One of the traces of this system is $\{\text{G} \mapsto \bot, \text{H} \mapsto \top, \text{E} \mapsto \bot\}\{\text{G} \mapsto \bot, \text{H} \mapsto \top, \text{E} \mapsto \top\}\{\text{G} \mapsto \top, \text{H} \mapsto \top, \text{E} \mapsto \top\}^\omega$, where H is failed initially, which causes failure of E in the second step, and the failures of H and E together cause a failure of G in the third step.

Fault propagation graphs are often used to identify sets of initial faults that can lead the system to a dangerous or unwanted state (usually called a *top level event*). Such sets of initial faults are called *cut sets*.

Definition 4 (Cut set). *Let G be a fault propagation graph $G = (C, canFail)$ and φ a positive Boolean formula, called* top level event. *The assignment $cs\colon C \to \mathbb{B}$ is called a* cut set *of G for φ if there is a trace π of G that starts in the state cs and there is some $k \geq 0$ such that $\pi_k \models \varphi$. A cut set cs is called* minimal cut set *if it is minimal with respect to the pointwise ordering of functions $\mathbb{B}^C$, i.e., there is no other cut set cs' such that $\{c \in C \mid cs'(c) = \top\} \subsetneq \{c \in C \mid cs(c) = \top\}$.*

For brevity, when talking about cut sets, we often mention only the components that are set to $\top$ by the cut set.

Example 2 ([7]). The minimal cut sets of the FPG from Example 1 for the top level event $\varphi = \text{G}$ are $\{\text{G}\}$, $\{\text{H}\}$, and $\{\text{E}\}$. These three cut sets are witnessed by the following traces:

1. $\{\text{G} \mapsto \top, \text{H} \mapsto \bot, \text{E} \mapsto \bot\}^\omega$,
2. $\{\text{G} \mapsto \bot, \text{H} \mapsto \top, \text{E} \mapsto \bot\}\{\text{G} \mapsto \bot, \text{H} \mapsto \top, \text{E} \mapsto \top\}\{\text{G} \mapsto \top, \text{H} \mapsto \top, \text{E} \mapsto \top\}^\omega$,
3. $\{\text{G} \mapsto \bot, \text{H} \mapsto \bot, \text{E} \mapsto \top\}\{\text{G} \mapsto \bot, \text{H} \mapsto \top, \text{E} \mapsto \top\}\{\text{G} \mapsto \top, \text{H} \mapsto \top, \text{E} \mapsto \top\}^\omega$.

Note that the FPG has also other cut sets, such as $\{\text{G}, \text{E}\}$, $\{\text{H}, \text{E}\}$, and $\{\text{G}, \text{H}, \text{E}\}$, which are not minimal.

In the following, we work with fault propagation graphs whose all *canFail* formulas are positive. Such fault propagation graphs are called *monotone*. Note that the definition of trace ensures that in each trace, if a component c is set to $\top$ in a state π_i, it is $\top$ in all the subsequent states π_j for $j > i$. This ensures that each trace eventually reaches a fixed point. Moreover, before reaching this fixed point, the trace can contain at most $|C|$ distinct states.

For monotone FPGs, there is an efficient algorithm for minimal cut set enumeration [7]. This approach consists in enumerating of the minimal models of a specific LRA formula, in which theory constraints are used only if the input FPG contains cycles (and which therefore is purely Boolean for acyclic FPGs).

3 Cyclic Redundancy Architectures

In this section, we describe the framework adopted to model redundancy architectures, in form of a restricted class of symbolic transition systems modulo UFLRA. We call this restricted class *transition systems with uninterpreted functions and voters (UF+V TS)*.[1] This modeling framework is more expressive than mere SMT formulas modulo UFLRA, which were used in the previous works on analysis of redundancy architectures [6], as it can express architectures that contain cyclic dependencies among the modules.

Definition 5 (UF+V transition system). *A* transition system with uninterpreted functions and voters *is a tuple* $(V_S, V_{\text{in}}, V_{\text{init}}, T_{\text{next}}, T_{\text{init}})$*, where*

- V_S *is a finite set of real-valued* signal variables*;*
- V_{in} *with* $V_S \cap V_{\text{in}} = \emptyset$ *is a finite set of real-valued* input variables*;*
- V_{init} *is a finite set of real-valued* initial value variables*;*
- $T_{\text{next}} \colon V_S \to$ *Expr is a transition function, where Expr is the set of all expressions of form* $f(x_1, x_2, \ldots, x_k)$ *for* $k \geq 0$*,* $x_i \in (V_S \cup V_{\text{in}})$*, and where* f *is either an uninterpreted function symbol of arity* k *or the function symbol* voter_k *with an odd* $k > 0$*;*
- T_{init} *is an* initial value mapping *that assigns an initial value variable* $T_{\text{init}}(v) \in V_{\text{init}}$ *to each signal* $v \in V_S$ *for which* $T_{\text{next}}(v) = f(\overline{x})$ *for an uninterpreted* f*.*

A UF+V transition system is called *well formed* if it does not contain cyclic dependencies among voters, i.e., there is no sequence $v_1 \ldots v_n$ of signal variables such that $v_1 = v_n$ and each v_i with $i > 0$ satisfies $T_{\text{next}}(v_i) = voter_k(x_1, \ldots, x_k)$ with $x_j = v_{i-1}$ for some $1 \leq j \leq k$. For well formed UF+V TS, we can define *voter depth* $vd \colon V_S \cup V_{\text{in}} \to \mathbb{N}$ as the unique solution to the following set of equations: $vd(in) = 0$ for each $in \in V_{\text{in}}$, $vd(s) = 0$ for each $v \in V_S$ such that $T_{\text{next}}(v) = f(x_1, x_2, \ldots, x_k)$, and $vd(v) = \max\{vd(x_i) \mid 1 \leq i \leq k\} + 1$ for each $v \in V_S$ such that $T_{\text{next}}(v) = voter_k(x_1, x_2, \ldots, x_k)$.

In the rest of the paper, we assume that all UF+V TS are well formed. In the rest of this section, let us fix an arbitrary well formed UF+V transition system $S = (V_S, V_{\text{in}}, V_{\text{init}}, T_{\text{next}}, T_{\text{init}})$.

We now give a formal definition of the behavior of the UF+V system in presence of faults. Intuitively, we are given the set Fa lts of faulty signal-producing components of the system, which do not have to behave correctly: a faulty component neither has to start in its specified initial value nor respect its transition function.

Definition 6 (Trace of UF+V TS). *A state of a UF+V transition system* S *is an arbitrary assignment of real numbers to signal and input variables* $s \colon (V_S \cup V_{\text{in}}) \to \mathbb{R}$*.*

[1] Note than although UF+V TS and the related concepts can be defined directly in terms of UFLRA symbolic transition systems, we chose to make the definition explicit to simplify the presentation and proofs.

The sequence of states $\pi = \pi_0\pi_1 \ldots \in (\mathbb{R}^{V_S \cup V_{\text{in}}})^\omega$ *is called a* trace of the system S for the fault set $\mathsf{Fa}\ \mathsf{Its} \subseteq V_S$, input stream $\iota = \iota_0\iota_1 \ldots \in (\mathbb{R}^{V_{\text{in}}})^\omega$, initial value assignment $\text{Init}\colon V_{\text{init}} \to \mathbb{R}$, and interpretation $[\![\_]\!]$, *which to each uninterpreted function symbol of arity* k *assigns a function* $[\![f]\!]\colon \mathbb{R}^k \to \mathbb{R}$, *if:*

- $\pi_i(in) = \iota_i(in)$ *for all* $i \geq 0$ *and* $in \in V_{\text{in}}$.
- *For* $v \in V_S \setminus \mathsf{Fa}\ \mathsf{Its}$ *such that* $T_{\text{next}}(v) = f(x_1, \ldots, x_k)$ *with an uninterpreted function symbol* f, *it is the case that* $\pi_0(v) = \text{Init}(T_{\text{init}}(v))$ *and all* $i > 0$ *satisfy* $\pi_i(v) = [\![f]\!](\pi_{i-1}(x_1), \ldots, \pi_{i-1}(x_k))$.
- *For all* $i \geq 0$ *and* $v \in V_S \setminus \mathsf{Fa}\ \mathsf{Its}$ *such that* $T_{\text{next}}(v) = voter_k(x_1, \ldots, x_k)$, *it is the case that* $\pi_i(v) = majority(\pi_i(x_1), \ldots, \pi_i(x_k))$.

Traces for the fault set $\mathsf{Fa}\ \mathsf{Its} = \emptyset$ *are called* nominal.

Note that each uninterpreted module needs one time step to compute its result, while the results of voters are instantaneous. The time delay for modules allows cyclic dependencies among modules, while no delay for voters gives the expected semantics to architectures where some replicas of a module are guarded by a voter and others are not, such as in schemas from Figures 2b and 2c.

Example 3. Consider the example from Figure 1, where the reference system with 3 modules M_1, M_2, and M_3 is extended with TMR such that the modules M_1 and M_2 are replaced by three replicas whose results are combined by a voter.

We can represent the redundancy version of the system as a UF+V TS as follows. The nominal behavior of the modules M_1, M_2, and M_3 is represented by binary uninterpreted functions f_1, f_2, and f_3, respectively. Further, we represent initial values of M_1, M_2, M_3 by variables $init_{m_1}$, $init_{m_2}$, and $init_{m_3}$ respectively. Finally, we represent the output of i-th replica of each module M_j by a signal variable x_j^i and the output of the voter corresponding to the module M_j by a signal variable x_j^v.

This gives the UF+V transition system $S = (V_S, \{in_1, in_2\}, V_{\text{init}}, T_{\text{next}}, T_{\text{init}})$, with $V_S = \{x_1^1, x_1^2, x_1^3, x_1^v, x_2^1, x_2^2, x_2^3, x_2^v, x_3^1\}$, $V_{\text{init}} = \{init_{m_j} \mid j \in \{1,2,3\}\}$, and

$$T_{\text{next}}(x_1^i) = f_1(in_1, x_2^v) \text{ for } 1 \leq i \leq 3, \qquad T_{\text{init}}(x_1^i) = init_{m_1} \text{ for } 1 \leq i \leq 3,$$
$$T_{\text{next}}(x_2^i) = f_2(in_2, x_1^v) \text{ for } 1 \leq i \leq 3, \qquad T_{\text{init}}(x_2^i) = init_{m_2} \text{ for } 1 \leq i \leq 3,$$
$$T_{\text{next}}(x_3^1) = f_3(x_1^v, x_2^v), \qquad T_{\text{init}}(x_3^1) = init_{m_3},$$
$$T_{\text{next}}(x_j^v) = voter_3(x_j^1, x_j^2, x_j^3) \text{ for } j \in \{1,2\}.$$

We define the class of *redundancy* transition systems, where the only purpose of all voters is to recognize and repair outputs of failed components; more specifically, if all components behave correctly, the voters are not necessary.

Definition 7 (Redundancy UF+V TS). *We call the system* S *a* redundancy UF+V transition system *if in all its nominal traces, all inputs of each voter are always identical. Formally, if* π *is any nominal trace of* S *and if* v *is a variable for which* $T_{\text{next}}(v) = voter_k(\overline{x})$, *then* $\left|\{\pi_i(x_j) \mid 1 \leq j \leq k\}\right| = 1$ *for all* $i \geq 0$.

Similarly to FPGs, a cut set is a set of faults that leads to the undesired behavior of the system. In particular, given a set of signals that are considered as *output signals* (or *outputs*) of the system, a cut set of the given UF+V TS is a set of faults that can cause an incorrect value of at least one output.

Definition 8 ((Minimal) cut set). *A fault set* $\mathsf{Fa}\ \mathsf{lts} \subseteq V_S$ *is called a* cut set *of* S *for a set of output signals* $V_{\text{out}} \subseteq V_S$ *if there exist an input stream, initial value assignment, and an interpretation such that values of output signals of some trace* π *for the fault set* $\mathsf{Fa}\ \mathsf{lts}$ *differ from the outputs of the nominal trace* π^{nom} *with the same input stream, initial values, and interpretation, i.e., there is* $c \geq 0$ *and* $o \in V_{\text{out}}$ *for which* $\pi_c(o) \neq \pi_c^{nom}(o)$*. A cut set is called minimal (*MCS*) if it is minimal in terms of set inclusion.*

Since the redundancy UF+V TS form a subclass of UFLRA transition systems, there is a straightforward procedure for minimal cut set enumeration. As in the case of combinational systems [6], one can construct a *miter system*, which consists of two copies of the architecture: the first is allowed to fail and the second is constrained to behave nominally. Minimal cut sets can then be obtained by using a technique based on symbolic model checking [3] to enumerate all minimal assignments to fault variables under which it is possible to reach some state in which the outputs of the two copies differ.

4 Reducing Redundancy UF+V TS to Fault Propagation Graphs

In this section, we show the main result of the paper, which is that minimal cut set enumeration of redundancy UF+V transition systems can be reduced to minimal cut set enumeration of Boolean fault propagation graphs, which is more efficient than MCS enumeration based on miter construction and model checking.

4.1 Reduction

We for each UF+V system S define a corresponding FPG S^B. The components of S^B correspond to the signal variables of the original system S. With a slight abuse of notation, we use the same names for the original real-valued signal variables of S and the components of S^B, although they have different types. Intuitively, the reduction ensures that each component v of S^B can fail if and only if there is a trace of S in which the value of the signal variable v deviates from its nominal value.

Definition 9. *Let* $S = (V_S, V_{\text{in}}, V_{\text{init}}, T_{\text{next}}, T_{\text{init}})$ *be a UF+V TS. We define a corresponding* FPG $S^B = (V_S, canFail)$*, where* $canFail(v) = \bigvee_{v' \in \overline{x} \cap V_S} v'$ *if* $T_{\text{next}}(v) = f(\overline{x})$ *and* $canFail(v) = atLeast_{\lceil k/2 \rceil}(\overline{x} \cap V_S)$ *if* $T_{\text{next}}(v) = voter_k(\overline{x})$*, using the definition* $atLeast_m(X) = \bigvee_{\substack{Y \subseteq X \\ |Y|=m}} \bigwedge_{y \in Y} y$.[2]

[2] Note that there are more efficient and compact encodings for the *atLeast* constraint [18]; we use the most simple one for presentation purposes.

Example 4. Consider the transition system S from Example 3. The corresponding fault propagation graph is $S^B = (\{x_1^1, x_1^2, x_1^3, x_1^v, x_2^1, x_2^2, x_2^3, x_2^v, x_3^1\}, \mathit{canFail})$, where

$$\mathit{canFail}(x_1^i) = x_2^v \text{ for all } 1 \le i \le 3, \qquad \mathit{canFail}(x_2^i) = x_1^v \text{ for all } 1 \le i \le 3,$$
$$\mathit{canFail}(x_3^1) = x_1^v \vee x_2^v,$$
$$\mathit{canFail}(x_1^v) = \mathit{atLeast}_2(x_1^1, x_1^2, x_1^3), \qquad \mathit{canFail}(x_2^v) = \mathit{atLeast}_2(x_2^1, x_2^2, x_2^3).$$

4.2 Correctness

We show that the reduction preserves the cut sets. In the rest of the section, let $S = (V_S, V_{\text{in}}, V_{\text{init}}, T_{\text{next}}, T_{\text{init}})$ be an arbitrary redundancy UF+V TS, $\mathsf{Fa\ lts} \subseteq V_S$ be an arbitrary fault set, and $V_{\text{out}} \subseteq V_S$ be an arbitrary set of output signals. First, we show that each cut set of S corresponds to a cut set of S^B.

Lemma 1. *If* $\mathsf{Fa\ lts}$ *is a cut set of* S *for the set of outputs* V_{out}*, then* cs *defined as* $cs(v) = \top$ *iff* $v \in \mathsf{Fa\ lts}$ *is a cut set of* S^B *for the top level event* $\bigvee_{o \in V_{\text{ou}}} o$*.*

Proof. Let $\mathsf{Fa\ lts}$ be a cut set of S for some trace π for some ι, Init, and $[\![\_]\!]$. Let π^{nom} be the corresponding nominal trace. Define the trace π^B of S^B as $\pi_0^B = cs$ and for all $i > 0$ define π_i^B by $\pi_i^B(v) = \top$ if $\pi_{i-1}^B(v) = \top$ and $\pi_i^B(v) = [\![\mathit{canFail}(v)]\!]_{\pi_{i-1}^B}$ if $\pi_{i-1}^B(v) = \bot$. In other words, π^B is the unique trace starting in cs in which all the components fail as soon as possible. By monotonicity, the trace π^B has a fixed point, i.e., there is n such that $\pi_n^B = \pi_{n'}^B$ for all $n' > n$.

We show that π^B satisfies $\pi_n^B(o) = \top$ for some $o \in V_{\text{out}}$ and thus cs is a cut set for the top level event $\bigvee_{o \in V_{\text{ou}}} o$. To do this, we prove by induction on i and on the voter depth $vd(v)$[3] that for all $v \in V_S$ and $i \ge 0$, $\pi_i(v) \neq \pi_i^{nom}(v)$ implies $\pi_n^B(v) = \top$. We distinguish three cases:

- If $v \in \mathsf{Fa\ lts}$, then $\pi_0^B(v) = \top$. From the definition of π^B, this implies that $\pi_l^B(v) = \top$ for all $l \ge 0$. In particular, $\pi_n^B(v) = \top$.
- If $v \notin \mathsf{Fa\ lts}$ and $T_{\text{next}}(v) = f(x_1, \ldots, x_k)$, we distinguish two cases:
  - If $i = 0$: since $\pi_0(v) \neq \pi_0^{nom}(v)$, then it must be the case that $\pi_0(v) \neq \text{Init}(T_{\text{init}}(v))$, therefore $v \in \mathsf{Fa\ lts}$. This is a contradiction.
  - If $i > 0$: then $\pi_i(v) \neq \pi_i^{nom}(v)$ by definition implies

$$[\![f]\!](\pi_{i-1}(x_1), \ldots, \pi_{i-1}(x_k)) \neq [\![f]\!](\pi_{i-1}^{nom}(x_1), \ldots, \pi_{i-1}^{nom}(x_k))$$

    and hence $\pi_{i-1}(x_j) \neq \pi_{i-1}^{nom}(x_j)$ for some $1 \le j \le k$ because $[\![f]\!]$ is a function. Since $\pi_{i-1}(in) = \pi_{i-1}^{nom}(in)$ holds for all $in \in V_{\text{in}}$, we know that $x_j \in V_S$. Therefore the induction hypothesis implies $\pi_n^B(x_j) = \top$ and thus $\pi_{n+1}^B(v) = \top$ because π_n^B satisfies $\mathit{canFail}(v)$. Since π_n^B was chosen as the fixed point of π^B, this implies $\pi_n^B(v) = \pi_{n+1}^B(v) = \top$.

[3] Induction on the voter depth is employed because UF+V transition systems propagate results of voters instantaneously.

– If $v \notin \mathsf{Fa\ lts}$ and $T_{\text{next}}(v) = \mathit{voter}_k(x_1, \ldots, x_k)$, then $\pi_i(v) \neq \pi_i^{nom}(v)$ for any $i \geq 0$ by definition implies

$$\mathit{majority}(\pi_i(x_1), \ldots, \pi_i(x_k)) \neq \mathit{majority}(\pi_i^{nom}(x_1), \ldots, \pi_i^{nom}(x_k)). \quad (1)$$

Since S is a redundancy TS, all $\pi_i^{nom}(x_j)$ are equal and the disequality (1) implies that $\pi_i(x_j) \neq \pi_i^{nom}(x_j)$ for at least $\lceil k/2 \rceil$ of x_j. All these x_j are not in V_{in} and must therefore be in V_S. By definition of voter depth, $vd(x_j) < vd(v)$ for all these x_j. Therefore by the induction hypothesis $\pi_n^B(x_j) = \top$ for at least $\lceil k/2 \rceil$ of x_j and thus $\pi_{n+1}^B(v) = \top$ because π_n^B satisfies $\mathit{canFail}(v)$. This again implies $\pi_n^B(v) = \pi_{n+1}^B(v) = \top$ because π_n^B is the fixed point of π^B.

This finishes the proof: if $\mathsf{Fa\ lts}$ is a cut set, $\pi_c(o) \neq \pi_c^{nom}(o)$ for some $c \geq 0$ and $o \in V_{\text{out}}$, and thus $\pi_n^B(o) = \top$. Therefore we know that $\pi_n^B \models \bigvee_{o \in V_{\text{ou}}} o$ and thus cs is a cut set of S^B. □

For the converse direction, we for each fault set devise a trace of the UF+V TS S that propagates all the possible deviations from the nominal value. We call this trace *maximally fault-propagating*. In this trace, all signal values are from the set $\{0, 1\}$, all nominal signal values are 0 and become 1 only as a result of a fault. Moreover, if there is a trace for the given fault set in which a signal deviates from its nominal value, the value of the corresponding signal in the maximally fault-propagating will be 1.

Definition 10 (Maximally fault-propagating trace). *Let S be a UF+V TS. Define*

– $\iota_i(v_{in}) = 0$ *for all* $i \geq 0$, $v_{in} \in V_{\text{in}}$, *i.e., ι is a stream of constant zero inputs;*
– $\text{Init}(v_{init}) = 0$ *for each* $v_{init} \in V_{\text{init}}$*; and*
– $[\![f]\!](x_1, \ldots, x_k) = 1 - \prod_{1 \leq i \leq k}(1 - x_i)$ *for each uninterpreted f, i.e., the output is 0 if all inputs are 0; it is 1 if at least one input is 1.*

The maximally fault-propagating trace of S *for a fault set* $\mathsf{Fa\ lts}$*, denoted as π^{fp}, is the unique trace of S for the above input stream, initial values, interpretation, and the given fault set that for all $i \geq 0$ and v satisfies $\pi_i^{fp}(v) = 1$ whenever $v \in \mathsf{Fa\ lts}$.*

Observe that the trace π^{fp} is *monotone*, i.e., once a signal gets set to 1, it stays set to 1 for the rest of the trace. This is formalized by the following lemma, which can be proven by induction on i, $j - i$, and voter depth of v.

Lemma 2. *Let S be a UF+V TS,* $\mathsf{Fa\ lts}$ *a fault set, and π^{fp} the corresponding maximally fault-propagating trace. Then $\pi_i^{fp}(v) = 1$ for each $i \geq 0$ and $v \in V_S$ implies $\pi_j^{fp}(v) = 1$ for all $j > i$.*

We can now show that if a trace of the FPG version S^B of a UF+V TS S triggers the top level event for some initial fault assignment, there is a trace in the original system S for the corresponding fault set whose output deviates from the nominal one; namely it is the trace π^{fp}.

Lemma 3. *If cs defined as* $cs(v) = \top$ *iff* $v \in$ Fa Its *is a cut set of* S^B *for the top level event* $\bigvee_{o \in V_{\text{ou}}} o$*, then* Fa Its *is a cut set of* S *for the set of outputs* V_{out}*.*

Proof. Suppose that the trace π^B of S^B with the initial state cs satisfies $\pi^B_c(o) = \top$ for some $c \geq 0$ and $o \in V_{\text{out}}$. We show that Fa Its is a cut set of S for the set of output signals V_{out}. Let π^{fp} be the maximally fault-propagating trace of S for Fa Its and π^{nom} the corresponding nominal trace.

We show that for each $i \geq 0$ and $v \in V_S$, the condition $\pi^B_i(v) = \top$ implies $\pi^{fp}_i(v) \neq \pi^{nom}_i(v)$. We proceed by induction on i:

- For $i = 0$: If $cs = \pi^B_0(v) = \top$, then $v \in$ Fa Its and thus $\pi^{fp}_0(v) \neq \pi^{nom}_0(v)$ because $\pi^{fp}_0(v) = 1$ and $\pi^{nom}_0(v) = 0$.
- For $i > 0$: Assume that $\pi^B_i(v) = \top$. We distinguish four cases:
  - If $v \in$ Fa Its then $\pi^{fp}_i(v) = 1$ and so $\pi^{fp}_i(v) \neq \pi^{nom}_i(v) = 0$.
  - If $\pi^B_{i-1}(v) = \top$, then we get that $\pi^{fp}_{i-1}(v) \neq \pi^{nom}_{i-1}(v)$ from the induction hypothesis, and thus $\pi^{fp}_i(v) \neq \pi^{nom}_i(v)$ by Lemma 2.
  - If $v \notin$ Fa Its, $\pi^B_{i-1}(v) = \bot$, and $T_{\text{next}}(v) = f(x_1, \ldots, x_k)$, then $\pi^B_i(v) = \top$ implies that $\pi^B_{i-1}(x_j) = \top$ for at least one $x_j \in V_S$. From the induction hypothesis, we get that $\pi^{fp}_{i-1}(x_j) \neq \pi^{nom}_{i-1}(x_j)$ and since $\pi^{nom}_{i-1}(x_j) = 0$, we know that $\pi^{fp}_{i-1}(x_j) = 1$. By the definition of $[\![f]\!]$ in π^{fp}_i, we know that also $\pi^{fp}_i(v) = 1$, which is not equal to $\pi^{nom}_i(v) = 0$.
  - If $v \notin$ Fa Its, $\pi^B_{i-1}(v) = \bot$, and $T_{\text{next}}(v) = voter_k(x_1, \ldots, x_k)$, then $\pi^B_i(v) = \top$ implies that at least $\lceil k/2 \rceil$ of $x_j \in V_S$ satisfy $\pi^B_{i-1}(x_j) = \top$. From the induction hypothesis we get that $\pi^{fp}_{i-1}(x_j) \neq \pi^{nom}_{i-1}(x_j)$ for these x_j and since $\pi^{nom}_{i-1}(x_j) = 0$, we know that $\pi^{fp}_{i-1}(x_j) = 1$ for at least $\lceil k/2 \rceil$ of x_j. By the definition of majority function, we know that also $\pi^{fp}_{i-1}(v) = 1$ and thus, by Lemma 2, also $\pi^{fp}_i(v) = 1 \neq 0 = \pi^{nom}_i(v)$.

Therefore $\pi^B_c(o) = \top$ implies $\pi^{fp}_c(o) \neq \pi^{nom}_c(o)$ and Fa Its is a cut set of S. □

Theorem 1. *For each fault set* Fa Its*, the following two claims are equivalent:*

1. *The set* Fa Its *is a cut set of* S *for the set of output signals* V_{out}*.*
2. *The assignment cs defined as* $cs(v) = \top$ *iff* $v \in$ Fa Its *is a cut set of* S^B *for the top level event* $\bigvee_{o \in V_{\text{ou}}} o$*.*

5 Related Work

Approaches to the analysis of redundant architectures include [6], which addresses the generation of the reliability function for a class of generic architectures including tree- and DAG-like structures. The computation of the reliability is based on predicate abstraction and BDDs. Our work extends and improves the approach of [6] in several directions. First, it supports cyclic architectures, to which predicate abstraction as defined in [6] cannot be applied. Second, it does not require that the redundancy is localized within small blocks (manually

defined by the user or in a library), to which the predicate abstraction can be applied. In contrast, our approach applies the abstraction directly on the level of individual modules and voters. Moreover, the approach of [6] needs to compute the abstracted versions of the specified blocks upfront by quantifier elimination. Finally, our approach outperforms the approach of [6].

Other works on redundant architecture analysis are either based on ad-hoc algorithms [13] which are not fully automated, and require discretization and additional input data from the user, or use simulation techniques such as Monte Carlo analysis [15], which do not examine the system behaviors exhaustively.

A classification of fault tolerant architectures is presented in [10]. The classification is based on three different patterns, namely comparison, voting, and sparing, that can be composed to define generic and possibly cyclic architectures. A follow-up work [11] builds upon these patterns and introduces strategies to evaluate several architectures at once (family-based analysis of redundant architectures) by reduction to Discrete Time Markov Chains. Our techniques are orthogonal, and could be applied on top of the approach proposed in [11].

The concept of *maximally fault-propagating trace* used to prove Lemma 3 is similar to the concept of *maximally diverse interpretations* [8], which can be used to efficiently reduce a formula in the positive fragment of EUF logic to a SAT formula. Both concepts restrict the interpretations of uninterpreted functions to a specific subclass, which exhibits all the relevant behaviors.

6 Experimental Evaluation

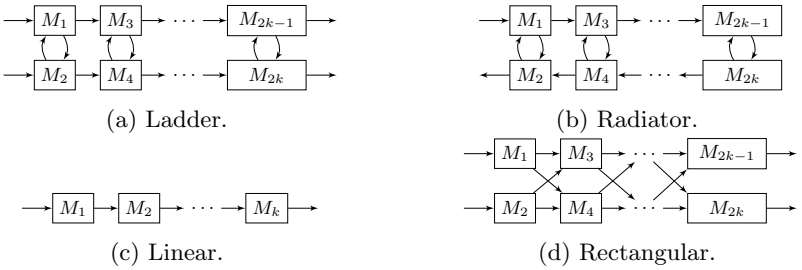
We have performed an experimental evaluation of the proposed approach for minimal cut set enumeration in order to answer the following research questions:

RQ1 How does the new approach scale on redundancy architectures *with cycles*?
RQ2 On redundancy architectures *with cycles*, how do the run-times compare against the approach based on the enumeration of minimal cut sets of the miter system by a model checker?
RQ3 On redundancy architectures *without cycles*, how do the run-times compare against the approach based on predicate abstraction (PA) and BDD-based enumeration [6]?
RQ4 On redundancy architectures *without cycles*, what part of the runtime difference is caused by the different reduction to a Boolean problem (FPG vs PA) and what part is caused by a different solving approach of the resulting Boolean problem (SAT-based vs BDD-based)?

6.1 Benchmarks and Setup

To answer these research questions, we used four sets of redundancy systems:

Scalable cyclic systems This benchmark set contains two kinds of benchmarks. For evaluation on redundancy architectures with a linear number



(a) Ladder. (b) Radiator.

(c) Linear. (d) Rectangular.

Fig. 3: Scalable architectures used in the experimental evaluation.

of cycles, we have generated ladder-shaped (Figure 3a) architectures of all lengths between 1 and 100. For evaluation on redundancy architectures with a large number of cycles, we have generated radiator-shaped (Figure 3b) architectures of all lengths between 1 and 50. For each of the architectures, we have generated its three redundant versions by replacing each module by a TMR block with one to three voters by using schemas from Figures 2b, 2d, and 2e. This yields systems with $2 \cdot \mathit{length} \cdot (3 + \mathit{numVoters})$ signals.

Random cyclic systems We have generated 250 random cyclic redundancy UF+V systems with 1 to 150 modules of arity between 1 and 3, randomly generated 1 to 6 replicas of each module, and 1 to 6 voters of arity 3 or 5, randomly connected to the replicas.

Scalable acyclic systems This benchmark set contains linear-shaped (Figure 3c) and rectangular-shaped (Figure 3d) architectures of all lengths between 1 and 200 that were used for evaluation of predicate abstraction technique [6]. As in the original paper, we have used redundant versions of the systems with the modules replaced by a TMR block with one to three voters.

Random acyclic systems We have used randomly generated acyclic architectures composed of randomly chosen TMR blocks that were also used in [6].

We have evaluated the following approaches for minimal cut set enumeration:

- For the systems *with cycles*, we have generated their FPG version as described in Section 4 and also the UFLRA transition system implementing the miter construction in the SMV format, For enumeration of the minimal cut sets of the fault propagation graphs, we have used the tool SMT-PGFDS [7] (denoted as FPG in the experiments); for enumeration of the minimal cut sets of miter systems, we have used the tool xSAP [2], which internally uses an algorithm based on parametric IC3 [3] (denoted as ParamIC3).
- For the systems *without cycles*, we have generated both their FPG version and the description in the format of the tool OCRA [9] as used in [6]. Although the FPGs could be solved by the tool SMT-PGFDS and the OCRA systems can be solved by predicate abstraction, which is implemented in xSAP, and its BDD-based engine [6], this would not compare only the effect of the reduction to the Boolean case, but also a confounding factor of the underlying

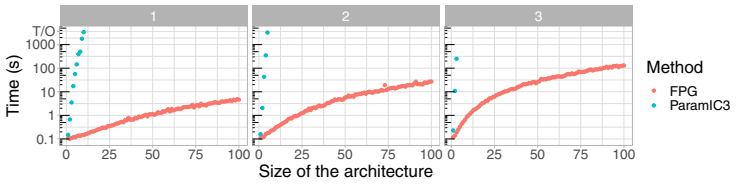


Fig. 4: Solving time on ladder-shaped benchmarks. Divided according to the number of voters per one reference module.

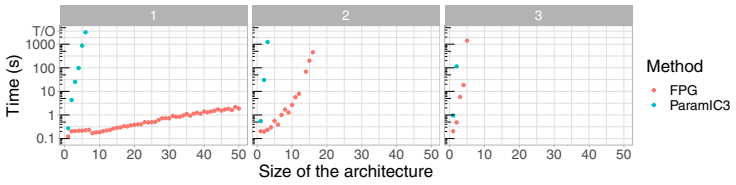


Fig. 5: Solving time on radiator-shaped benchmarks. Divided according to the number of voters per one reference module.

backend (SAT-based in SMT-PGFDS and BDD-based in xSAP). To answer RQ4, we have thus performed more fine-grained analysis as follows.
From each FPG, we generated the corresponding Boolean formula, which is possible since the graph is acyclic [7]. We also generated the Boolean formula obtained by predicate abstraction from each OCRA encoding. We thus obtained two Boolean formulas for each system: one by reduction to fault propagation (FP), and one by reduction by predicate abstraction (PA). We have then used the SAT-based enumeration algorithm of SMT-PGFDS and also BDD-based enumeration algorithm of xSAP on both of these Boolean formulas. This gives 4 combinations: FP-SAT, FP-BDD, PA-SAT, PA-BDD.

All experiments were executed on a cluster of 9 computational nodes, each with Intel Xeon CPU X5650 @ 2.67GHz, 12 CPU and 96 GiB of RAM. We have used time limit 1 hour of wall-clock time and memory limit 16 GB for each benchmark-solver pair. The detailed experimental results can be found at https://es-static.fbk.eu/people/mjonas/papers/tacas22_redarchs/.

6.2 Results for Cyclic Benchmarks

The comparison of running times of FPG-based and of model-checking-based approaches on the scalable cyclic benchmarks is shown in Figures 4 and 5. Figure 4 shows a significant benefit of the technique based on fault propagation on the ladder-shaped benchmarks; not only that it can enumerate cut sets of all the used benchmarks, but its run-times are dramatically better. However, as can be seen on Figure 5, the situation is different on the radiator-shaped benchmarks, which contain a large number of cycles. Although the performance of technique based

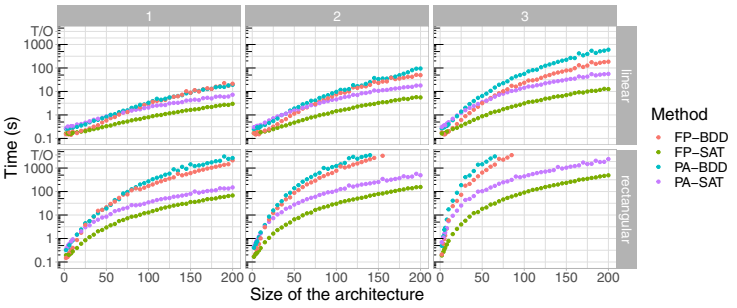


Fig. 7: Solving time on scalable acyclic benchmarks. Divided by the architecture and number of voters per one reference module.

on fault propagation is still superior to the model-checking-based technique, it scales poorly on the systems with 2 and 3 voters per one TMR block. The answer to RQ1 is thus that the proposed approach scales well if the number of cycles in the system is not too large; if the number of cycles is large, the technique scales worse, but nevertheless significantly better than the state-of-the-art technique based on miter construction and model checking [3].

The run-times on random cyclic benchmarks are shown in Figure 6. The figure shows that the performance of the proposed technique is better by several orders of magnitude and can enumerate minimal cut sets of 59 random systems that are out of reach for the technique based on model checking. Note that some of the systems are hard for both of the approaches: both approaches timed out on 66 of the 250 benchmarks. Together with the results for the ladder-shaped and radiator-shaped systems, this answers RQ2: the technique proposed in this paper has significantly better performance than the state-of-the-art technique based on model checking.

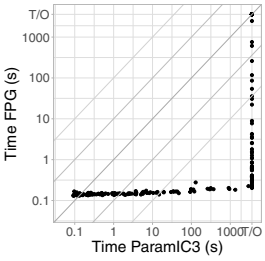


Fig. 6: Solving time on random cyclic benchmarks.

There are two reasons of the observed performance difference. First is the reduction of UFLRA transition system to the Boolean one, which has been also observed to bring significant benefit on acyclic systems in the case of predicate abstraction [6]. Second is the underlying MCS-enumeration technique applied the resulting FPG. This technique reduces the expensive sequential reasoning to an enumeration of minimal models of a single SMT formula, which can significantly improve performance [7].

6.3 Results for Acyclic Benchmarks

The comparison of the performance on acyclic scalable benchmarks is shown in Figure 7. The results are divided according to the method used to reduce the

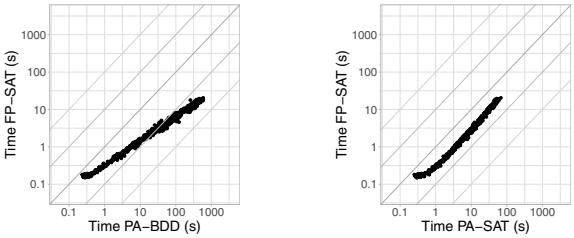


Fig. 8: Solving time on random acyclic benchmarks.

problem to Boolean case (FP vs. PA) and the technique used to enumerate the minimal cut sets of the Boolean system (SAT vs. BDD). Scatter plots of solving times on random acyclic benchmarks can be seen on Figure 8.

The results show that the reduction of the problem to fault propagation and using an off-the-shelf solver for enumeration of minimal cut sets of the resulting Boolean system (i.e., FP-SAT) is clearly superior to the state-of-the-art approach based on predicate abstraction and BDD-based MCS enumeration (i.e., PA-BDD). The difference between these two approaches is even several orders of magnitude on scalable benchmarks and grows with the size of the system and its complexity. The performance is also significantly better on the random benchmarks. This answers RQ3 in favor of the technique proposed in this paper.

As for RQ4, Figures 7 and 8 show that both the different reduction technique (FP vs. PA) and the solving technique (SAT vs. BDD) play a role in this difference. However, the larger part of the runtime difference between the proposed approach (FP-SAT) and the state-of-the-art approach (PA-BDD) [6] is due to better performance of SAT-based enumeration. This insight is additional interesting outcome of our our experiments. Nevertheless, for both of the enumeration approaches, the proposed reduction based on fault propagation provides better performance than the state-of-the-art reduction by predicate abstraction.

7 Conclusions and Future Work

We have presented a framework for modeling redundancy architectures with possible cyclic dependencies among the computational modules and we have developed an efficient approach for enumeration of minimal cut sets of such architectures. The experimental evaluation has shown that this approach dramatically outperforms the state-of-the-art approach based on model checking on cyclic redundancy architectures and has a better performance than the state-of-the-art approach based on predicate abstraction on acyclic architectures.

In the future, we plan to extend the approach to a more general class of voters than majority voters. We also plan to extend the approach to support common cause analysis for different component faults and possibly to synthesize an optimal distribution of the modules of the architecture between the computational nodes of a system such as Integrated Modular Avionics.

References

1. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-825
2. Bittner, B., Bozzano, M., Cavada, R., Cimatti, A., Gario, M., Griggio, A., Mattarei, C., Micheli, A., Zampedri, G.: The xSAP safety analysis platform. In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9636, pp. 533–539. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_31
3. Bozzano, M., Cimatti, A., Griggio, A., Mattarei, C.: Efficient anytime techniques for model-based safety analysis. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 603–621. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_41
4. Bozzano, M., Cimatti, A., Mattarei, C.: Automated analysis of reliability architectures. In: 2013 18th International Conference on Engineering of Complex Computer Systems, Singapore, July 17-19, 2013. pp. 198–207. IEEE Computer Society (2013). https://doi.org/10.1109/ICECCS.2013.37
5. Bozzano, M., Cimatti, A., Mattarei, C.: Efficient analysis of reliability architectures via predicate abstraction. In: Bertacco, V., Legay, A. (eds.) Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8244, pp. 279–294. Springer (2013). https://doi.org/10.1007/978-3-319-03077-7_19
6. Bozzano, M., Cimatti, A., Mattarei, C.: Formal reliability analysis of redundancy architectures. Formal Aspects Comput. **31**(1), 59–94 (2019). https://doi.org/10.1007/s00165-018-0475-1
7. Bozzano, M., Cimatti, A., Pires, A.F., Griggio, A., Jonáš, M., Kimberly, G.: Efficient SMT-Based Analysis of Failure Propagation. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 209–230. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_10
8. Bryant, R.E., German, S.M., Velev, M.N.: Exploiting positive equality in a logic of equality with uninterpreted functions. In: Halbwachs, N., Peled, D.A. (eds.) Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1633, pp. 470–482. Springer (1999). https://doi.org/10.1007/3-540-48683-6_40
9. Cimatti, A., Dorigatti, M., Tonetta, S.: OCRA: A tool for checking the refinement of temporal contracts. In: Denney, E., Bultan, T., Zeller, A. (eds.) 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. pp. 702–705. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693137

10. Ding, K., Morozov, A., Janschek, K.: Classification of hierarchical fault-tolerant design patterns. In: 15th IEEE Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2017, Orlando, FL, USA, November 6-10, 2017. pp. 612–619. IEEE Computer Society (2017). https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.108
11. Dubslaff, C., Ding, K., Morozov, A., Baier, C., Janschek, K.: Breaking the limits of redundancy systems analysis. CoRR **abs/1912.05364** (2019), http://arxiv.org/abs/1912.05364
12. Haeussermann, W.: Description and Performance of the Saturn Launch Vehicle's Navigation, Guidance, and Control System. IFAC Proceedings Volumes **3**(1), 275–312 (1970). https://doi.org/https://doi.org/10.1016/S1474-6670(17)68785-8, 3rd International IFAC Conference on Automatic Control in Space, Toulouse, France, March 2-6, 1970
13. Hamamatsu, M., Tsuchiya, T., Kikuno, T.: On the Reliability of Cascaded TMR Systems. In: Ishikawa, Y., Tang, D., Nakamura, H. (eds.) 16th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2010, Tokyo, Japan, December 13-15, 2010. pp. 184–190. IEEE Computer Society (2010). https://doi.org/10.1109/PRDC.2010.45
14. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 424–437. Springer (2006). https://doi.org/10.1007/11817963_39
15. Lee, S., Jung, J., Lee, I.: Voting structures for cascaded triple modular redundant modules. IEICE Electronic Express **4**(21), 657–664 (2007). https://doi.org/10.1587/elex.4.657
16. Prisaznuk, P.J.: Integrated modular avionics. In: Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@ m_NAECON 1992. pp. 39–45. IEEE (1992)
17. Ruijters, E., Stoelinga, M.: Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15**, 29–62 (2015). https://doi.org/10.1016/j.cosrev.2015.03.001
18. Wynn, E.: A comparison of encodings for cardinality constraints in a SAT solver. CoRR **abs/1810.12975** (2018), http://arxiv.org/abs/1810.12975
19. Yeh, Y.: Triple-triple redundant 777 primary flight computer. In: 1996 IEEE Aerospace Applications Conference. Proceedings. vol. 1, pp. 293–307 vol.1 (1996). https://doi.org/10.1109/AERO.1996.495891

Tools | Optimizations, Repair and Explainability


Adiar

Binary Decision Diagrams in External Memory

Steffan Christ Sølvsten (✉), Jaco van de Pol,
Anna Blume Jakobsen, and Mathias Weller Berg Thomasen

Aarhus University, Denmark {soelvsten,jaco}@cs.au.dk

Abstract. We follow up on the idea of Lars Arge to rephrase the Reduce and Apply operations of Binary Decision Diagrams (BDDs) as iterative I/O-efficient algorithms. We identify multiple avenues to simplify and improve the performance of his proposed algorithms. Furthermore, we extend the technique to other common BDD operations, many of which are not derivable using Apply operations alone. We provide asymptotic improvements to the few procedures that can be derived using Apply.
Our work has culminated in a BDD package named Adiar that is able to efficiently manipulate BDDs that outgrow main memory. This makes Adiar surpass the limits of conventional BDD packages that use recursive depth-first algorithms. It is able to do so while still achieving a satisfactory performance compared to other BDD packages: Adiar, in parts using the disk, is on instances larger than 9.5 GiB only 1.47 to 3.69 times slower compared to CUDD and Sylvan, exclusively using main memory. Yet, Adiar is able to obtain this performance at a fraction of the main memory needed by conventional BDD packages to function.

Keywords: Time-forward Processing · External Memory Algorithms · Binary Decision Diagrams

1 Introduction

A Binary Decision Diagram (BDD) provides a canonical and concise representation of a boolean function as an acyclic rooted graph. This turns manipulation of boolean functions into manipulation of graphs [10, 11].

Their ability to compress the representation of a boolean function has made them widely used within the field of verification. BDDs have especially found use in model checking, since they can efficiently represent both the set of states and the state-transition function [11]. Examples are the symbolic model checkers NuSMV [14, 15], MCK [17], LTSmin [19], and MCMAS [24] and the recently envisioned symbolic model checking algorithms for CTL* in [3] and for CTLK in [18]. Hence, continuous research effort is devoted to improve the performance of this data structure. For example, despite the fact that BDDs were initially envisioned back in 1986, BDD manipulation was first parallelised in 2014 by Velev and Gao [35] for the GPU and in 2016 by Van Dijk and Van de Pol [16] for multi-core processors [12].

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 295–313, 2022.
https://doi.org/10.1007/978-3-030-99527-0_16

The most widely used implementations of decision diagrams make use of recursive depth-first algorithms and a unique node table [16, 23, 34]. Lookup of nodes in this table and following pointers in the data structure during recursion both pause the entire computation while missing data is fetched [21,26]. For large enough instances, data has to reside on disk and the resulting I/O-operations that ensue become the bottle-neck. So in practice, the limit of the computer's main memory becomes the upper limit on the size of the BDDs.

Related Work. Prior work has been done to overcome the I/Os spent while computing on BDDs. David Long [25] achieved a performance increase of a factor of two by blocking all nodes in the unique node table based on their time of creation, i.e. with a depth-first blocking. But, in [6] this was shown to only improve the worst-case behaviour by a constant. Ochi, Yasuoka, and Yajima [28] designed in 1993 breadth-first BDD algorithms that exploit a levelwise locality on disk. Their technique was improved by Ashar and Cheong [8] in 1994 and by Sanghavi et al. [31] in 1996. The fruits of their labour was the BDD library CAL capable of manipulating BDDs larger than available main memory. Kunkle, Slavici and Cooperman [22] extended in 2010 the breadth-first approach to distributed BDD manipulation.

The breadth-first algorithms in [8, 28, 31] are not optimal in the I/O-model, since they still use a single hash table for each level. This works well in practice, as long as a single level of the BDD can fit into main memory. If not, they still exhibit the same worst-case I/O behaviour as other algorithms [6].

In 1995, Arge [5, 6] proposed optimal I/O algorithms for the basic BDD operations Apply and Reduce. To this end, he dropped all use of hash tables. Instead, he exploited a total and topological ordering of all nodes within the graph. This is used to store all recursion requests in priority queues, so they get synchronized with the iteration through the sorted input stream of nodes. Martin Šmérek implemented these algorithms in 2009 as they were described, but the performance was disappointing, since the intermediate unreduced BDD grew too large to handle in practice [personal communication, Sep 2021].

Contributions. Our work directly follows up on the theoretical contributions of Arge in [5, 6]. We simplified and improved on his I/O-optimal Apply and Reduce algorithms. In particular, we modified and pruned the intermediate representation, to prevent data duplication and to save on the number of sorting operations. We also provide I/O-efficient versions of several other standard BDD operations, where we obtain asymptotic improvements for the operations that are derivable from Apply.

Our proposed algorithms and data structures have been implemented to create a new easy-to-use and open-source BDD package, named Adiar. Our experimental evaluation shows that Adiar is able to manipulate BDDs larger than the given main memory available, with only an acceptable slowdown compared to a conventional BDD library running exclusively in main memory.

1.1 Overview

The rest of the paper is organised as follows. Section 2 covers preliminaries on the I/O-model and Binary Decision Diagrams. We present our algorithms for I/O-efficient BDD manipulation in Section 3. Section 4 provides an overview of the resulting BDD package, Adiar, and Section 5 contains an experimental evaluation of it. Our conclusions and future work are in Section 6.

2 Preliminaries

2.1 The I/O-Model

The I/O-model [1] allows one to reason about the number of data transfers between two levels of the memory hierarchy, while abstracting away from technical details of the hardware, to make a theoretical analysis manageable.

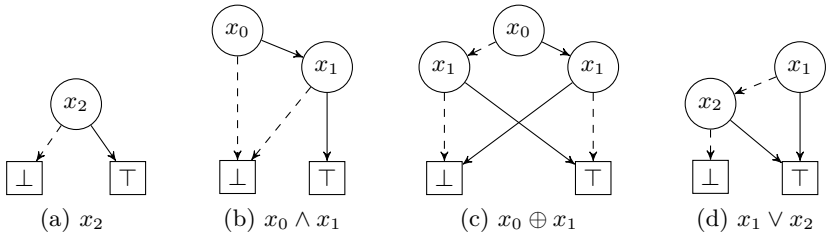
An I/O-algorithm takes inputs of size N, residing on the higher level of the two, i.e. in *external storage* (e.g. on a disk). The algorithm can only do computations on data that reside on the lower level, i.e. in *internal storage* (e.g. main memory). This internal storage can only hold a smaller and finite number of M elements. Data is transferred between these two levels in blocks of B consecutive elements [1]. Here, B is a constant size not only encapsulating the page size or the size of a cache-line but more generally how expensive it is to transfer information between the two levels. The cost of an algorithm is the number of data transfers, i.e. the number of *I/O-operations*, or just *I/Os*, it uses.

For all realistic values of N, M, and B we have that $N/B < \text{sort}(N) \ll N$, where $\text{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B)$ [1, 7] is the sorting lower bound, i.e. it takes $\Omega(\text{sort}(N))$ I/Os in the worst-case to sort a list of N elements [1]. With an M/B-way merge sort algorithm, one can obtain an optimal $O(\text{sort}(N))$ I/O sorting algorithm [1], and with the addition of buffers to lazily update a tree structure, one can obtain an I/O-efficient priority queue capable of inserting and extracting N elements in $O(\text{sort}(N))$ I/Os [4].

TPIE. The TPIE library [36] provides an implementation of I/O-efficient algorithms and data structures such that the use of B-sized buffers is completely transparent to the programmer. Elements can be stored in files that act like lists. One can **push** new elements to the end of a file and read the **next** elements from the file in either direction, provided **has_next** returns true. One can also **peek** the next element without moving the read head. TPIE provides an optimal $O(\text{sort}(N))$ external memory merge sort algorithm for its files. Furthermore, it provides an implementation of the I/O-efficient priority queue of [30] as developed in [29], which supports the **push**, **top** and **pop** operations.

2.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [10], as depicted in Fig. 1, is a rooted directed acyclic graph (DAG) that concisely represents a boolean function $\mathbb{B}^n \to \mathbb{B}$,



(a) x_2 (b) $x_0 \wedge x_1$ (c) $x_0 \oplus x_1$ (d) $x_1 \vee x_2$

Fig. 1: Examples of Reduced Ordered Binary Decision Diagrams. Leaves are drawn as boxes with the boolean value and internal nodes as circles with the decision variable. *Low* edges are drawn dashed while *high* edges are solid.

where $\mathbb{B} = \{\top, \bot\}$. The leaves contain the boolean values $\bot$ and $\top$ that define the output of the function. Each internal node contains the *label* i of the input variable x_i it represents, together with two outgoing arcs: a *low* arc for when $x_i = \bot$ and a *high* arc for when $x_i = \top$. We only consider Ordered Binary Decision Diagrams (OBDD), where each unique label may only occur once and the labels must occur in sorted order on all paths. The set of all nodes with label j is said to belong to the jth *level* in the DAG.

If one exhaustively (1) skips all nodes with identical children and (2) removes any duplicate nodes, then one obtains the *Reduced Ordered Binary Decision Diagram* (ROBDD) of the given OBDD. If the variable order is fixed, this reduced OBDD is a unique canonical form of the function it represents [10].

The two primary algorithms for BDD manipulation are called Apply and Reduce. The Apply computes the OBDD $h = f \odot g$ where f and g are OBDDs and $\odot$ is a function $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$. This is essentially done by recursively computing the product construction of the two BDDs f and g and applying $\odot$ when recursing to pairs of leaves. The Reduce applies the two reduction rules on an OBDD bottom-up to obtain the corresponding ROBDD [10].

Common implementations of BDDs use recursive depth-first procedures that traverse the BDD and the unique nodes are managed through a hash table [9, 16, 20, 23, 34]. The latter allows one to directly incorporate the Reduce algorithm of [10] within each node lookup [9, 27]. They also use a memoisation table to minimise the number of duplicate computations [16, 23, 34]. If the size N_f and N_g of two BDDs are considerably larger than the memory M available, each recursion request of the Apply algorithm will in the worst case result in an I/O, caused by looking up a node within the memoisation and following the low and high arcs [6, 21]. Since there are up to $N_f \cdot N_g$ recursion requests, this results in up to $O(N_f \cdot N_g)$ I/Os in the worst case. The Reduce operation transparently built into the unique node table with a *find-or-insert* function can also cause an I/O for each lookup within this table [21]. This adds yet another $O(N)$ I/Os, where N is the number of nodes in the unreduced BDD.

Lars Arge provided in [5,6] a description of an Apply algorithm that is capable of only using $O(\mathrm{sort}(N_f \cdot N_g))$ I/Os and a Reduce that uses $O(\mathrm{sort}(N))$ I/Os

(see [6] for a detailed description). He also proved this to be optimal for both algorithms, assuming a levelwise ordering of nodes on disk [6]. Our algorithms, implemented in Adiar, differ from Arge's in subtle non-trivial ways. We will not elaborate further on his original proposal, since our algorithms are simpler and better at conveying the *time-forward processing* technique he used. Instead, we will mention where our Reduce and Apply algorithms differ from his.

3 BDD Manipulation by Time-forward Processing

Our algorithms exploit the total and topological ordering of the internal nodes in the BDD depicted in (1) below, where parents precede their children. It is topological by ordering a node by its *label*, $i : \mathbb{N}$, and total by secondly ordering on a node's *identifier*, $id : \mathbb{N}$. This identifier only needs to be unique on each level as nodes are still uniquely identifiable by the combination of their label and identifier.

$$(i_1, id_1) < (i_2, id_2) \equiv i_1 < i_2 \vee (i_1 = i_2 \wedge id_1 < id_2) \tag{1}$$

We write the *unique identifier* $(i, id) : \mathbb{N} \times \mathbb{N}$ for a node as $x_{i,id}$.

BDD nodes do not contain an explicit pointer to their children but instead the children's unique identifier. Following the same notion, leaf values are stored directly in the leaf's parents. This makes a node a triple $(uid, low, high)$ where $uid : \mathbb{N} \times \mathbb{N}$ is its unique identifier and low and $high : (\mathbb{N} \times \mathbb{N}) + \mathbb{B}$ are its children. The ordering in (1) is lifted to compare the uids of two nodes, and so a BDD is represented by a file with BDD nodes in sorted order. For example, the BDDs in Fig. 1 would be represented as the lists depicted in Fig. 2.

The Apply algorithm in [6] produces an unreduced OBDD, which is turned into an ROBDD with Reduce. The original algorithms of Arge solely work on a node-based representation. Arge briefly notes that with an arc-based representation, the Apply algorithm is able to output its arcs in the order needed by the following Reduce, and vice versa. Here, an arc is a triple $(source, is\_high, target)$ (written as $source \xrightarrow{is\_high} target$) where $source : \mathbb{N} \times \mathbb{N}$, $is\_high : \mathbb{B}$, and $target : (\mathbb{N} \times \mathbb{N}) + \mathbb{B}$, i.e. $source$ and $target$ contain the level and identifier of internal nodes. We have further pursued this idea of an arc-based representation and can conclude that the algorithms indeed become simpler and more efficient with an arc-based output from Apply. On the other hand, we see no such benefit over the more compact node-based representation in the case of Reduce. Hence as is depicted in Fig. 3, our algorithms work in tandem by cycling between the node-based and arc-based representation.

1a: $[\ (x_{2,0}, \bot, \top)\ ]$
1b: $[\ (x_{0,0}, \bot, x_{1,0})\ , (x_{1,0}, \bot, \top)\ ]$
1c: $[\ (x_{0,0}, x_{1,0}, x_{1,1})\ , (x_{1,0}, \bot, \top)\ , (x_{1,1}, \top, \bot)\ ]$
1d: $[\ (x_{1,0}, x_{2,0}, \top)\ , (x_{2,0}, \bot, \top)\ ]$

Fig. 2: In-order representation of BDDs of Fig. 1

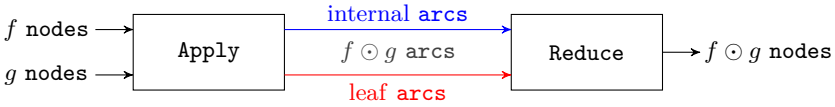
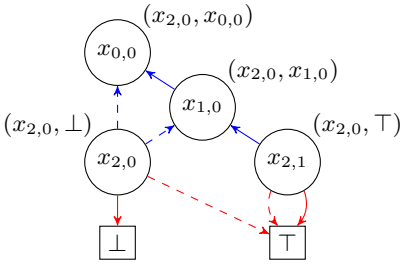


Fig. 3: The Apply–Reduce pipeline of our proposed algorithms



(a) Semi-transposed graph. (pairs indicate nodes in Fig. 1a and 1b, respectively)

internal arcs	leaf arcs
$[\ x_{0,0} \xrightarrow{\top} x_{1,0}$,	$[\ x_{2,0} \xrightarrow{\bot} \top$,
$x_{1,0} \xrightarrow{\bot} x_{2,0}$,	$x_{2,0} \xrightarrow{\top} \bot$,
$x_{0,0} \xrightarrow{\bot} x_{2,0}$,	$x_{2,1} \xrightarrow{\bot} \top$,
$x_{1,0} \xrightarrow{\top} x_{2,1}\ ]$	$x_{2,1} \xrightarrow{\top} \top\ ]$

(b) In-order arc-based representation.

Fig. 4: Unreduced output of Apply when computing $x_2 \Rightarrow (x_0 \wedge x_1)$

Notice that our Apply outputs two files containing arcs: arcs to internal nodes (blue) and arcs to leaves (red). Internal arcs are output at the time their targets are processed, and since nodes are processed in ascending order, internal arcs end up being sorted with respect to the unique identifier of their target. This groups all in-going arcs to the same node together and effectively reverses internal arcs. Arcs to leaves, on the other hand, are output when their source is processed, which groups all out-going arcs to leaves together. These two outputs of Apply represent a semi-transposed graph, which is exactly of the form needed by the following Reduce. For example, the Apply on the node-based ROBDDs in Fig. 1a and 1b with logical implication as the operator will yield the arc-based unreduced OBDD depicted in Fig. 4.

For simplicity, we will ignore any cases of leaf-only BDDs in our presentation of the algorithms. They are easily extended to also deal with those cases.

3.1 Apply

Our Apply algorithm works by a single top-down sweep through the input DAGs. Internal arcs are reversed due to this top-down nature, since an arc between two internal nodes can first be resolved and output at the time of the arc's target. These arcs are placed in the file $F_{internal}$. Arcs from nodes to leaves are placed in the file F_{leaf}.

The algorithm itself essentially works like the standard Apply algorithm. Given a recursion request for a pair of input nodes v_f from f and v_g from g, a single node is created with label $\min(v_f.uid.label, v_g.uid.label)$ and recursion requests r_{low} and r_{high} are created for its two children. If the label of $v_f.uid$ and

```
1  Apply(f , g , ⊙)
2    F_internal ← [ ] ; F_leaf ← [ ] ; Q_app:1 ← ∅ ; Q_app:2 ← ∅
3    v_f ← f . next ( ) ; v_g ← g . next ( ) ; id ← 0 ; label ← undefined
4
5    /* Insert request for root (v_f, v_g) */
6    Q_app:1 . push (NIL --undefined--> (v_f.uid, v_g.uid))
7
8    /* Process requests in topological order */
9    while Q_app:1 ≠ ∅ ∨ Q_app:2 ≠ ∅ do
10     (s --is_high--> (t_f, t_g) , low , high) ← TopOf(Q_app:1 , Q_app:2 )
11
12     t_seek ← if low , high = NIL then min(t_f , t_g) else max(t_f , t_g)
13     while v_f . uid < t_seek ∧ f . has_next ( ) do v_f ← f . next ( ) od
14     while v_g . uid < t_seek ∧ g . has_next ( ) do v_g ← g . next ( ) od
15
16     if low = NIL ∧ high = NIL ∧ t_f ∉ {⊥, ⊤} ∧ t_g ∉ {⊥, ⊤}
17                  ∧ t_f . label = t_g . label ∧ t_f . id ≠ t_g . id
18     then /* Forward information of min(t_f, t_g) to max(t_f, t_g) */
19       v ← if t_seek = v_f then v_f else v_g
20       while Q_app:1 . top ( ) matches _ → (t_f, t_g) do
21         (s --is_high--> (t_f, t_g)) ← Q_app:1 . pop ( )
22         Q_app:2 . push (s --is_high--> (t_f, t_g) , v . low , v . high)
23       od
24     else /* Process request (t_f, t_g) */
25       id ← if label ≠ t_seek . label then 0 else id+1
26       label ← t_seek . label
27
28       /* Forward or output out-going arcs */
29       r_low , r_high ← RequestsFor((t_f , t_g) , v_f , v_g , low , high , ⊙)
30       (if r_low ∈ {⊥, ⊤} then F_leaf else Q_app:1 ) . push (x_label,id --⊥--> r_low)
31       (if r_high ∈ {⊥, ⊤} then F_leaf else Q_app:1 ) . push (x_label,id --⊤--> r_high)
32
33       /* Output in-going arcs */
34       while Q_app:1 ≠ ∅ ∧ Q_app:1 . top ( ) matches (_ → (t_f, t_g)) do
35         (s --is_high--> (t_f, t_g)) ← Q_app:1 . pop ( )
36         if s ≠ NIL then F_internal . push (s --is_high--> x_label,id)
37       od
38       while Q_app:1 ≠ ∅ ∧ Q_app:2 . top ( ) matches (_ → (t_f, t_g) , _ , _) do
39         (s --is_high--> (t_f, t_g) , _ , _) ← Q_app:2 . pop ( )
40         if s ≠ NIL then F_internal . push (s --is_high--> x_label,id)
41       od
42   od
43   return F_internal , F_leaf
```

Fig. 5: The Apply algorithm

$v_g.uid$ are equal, then $r_{low} = (v_f.low, v_g.low)$ and $r_{high} = (v_f.high, v_g.high)$. Otherwise, r_{low}, resp. r_{high}, contains the *uid* of the low child, resp. the high child, of $\min(v_f, v_g)$, whereas $\max(v_f.uid, v_g.uid)$ is kept as is.

The pseudocode for the Apply procedure is shown in Fig. 5, where the **RequestsFor** function computes r_{low} and r_{high} for the pair of nodes (t_f, t_g). The goal of the rest of the algorithm is to obtain the information that **RequestsFor** needs in an I/O-efficient way. To this end, the two priority queues $Q_{app:1}$ and $Q_{app:2}$ are used to synchronise recursion requests for a pair of nodes (t_f, t_g) with the sequential order of reading nodes in f and g. $Q_{app:1}$ has elements of the form $(s \xrightarrow{is\_high} (t_f, t_g))$ and $Q_{app:2}$ has elements $(s \xrightarrow{is\_high} (t_f, t_g), low, high)$. The boolean *is_high* and the unique identifer s, being the request's origin, are used on lines 33 – 41, to output all ingoing arcs when the request is resolved.

Elements in $Q_{app:1}$ are sorted in ascending order by $\min(t_f, t_g)$, i.e. the node encountered first from f and g. Requests to the same (t_f, t_g) are grouped together by secondarily sorting the tuple lexicographically. $Q_{app:2}$ is sorted in ascending order by $\max(t_f, t_g)$, i.e. the second of the two to be visited, and ties are again broken lexicographically. This second priority queue is used in the case where $t_f.label = t_g.label$ but $t_f.id \neq t_g.id$, i.e. when both are needed to resolve the request but they are not necessarily available at the same time. To this end, the given request is moved from $Q_{app:1}$ into $Q_{app:2}$ on lines 19 – 23. Here, the request is extended with the unique identifiers *low* and *high* of $\min(v_f, v_g)$, which makes the children of $\min(v_f, v_g)$ available at $\max(v_f, v_g)$.

The next request to process from $Q_{app:1}$ or $Q_{app:2}$ is dictated by the **TopOf** function on line 10. In the case that both $Q_{app:1}$ and $Q_{app:2}$ are non-empty, let $r_1 = (s_1 \xrightarrow{is\_high_1} (t_{f:1}, t_{g:1}))$ be the top element of $Q_{app:1}$ and let the top element of $Q_{app:2}$ be $r_2 = (s_2 \xrightarrow{is\_high_2} (t_{f:2}, t_{g:2}), low, high)$. **TopOf**$(Q_{app:1}, Q_{app:2})$ returns $(r_1, \texttt{Nil}, \texttt{Nil})$ if $\min(t_{f:1}, t_{g:1}) < \max(t_{f:2}, t_{g:2})$ and r_2 otherwise. If either one is empty, then it equivalently outputs the top request of the other.

The arc-based output greatly simplifies the algorithm compared to the original proposal of Arge in [6]. Our algorithm only uses two priority queues rather than four. Arge's algorithm, like ours, resolves a node before its children, but due to the node-based output it has to output this entire node before its children. Hence, it has to identify all nodes by the tuple (t_f, t_g), doubling the space used. Instead, the arc-based output allows us to output the information at the time of the children and hence we are able to generate the label and its new identifier for both parent and child. Arge's algorithm also did not forwarded a request's source s, so repeated requests to the same pair of nodes were merely discarded upon retrieval from the priority queue, since they carried no relevant information. Our arc-based output, on the other hand, makes every element placed in the priority queue forward the source s, vital for the creation of the semi-transposed graph.

Proposition 1 (Following Arge 1996 [6]). *The Apply algorithm in Fig. 5 has I/O complexity $O(sort(N_f \cdot N_g))$ and $O((N_f \cdot N_g) \cdot \log(N_f \cdot N_g))$ time complexity, where N_f and N_g are the respective sizes of the BDDs for f and g.*

See the full paper [33] for the proof.

Pruning by shortcutting the operator The Apply procedure above, like Arge's original algorithm, follows recursion requests until a pair of leaves is met. Yet, for example in Fig. 4 the node for the request $(x_{2,0}, \top)$ is unnecessary to resolve, since all leaves of this subgraph trivially will be $\top$ due to the implication operator. The subsequent Reduce will remove this node and its children in favour of the $\top$ leaf. Hence, the **RequestsFor** function can instead immediately create a request for the leaf. We implemented this in Adiar, since it considerably decreases the size of $Q_{app:1}$, $Q_{app:2}$, and of the output.

3.2 Reduce

Our Reduce algorithm in Fig. 6 works like other explicit variants with a single bottom-up sweep through the OBDD. Since the nodes are resolved and output in a bottom-up descending order, the output is exactly in the reverse order as it is needed for any following Apply. We have so far ignored this detail, but the only change necessary to the Apply algorithm in Section 3.1 is for it to read the list of nodes of f and g in reverse.

The priority queue Q_{red} is used to forward the reduction result of a node v to its parents in an I/O-efficient way. Q_{red} contains arcs from unresolved sources s in the given unreduced OBDD to already resolved targets t' in the ROBDD under construction. The bottom-up traversal corresponds to resolving all nodes in descending order. Hence, arcs $s \xrightarrow{is\_high} t'$ in Q_{red} are first sorted on s and secondly on *is_high*; the latter simplifies retrieving the low and high arcs on lines 8 and 9. The base-cases for the Reduce algorithm are the arcs to leaves in F_{leaf}, which follow the exact same ordering. Hence, on lines 8 and 9, arcs in Q_{red} and F_{leaf} are merged using the **PopMax** function that retrieves the arc that is maximal with respect to this ordering.

Since nodes are resolved in descending order, $F_{internal}$ follows this ordering on the arc's target when elements are read in reverse. The reversal of arcs in $F_{internal}$ makes the parents of a node v, to which the reduction result is to be forwarded, readily available on lines 26 – 32.

The algorithm otherwise proceeds similarly to the standard Reduce algorithm [10]. For each level j, all nodes v of that level are created from their high and low arcs, e_{high} and e_{low}, taken out of Q_{red} and F_{leaf}. The nodes are split into the two temporary files $F_{j:1}$ and $F_{j:2}$ that contain the mapping $[uid \mapsto uid']$ from a node in the given unreduced OBDD to its equivalent node in the output. $F_{j:1}$ contains the nodes v removed due to the first reduction rule and is populated on lines 7 – 12: if both children of v are the same then $[v.uid \mapsto v.low]$ is pushed to this file. $F_{j:2}$ contains the mappings for the second rule and is populated on lines 15 – 24. Nodes not placed in $F_{j:1}$ are placed in an intermediate file F_j and sorted by their children. This makes duplicate nodes immediate successors. Every unique node encountered in F_j is output to F_{out} before mapping itself and all its duplicates to it in $F_{j:2}$. Since nodes are output out-of-order compared to the input and it is unknown how many will be output for said level, they are given new decreasing identifiers starting from the maximal possible value `MAX_ID`. Finally, $F_{j:2}$ is sorted back in order of $F_{internal}$ to forward the results

```
1  Reduce(F_internal , F_leaf)
2    F_out ← [] ; Q_red ← ∅
3    while Q_red ≠ ∅ do
4      j ← Q_red . top() . source . label ; id ← MAX_ID;
5      F_j ← [] ; F_j:1 ← [] ; F_j:2 ← []
6
7      while Q_red . top() . source . label = j do
8        e_high ← PopMax(Q_red , F_leaf)
9        e_low ← PopMax(Q_red , F_leaf)
10       if e_high . target = e_low . target
11       then F_j:1 . push([e_low . source ↦ e_low . target])
12       else F_j . push((e_low . source , e_low . target , e_high . target))
13     od
14
15     sort v ∈ F_j by v . low and secondly by v . high
16     v' ← undefined
17     for each v ∈ F_j do
18       if v' is undefined or v . low ≠ v' . low or v . high ≠ v' . high
19       then
20         id ← id − 1
21         v' ← (x_j,id , v . low , v . high)
22         F_out . push(v)
23       F_j:2 . push([v . uid ↦ v' . uid])
24     od
25
26     sort [uid ↦ uid'] ∈ F_j:2 by uid in descending order
27     for each [uid ↦ uid'] ∈ MergeMaxUid(F_j:1 , F_j:2) do
28       while arcs from F_internal . peek() matches _ -→ uid do
29         (s -is_high→ uid) ← F_internal . next()
30         Q_red . push(s -is_high→ uid')
31       od
32     od
33   od
34   return F_out
```

Fig. 6: The Reduce algorithm

in both $F_{j:1}$ and $F_{j:2}$ to their parents on lines 26 – 32. Here, **MergeMaxUid** merges the mappings $[uid \mapsto uid']$ in $F_{j:1}$ and $F_{j:2}$ by always taking the mapping with the largest uid from either file.

Since the original algorithm of Arge in [6] takes a node-based OBDD as an input and internally uses node-based auxiliary data structures, his Reduce algorithm had to create two copies of the input to reverse all internal arcs: a copy sorted by the nodes' low child and one sorted by their high children. Since $F_{internal}$ already has its arcs reversed, our design eliminates two expensive sorting steps and more than halves the memory used.

Another consequence of Arge's node-based representation is that his algorithm had to move all arcs to leaves into Q_{red} rather than merging requests from Q_{red} with the base-cases from F_{leaf}. The semi-transposed input allows us to decrease the number of I/Os due to Q_{red} by $\Theta(\text{sort}(N_\ell))$ where N_ℓ are the number of arcs to leaves (see [33] for the proof). In practice, together with pruning the recursion during Apply, this can provide up to a factor 2 speedup [33].

Proposition 2 (Following Arge 1996 [6]). *The Reduce algorithm in Fig. 6 has an $O(sort(N))$ I/O complexity and an $O(N \log N)$ time complexity.*

See the full paper [33] for the proof. Arge proved in [6] that this $O(\text{sort}(N))$ I/O complexity is optimal for the input, assuming a levelwise ordering of nodes.

3.3 Other BDD Algorithms

By applying the above algorithmic techniques, one can obtain all other singly-recursive BDD algorithms; see [33] for the details. We now design asymptotically better variants of Negation and Equality Checking than what is possible by deriving them using Apply.

Negation A BDD is negated by inverting the value in its nodes' leaf children. This is an $O(1)$ I/O-operation if a *negation flag* is used to mark whether the nodes should be negated on-the-fly as they are read from the stream.

Proposition 3. *Negation has I/O, space, and time complexity $O(1)$.*

This is an improvement over the $O(\text{sort}(N))$ I/Os spent by Apply to compute $f \oplus \top$, where $\oplus$ is exclusive or. Furthermore, disk space is shared between BDDs.

Equality Checking To check for $f \equiv g$ one has to check the DAG of f being isomorphic to the one for g [10]. This makes f and g trivially inequivalent when the number of nodes, number of levels, or the label or size of each of the L levels do not match. This can be checked in $O(1)$ and $O(L/B)$ I/Os if the Reduce algorithm in Fig. 6 is made to also output the relevant meta-information.

If $f \equiv g$, the isomorphism relates the roots of the BDDs for f and g. For any node v_f of f and v_g of g, if (v_f, v_g) is uniquely related by the isomorphism, then so should $(v_f.low, v_g.low)$ and $(v_f.high, v_g.high)$. Hence, one can check for equality by traversing the product of both BDDs (as in Apply) and check for one of the following two conditions being violated.

- The children of the given recursion request (t_f, t_g) should either both be the same leaf or an internal node with the same label.
- On level i, exactly N_i unique recursion requests should be retrieved from the priority queues, where N_i are the number of nodes on level i.

If the first condition is never violated, it is guaranteed that $f \equiv g$, and so $\top$ is output. The second ensures that the algorithm terminates earlier on negative cases and lowers the provable complexity bound; see [33] for the proof.

Proposition 4. *Equality Checking has I/O complexity $O(sort(N))$ and time complexity $O(N \log N)$, where $N = \min(N_f, N_g)$ is the minimum of the respective sizes of the BDDs for f and g.*

If (1) on page 5 is extended such that $\bot, \top$ succeed all unique identifiers and $\bot < \top$, then Fig. 6 actually enforces a much stricter ordering; it outputs nodes in an order purely based on their label and the unique identifier of their children.

Proposition 5. *If G_f and G_g are outputs of Reduce in Fig. 6, then $f \equiv g$ if and only if the ith nodes of G_f and G_g match numerically.*

See the full paper [33] for the proof. The negation operation breaks this property by changing the leaf values without changing their order. So, in the case where f or g, but not both, have their negation flag set, one still has to use the $O(\text{sort}(N))$ algorithm above, but otherwise a simple linear scan of both BDDs suffices.

Corollary 1. *If the negation flag of the BDDs for f and g are equal, then Equality Checking can be done in $2 \cdot N/B$ I/Os and $O(N)$ time, where $N = \min(N_f, N_g)$ is the minimum of the respective sizes of the BDDs for f and g.*

Both Proposition 4 and Corollary 1 are an asymptotic improvement on the $O(\text{sort}(N^2))$ equality checking algorithm by computing $f \leftrightarrow g$ with Apply and Reduce and then test whether the output is the $\top$ leaf.

4 Adiar: An Implementation

The algorithms and data structures described in Section 3 have been implemented in a new BDD package, named Adiar[1, 2]. The most important operations are shown in Table 1. Interaction with the BDD package is done through C++ programs that include the `<adiar/adiar.h>` header file and are built and linked with CMake. Its two dependencies are the Boost library and the TPIE library; the latter is included as a submodule of the Adiar repository, leaving it to CMake to build TPIE and link it to Adiar.

Adiar is initialised with the `adiar_init(memory, temp_dir)` function, where `memory` is the memory (in bytes) dedicated to Adiar and `temp_dir` is the directory where temporary files will be placed, e.g. a dedicated harddisk. The BDD package is deinitialised by calling the `adiar_deinit()` function.

The `bdd` object in Adiar is a container for the underlying files for each BDD, while a `__bdd` object is used for possibly unreduced arc-based OBDDs. Reference counting on the underlying files is used to reuse the same files and to immediately delete them when the reference count decrements to 0. Files are deleted as early as possible by use of implicit conversions between the `bdd` and `__bdd` objects and an overloaded assignment operator, making the concurrently occupied space on disk minimal.

[1] **adiar** ⟨ portuguese ⟩ (*verb*) : to defer, to postpone

[2] Source code is publicly available at github.com/ssoelvsten/adiar

Adiar function	Operation	I/O complexity	Justification
`bdd_apply(`f`,`g`,`$\odot$`)`	$f \odot g$	$O(\text{sort}(N_f N_g))$	Prop. 1, 2
`bdd_ite(`f`,`g`,`h`)`	$f \ ? \ g \ : \ h$	$O(\text{sort}(N_f N_g N_h))$	[33], Prop. 2
`bdd_restrict(`f`,`i`,`v`)`	$f\|_{x_i=v}$	$O(\text{sort}(N_f))$	[33], Prop. 2
`bdd_exists(`f`,`i`)`	$\exists v : f\|_{x_i=v}$	$O(\text{sort}(N_f^2))$	[33], Prop. 2
`bdd_forall(`f`,`i`)`	$\forall v : f\|_{x_i=v}$	$O(\text{sort}(N_f^2))$	[33], Prop. 2
`bdd_not(`f`)`	$\neg f$	$O(1)$	Prop. 3
`bdd_satcount(`f`)`	$\#x : f(x)$	$O(\text{sort}(N_f))$	[33]
`bdd_nodecount(`f`)`	N_f	$O(1)$	Section 3.3
f `==` g	$f \equiv g$	$O(\text{sort}(\min(N_f, N_g)))$	Prop. 4

Table 1: Some of the operations supported by Adiar and their I/O-complexity.

5 Experimental Evaluation

While time-forwarding may be an asymptotic improvement over the recursive approach in the I/O-model, its usability in practice is another question entirely. We have compared *Adiar* 1.0.1 to the recursive BDD packages *CUDD* 3.0.0 [34] and *Sylvan* 1.5.0 [16] (in single-core mode). We constructed BDDs for some benchmarks in all tools in a similar manner, ensuring the same variable ordering.

The experimental results[3] were obtained on server nodes of the *Grendel* cluster at the Centre for Scientific Computing Aarhus. Each node has two 48-core 3.0 GHz Intel Xeon Gold 6248R processors, 384 GiB of RAM, 3.5 TiB of available SSD disk, run CentOS Linux, and compile code with GCC 10.1.0. We report the *minimum* measured running time, since it minimises any error caused by the CPU, memory and disk [13]; using the average or median does not significantly change any of our results. For comparability all compute nodes are set to use 350 GiB of the available RAM, while each BDD package is given 300 GiB of it. Sylvan was set to not use any parallelisation, given a ratio between the node table and the cache of 64:1 and set to start its data structures 2^{12} times smaller than the final 262 GiB it may occupy, i.e. at first with a table and cache that occupies 66 MiB. The size of the CUDD cache was set such it would have the same node table to cache ratio when reaching 300 GiB.

5.1 Queens

The solution to the Queens problem is the number of arrangements of N queens on an $N \times N$ board, such that no queen is threatened by another. Our benchmark follows the description in [22]: the variable x_{ij} represents whether a queen is placed on the ith row and the jth column and the solution to the problem then corresponds to the number of satisfying assignments to the formula $\bigwedge_{i=0}^{N-1} \bigvee_{j=0}^{N-1} (x_{ij} \wedge \neg \mathit{has\_threat}(i,j))$, where $\mathit{has\_threat}(i,j)$ is true, if a queen is placed on a tile (k,l), that would be in conflict with a queen placed on (i,j).

[3] Available at Zenodo [32] and at github.com/ssoelvsten/bdd-benchmark

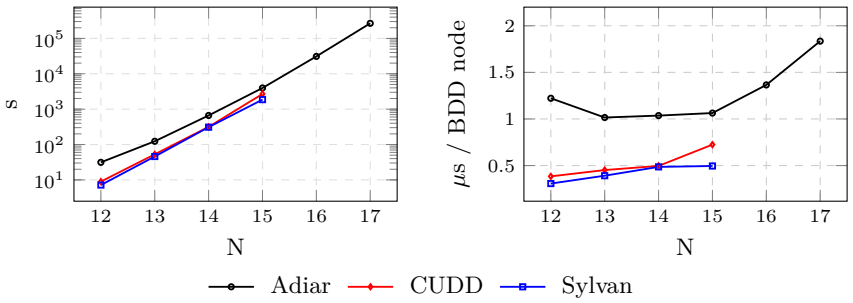


Fig. 7: Running time solving N-Queens (lower is better).

The ROBDD of the innermost conjunction can be directly constructed, without any BDD operations.

The current version of Adiar is implemented purely using external memory algorithms. These perform poorly when given small amounts of data. Hence, it is not meaningful to compare performance for $N < 12$ where the BDDs involved are 23.5 MiB or smaller. For $N \geq 12$, Fig. 7 shows how the gap in running time between Adiar and other BDD packages shrinks as instances grow. At $N = 15$, which is the largest instance solved by Sylvan and CUDD, Adiar is 1.47 times slower than CUDD and 2.15 times slower than Sylvan.

The largest instance solved by Adiar is $N = 17$ where the largest BDD constructed is 719 GiB in size. In contrast, Sylvan only constructed a 12.9 GiB sized BDD for $N = 15$. Even though Adiar has to use disk, it only becomes 1.8 times slower per processed node compared to its highest performance at $N = 13$. Conversely, Adiar is able to solve the $N = 15$ problem with much less main memory than both Sylvan and CUDD. Fig. 8 shows the running time on the same machine with its memory, including its file system cache, limited with *cgroups* to be 1 GiB more than given to the BDD package. Yet, Adiar is only 1.39 times slower when decreasing its memory down to 2 GiB, while Sylvan cannot function with less than 56 GiB of memory available.

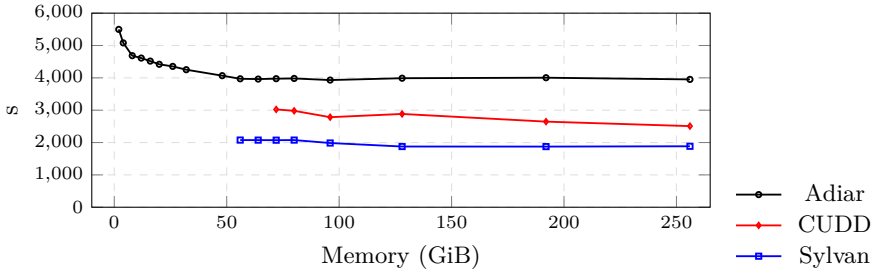


Fig. 8: Running time of 15-Queens with variable memory (lower is better).

We also ran experiments on counting the number of draw positions in a 3D-version of Tic-Tac-Toe, derived from [22]. Our results [33] paint a similar picture: Adiar is only 2.50 times slower than Sylvan for Sylvan's largest solved instance; Sylvan only creates BDDs of up to 34.4 GiB in size, whereas Adiar constructs a 902 GiB sized BDD; Adiar only slows down by a factor of 2.49 per processed node when using the disk extensively to solve the larger instances.

5.2 Combinatorial Circuit Verification

The *EPFL* Combinational Benchmark Suite [2] consists of 23 combinatorial circuits designed for logic optimisation and synthesis. 20 of these are split into the two categories *random/control* and *arithmetic*, and each of these original circuits C_o is distributed together with one circuit optimised for size C_s and one circuit optimised for depth C_d. The last three are the *More than a Million Gates* benchmarks, which we will ignore as they come without optimised versions.

Based on the approach of the *Nanotrav* program as distributed with CUDD, we verify the functional equivalence between each output gate of C_o and the corresponding gate in each optimised circuits C_d, and C_s. The BDDs are computed by representing every input gate by a decision variable, and computing the BDD of all other gates from the BDDs of their input wires. Finally, the BDDs for every pair of corresponding output gates are tested for equality. Memoisation ensures that the same gate is not computed twice, while a reference counter is maintained for each gate such that dead BDDs in the memoisation table may be garbage collected. Recall that Adiar stores each BDD in a separate file, while Sylvan and CUDD share nodes between different BDDs in a forest.

Table 2 shows the number of verified instances with each BDD package within a 15 days time limit. Adiar is able to verify three more benchmarks than both other BDD packages. This is despite the fact that most instances include hundreds of concurrent BDDs, while the disk is only 12 times larger than main memory. For example, the largest verified benchmark, *mem_ctrl*, has up to 1231 BDDs existing at the same time.

Table 3 shows the time it took Adiar to verify equality between the original and each of the optimised circuits, for the three largest cases verified. The table also shows the sum of the sizes of the output BDDs that represent each circuit. Throughout all solved benchmarks, equality checking took less than 1.47% of the total construction time and the $O(N/B)$ algorithm could be used in 71.6% of all BDD comparisons. The *voter* benchmark with its single output shows that

	# solved	# out-of-space	# time-out
Adiar	23	6	11
CUDD	20	19	1
Sylvan	20	13	7

Table 2: Number of verified *arithmetic* and *random/control* circuits from [2]

	depth	size
Time (s)	5862	5868
$O(\text{sort}(N))$	496	476
$O(N/B)$	735	755
N (MiB)	614313	

(a) *mem_ctrl*

	depth	size
Time (s)	3.89	3.27
$O(\text{sort}(N))$	22	22
$O(N/B)$	3	3
N (MiB)	3589	

(b) *sin*

	depth	size
Time (s)	0.058	0.006
$O(\text{sort}(N))$	1	0
$O(N/B)$	0	1
N (MiB)	5.74	

(c) *voter*

Table 3: Running time for equivalence testing. $O(\text{sort}(N))$ and $O(N/B)$ is the number of times the respective algorithm in Section 3.3 was used.

the $O(N/B)$ algorithm is about 10 times faster than the $O(\text{sort}(N))$ algorithm and can compare at least $2 \cdot 5.75$ MiB/0.006 s = 1.86 GiB/s.

6 Conclusions and Future Work

Adiar provides an I/O-efficient implementation of BDDs. The iterative BDD algorithms exploit a topological ordering of the BDD nodes in external memory, by use of priority queues and sorting algorithms. All recursion requests for a single node are processed together, eliminating the need for a memoisation table.

The performance of Adiar is very promising in practice for instances larger than a few hundred MiB. As the size of the BDDs increase, the performance of Adiar gets closer to conventional recursive BDD implementations – for BDDs larger than a few GiB the use of Adiar has at most resulted in a 3.69 factor slowdown. Simultaneously, the design of our algorithms allow us to compute on BDDs that outgrow main memory with only a 2.49 factor slowdown, which is negligible compared to use of swap memory with conventional BDD packages.

This performance comes at the cost of Adiar not being able to share nodes between BDDs. Yet, this increase in space usage is not a problem in practice and it makes garbage collection a trivial and cheap deletion of files on disk. On the other hand, the lack of sharing makes it impossible to check for functional equivalence with a mere pointer comparison. Instead, one has to explicitly check for the two DAGS being isomorphic. We have improved the asymptotic and practical performance of equality checking such that it is negligible in practice.

This lays the foundation on which we intend to develop external memory versions of the BDD algorithms that are still missing for symbolic model checking. Specifically, we intend to improve the performance of quantifying multiple variables and designing a relational product operation. Furthermore, we will improve performance for small instances that fit entirely into internal memory.

Acknowledgements

Thanks to the late Lars Arge, to Gerth S. Brodal, and to Mathias Rav for their inputs. Furthermore, thanks to the Centre for Scientific Computing Aarhus (phys.au.dk/forskning/cscaa/) for running our experiments on their cluster.

References

1. Aggarwal, A., Vitter, Jeffrey, S.: The input/output complexity of sorting and related problems. Communications of the ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
2. Amarú, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. In: 24th International Workshop on Logic & Synthesis (2015)
3. Amparore, E., Donatelli, S., Gallà, F.: A CTL* model checker for Petri nets. In: Application and Theory of Petri Nets and Concurrency. Lecture Notes in Computer Science, vol. 12152, pp. 403–413. Springer (2020). https://doi.org/10.1007/978-3-030-51831-8_21
4. Arge, L.: The buffer tree: A new technique for optimal I/O-algorithms. In: Workshop on Algorithms and Data Structures (WADS). Lecture Notes in Computer Science, vol. 955, pp. 334–345. Springer, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-60220-8_74
5. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: 6th International Symposium on Algorithms and Computations (ISAAC). Lecture Notes in Computer Science, vol. 1004, pp. 82–91 (1995). https://doi.org/10.1007/BFb0015411
6. Arge, L.: The I/O-complexity of ordered binary-decision diagram. In: BRICS RS preprint series. vol. 29. Department of Computer Science, University of Aarhus (1996). https://doi.org/10.7146/brics.v3i29.20010
7. Arge, L.: External geometric data structures. In: 10th International Computing and Combinatorics Conference (COCOON). Lecture Notes in Computer Science, vol. 3106 (2004). https://doi.org/10.1007/978-3-540-27798-9_1
8. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 622–627. IEEE Computer Society Press (1994). https://doi.org/10.1109/ICCAD.1994.629886
9. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: 27th Design Automation Conference (DAC). pp. 40–45. Association for Computing Machinery (1990). https://doi.org/10.1109/DAC.1990.114826
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
11. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
12. Bryant, R.E.: Binary decision diagrams. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 191–217. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
13. Chen, J., Revels, J.: Robust benchmarking in noisy environments. arXiv (2016), https://arxiv.org/abs/1608.04295
14. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
15. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer **2**, 410–425 (2000). https://doi.org/10.1007/s100090050046

16. Van Dijk, T., Van de Pol, J.: Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer **19**, 675–696 (2016). https://doi.org/10.1007/s10009-016-0433-2
17. Gammie, P., Van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 3114, pp. 479–483. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_41
18. He, L., Liu, G.: Petri net based symbolic model checking for computation tree logic of knowledge. arXiv (2020), https://arxiv.org/abs/2012.10126
19. Kant, G., Laarman, A., Meijer, J., Van de Pol, J., Blom, S., Van Dijk, T.: LTSmin: High-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
20. Karplus, K.: Representing Boolean functions with if-then-else DAGs. Tech. rep., University of California at Santa Cruz, USA (1988)
21. Klarlund, N., Rauhe, T.: BDD algorithms and cache misses. In: BRICS Report Series. vol. 26 (1996). https://doi.org/10.7146/brics.v3i26.20007
22. Kunkle, D., Slavici, V., Cooperman, G.: Parallel disk-based computation for large, monolithic binary decision diagrams. In: 4th International Workshop on Parallel Symbolic Computation (PASCO). pp. 63–72 (2010). https://doi.org/10.1145/1837210.1837222
23. Lind-Nielsen, J.: BuDDy: A binary decision diagram package. Tech. rep., Department of Information Technology, Technical University of Denmark (1999)
24. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. International Journal on Software Tools for Technology Transfer **19**, 9–30 (2017). https://doi.org/10.1007/s10009-015-0378-x
25. Long, D.E.: The design of a cache-friendly BDD library. In: Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 639–645. Association for Computing Machinery (1998)
26. Minato, S.i., Ishihara, S.: Streaming BDD manipulation for large-scale combinatorial problems. In: Design, Automation and Test in Europe Conference and Exhibition. pp. 702–707 (2001). https://doi.org/10.1109/DATE.2001.915104
27. Minato, S.i., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In: 27th Design Automation Conference (DAC). pp. 52–57. Association for Computing Machinery (1990). https://doi.org/10.1145/123186.123225
28. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: International Conference on Computer Aided Design (ICCAD). pp. 48–55. IEEE Computer Society Press (1993). https://doi.org/10.1109/ICCAD.1993.580030
29. Petersen, L.H.: External Priority Queues in Practice. Master's thesis, Department of Computer Science, University of Aarhus (2007)
30. Sanders, P.: Fast priority queues for cached memory. ACM Journal of Experimental Algorithmics **5**, 7–32 (2000). https://doi.org/10.1145/351827.384249
31. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package by exploiting memory hierarchy. In: 33rd Design Automation Conference (DAC). pp. 635–640. Association for Computing Machinery (1996). https://doi.org/10.1145/240518.240638
32. Sølvsten, S.C., Van de Pol, J.: Adiar v1.0.1 : TACAS 2022 artifact. Zenodo (2021). https://doi.org/10.5281/zenodo.5638335

33. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Efficient binary decision diagram manipulation in external memory. arXiv (2021), https://arxiv.org/abs/2104.12101
34. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Tech. rep., University of Colorado at Boulder (2015)
35. Velev, M.N., Gao, P.: Efficient parallel GPU algorithms for BDD manipulation. In: 19th Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 750–755. IEEE Computer Society Press (2014). https://doi.org/10.1109/ASPDAC.2014.6742980
36. Vengroff, D.E.: A Transparent Parallel I/O Environment. In: DAGS Symposium on Parallel Computation. pp. 117–134 (1994)

Forest GUMP: A Tool for Explanation

Alnis Murtovi(✉), Alexander Bainczyk, and Bernhard Steffen

Chair for Programming Systems, TU Dortmund University, Dortmund, Germany
{alnis.murtovi,alexander.bainczyk,bernhard.steffen}@tu-dortmund.de

Abstract. In this paper, we present Forest GUMP (for Generalized, Unifying Merge Process) a tool for providing tangible experience with three concepts of explanation. Besides the well-known *model explanation* and *outcome explanation*, Forest GUMP also supports *class characterization*, i.e., the precise characterization of all samples with the same classification. Key technology to achieve these results is algebraic aggregation, i.e., the transformation of a Random Forest into a semantically equivalent, concise white-box representation in terms of Algebraic Decision Diagrams (ADDs). The paper sketches the method and illustrates the use of Forest GUMP along an illustrative example taken from the literature. This way readers should acquire an intuition about the tool, and the way how it should be used to increase the understanding not only of the considered dataset, but also of the character of Random Forests and the ADD technology, here enriched to comprise infeasible path elimination.

Keywords: Random Forest, Binary/Algebraic Decision Diagram, Aggregation, Infeasible Paths, Explainability, Random Seed

1 Introduction

Random Forests are one of the most widely known classifiers in machine learning [3,17]. The method is easy to understand and implement, and at the same time achieves impressive classification accuracies in many applications. Compared to other methods, Random Forests are fast to train and they are clearly more suitable for smaller datasets. In contrast to a single decision tree, Random Forests, a collection of many trees, do not overfit as easily on a dataset and their variance decreases with their size. On the other hand, Random Forests are considered black-box models because of their highly parallel nature: following the execution of Random Forests means, in particular, following the execution in all the involved trees. Such black-box executions are hard to explain to a human user even for very small examples.

In contrast, decision trees are considered white-box models because of their sequential evaluation nature. Even if a tree is large in size, a human can easily follow its computation step by step by evaluating (simple) decisions at each node from the root to a leaf. Indeed, the set of decisions along such an execution path precisely explains why a certain choice has been taken.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 314–331, 2022.
https://doi.org/10.1007/978-3-030-99527-0_17

Popular methods towards explainability try to establish some user intuition. For example, they may hint at the most influential input data, like highlighting or framing the area of a picture where a face has been identified. Such information is very helpful, and it helps in particular to reveal some of the "popular" drastic mismatches incurred by neural networks: if the framed area of the image does not contain the "tagged" object, the identification is clearly questionable. However, even in a correct classification, the tag by itself gives no reason why the identification is indeed correct.

More ambitious are methods that try to turn black-box model into white-box models, ideally preserving the semantics of the classification function. For Random Forests this has been achieved for the first time in [10,14] using the 'aggregating power' of Algebraic Decision Diagrams (ADDs) and Binary Decision Diagrams (BDDs). ADDs are essentially decision trees whose leaves are labelled with elements of some algebra, whereas BBDs are the special case for the algebra of Boolean values. Lifting the algebraic operations from the leaves to the entire ADDs/BDDs allows one to aggregate entire Random Forests into single semantically equivalent ADDs, the precondition for solving three explainability problems:

- The *Model Explanation Problem* [15], i.e. the problem of making the model as a whole interpretable, is solved in terms of an ADD that specifies precisely the same classification function as the original Random Forest (cf. Section 6.2).
- The *Class Characterization Problem*, i.e. the problem, given a class c, characterizing the set of all samples that are classified by the Random Forest as c. This problem is solved in terms of a BDD which precisely characterizes this set of samples (cf. Section 6.3).
- The *Outcome Explanation Problem* [15], i.e. the problem of explaining a concrete classification, is solved in terms of a minimal conjunction of (negated) decisions that are sufficient to guide the sample into the considered class (cf. Section 6.4).

In this paper, we present Forest GUMP (for Generalized, Unifying Merge Process) a tool for providing a tangible experience with the described concepts of explanation. Experimentation with Forest GUMP does not only yield semantically equivalent, concise white-box representations for a given Random Forest which reveal characteristics of the underlying datasets, but it also allows one to experience, e.g., the impact of random seeds on both the quality of prediction and the size of the explaining models (cf. Section 6). Our implementation relies on the standard Random Forest implementation in Weka [28] and on the ADD implementation of the ADD-Lib [9,12,26]. For a more detailed description of the transformations and a quantitative analysis we refer the reader to [10,11,14].

Related Work: Various methods for making Random Forests interpretable exist such as extracting decision rules from the considered black-box model [6], methods that are agnostic to the black-box model under consideration [20,24] or by deriving a single decision tree from the black-box model [5,7,16,27,29]. In this

context, single decision trees are considered key to a solution of both, the model explanation and outcome explanation problem. State of the art solutions to derive a single decision tree from a Random Forest are approximative [5,7,16,27,29]. Thus, their derived explanations are not fully faithful to the original semantics of the considered Random Forest. This is in contrast to our ADD-based aggregation, which precisely reflects the semantics of the original Random Forest.

After a short introduction to Random Forests in Section 2, we present our approach to their aggregation in Section 3 which is followed by an elimination of redundant predicates from the decision diagrams in Section 4 and a non-compositional abstraction in Section 5. Section 6 introduces Forest GUMP and solutions to the three explainability problems. In the end, we summarize the lessons we have learned using Forest GUMP in Section 7 which is followed by a conclusion and direction to future work in Section 8.

2 Random Forests

Learning Random Forests is a quite popular, and algorithmically relatively simple classification technique that yields good results for many real-world applications. Its decision model generalises a training dataset that holds examples of input data labelled with the desired output, also called *class*. As its name suggests, an ensemble of decision trees constitutes a Random Forest. Each of these trees is itself a classifier that was learned from a random sample of the training dataset. Consequently, all trees are different in structure, they represent different decision functions, and can yield different decisions for the very same input data.

To apply a Random Forest to previously unseen input data, every decision tree is evaluated separately: Tracing the trees from their root down to one of the leaves yields one decision per tree, i.e. the predicted class. The overall decision of the Random Forest is then derived as the most frequently chosen class, an aggregation commonly referred to as *majority vote*. The key advantage of this approach is, compared to single decision trees, the reduced variance. A detailed introduction to Random Forests, decision trees, and their learning procedures can be found in [3,17,23].

In this paper, we use Weka [28] as our reference implementation of Random Forests. However, our approach does not depend on implementation details and can be easily adapted to other implementations.

Figure 1 shows a small Random Forests that was learned from the popular Iris dataset [8]. The dataset lists dimensions of Iris flowers' sepals and petals for three different species. Using this forest to decide the species on the basis of given measurements requires to first evaluate the three trees individually and to subsequently determine the majority vote. This effort clearly grows linearly with the size of the forest. In the following we use this example to illustrate our approach of forest aggregation for explainability.

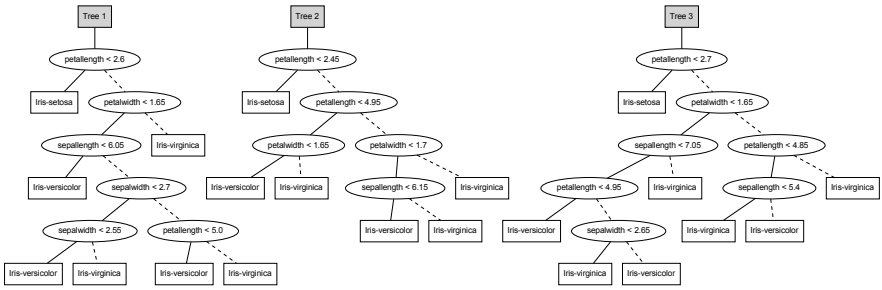


Fig. 1. Random Forest learned from the Iris dataset [8] (39 nodes).

Key idea behind our approach is to partially evaluate the Random Forests at construction time which, in particular, eliminates redundancies between the individual trees of a Random Forest. E.g., in our accompanying Iris flower example (cf. Fig. 1) the predicate $petalwidth < 1.65$ is used in all three trees. This can easily lead to cases where the same predicate is evaluated many times in the classification process. The partial evaluation proposed in this paper transforms Random Forests into decision structures where such redundancies are totally eliminated.

An adequate data structure to achieve this goal for binary decisions are Binary Decision Diagrams [1,4,19] (BDDs): For a given predicate ordering, they constitute a normal form where each predicate is evaluated at most once, and only if required to determine the final outcome.

Algebraic Decision Diagrams (ADDs) [2] generalise BDDs to capture functions of the type $\mathbb{B}^{\mathcal{P}} \to \mathcal{C}^n$ which are exactly what we need to specify the semantics of Random Forests for a classification domain $\mathcal{C}$. Moreover, in analogy to BDDs, which inherit the algebraic structure of their co-domain $\mathbb{B}$, ADDs also inherit the algebraic structure of their co-domains if available.

We exploit this property during the partial evaluation of Random Forests by considering the class vector co-domain (cf. Sect. 3). The aggregation to achieve the corresponding optimised decision structures is then a straightforward consequence of the used ADD technology.

3 Class Vector Aggregation

Class vectors faithfully represent the information about how many trees of the original Random Forest voted for a certain outcome. Obviously, this information is sufficient to obtain the precise results of a corresponding majority vote. Formally, the domain of *class vectors* forms a monoid

$$V := (\mathbb{N}^{|\mathcal{C}|}, +, \mathbf{0})$$

where addition $+$ is defined component-wise and $\mathbf{0}$ is the neutral element.

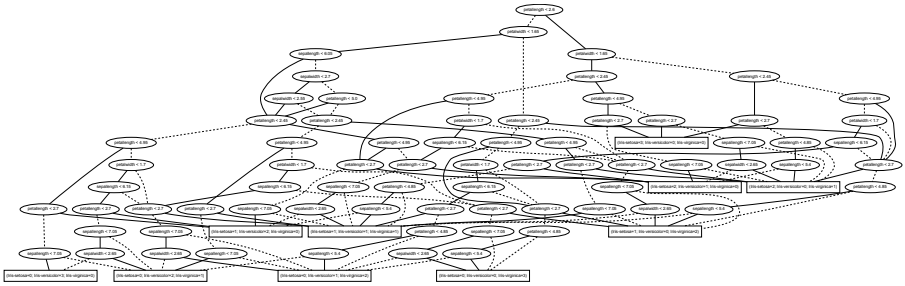


Fig. 2. Class vector aggregation of the Random Forest (83 nodes).

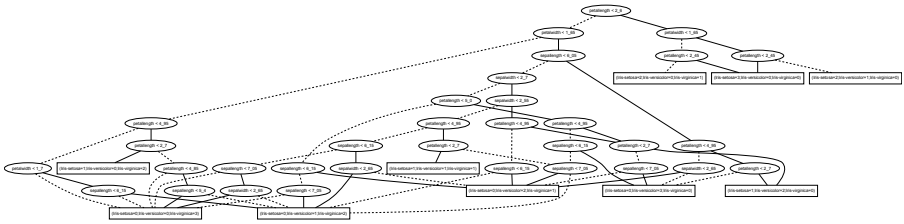


Fig. 3. Class vector aggregation of the Random Forest without semantically redundant nodes (43 nodes).

With the compositionality of the algebraic structure V and the corresponding ADDs $\mathcal{D}_V$, we can transform any Random Forest incrementally into a semantically equivalent ADD. Starting with the empty Random Forest, i.e. the neutral element $\mathbf{0}$, we consider one tree after the other, aggregating a growing sequence of decision trees until the entire forest is entailed in the new decision diagram. The details of this transformation are described in [14]. Figure 2 shows the result of this transformation for our running example.

4 Infeasible Path Elimination

When aggregating the trees of a Random Forest they all use varying sets of predicates. In contrast to simple Boolean variables, predicates are not independent on one another, i.e. the evaluation of one predicate may yield some degree of knowledge about other predicates. E.g., the predicate $petallength < 2.45$ induces knowledge about other predicates that reason about *petallength*: When the petal length is smaller than 2.45 it cannot possibly be greater or equal to 2.7 at the same time. This is not taken care of by the symbolic treatment of predicates we followed until now. In fact, predicates are typically considered independent in the ADD/BDD community.

Infeasible path elimination, as illustrated by the difference between Figure 2 and Figure 3 for our running example, leverages the potential of a semantic

treatment of predicates with significant effect on the size of the resulting ADDs. In fact, the experiments with thousands of trees reported in [14] would not have been successful without infeasible path elimination.

Please note that infeasible path elimination

- is only required after aggregation: The trees in the original Random Forest have no infeasible paths by construction. They are introduced in the course of our *symbolic* aggregation, which is insensitive to semantic properties.
- is compositional and can therefore be applied during the stepwise transformation, before the final most frequent label abstraction (cf. Sect. 5), and at the very end.
- does not support normal forms: Whereas class vector abstraction is canonical for a given variable ordering, infeasible path elimination is not! Thus our approach may yield different decision diagrams depending on the order of tree aggregation. It is guaranteed, however, that the resulting decision diagrams are minimal.

Infeasible path elimination is a hard problem in general.[1] Our corresponding implementation uses SMT-solving [21] to eliminate all infeasible paths. An in-depth discussion of infeasible path elimination is a topic in its own and beyond the scope of this paper.

Class vector aggregation and infeasible path elimination are both compositional and can therefore be applied in arbitrary order without changing the semantics. The majority vote at compile time described in the next section is not compositional and must therefore be applied at the very end.

5 Majority Vote at Compile Time

As mentioned above, maintaining the information about the result of the majority votes is not compositional. In fact, knowing the result of the majority votes for two Random Forest gives no clue about the majority vote of the combined forest. Thus the majority vote abstraction can only be applied at the very end, after the entire aggregation has been computed compositionally.

The result of the compositional aggregation process, including infeasible path elimination, is a decision diagram $d \in \mathcal{D}_V$ with class vectors in its terminal nodes. The majority vote abstraction $\Delta_C : \mathcal{D}_V \to \mathcal{D}_C$ can now be defined as the lifted version of the majority vote abstraction on class vectors $\mathbf{v} \in \mathbb{N}^{|\mathcal{C}|}$ (cf. [14]):

$$\delta_C(\mathbf{v}) := \arg\max_{c \in \mathcal{C}} \mathbf{v}_c.$$

Note that δ_C does not project into the same carrier set but rather from one algebraic structure V into another C. However, these transformations can be applied to the corresponding decision diagrams in the very same way. Fig. 4 shows the result of the most frequent class abstraction for our running example.

[1] For the cases considered here it is polynomial, but there are of course theories for which it becomes exponentially hard or even undecidable.

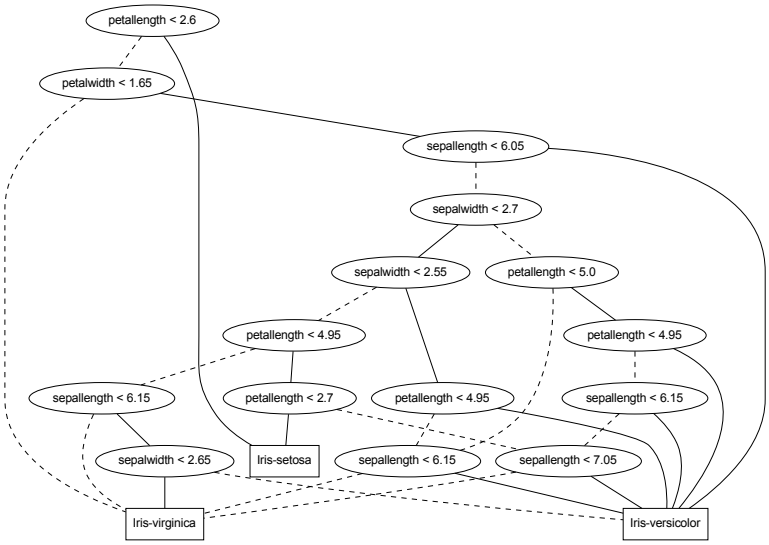


Fig. 4. Most frequent label abstraction of the aggregated Random Forest (majority vote) without semantically redundant nodes (18 nodes).

6 Forest GUMP and Three Problems of Explanability

Forest GUMP[2] (Generalized Unifying Merge Process) is a tool we developed to illustrate the power of algebraic aggregation for the optimization and explanation of Random Forests. It is designed to allow everyone, in particular people without IT or machine learning knowledge, to experience the nature of Random Forests. To avoid unnecessary entry hurdles, we decided to implement Forest GUMP as a simple to use web application. It allows the user to experience the methods described in the previous sections and the proposed solutions to the explanability problems which will be illustrated in the following sections. We will first give a brief overview of Forest GUMP and then showcase its potential in the following sections.

Forest GUMP's user interface (see Figure 5) is essentially divided into two parts. On the left side the user can input the necessary data to learn a Random Forest and subsequently visualize it while the currently chosen representation will be visualized on the right side. First, the user has to upload a dataset or choose one of six datasets that we provide (cf. (1) in Fig. 5) on which the Random Forest will be learned. Next, the hyperparameters necessary for the learning procedure have to be selected, such as the number of trees to be learned (cf. (2) in Fig. 5). Then, one can choose different aggregation methods, i.e. the ones

[2] A link to a running instance of Forest GUMP is available at https://gitlab.com/scce/forest-gump.

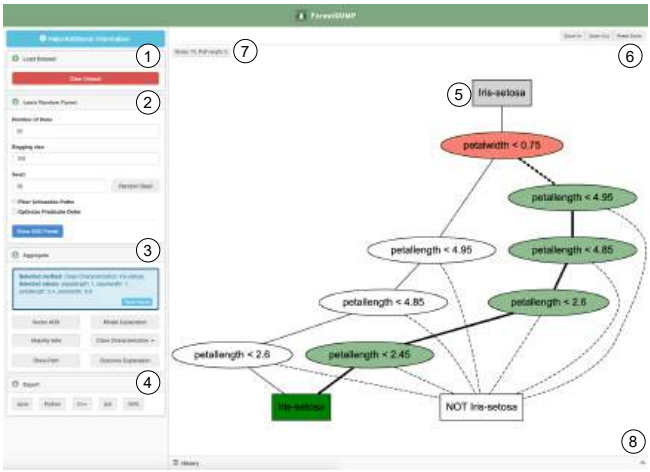


Fig. 5. Overview of Forest GUMP. The visualized ADD is our solution to the class class characterization problem (cf. Sect. 6.3) for the class Iris-Setosa.

Fig. 6. The execution history in Forest GUMP.

mentioned in the previous sections and further ones which will be explained in the following Sections (cf. (3) in Fig. 5). It it also possible to input a sample, classify it with the ADD and highlight the path from the root the leaf (satisfied predicates are highlighted in green, unsatisfied predicates are highlighted in red). In the end, the currently visualized ADD can be exported as Forest GUMP provides code generators for Java, C++, Python and GraphViz's dot format (cf. (4) in Fig. 5). Additionally, the currently visualized ADD can be exported as an SVG to be viewed locally (cf. (4) in Fig. 5).

The grey rectangle (cf. (6) in Fig. 5) points to the root of the currently visualized ADD. One can zoom into/out which can be helpful when the ADDs are rather large (cf. (6) in Fig. 5). On the top left the number of nodes and the length of the currently highlighted path are displayed (cf. (7) in Fig. 5). On the bottom right, one can open a history of all the representations one chose to visualize (cf. (8) in Fig. 5).

Figure 6 shows the expanded execution history. For each visualized ADD, the execution history lists the aggregation variant, the hyperparameters used to learn

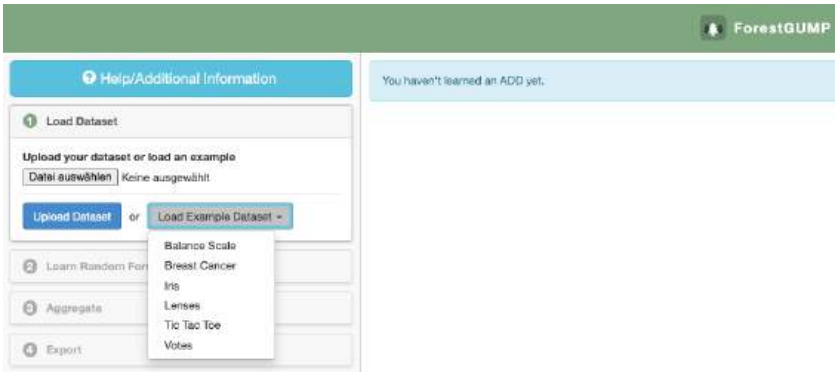


Fig. 7. The user can either choose to upload their own dataset or select one of six exemplary datasets.

the Random Forest and the size (i.e. the number of nodes) and the maximum depth which is the longest path from root to leaf. The execution history also allows one to replay an experiment by clicking on the button on the right side of a row which allows one to compare different ADD variants. One can also delete the individual entries or the whole history and export the history to a CSV.

6.1 A Walkthrough of Forest GUMP

In the following we will see how hard it is to understand how a Random Forest comes to its decision and provide methods for solving the three explainability problems with absolute precision.

Learning a Random Forest To begin, we need a Random Forest which requires a dataset on which it will be learned. In Forest GUMP, the user can upload their own dataset in the Attribute-Relation File Format (ARFF) [28]. Alternatively, we provide six exemplary datasets from which a user can select one to directly start using the tool. Figure 7 illustrates how this looks like in Forest GUMP. Having chosen a dataset, next, the hyperparameters necessary for the learning procedure of the Random Forest have to be specified (see Figure 8). The inputs are the following:

- the *number of trees* to be learned,
- the *bagging size*, i.e. the fraction of samples to be used to learn each tree and
- a *seed* to be able to reproduce the setting.[3]

Additionally, the user can decide to eliminate the infeasible paths as this can strongly reduce the size of the ADDs (see Section 4). While the predicate order is fixed by default, the user can decide to let Forest GUMP optimize the predicate order as the order can also greatly impact the size of the ADDs. A more in

[3] One can generate a random seed by clicking on the button next to the input field.

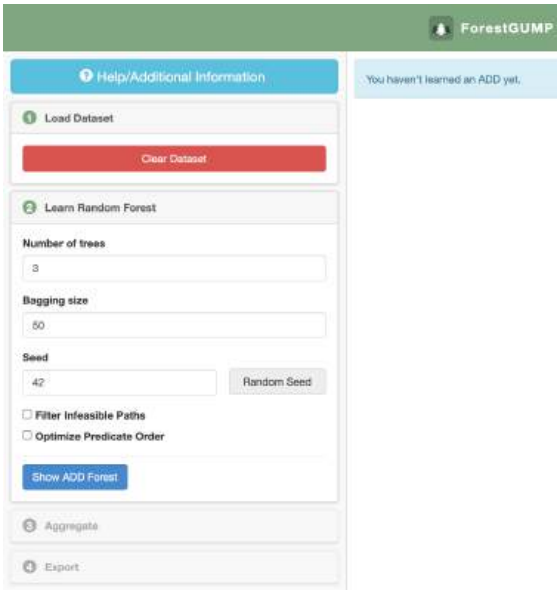


Fig. 8. The user has to specify the necessary hyperparameters to be able to learn a Random Forest. While the first three hyperparameters are needed for the learning procedure, the elimination of the infeasible paths and the optimization of the predicate order are specific to our aggregation method.

depth discussion on the interplay between the infeasible path elimination and the predicate order will follow. Figure 9 shows a Random Forest that was learned on the Iris dataset, consisting of 20 trees[4], a bagging size of 100% and 58 as the seed. If we now want to classify a given input, for each tree we would have to traverse from the root to the leaf and receive one predicted class per tree. The class which was predicted most often is the final result. Trying to understand why the Random Forest predicted this specific class is seemingly impossible. In the following we will show how we can do better.

6.2 Model Explanation Problem

The canonical white-box model corresponding to the Random Forest of Figure 9 can be constructed through the most frequent label abstraction (see Sect. 5) of the aggregated Random Forest (see Sect. 3), whose infeasible paths are eliminated (see Sect. 4). This solves the Model Explanation Problem.

Figure 10 sketches the result of this construction: A canonical white-box model with 310 nodes. Admittedly, this model is still frightening, but given a sample, it allows one to easily follow the corresponding classification process, and in this case it may require at most 19 individual decisions based on the petal

[4] Note that each decision tree is represented as an ADD.

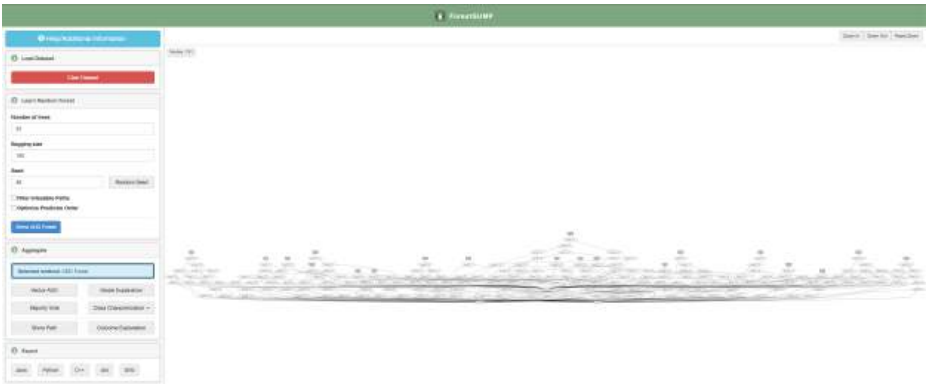


Fig. 9. A Random Forest consisting of 20 individual decision trees (191 number of nodes, longest path consists of 9 nodes). Note that each decision tree is represented as an ADD and that all ADDs share common subfunctions, i.e. it is essentially a shared ADD forest. The actual Random Forest, where nothing is shared, contains 284 nodes.

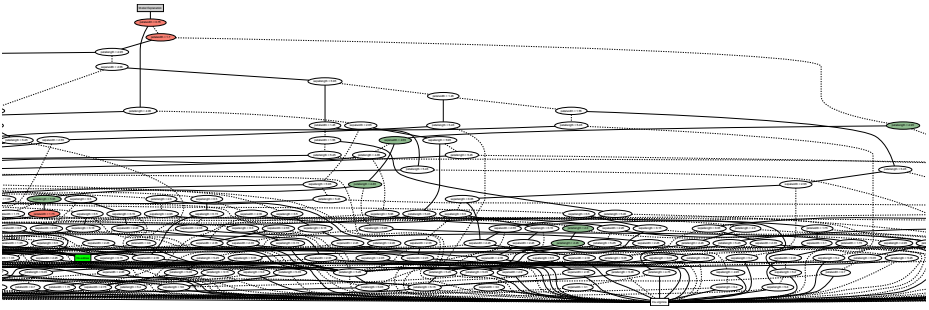


Fig. 10. An extract of the model explanation. The ADD is constructed from the most frequent label abstraction of the aggregated Random Forest following an elimination of all infeasible paths (310 nodes, longest path with length 19, the highlighted path has a length of 9).

and sepal characteristics. This decision set is our set of predicates. The conjunction of these predicates is a solution to the Outcome Explanation Problem. However, more concise explanations are derived from the class characterization BDD discussed in the following section.

Given the sample $petallength = 2.4$, $petalwidth = 1.8$, $sepallength = 5.9$, $sepalwidth = 2.5$, the outcome explanation given by the model explanation consists of the following 9 predicates (in Figure 10 satisfied predicates are highlighted in green, unsatisfied predicates are highlighted in red):

$$\begin{aligned} &\neg(petalwidth < 0.75) \wedge \neg(petalwidth < 1.7) \wedge (petallength < 4.95) \wedge \\ &\quad (sepalwidth < 2.65) \wedge (petallength < 4.85) \wedge (sepallength < 5.95) \wedge \\ &\neg(petalwidth < 1.75) \wedge \quad (petallength < 2.6) \wedge (petallength < 2.45) \end{aligned}$$

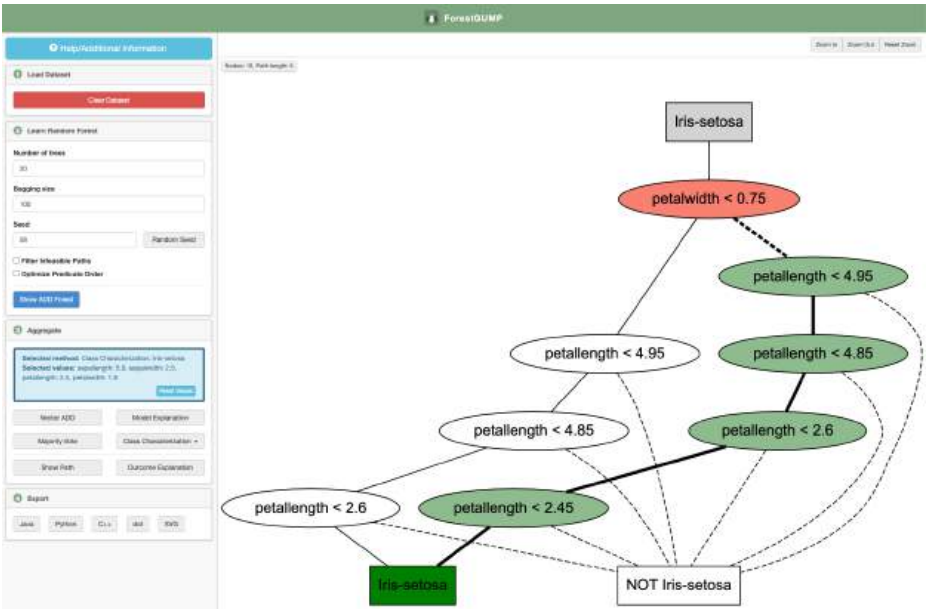


Fig. 11. The class characterization for the class Iris-Setosa (10 nodes, the highlighted path is also the longest path with length 5). The leaf corresponding to Iris-Setosa is highlighted in green, the leaf representing all other classes (i.e. Iris-Virginica and Iris-Versicolor) is highlighted in red.

While this is already an improvement compared to the Random Forest, where you would have to traverse all 20 decision trees, we will see how we can improve even more in the following.

6.3 Class Characterization Problem

The class characterization problem is particularly interesting because it allows one to 'reverse' the classification process. While the direct problem is 'given a sample, provide its classification', the reverse problem sounds 'given a class, what are the characteristics of all the samples belonging to this class?'

BDD-based Class Characterisation can be defined via the following simple transformation function: Given a class $c \in \mathcal{C}$, we define a corresponding projection function $\delta_B(c) : \mathcal{C} \to \mathbb{B}$ on the co-domain as

$$\delta_B(c)(c') := \begin{cases} 1 & \text{if } c' = c \\ 0 & \text{otherwise.} \end{cases}$$

for $c' \in \mathcal{C}$. Again, the function $\delta_B(c)$ can be lifted to operate on ADDs, yielding $\Delta_B(c) : \mathcal{D}_C \to \mathcal{D}_{\mathbb{B}}$.

The BDD shown in Figure 11 is a minimal characterization of the set of all the samples that are guaranteed to be classified as Iris-Setosa.

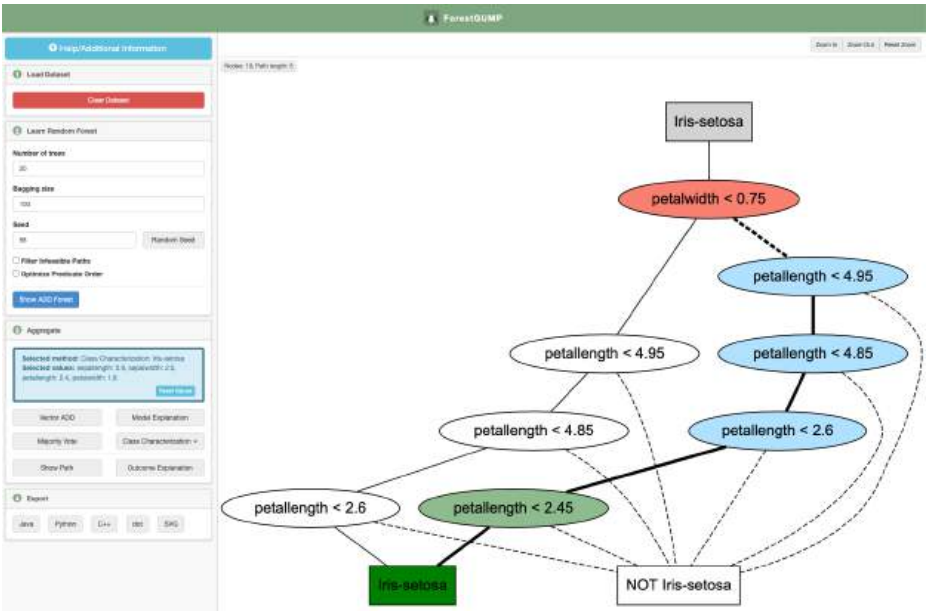


Fig. 12. The outcome explanation for the input $petallength = 2.4$, $petalwidth = 1.8$, $sepallength = 5.9$, $sepalwidth = 2.5$ (10 nodes, hightlighted path of length 5).

Being able to reverse a learned classification function has a major practical importance. Think, e.g., of a marketing research scenario where data have been collected with the aim to propose bestfitting product offers to customers according to their user profile. This scenario can be considered as a classification problem where the offered product plays the role of the class. Now, being able to reverse the customer $\rightarrow$ product classification function provides the marketing team with a tailored product $\rightarrow$ customer promotion process: for a given product, it addresses all customers considered to favor this very product as in the corresponding patent [18].

The path highlighted in Figure 11 is the path from the root to the leaf for the same sample $petallength = 2.4$, $petalwidth = 1.8$, $sepallength = 5.9$, $sepalwidth = 2.5$. Compared to the path with length 9 in the model explanation, we now have a path of length 5 with the following predicates:

$$\neg(petalwidth < 0.75) \wedge (petallength < 4.95) \wedge (petallength < 4.85) \wedge (petallength < 2.6) \wedge (petallength < 2.45)$$

6.4 Outcome Explanation Problem

The previous classification formula expresses the collection of 'conditions' that this sample satisfies, and it provides therefore a precise justification why it is classified in this class. Despite the fact that the class characterization BDD is

canonical, it is easy to see that there are some redundancies in the formula. For example, a $petallength < 2.45$ is also inherently smaller than 2.6, 4.85 and 4.95; therefore, for this specific sample those three predicates are redundant. This is the result of the imposed predicate ordering in BDDs: all the BDD predicates are listed, and they are listed in a fixed order. After eliminating these redundancies, we are left with the following precise minimal outcome explanation: this sample is recognized as belonging to the class Iris-Setosa because it has the properties $\neg(petalwidth < 0.75) \wedge (petallength < 2.45)$.

In Forest GUMP we make these redundant predicates explicit by highlighting them in blue (see Figure 12). From 9 predicates in the model explanation to 5 predicates in the class characterization, we have now arrived at an explanation that only consists of 2 predicates.

7 Lessons Learned

Playing with Forest GUMP led to interesting observations not only concerning the analyzed data domains but also concerning Random Forest Learning and the applied ADD technology.

Random Forest Learning. Changing the random seed for the learning process had a significant impact on the size of the explanation models and the class characterizations. The observed sizes of the explanation models ranged from 138 to 519. Interesting was that the larger sizes did not necessarily imply a better prediction quality. The same also applied to the class characterizations. In fact, we observed a 100% prediction quality for a class characterization of only 3 nodes, while a class characterization for the same species with 40 nodes only scored 33% prediction.

Analyzed Data Domain. The class characterizations for the three iris species differed quite a bit. For two species the observed sizes were much bigger than the sizes of the third species, independently of the chosen random seed and bagging size. In fact, for Iris-Setosa we observed a class characterization with only 3 nodes implying an outcome explanation for our chosen sample with only one predicate. Figure 13 serves for the corresponding explanation. Put it differently, class characterizations seem to be good indications for 'tightness': The closer the samples lie the more criteria are required for separation.

ADD Technology. ADDs are canonical as soon as one has chosen a predicate/variable ordering. Although we could observe the effect of corresponding optimization heuristics[5], the impact was moderate and helpful mainly for model explanation and class characterization. Figure 14 shows the the outcome explanation for the same problem but where the ADD, representing the class characterization for the class Iris-Setosa, is reordered.[6] While the reordering

[5] CUDD [25] provides a number of heuristics for optimizing variable orders.

[6] The used reordering method is named CUDD_REORDER_GROUP_SIFT_CONV as it was both, fast and effective, in our experiments.

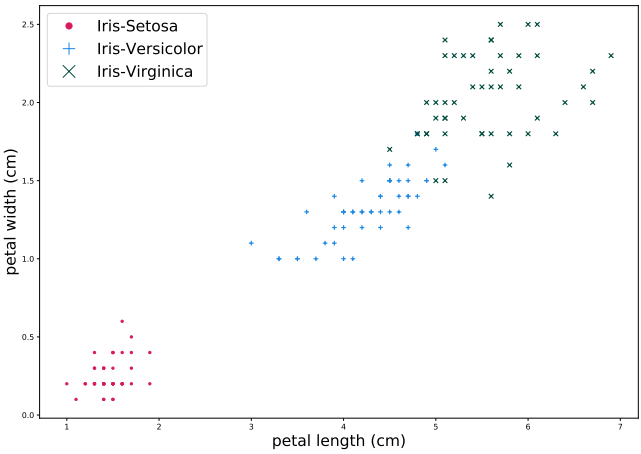


Fig. 13. Visualization of the iris dataset using only the petal length and petal width.

reduces the class characterization size from 10 to 8 nodes, the length of the outcome explanation is unchanged. For the model explanation of Figure 10, the size can be reduced from 310 nodes to 196 nodes while the path for the sample $petallength = 2.4$, $petalwidth = 1.8$, $sepallength = 5.9$, $sepalwidth = 2.5$ actually increased by 1 (from 9 to 10). Thus the outcome explanation may even be impaired. This is not too surprising as these optimizations aim a size reduction and not depth reduction of the considered ADDs. We are currently investigating good heuristics for depth reduction.

More striking was the impact of infeasible path elimination. In fact, this optimization can be regarded key for scalability when increasing the forest size. [14] reports results about forests with 10.000 trees. Without infeasible path reduction already 100 trees are problematic.

Standard ADD frameworks work on Boolean variables rather than predicates. Thus in their setting infeasible paths do not occur. The problem of infeasible path reduction in ADDs was first discussed in [13,14]. Our current corresponding solution is still basic. We are currently generalizing our solution using more involved SMT technology.

Of course, these observations where made on rather small datasets and it has to be seen how well they tranfer to more complex scenarios. We believe, however, that they indicate general phenomena whose essence remains true in larger setting.

8 Conclusion and Perspectives

We have presented Forest GUMP (for Generalized, Unifying Merge Process) a tool for providing tangible experience with three concepts of explanation: *model*

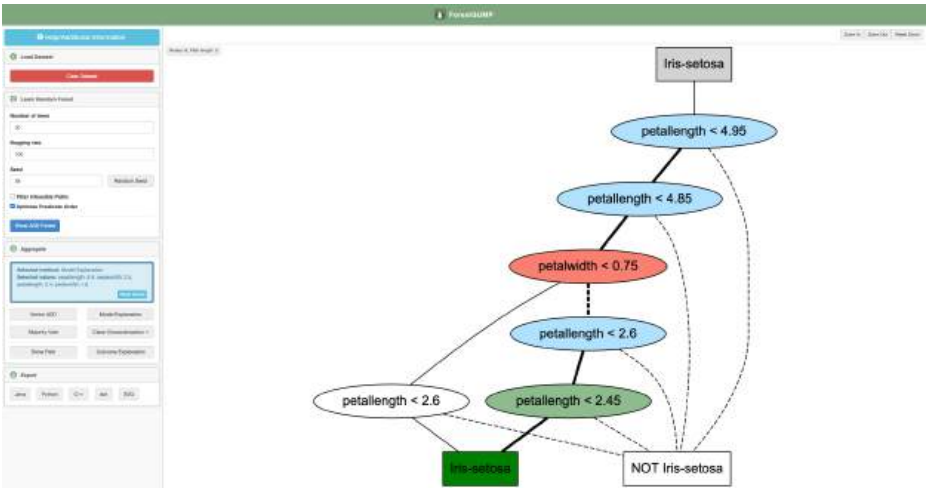


Fig. 14. The outcome explanation for the input $petallength = 2.4$, $petalwidth = 1.8$, $sepallength = 5.9$, $sepalwidth = 2.5$ (8 nodes, highlighted path of length 5) where the class characterization from Figure 11 is reordered.

explanation, *outcome explanation*, and *class characterization*. Key technology to achieve model explanation is algebraic aggregation, i.e. the transformation of a Random Forest into a semantically equivalent, concise white-box representation in terms of Algebraic Decision Diagrams. Class characterization is then achieved in terms of BDDs where the structure unnecessary to distinguish the considered class is collapsed. This abstraction is not only interesting in itself to better understand how easily the classes can be separated, but it also leads to highly optimized outcome explanations. Together with infeasible path elimination and the suppression of redundant predicates on a path, we observe reductions of outcome explanations by more than an order of magnitude. Forest GUMP allows even newcomers to easily experience these phenomena without much training.

Of course, these are first steps in a very ambitious new direction and it has to be seen how far the approach carries. Scalability will probably require decomposition methods, perhaps in a similar fashion as illustrated by the difference between model explanation and the considerably smaller class characterization. More work is needed also on techniques that aim at limiting the number of involved predicates.

Data Availability Statement: The artifact is available in the Zenodo repository [22].

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **27**(6), 509–516 (1978)
2. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Proceedings of 1993 Inter-

national Conference on Computer Aided Design (ICCAD). pp. 188–191 (1993). https://doi.org/10.1109/ICCAD.1993.580054
3. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (Oct 2001). https://doi.org/10.1023/A:1010933404324
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
5. Chipman, H.A., George, E.I., McCulloch, R.E.: Making sense of a forest of trees (1999)
6. Deng, H.: Interpreting tree ensembles with intrees. Int. J. Data Sci. Anal. **7**(4), 277–287 (2019). https://doi.org/10.1007/s41060-018-0144-8
7. Domingos, P.M.: Knowledge discovery via multiple models. Intell. Data Anal. **2**(1-4), 187–202 (1998). https://doi.org/10.1016/S1088-467X(98)00023-7
8. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of eugenics **7**(2) (1936)
9. Gossen, F., Margaria, T., Murtovi, A., Naujokat, S., Steffen, B.: Dsls for decision services: A tutorial introduction to language-driven engineering. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11244, pp. 546–564. Springer (2018). https://doi.org/10.1007/978-3-030-03418-4_33
10. Gossen, F., Margaria, T., Steffen, B.: Towards explainability in machine learning: The formal methods way. IT Prof. **22**(4), 8–12 (2020). https://doi.org/10.1109/MITP.2020.3005640
11. Gossen, F., Margaria, T., Steffen, B.: Formal methods boost experimental performance for explainable AI. IT Prof. **23**(6), 8–12 (2021). https://doi.org/10.1109/MITP.2021.3123495, https://doi.org/10.1109/MITP.2021.3123495
12. Gossen, F., Murtovi, A., Linden, J., Steffen, B.: The java library for algebraic decision diagrams. https://add-lib.scce.info, accessed: 2022-01-13
13. Gossen, F., Steffen, B.: Large random forests: Optimisation for rapid evaluation. CoRR **abs/1912.10934** (2019), http://arxiv.org/abs/1912.10934
14. Gossen, F., Steffen, B.: Algebraic aggregation of random forests: towards explainability and rapid evaluation. International Journal on Software Tools for Technology Transfer (Sep 2021). https://doi.org/10.1007/s10009-021-00635-x
15. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A survey of methods for explaining black box models. ACM Comput. Surv. **51**(5), 93:1–93:42 (2019). https://doi.org/10.1145/3236009
16. Hara, S., Hayashi, K.: Making tree ensembles interpretable: A bayesian model selection approach. In: Storkey, A.J., Pérez-Cruz, F. (eds.) International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain. Proceedings of Machine Learning Research, vol. 84, pp. 77–85. PMLR (2018), http://proceedings.mlr.press/v84/hara18a.html
17. Ho, T.K.: Random decision forests. In: Proceedings of 3rd International Conference on Document Analysis and Recognition. vol. 1, pp. 278–282 vol.1 (1995). https://doi.org/10.1109/ICDAR.1995.598994
18. Hungar, H., Steffen, B., Margaria, T.: Methods for generating selection structures, for making selections according to selection structures and for creating selection descriptions. https://patents.justia.com/patent/9141708 (Sep 2015), USPTO Patent number: 9141708

19. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell System Technical Journal **38**(4), 985–999 (1959)
20. Lou, Y., Caruana, R., Gehrke, J.: Intelligible models for classification and regression. In: Yang, Q., Agarwal, D., Pei, J. (eds.) The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012. pp. 150–158. ACM (2012). https://doi.org/10.1145/2339530.2339556
21. 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). https://doi.org/10.1007/978-3-540-78800-3_24
22. Murtovi, A., Bainczyk, A., Steffen, B.: Forest gump: A tool for explanation (tacas 2022 artifact) (Nov 2021). https://doi.org/10.5281/zenodo.5733107
23. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
24. Ribeiro, M.T., Singh, S., Guestrin, C.: "why should I trust you?": Explaining the predictions of any classifier. In: Krishnapuram, B., Shah, M., Smola, A.J., Aggarwal, C.C., Shen, D., Rastogi, R. (eds.) Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016. pp. 1135–1144. ACM (2016). https://doi.org/10.1145/2939672.2939778
25. Somenzi, F.: Cudd: Cu decision diagram package release 3.0 (2015)
26. Steffen, B., Gossen, F., Naujokat, S., Margaria, T.: Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages, pp. 311–344. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_17
27. Van Assche, A., Blockeel, H.: Seeing the forest through the trees: Learning a comprehensible model from an ensemble. In: Kok, J.N., Koronacki, J., Mantaras, R.L.d., Matwin, S., Mladenič, D., Skowron, A. (eds.) Machine Learning: ECML 2007. pp. 418–429. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
28. Witten, I.H., Frank, E., Hall, M.A., Pal, C.J.: Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edn. (2016)
29. Zhou, Y., Hooker, G.: Interpreting models via single tree approximation (2016)

Alpinist: an Annotation-Aware GPU Program Optimizer*

Ömer Şakar[1](✉), Mohsen Safari[1], Marieke Huisman[1], and Anton Wijs[2]

[1] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{o.f.o.sakar,m.safari,m.huisman}@utwente.nl
[2] Software Engineering & Technology, Eindhoven University of Technology, Eindhoven, The Netherlands
a.j.wijs@tue.nl

Abstract. GPU programs are widely used in industry. To obtain the best performance, a typical development process involves the manual or semi-automatic application of optimizations prior to compiling the code. To avoid the introduction of errors, we can augment GPU programs with (pre- and postcondition-style) annotations to capture functional properties. However, keeping these annotations correct when optimizing GPU programs is labor-intensive and error-prone.
This paper introduces Alpinist, an annotation-aware GPU program optimizer. It applies frequently-used GPU optimizations, but besides transforming code, it also transforms the annotations. We evaluate Alpinist, in combination with the VerCors program verifier, to automatically optimize a collection of verified programs and reverify them.

Keywords: GPU · Optimization · Deductive verification · Annotation-aware · Program transformation

1 Introduction

Over the course of roughly a decade, graphics processing units (GPUs) have been pushing the computational limits in fields as diverse as computational biology [64], statistics [35], physics [7], astronomy [24], deep learning [29], and formal methods [17,43,44,65,67]. Dedicated programming languages such as CUDA [34] and OpenCL [42] can be used to write GPU source code. To achieve the most performance out of GPUs, developer should apply incremental optimizations, tailored to the GPU architecture. Unfortunately, this is to a large extent a *manual* activity. The fact that for different GPU devices, the same code tends to require a different sequence of transformations [21] makes this procedure even more time consuming and error-prone. Recently, automating this has received some attention, for instance by applying machine learning [3].

* This work is supported by NWO grant 639.023.710 for the Mercedes project and by NWO TTW grant 17249 for the ChEOPS project

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 332–352, 2022.
https://doi.org/10.1007/978-3-030-99527-0_18

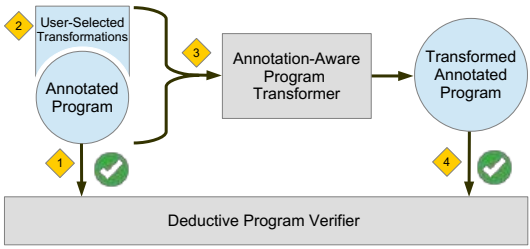


Fig. 1: Annotation-Aware Program Transformation.

Reasoning about the correctness of GPU software is hard, but necessary. Multiple verification techniques and tools have been developed to aid in this task aimed at detecting data races, see [8, 10, 14, 32, 33], and for a recent overview, see [22]. Some of these techniques apply deductive program verification, which requires a program to be *manually* augmented with pre- and postcondition annotations. However, annotating a program is time consuming. The more complex a program is, the more challenging it becomes to annotate it. In particular, as a program is being optimized repeatedly, its annotations tend to change frequently.

This paper presents Alpinist, a tool that can apply *annotation-aware transformations* [26] on annotated GPU programs. It can be used with the deductive program verifier VerCors [9]. VerCors can verify the *functional correctness* of GPU programs [10]. It allows the verification of many typical GPU computations, see e.g., [48,50,51]. The purpose of Alpinist is twofold (see Fig. 1): First, it automates the optimization of GPU code, to the extent that the developer needs to indicate which optimization needs to be applied where, and the tool performs the transformation. Interestingly, the presence of annotations is exploited by Alpinist to determine whether an optimization is actually applicable, and in doing so, can sometimes apply an optimization where a compiler cannot. Second, as it applies a code transformation, it also transforms the related annotations, which means that once the developer has annotated the unoptimized, simpler code, any further optimized version of that code is automatically annotated with updated pre- and postconditions, making it reverifiable. This avoids having to re-annotate the program every time it is optimized for a specific GPU device.

Alpinist supports GPU code optimizations that are used frequently in practice, namely loop unrolling, tiling, kernel fusion, iteration merging, matrix linearization and data prefetching. In the current paper, we discuss how Alpinist has been implemented, how it can be applied on annotated GPU code, and how some of the more complex optimizations work. In addition, we evaluate the effect of applying several of these optimizations, both in terms of annotation size and time needed to verify a program, to a collection of examples including the verified case studies in [48, 49, 51].

Outline. Section 2 demonstrates how Alpinist optimizes a verified GPU program while preserving its provability. Section 3 discusses the architecture of Alpinist. Section 4 discusses the most complex optimizations supported by

```
1  /*@ context_everywhere N > 0 && N < a.length;
2  req (\forall* int i; 0 <= i < a.length; Perm(a[i], 1));
3  ens (\forall* int i; 0 <= i < a.length; i != a.length-1 ==> Perm(a[i+1], 1));
4  ens (\forall* int i; 0 <= i < a.length; i == a.length-1 ==> Perm(a[0], 1));
5  ens (\forall int i; 0 <= i < a.length-1; a[i+1] == N*i);
6  ens a[0] == N*(a.length-1); @*/
7  void Host(int[] a, int size, int N) {
8    par Kernel1 (int tid = 0 .. a.length)
9     /*@ context Perm(a[tid], 1);
10    ens a[tid] == 0; @*/
11   { a[tid] = 0; }
12   par Kernel2 (int tid = 0 .. a.length)
13    /*@ context tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
14    req tid != a.length-1 ? a[tid+1] == 0 : a[0] == 0;
15    ens tid != a.length-1 ? a[tid+1] == N*tid : a[0] == N*tid; @*/
16   { /*@ inv k >= 0 && k <= N;
17     inv tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
18     inv tid != a.length-1 ? a[tid+1] == k*tid : a[0] == k*tid;@*/
19     for(int k = 0; k < N; k++) {
20       if (tid != a.length-1) { a[tid+1] = a[tid+1] + tid; }
21       else { a[0] = a[0] + tid; }
22 } } }
```

Fig. 2: A verified GPU-style program

Alpinist in detail, namely loop unrolling, tiling and kernel fusion, and briefly discusses the remaining three. Section 5 presents the results of experiments in which the tool has been applied on a collection of programs. Section 6 discusses related work and Section 7 concludes the paper, and discusses future work.

2 Annotation-Aware Optimization using Alpinist

This section illustrates how Alpinist can optimize a verified GPU program while preserving its provability. Fig. 2 shows a GPU program with annotations [10] that is verified by VerCors. The example is written in a simplified version of VerCors' own language PVL. The program initializes an array `a`, and subsequently updates the values in `a`, `N` times. The workflow of a GPU program in general is that the host (i.e., CPU) invokes a *kernel*, i.e., a GPU function, executed by a specified number of GPU threads. These threads are organized in one or more *thread blocks*. In this program, there are two kernels, both executed by one thread block of `a.length` threads (lines 8 and 12 (l.8, l.12))[3]. Each thread has a unique identifier, in the example called `tid`. In the first kernel (l.8-l.11), each thread initializes `a[tid]` to 0. In the second kernel (l.12-l.22), each thread updates `a[tid+1]` (modulo `a.length`) `N` times, by adding `tid` to it. In the main `Host` function, `Kernel1` is called, followed by `Kernel2`.

The kernels, the for-loop and the host function are annotated for verification (in blue), using permission-based separation logic [6,11,12]. Permissions capture which memory locations may be accessed by which threads; they are fractional values in the interval (0, 1] (cf. Boyland [12]): any fraction in the interval (0,

[3] In practice, the size of a block cannot exceed a specific upper-bound, but for this example, we assume that `a.length` is sufficiently small.

```
1  /*@ context_everywhere N > 0 && N < a.length;
2   req (\forall* int i; 0 <= i < a.length; Perm(a[i], 1));
3   ens (\forall* int i; 0 <= i < a.length; i != a.length-1 ==> Perm(a[i+1], 1));
4   ens (\forall* int i; 0 <= i < a.length; i == a.length-1 ==> Perm(a[0], 1));
5   ens (\forall int i; 0 <= i < a.length-1; a[i+1] == N*i);
6   ens a[0] == N*(a.length-1); @*/
7  void Host(int[] a,int size,int N){
8    par Fused_Kernel(int tid = 0 .. a.length)
9     /*@ req Perm(a[tid], 1);
10     ens tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
11     ens tid != a.length-1 ? a[tid+1] == N*tid : a[0] == N*tid; @*/
12   {
13     a[tid] = 0;
14     /*@ req Perm(a[tid], 1);
15      req a[tid] == 0;
16      ens tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
17      ens tid != a.length-1 ? a[tid+1] == 0 : a[0] == 0; @*/
18     barrier(Fused_Kernel)
19
20     int a_reg_0, a_reg_1;
21     if (tid != a.length-1) { a_reg_1 = a[tid+1] } else { a_reg_0 = a[0] }
22     int k = 0;
23     if (tid != a.length-1) { a_reg_1 = a_reg_1 + tid; }
24     else { a_reg_0 = a_reg_0 + tid; }
25     k ++;
26     /*@ inv k >= 0 + 1 && k <= N;
27      inv tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
28      inv tid != a.length-1 ? a_reg_1 == k*tid : a_reg_0 == k*tid; @*/
29     for(k; k < N; k++) {
30       if (tid != a.length-1) { a_reg_1 = a_reg_1 + tid; }
31       else { a_reg_0 = a_reg_0 + tid; }
32     }
33     if (tid != a.length-1) { a[tid+1] = a_reg_1 } else { a[0] = a_reg_0 };
34 } }
```

Fig. 3: An optimized GPU-style program, annotated for verification

1) indicates a read permission, while 1 indicates a write permission. A write permission can be split into multiple read permissions and read permissions can be added up, and transformed into a write permission if they add up to 1. The soundness of the logic ensures that for each memory location, the total number of permissions among all threads does not exceed 1.

To specify permissions, predicates are used of the form `Perm(L, π)` where `L` is a heap location and π a fractional value in the interval (0, 1] (e.g., 1\3). Pre- and postconditions, denoted by keywords `req` and `ens`, should hold at the beginning and the end of an annotated function, respectively. The keyword `context` abbreviates both `req` and `ens` (l.9, l.13). The keyword `context_everywhere` is used to specify a property that must hold throughout the function (l.1). Note that `\forall*` is used to express a universal separating conjunction over permission predicates (l.2-l.4) and `\forall` is used as standard universal conjunction over logical predicates (l.5). For logical conjunction, `&&` is used and $**$ is used as separating conjunction in separation logic.

In the example, write permissions are required for all locations in `a` (l.2). The pre- and postconditions of the first kernel specify that each thread needs write permission for `a[tid]` (l.9). The postcondition states that `a[tid]` is set to 0 (l.10). In the second kernel, all threads have write permission for `a[tid+1]`,

except thread `a.length-1` which has write permission for `a[0]` (l.13). Moreover, it is required that `a[tid+1]` (modulo `a.length`) is 0 (l.14). For the for-loop (l.19-l.22), loop invariants are specified: `k` is in the range $[0, N]$ (l.16), each thread has write permission for `a[tid+1]` (modulo `a.length`) (l.17) and this location always has the value `k*tid` (l.18). The postconditions of the second kernel and the host function are similar to this latter invariant.

Fig. 3 shows an optimized version of the program, with updated annotations to make it verifiable. Alpinist has applied three optimizations:

1. *Fusing the two kernels*: in GPU programs, the only *global* synchronisation points (used, for instance, to avoid data races) exist implicitly between kernel launches. However, if such a global synchronisation point is not really needed between two specific kernels, then fusing them gives several benefits, in particular the ability to store intermediate results in (fast) thread-local register memory as opposed to (slow) GPU global memory, and it has a positive effect on power consumption [62]. In the example, the kernels are combined into `Fused_Kernel`, and a *thread block-local* barrier is introduced (l.18) to avoid data races within the single thread block executing the code.
2. *Using register memory*; register variables can be used to reduce the number of global memory accesses. Here, the use of `a_reg_0` and `a_reg_1` has been enabled by kernel fusion.
3. *Unrolling the for-loop*; the for-loop has been unrolled once here (l.20-l.25). Since GPU threads are very light-weight, compared to CPU threads, any checking of conditions that can be avoided benefits performance. When unrolling a loop, this means that fewer checks of the loop-condition are needed. Note that here, Alpinist benefits from the knowledge that $N > 0$ (l.1), so it knows that the for-loop can be unrolled at least once.

To preserve provability of the optimized program, Alpinist changed the annotations, in particular the pre- and postcondition of the fused kernel and the loop invariants (highlighted in Fig. 3). Moreover, Alpinist introduced an annotated barrier (l.14-l.18). Since threads synchronize at a barrier, it is possible to redistribute the permissions. In the rest of the paper, we discuss how Alpinist performs these annotation-aware transformations.

3 The Design of Alpinist

This section gives a high-level overview of the design of Alpinist. The optimizations supported by Alpinist are discussed in Section 4. To understand the design of Alpinist, we first explain the architecture of the VerCors verifier.

3.1 VerCors' Architecture

VerCors is a deductive program verifier, which is designed to work for different input languages (e.g., Java and OpenCL). It takes as input an annotated program,

which is then transformed in several steps into an annotated Silver program. Silver is an intermediate verification language, used as input for Viper [37, 60]. Viper then generates proof obligations, which can be discharged by an automated theorem prover, such as Z3 [36].

The internal transformations in VerCors are defined over our internal AST representation (written in the Common Object Language or COL [52]), which captures the features of all input languages. Some of the transformations are generic (e.g., splitting composite variable declarations) and others are specific to verification (e.g., transforming contracts). The transformations implemented as part of Alpinist are also applied on the COL AST, but they are developed with a different goal in mind, and in particular several of the transformation are specific to the supported optimizations.

Using VerCors and its architecture to implement Alpinist gives us some benefits. First, existing helper functions can be reused, which simplifies tasks such as gathering information regarding specific AST nodes. Second, some generic transformations of VerCors can be reused, such as splitting composite variable declarations or simplifying expressions. This helps to simplify the implementation of the optimizations. Third, using the architecture of VerCors allows us to prove assertions that we generate relatively easily by invoking VerCors internally.

3.2 Alpinist's Architecture

Alpinist takes a verified file as its input, annotated with special optimization annotations that indicate where specific optimizations should be applied. Alpinist is written in Java and Scala and runs on Windows, Linux and macOS. Fig. 4 gives a high-level overview of the internal design of Alpinist. The input program goes through four phases: the *parsing* phase, the *applicability checking* phase, the *transformation* phase and the *output* phase.

The *parsing phase* transforms the input file into a COL AST, after which the *applicability checking phase* checks if the optimization can be applied. Some optimizations, such as tiling (see Section 4.2), are always applicable, hence their applicability check always passes. For other optimizations, prerequisites must be established. Sometimes, a syntactical analysis of the AST suffices, e.g., kernel fusion (see Section 4.3). For this optimization, it must be determined whether there is any data dependency between two selected kernels. When analysis of the AST is not enough, VerCors can be used to perform more complex reasoning. An example of this is loop unrolling (see Section 4.1). Its prerequisite is that for the loop to be unrollable `k` times, it is guaranteed that the loop executes at least `k` times. This prerequisite is encoded as an assertion to be proven by VerCors.

The applicability checking phase is one of the strengths of Alpinist. It exploits the fact that the input program is annotated to determine whether an optimization is applicable, and relies on the fact that VerCors can perform complex reasoning. Moreover, this approach allows to distinguish failure due to unsatisfied prerequisites and due to mistakes in the transformation procedure.

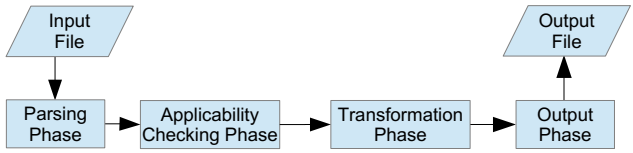


Fig. 4: The internal design of ALPINIST.

If the applicability check passes (i.e., the optimization is applicable), the transformation phase is next, otherwise a message is generated that the prerequisites could not be proven.

The *transformation* phase applies the optimizations to the input AST. The *output phase* either prints the optimized program in the same language as the input program, or a message is printed, signifying either a failure in optimizing or a verification failure in the applicability checking phase.

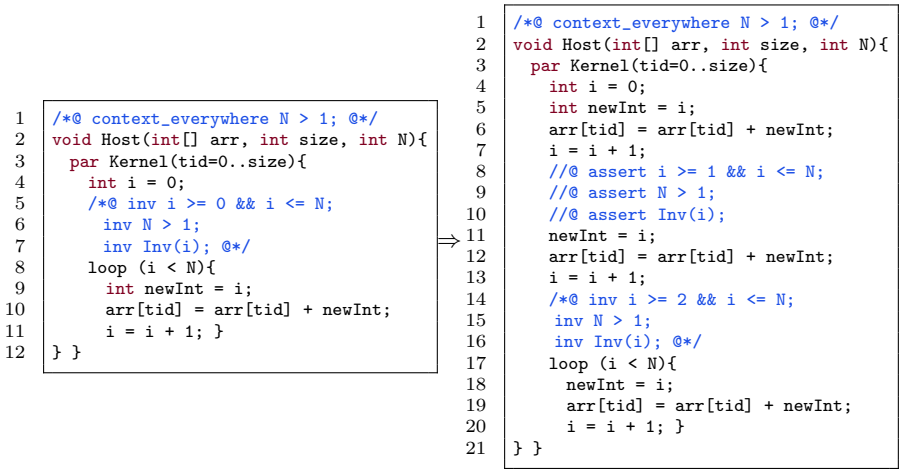
4 GPU Optimizations

ALPINIST supports six frequently-used GPU optimizations, namely loop unrolling, tiling, kernel fusion, iteration merging, matrix linearization and data prefetching. This section discusses loop unrolling, tiling, and kernel fusion in detail. The other optimizations follow the same approach in spirit and are discussed briefly, which can be found in the ALPINIST implementation [16]. Each optimization is introduced in the context of GPU programs. Then, we discuss how to apply them. Interesting insights are discussed where relevant.

4.1 Loop Unrolling

Loop unrolling is a frequently-used optimization technique that is applicable to both GPU and CPU programs. It unrolls some iterations of a loop, which increases the code size, but can have a positive impact on program performance; e.g., see [21, 38, 46, 59, 63] for its impact, specifically on GPU programs. Fig. 5 shows an example of unrolling an (annotated) loop twice: the body of the loop is duplicated twice before the loop. This has the following effect on the annotations: the loop invariant bounding the loop variable (l.5) changes in the optimized program (l.14). Note that the other loop invariants (i.e., `Inv(i)`) remain the same. Moreover, after each unrolling part, we add all invariants as assertions (l.8-l.10) except after the last unroll. This captures that the code produced by unrolling the loop should still satisfy the original loop invariants.

Our approach to loop unrolling is more general than optimization techniques during compilation. For instance, the `unroll` pragma in CUDA [55] and the unroll function in Halide [56] unroll loops by calculating the number of iterations to see if unrolling is possible, i.e., it should be computable at compile time. This difference is illustrated in Fig. 5 where `N` (i.e., the number of iterations) is unknown at compile time. Their approach *cannot automatically* handle this



```
1  /*@ context_everywhere N > 1; @*/
2  void Host(int[] arr, int size, int N){
3    par Kernel(tid=0..size){
4      int i = 0;
5      /*@ inv i >= 0 && i <= N;
6        inv N > 1;
7        inv Inv(i); @*/
8      loop (i < N){
9        int newInt = i;
10       arr[tid] = arr[tid] + newInt;
11       i = i + 1; }
12 } }
```

⇒

```
1  /*@ context_everywhere N > 1; @*/
2  void Host(int[] arr, int size, int N){
3    par Kernel(tid=0..size){
4      int i = 0;
5      int newInt = i;
6      arr[tid] = arr[tid] + newInt;
7      i = i + 1;
8      //@ assert i >= 1 && i <= N;
9      //@ assert N > 1;
10     //@ assert Inv(i);
11     newInt = i;
12     arr[tid] = arr[tid] + newInt;
13     i = i + 1;
14     /*@ inv i >= 2 && i <= N;
15      inv N > 1;
16      inv Inv(i); @*/
17     loop (i < N){
18       newInt = i;
19       arr[tid] = arr[tid] + newInt;
20       i = i + 1; }
21 } }
```

Fig. 5: An example of unrolling a loop 2 times.

```
1  void Host(int[] array, int size){
2    par Kernel(tid=0..size){
3      int i = init; // The loop variable
4      :
5      //@ assert (i == a) || (i == b); // Depending on initialization of i only one
6                                       // of the conditions is specified
7      /*@ inv i >= a && i <= b; // The lowerbound of i (a), The upperbound of i (b)
8       inv Inv(i); @*/ // Additional loop invariants
9      loop (cond(i)) { // The loop condition
10       body(i); // The loop body, a sequence of statements in the i^th iteration.
11       i = upd(i); } // The update function of i, restricted to (i + c), (i − c),
12 } }                 // (i × c) or (i/c) where c is a positive integer constant[4].
```

Fig. 6: A general template of a loop inside a kernel.

case, while our approach *can automatically* unroll the loop, since annotations (l.1, l.6) specify the lower-bound of `N` (provided by the programmer, who knows that this is a valid lower-bound). VerCors verifies that the unrolling is valid.

Fig. 6 shows a loop template in a verified GPU program. We would like to automatically unroll the loop `k` times and preserve the provability of the program. To accomplish this, we follow a procedure consisting of three parts: the main, checking and updating part. In the *main part*, an annotated (verified) GPU program and positive `k` are given as input. Next we go to the *checking part*, to see if it is possible to unroll the loop `k` times. This part corresponds with the applicability checking phase. Thus, we statically calculate the number of loop iterations, by counting how many times the condition (`cond(i)`) holds starting from either `a` (as the lowerbound of `i`) or `b` (as the upperbound of `i`), depending on the operation of `upd(i)`. If `k` is greater than the total number of loop iterations at the end of the checking part, then we report an error. Otherwise

[4] If `c` was negative, for the multiplication and division, `i` would oscillate between positive and negative values and hence would not always be useful as array index. Hence we consider `c` to be positive.

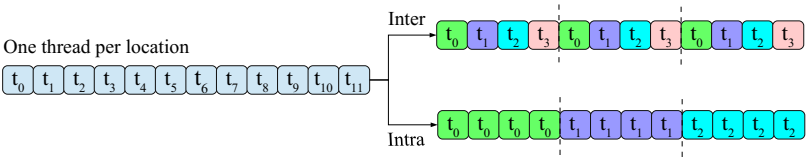


Fig. 7: Inter- and intra-tiling of an array as `T` = 12, `N` = 4 and $\lceil$`T/N`$\rceil$ = 3.

```
1  void Host(int[] a, int T){
2    par Kernel(tid = 0..T)
3    /*@ // Preconditions related to permissions and functional correctness
4        req prePerm(a[tid]) ** preFunc(a[tid]);
5        // Postconditions related to permissions and functional correctness
6        ens postPerm(a[tid]) ** postFunc(a[tid]); @*/
7    { body(a[tid]); } }
```

Fig. 8: A general unoptimized GPU program to apply for tiling.

we go to the *updating part*, in which we update either `a` or `b` according to the operation in `upd(i)`. If the operation is addition or multiplication, then the loop variable `i` (in the unoptimized program) goes from `a` to `b`. That means, after unrolling, `a` should be updated according to the constant `c` from the update expression and `k`. If the operation is subtraction or division, `i` goes from `b` to `a`. Thus, after unrolling, `b` should be updated. After the updating part, we return to the main part to unroll the loop `k` times.

4.2 Tiling

Tiling is another well-known optimization technique for GPU programs. It increases the workload of the threads to fully utilize GPU resources by assigning more data to each thread. Concretely, we assume there are `T` threads and a one-dimensional array of size `T` in the unoptimized GPU program where each thread is responsible for one location in that array (Fig. 8). To apply the optimization, we first divide the array into $\lceil$`T/N`$\rceil$ chunks, each of size `N` ($1 \leq$ `N` $\leq$ `T`)[5]. There are two different ways to create and assign threads to array cells (as in Fig. 7):

- *Inter-Tiling* We define `N` threads and assign them to one specific location in each chunk. That means each thread serially iterates over all chunks to be responsible for a specific location in each chunk.
- *Intra-Tiling* We define $\lceil$`T/N`$\rceil$ threads and assign one thread to one chunk (i.e., 1-to-1 mapping) to serially iterate over all cells in that chunk.

Both forms of tiling can have a positive impact on GPU program performance; e.g., see [25, 28, 47, 69] for the impact of this optimization.

Fig. 9 shows the optimized version of Fig. 8 by applying inter-tiling. Regarding program optimization, two major changes happen: 1) the total number of threads has reduced (l.2), and 2) the body is encapsulated inside a loop (l.16-l.18). As mentioned, in inter-tiling, we define `N` threads instead of `T`. The number

[5] Since `N` is in the range $1 \leq$ `N` $\leq$ `T`, the last chunk might have fewer cells.

```
1  void Host(int[] a, int T){
2    par Kernel(tid = 0..N)
3    /*@ req (\forall* int i; 0 <= i && i < ceiling(T, N) && tid+i×N < T;
4           pre(a[tid+i×N]));
5        ens (\forall* int i; 0 <= i && i < ceiling(T, N) && tid+i×N < T;
6           post(a[tid+i×N])); @*/
7    {
8      int j = 0;
9      /*@ inv j >= 0 && j <= ceiling(T, N);
10     inv (\forall* int i; 0 <= i && i < ceiling(T, N) && tid+i×N < T;
11                    prePerm(a[tid+i×N]));
12     inv (\forall int i; j <= i && i < ceiling(T, N) && tid+i×N < T;
13                    preFunc(a[tid+i×N]));
14     inv (\forall* int i; 0 <= i && i < j && tid+i×N < T;
15                    postFunc(a[tid+i×N])); @*/
16     loop (tid+j×N < T){
17       body(a[tid+j×N]);
18       j = j + 1; }
19 } }
```

Fig. 9: Optimized version of the GPU program of Fig. 8 after applying inter-tiling.

of chunks is indicated by the function `ceiling(T, N)`. Each thread in the newly added loop iterates over all chunks (in the range 0 to `ceiling(T, N)`-1) to be responsible for a specific location. This happens by the loop variable `j` and the loop condition `tid+j×N < T`. This means, each thread `tid` can access its own location at index `tid` in each chunk. To preserve verifiability, we add invariants to the loop (l.9-l.17). Therefore, we specify:

- the boundaries of the loop variable `j`, which iterates over all chunks.
- a permission-related invariant for each thread in each chunk (l.10). This comes from the precondition of the kernel and is quantified over all chunks.
- an invariant to indicate functional properties of the locations that have not yet been updated by threads in the body of the loop (l.12). This comes from the functional precondition of the kernel and is quantified over all chunks.
- an invariant to specify how each thread updates the array in each chunk (l.14). This comes from the functional property as the postcondition of the kernel and is quantified over all chunks.

Moreover, we modify the specification of the kernel (l.3-l.6). Note that we have the condition `tid+j×N < T` in all universally quantified invariants, because the last chunk might have fewer cells than `N`. We quantified the pre- and postcondition of the kernel over the chunks in the same way as the invariants.

Intra-tiling is in essence similar to inter-tiling with two major differences: 1) the total number of threads is `ceiling(T, N)`, and 2) each thread in the loop iterates over cells within its own chunk. Therefore, we have different conditions in the loop and the quantified invariants. ALPINIST also supports this.

Above, each thread is assigned to one cell. This can easily be generalized to have each thread assigned to one or more consecutive cells (i.e., a task). A similar procedure can be applied as long as the tasks do not overlap, i.e., each cell is assigned to at most one thread.

4.3 Kernel Fusion

Kernel fusion is a GPU optimization where we merge two or more consecutive kernels into one. It increases the potential to use thread-local registers to store intermediate results (see Section 2) and can lead to less power consumption. See [2,19,61,62,68] for the impact of kernel fusion on GPU programs. We provide a generalized procedure to fuse an *arbitrary number* of consecutive kernels while considering *data dependency* between them. The idea is to fuse them by repeatedly fusing the first two kernels (i.e., kernel reduction). In each iteration, if there is no data dependency between the two kernels, we safely fuse them. Else if there is only one thread block then we fuse the two kernels by inserting a barrier between the bodies, else fusion fails.

A benefit of this approach is that it only considers two kernels at a time. In this way, it can be determined whether a barrier is necessary between two specific kernels, and we do not miss any possible fusion optimization. Another benefit of this approach is that when a data dependency between two kernels P and $P+1$ ($1 < P < \#\text{kernels}-1$) is detected, the output of the approach is the fusion of the first P kernels, and the remaining unfused kernels after P. This allows the user to not only find out that there is a data dependency between P and $P+1$, but also to obtain fused kernels where possible.

There are multiple challenges in this transformation: (1) how to detect data dependency between two kernels? (2) how to collect the pre- and postconditions for the fused kernel? and (3) how to deal with permissions so that in the fused kernel the permission for a location does not exceed 1? The main difficulty in addressing these challenges is that we have to consider many different possible scenarios. Fortunately, we can use the information from the contract of the two kernels. The permission patterns in the contract indicate for each thread which locations it reads from and writes to. We provide procedures to separately collect pre- and postconditions related to permissions and to functional correctness. Due to space limitations, we only discuss the essential steps to collect the precondition related to permissions for array accesses of the fused kernel in Alg. 1. Collecting the rest of the contract uses a similar procedure.

Alg. 1 requires kernels `k1` and `k2` to not lose any permissions, only possibly redistribute them (using a barrier). Furthermore, for ease of presentation, we assume that in both `k1` and `k2`, each thread accesses at most one cell of array `a`, and that the expressions used to compute array indices only combine constants and thread ID variables, using standard arithmetic operators.

We compare the postcondition of `k1` and the precondition of `k2` (l.2) to understand how to add permissions of the preconditions of `k1` and `k2` to the precondition of the fused kernel. Note that `prePerm` and `postPerm` correspond to a permission-related pre- and postcondition, respectively. We use the postcondition of `k1` for this comparison since the permission at the end of `k1` needs to be sufficient to satisfy the precondition of `k2`. If the index expressions `e1` and `e2` to access an array `a` are syntactically the same, then they refer to the same array cell. In that case, we first add to the precondition of the fused kernel the *original* permission from the precondition of `k1` that corresponds to the permis-

Algorithm 1 Kernel fusion procedure for collecting precondition permissions.

```
1: Add all precondition permissions related to non-shared arrays (i.e., accessed by only one of the
   two kernels) into the contract of the fused kernel kf.
2: for each shared array a with a permission postPerm(a[e1], p1) in the postcondition of the first
   kernel k1 and a permission prePerm(a[e2], p2) in the precondition of the second kernel k2 do
3:     if patterns e1 and e2 are syntactically the same then
4:         Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
5:         if p1 < p2 then
6:             Add prePerm(a[e2], p2-p1) as pre. to kf
7:     else if patterns e1 and e2 are not syntactically the same then
8:         if p1 + p2 ≤ 1 then
9:             Add pre. of k1 corresp. to postPerm(a[e1], p1) and prePerm(a[e2], p2) as pre. in kf
10:        else if p1 + p2 > 1 && p1 < 1 && p2 < 1 then
11:            Add pre. of k1 corresp. to postPerm(a[e1], p1) with permission p3 and prePerm(a[e2],
12:            p4) as pre. s.t. p3 + p4 == 1
13:        else if p1 == 1 (i.e., write) then                    ▷ Data dependency, add barrier
14:            Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
15:        else p2 == 1                                          ▷ Data dependency, add barrier
16:            Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
17:            Add prePerm(a[e2], 1-p1) as pre. to kf
```

sion for `a[e1]` in the postcondition of `k1` (remember that the latter permission may have been obtained in `k1` after permission redistribution). Second, if `p1` is not sufficient for the precondition of `k2` (l.5), we add additional permission to the precondition of the fused kernel to satisfy the precondition of `k2` (l.6).

The remaining different cases in the algorithm correspond to the different edge cases that we should consider when `e1` and `e2` are not syntactically the same. In particular, data dependency happens when the accumulated permission (in both kernels) for one location is greater than 1, and there is at least one write permission. Therefore, we have to distinguish multiple cases: 1) `p1` + `p2` does not exceed 1 (l.8), 2) `p1` + `p2` exceeds 1, but no write permission is involved (l.10), or 3) and 4) at least one write is involved (l.13 and l.15). In the latter two cases, a barrier must be introduced to take care of distributing permissions from the access in `k1` to the access in `k2`, and possibly additional permission for the latter must be added to the precondition of the fused kernel (l.17). After constructing the contract of the fused kernel, we check for data dependency.

Fig. 10 shows an example of fusing two kernels. We only present the permission precondition expressions which are collected with Alg. 1. There are two shared arrays `a` and `b`. To collect permission preconditions in the fused kernel, we follow steps {l.2→l.3→l.4} for array `a` and steps {l.2→l.3→l.4→l.5→l.6} for array `b`. As there is no data dependency, we can safely fuse the two kernels.

Implementing Data Dependency Detection. One of the implementation challenges of kernel fusion is to check data dependency in the applicability checking phase. Our idea of detecting kernel dependencies is similar to detecting loop iteration dependencies, see [1]. To detect data dependency for a specific shared array, the function `SV` is used. Fig. 11 shows an example of the output of `SV`. The kernel has `1\2` permission for `a[tid+1]` and `1\3` permission for `a[0]` if `tid+1` is out of bounds. `SV` takes an array name and the pre- and postconditions of a kernel (of the form `cond(tid) => Perm(a[patt(tid)], p)`) on l.3-l.6, and returns a mapping from indices `patt(tid)` to the permissions `p` (in Fig. 11: right).

```
1  void Host(...){
2    par Kernel1(tid1 = 0..T)
3    /*@ context Perm(a[tid1], 1);
4    context Perm(b[tid1], 1\2);@*/
5    { a[tid1] = 2*b[tid1]; }
6      par Kernel2(tid2 = 0..T)
7    /*@ context Perm(a[tid2], 1\2);
8    context Perm(b[tid2], 1);@*/
9    { b[tid2] = a[tid2]+1; } }
```

$\Longrightarrow$

```
1  void Host(...){
2    par Fused_Kernel(tid = 0..T)
3    /*@ req Perm(a[tid], 1);
4    req Perm(b[tid], 1\2);
5    req Perm(b[tid], 1\2);@*/
6    { a[tid] = 2*b[tid];
7      b[tid] = a[tid]+1; } }
```

Fig. 10: An example of collecting preconditions in fusing two kernels.

```
1  void Host(...){
2    par Kernel1(tid1 = 0..T)
3    /*@ context (tid != a.length-1 =>
4      Perm(a[(tid + 1)], 1\2));
5     context (tid == a.length-1 =>
6      Perm(a[0], 1\3)); @*/
7    { ... } }
```

$\Longrightarrow$

Output SV(a, spec kernel)

index	0	1	2	3	4
permission	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Fig. 11: Example output of the SV function for array `a`.

If the function `SV` is executed for two kernels to fuse with the same shared array `a`, the results $\texttt{SV}_1\texttt{(a)}$ and $\texttt{SV}_2\texttt{(a)}$ can be compared to determine whether there is data dependency between the two kernels. This comparison is described generally at l.8-l.16 in Algorithm 1. For each corresponding location in $\texttt{SV}_1\texttt{(a)}$ and $\texttt{SV}_2\texttt{(a)}$, we can determine, for example, whether both permissions combined do not exceed 1 (l.8) or whether the location in `k1` has write permission (l.12).

4.4 Other Optimizations

We briefly discuss the three remaining optimizations supported by Alpinist. Iteration merging is an optimization technique related to loop unrolling that is applicable to both GPU and CPU programs[6]. Iteration merging reduces the number of loop iterations by extending the loop body with multiple copies of it, as opposed to creating copies of it outside the loop, as is done in loop unrolling. Iteration merging can have a positive performance impact; see [38,46,53] for the effectiveness of this optimization on GPU programs.

Matrix linearization is an optimization where we transform two-dimensional arrays into one dimension ones. This optimization can result in better memory access patterns, thereby improving caching. See [5,13,54] for the impact of matrix linearization on GPU programs.

The last optimization implemented in Alpinist is data prefetching. Suppose there is a verified GPU program where each thread accesses an array location in global memory multiple times. In this optimization, we prefetch the values of those locations that are in global memory into registers which are local to each thread. A similar optimization, in which intermediate results are stored in register memory, is applied in Section 2. Therefore, instead of multiple accesses to the high latency global memory, we benefit from low-latency registers. Data prefetching can have a positive performance impact; see [4,58,70].

[6] Iteration merging is also referred to as loop unrolling/vectorization in the literature.

Table 1: A summary of the optimization and verification times for all optimizations.

Optimization	Optim. time (s)				Verif. time (orig.) (s)				Verif. time (opt.) (s)			
	min.	max.	avg.	med.	min.	max.	avg.	med.	min.	max.	avg.	med.
Loop unrolling	0.067	0.238	0.116	0.098	7.6	50.7	18.2	14.3	7.6	57.5	20.8	17.3
Tiling	0.044	0.052	0.048	0.047	16.7	21.5	18.7	18.1	19.3	31.4	24.7	20.8
Kernel fusion	0.099	0.338	0.173	0.137	16.7	54.5	24.6	20.0	14.9	22.3	19.0	19.5
Iteration merging	0.042	0.592	0.152	0.097	6.9	51	17.0	12.7	7.3	64	20.0	13.8
Matrix linearization	0.011	0.044	0.022	0.017	11.6	16	14.3	14.1	11.5	16.8	14.4	15.1
Data prefetching	0.010	0.068	0.051	0.053	9.7	23	14.0	13.4	10.4	23	13.5	12.7

5 Evaluation

This section describes the evaluation of Alpinist. The goal is to

Q1 test whether Alpinist works on GPU programs.
Q2 investigate how long it takes for Alpinist to transform GPU programs and how this affects the verification time.
Q3 investigate the usability of Alpinist on real-world complex examples.

5.1 Experiment Setup

Alpinist is evaluated on examples from three different sources. The first source consists of hand-made examples that cover different scenarios for each optimization. The second source is a collection of verified programs from VerCors' example repository[7]. The third source consists of complex case studies that are already verified in VerCors: two parallel prefix sum algorithms [51], parallel stream compaction and summed-area table algorithms [48], a variety of sorting algorithms [49], a solution [27] to the VerifyThis 2019 challenge 1 [18] and a Tic-Tac-Toe example [57] based on [23]. In total, we applied the optimizations 30 times in the first category, 23 times in the second category and 17 times in the third category (in total 70 experiments). All the examples are annotated with special optimization annotations such that Alpinist can apply those optimizations automatically. All these examples are publicly available at [15]. All the experiments were conducted on a MacBook Pro 2020 (macOS 11.3.1) with a 2.0GHz Intel Core i5 CPU. Each experiment was performed ten times, after which the average times, i.e., optimization and verification times, of those executions were recorded for the experiment.

5.2 Results & Discussion

Q1 To test whether Alpinist works on GPU programs, we applied the six optimizations in all 70 experiments and used VerCors to reverify all the resulting programs. All these tests were successful.
Q2 To investigate how long it takes for Alpinist to transform GPU programs, we recorded the transformation time for each optimization applied to all the

[7] The example repository of VerCors is available at https://github.com/utwente-fmt/vercors/tree/dev/examples.

Table 2: An overview of optimizing case studies, where # is the unroll factor (for loop unrolling) or the merge factor (for iteration merging), **OT** the time it takes to optimize, **VB** the original verification time (Verification Before) and **VA** the optimized verification time (Verification After). All times are in seconds.

Case	Loop unrolling				Iter. merging				Matrix lin.			Data pref.		
	#	OT	VB	VA	#	OT	VB	VA	OT	VB	VA	OT	VB	VA
BubbleSort [49]	1	0.101	25.4	27.3	4	0.170	29.8	34.1	N/A	N/A	N/A	N/A	N/A	N/A
InsertionSort [49]	1	0.134	25.6	25.8	3	0.225	24.1	28.0	N/A	N/A	N/A	N/A	N/A	N/A
SelectionSort [49]	1	0.107	23.5	25.7	2	0.592	22.8	27.7	N/A	N/A	N/A	N/A	N/A	N/A
TimSort [49]	2	0.216	29.3	38.5	3	0.182	29.1	37.9	N/A	N/A	N/A	N/A	N/A	N/A
Blelloch [51]	1	0.129	50.7	57.5	3	0.355	51.0	64.0	N/A	N/A	N/A	N/A	N/A	N/A
Kogge-Stone [51]	1	0.238	23.0	25.6	2	0.082	21.8	25.6	N/A	N/A	N/A	0.103	23.0	23.0
TicTacToe [57]	3	0.106	19.8	21.0	2	0.076	17.3	19.6	N/A	N/A	N/A	N/A	N/A	N/A
VerifyThis [27]	1	0.144	26.2	28.7	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Transpose [48]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.022	16.0	16.0	N/A	N/A	N/A

examples. Table 1 summarizes the best and worst optimization times for the six optimizations (as reported by Alpinist). To investigate the impact on the verification time, the table also shows the (best and worst) verification times of the original and optimized programs (as reported by VerCors). The table shows the minimum, maximum, average and median times of all examples. It can be observed that Alpinist takes insignificant time to apply each optimization to all the examples. Moreover, the verification time after optimizing generally increases. For loop unrolling, tiling and iteration merging, the verification time increases. This can be attributed to the additional code that is generated. For kernel fusion, the verification time decreases. This is due to verifying fewer kernels. For matrix linearization and data prefetching, the verification time slightly increases. This can be attributed to the linear expressions in matrix linearization and the extra statements to read from/write to the registers in data prefetching.

Q3 To investigate the usability of Alpinist on real-world examples, we successfully applied it on the third category with the complex case studies. Table 2 shows the optimization and verification times of applying loop unrolling, iteration merging, matrix linearization and data prefetching to these case studies. Note that in the case studies only these four optimizations could be applied. In the table, `N/A` indicates that the optimization is not applicable to the example.

6 Related Work

To the best of our knowledge, this is the first paper to showcase a tool that implements annotation-aware transformations. We categorize the related work into three parts, covering both tools and optimizations.

Automatic Optimizations without Correctness. There is a large body of related work, see e.g., [2, 4, 19, 25, 28, 47, 61, 62, 68–70], that shows the impact of automated optimizations on GPU programs, but does not consider *correctness*, or the preservation of it. Our tool can potentially complement these approaches by preserving the provability of the optimized programs.

Correctness Proofs for Transformations. Another body of related work focuses on different approaches to preserve provability not specific to GPU programs. COMPCert [30, 31] is a formally verified C compiler which preserves semantic equivalence of the source and compiled program, by proving correctness of each transformation in the compilation process. Wijs and Engelen [66] and De Putter and Wijs [45] prove the preservation of functional properties over transformations on models of concurrent systems. They prove preservation of model-independent properties. This approach differs from ours as they work on models instead of concrete programs.

Compiler Optimization Correctness. Finally, there is related work that focusses on the compilation of sequential programs, performing transformations from high-level source code to lower-level machine code while preserving the semantics. These approaches neither consider parallelization, nor target different architectures. In GPU programming, the optimizations often need to be applied manually rather than during the compilation process.

Namjoshi and Xu [41] use a proof checker to show equivalence between an original WebAssembly program and optimized program. An equivalence proof is generated based on the transformations. Namjoshi and Singhania [40] created a semi-automatic loop optimizer with user-directives. The loops are verified during compilation. For each transformation, semantics are defined to guarantee semantical equivalence to the original program. Namjoshi and Pavlinovic [39] focus on recovering from precision loss due to semantics-preserving program transformations and propose systematic approaches to simplify analysis of the transformed program. Finally Gjomemo et al. [20] help compiler optimizations by supplying high-level information gathered by external static analysis (e.g., Frama-C). This information is used by the compiler for better reasoning.

7 Conclusion

In this paper, we presented Alpinist, the annotation-aware GPU program optimizer. Given an unoptimized, annotated GPU program, we showed how Alpinist transforms both the code and the annotations, with the goal to preserve the provability of the optimized GPU program. Alpinist supports loop unrolling, tiling, kernel fusion, iteration merging, matrix linearization and data prefetching, of which the first three are discussed in detail. We discussed the design and implementation of Alpinist, and we validated it by verifying a set of examples and reverifying their optimized counterparts.

For future work, there are other optimizations that could be supported, such as data prefetching for all memory patterns as mentioned by Ayers et al. [4]. Another open question is if and how this approach can be used in program compilation. We also plan to extend this approach to preserve the provability of transpiled code, e.g., CUDA to OpenCL conversions. Moreover, we plan to investigate how Alpinist can be combined with techniques such as *autotuning* that automatically detect the potential for applying specific optimizations and identify optimal parameter configurations [3, 63].

References

1. Allen, R., Kennedy, K.: Automatic translation of Fortran programs to vector form. ACM Transactions on Programming Languages and Systems (TOPLAS) **9**(4), 491–542 (1987)
2. Ashari, A., Tatikonda, S., Boehm, M., Reinwald, B., Campbell, K., Keenleyside, J., Sadayappan, P.: On optimizing machine learning workloads via kernel fusion. ACM SIGPLAN Notices **50**(8), 173–182 (2015)
3. Ashouri, A., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A Survey on Compiler Autotuning using Machine Learning. ACM Computing Surveys **51**(5), 96:1–96:42 (2018)
4. Ayers, G., Litz, H., Kozyrakis, C., Ranganathan, P.: Classifying memory access patterns for prefetching. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 513–526 (2020)
5. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. Tech. rep., Citeseer (2008)
6. Berdine, J., Calcagno, C., O'Hearn, P.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W. (eds.) FMCO. LNCS, vol. 4111, pp. 115–137. Springer (2005)
7. Bertolli, C., Betts, A., Mudalige, G., Giles, M., Kelly, P.: Design and Performance of the OP2 Library for Unstructured Mesh Applications. In: Proceedings of the 1st Workshop on Grids, Clouds and P2P Programming (CGWS). Lecture Notes in Computer Science, vol. 7155, pp. 191–200. Springer (2011). https://doi.org/10.1007/978-3-642-29737-3_22
8. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA. pp. 113–132. ACM (2012)
9. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors Tool Set: Verification of Parallel and Concurrent Software. In: iFM. LNCS, vol. 10510, pp. 102 – 110. Springer (2017)
10. Blom, S., Huisman, M., Mihelčić, M.: Specification and Verification of GPGPU programs. Science of Computer Programming **95**, 376–388 (2014)
11. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL). pp. 259–270 (2005)
12. Boyland, J.: Checking Interference with Fractional Permissions. In: SAS. LNCS, vol. 2694, pp. 55–72. Springer (2003)
13. Catanzaro, B., Keller, A., Garland, M.: A decomposition for in-place matrix transposition. ACM SIGPLAN Notices **49**(8), 193–206 (2014)
14. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic testing of OpenCL code. In: Haifa Verification Conference. pp. 203–218. Springer (2011)
15. Şakar, O., Safari, M., Huisman, M., Wijs, A.: The repository for the examples used in Alpinist, https://github.com/OmerSakar/Alpinist-Examples.git
16. Şakar, O., Safari, M., Huisman, M., Wijs, A.: The repository for the implementations of Alpinist, https://github.com/utwente-fmt/vercors/tree/gpgpu-optimizations/src/main/java/vct/col/rewrite/gpgpuoptimizations
17. DeFrancisco, R., Cho, S., Ferdman, M., Smolka, S.: Swarm Model Checking on the GPU. International Journal on Software Tools for Technology Transfer **22**, 583–599 (2020). https://doi.org/10.1007/s10009-020-00576-x

18. Dross, C., Furia, C.A., Huisman, M., Monahan, R., Müller, P.: Verifythis 2019: a program verification competition. International Journal on Software Tools for Technology Transfer pp. 1–11 (2021)
19. Filipovič, J., Madzin, M., Fousek, J., Matyska, L.: Optimizing CUDA code by kernel fusion: application on BLAS. The Journal of Supercomputing **71**(10), 3934–3957 (2015)
20. Gjomemo, R., Namjoshi, K.S., Phung, P.H., Venkatakrishnan, V., Zuck, L.D.: From verification to optimizations. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 300–317. Springer (2015)
21. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a High-Level Language Targeted to GPU Codes. In: Proc. 2012 Innovative Parallel Computing (InPar). pp. 1–10. IEEE (2012). https://doi.org/10.1109/InPar.2012.6339595
22. van den Haak, L., Wijs, A., M.G.J. van den Brand, Huisman, M.: Formal Methods for GPGPU Programming: Is The Demand Met? In: Proceedings of the 16th International Conference on Integrated Formal Methods (IFM 2020). Lecture Notes in Computer Science, vol. 12546, pp. 160–177. Springer (2020). https://doi.org/10.1007/978-3-030-63461-2_9
23. Hamers, R., Jongmans, S.S.: Safe sessions of channel actions in Clojure: a tour of the discourje project. In: International Symposium on Leveraging Applications of Formal Methods. pp. 489–508. Springer (2020)
24. Herrmann, F., Silberholz, J., Tiglio, M.: Black Hole Simulations with CUDA. In: GPU Computing Gems Emerald Edition, chap. 8, pp. 103–111. Morgan Kaufmann (2011)
25. Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., Sadayappan, P.: Adaptive sparse tiling for sparse matrix multiplication. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 300–314 (2019)
26. Huisman, M., Blom, S., Darabi, S., Safari, M.: Program correctness by transformation. In: 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). LNCS, vol. 11244. Springer (2018)
27. Huisman, M., Joosten, S.: A solution to VerifyThis 2019 challenge 1, https://github.com/utwente-fmt/vercors/blob/97c49d6dc1097ded47a5ed53143695ace6904865/examples/verifythis/2019/challenge1.pvl
28. Konstantinidis, A., Kelly, P.H., Ramanujam, J., Sadayappan, P.: Parametric GPU code generation for affine loop programs. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 136–151. Springer (2013)
29. Le, Q., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Ng, A.: On Optimization Methods for Deep Learning. In: Proceedings of the 28th International Conference on Machine Learning (ICML). pp. 265–272. Omnipress (2011)
30. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 42–54 (2006)
31. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009)
32. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: SIGSOFT FSE 2010, Santa Fe, NM, USA. pp. 187–196. ACM (2010)
33. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: ACM SIGPLAN Notices. vol. 47, pp. 215–224. ACM (2012)

34. Lindholm, L., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro **28**(2), 39–55 (2008). https://doi.org/10.1109/MM.2008.31
35. Liu, X., Tan, S., Wang, H.: Parallel Statistical Analysis of Analog Circuits by GPU-Accelerated Graph-Based Approach. In: Proceedings of the 2012 Conference and Exhibition on Design, Automation & Test in Europe (DATE). pp. 852–857. IEEE Computer Society (2012). https://doi.org/10.1109/DATE.2012.6176615
36. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
37. Müller, P., Schwerhoff, M., Summers, A.: Viper - a verification infrastructure for permission-based reasoning. In: VMCAI (2016)
38. Murthy, G.S., Ravishankar, M., Baskaran, M.M., Sadayappan, P.: Optimal loop unrolling for GPGPU programs. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). pp. 1–11. IEEE (2010)
39. Namjoshi, K.S., Pavlinovic, Z.: The impact of program transformations on static program analysis. In: International Static Analysis Symposium. pp. 306–325. Springer (2018)
40. Namjoshi, K.S., Singhania, N.: Loopy: Programmable and formally verified loop transformations. In: International Static Analysis Symposium. pp. 383–402. Springer (2016)
41. Namjoshi, K.S., Xue, A.: A Self-certifying Compilation Framework for WebAssembly. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 127–148. Springer (2021)
42. The OpenCL 1.2 specification (2011)
43. Osama, M., Wijs, A.: Parallel SAT Simplification on GPU Architectures. In: TACAS, Part I. LNCS, vol. 11427, pp. 21–40. Springer (2019)
44. Osama, M., Wijs, A., Biere, A.: SAT Solving with GPU Accelerated Inprocessing. In: Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I. Lecture Notes in Computer Science, vol. 12651, pp. 133–151. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_8
45. de Putter, S., Wijs, A.: Verifying a verifier: on the formal correctness of an LTS transformation verification technique. In: International Conference on Fundamental Approaches to Software Engineering. pp. 383–400. Springer (2016)
46. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices **48**(6), 519–530 (2013)
47. Rocha, R.C., Pereira, A.D., Ramos, L., Góes, L.F.: Toast: Automatic tiling for iterative stencil computations on GPUs. Concurrency and Computation: Practice and Experience **29**(8), e4053 (2017)
48. Safari, M., Huisman, M.: Formal verification of parallel stream compaction and summed-area table algorithms. In: International Colloquium on Theoretical Aspects of Computing. pp. 181–199. Springer (2020)
49. Safari, M., Huisman, M.: A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In: International Conference on Integrated Formal Methods. pp. 257–275. Springer (2020)
50. Safari, M., Oortwijn, W., Huisman, M.: Automated verification of the parallel Bellman–Ford algorithm. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. (eds.) Static Analysis. pp. 346–358. Springer International Publishing, Cham (2021)
51. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: NASA Formal Methods Symposium. pp. 170–186. Springer (2020)

52. Şakar, O.: Extending support for axiomatic data types in vercors (April 2020), http://essay.utwente.nl/80892/
53. Shimobaba, T., Ito, T., Masuda, N., Ichihashi, Y., Takada, N.: Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL. Optics express **18**(10), 9955–9960 (2010)
54. Sundfeld, D., Havgaard, J.H., Gorodkin, J., De Melo, A.C.: CUDA-Sankoff: using GPU to accelerate the pairwise structural RNA alignment. In: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 295–302. IEEE (2017)
55. The CUDA team: Documentation of the CUDA unroll pragma (Accessed Oct 6, 2021), https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#pragma-unroll
56. The Halide team: Documentation of the Halide unroll function (Accessed Oct 6, 2021), https://halide-lang.org/docs/class_halide_1_1_func.html#a05935caceb6efb8badd85f306dd33034
57. The verification of tictactoe program, https://github.com/utwente-fmt/vercors/blob/0a2fdc24419466c2d3b7a853a2908c37e7a8daa7/examples/session-generate/MatrixGrid.pvl
58. Unkule, S., Shaltz, C., Qasem, A.: Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In: International Conference on Compiler Construction. pp. 21–40. Springer (2012)
59. Van Werkhoven, B., Maassen, J., Bal, H.E., Seinstra, F.J.: Optimizing convolution operations on GPUs using adaptive tiling. Future Generation Computer Systems **30**, 14–26 (2014)
60. Viper project website: (2016), http://www.pm.inf.ethz.ch/research/viper
61. Wahib, M., Maruyama, N.: Scalable kernel fusion for memory-bound GPU applications. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 191–202. IEEE (2014)
62. Wang, G., Lin, Y., Yi, W.: Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In: 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing. pp. 344–350. IEEE (2010)
63. Werkhoven, B.v.: Kernel Tuner: A search-optimizing GPU code auto-tuner. Future Generation Computer Systems **90**, 347–358 (2019)
64. Wienke, S., Springer, P., Terboven, C., Mey, D.: OpenACC - First Experiences with Real-World Applications. In: Proceedings of the 18th European Conference on Parallel and Distributed Computing (EuroPar). Lecture Notes in Computer Science, vol. 7484, pp. 859–870. Springer (2012). https://doi.org/10.1007/978-3-642-32820-6_85
65. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In: CAV, Part II. LNCS, vol. 9780, pp. 472–493. Springer (2016)
66. Wijs, A., Engelen, L.: REFINER: Towards Formal Verification of Model Transformations. In: NFM. LNCS, vol. 8430, pp. 258–263. Springer (2014)
67. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking. In: Proceedings of the 21st International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 9995, pp. 694–701. Springer (2016). https://doi.org/10.1007/978-3-319-48989-6_42
68. Wu, H., Diamos, G., Wang, J., Cadambi, S., Yalamanchili, S., Chakradhar, S.: Optimizing data warehousing applications for GPUs using kernel fusion/fission. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. pp. 2433–2442. IEEE (2012)

69. Xu, C., Kirk, S.R., Jenkins, S.: Tiling for performance tuning on different models of GPUs. In: 2009 Second International Symposium on Information Science and Engineering. pp. 500–504. IEEE (2009)
70. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. ACM Sigplan Notices **45**(6), 86–97 (2010)

Automatic Repair for Network Programs

Lei Shi[1]✉, Yuepeng Wang[2], Rajeev Alur[1], and Boon Thau Loo[1]

[1] University of Pennsylvania, Philadelphia, USA
[2] Simon Fraser University, Burnaby, Canada
{shilei,alur,boonloo}@seas.upenn.edu yuepeng@sfu.ca

Abstract. Debugging imperative network programs is a difficult task for operators as it requires understanding various network modules and complicated data structures. For this purpose, this paper presents an automated technique for repairing network programs with respect to unit tests. Given as input a faulty network program and a set of unit tests, our approach localizes the fault through symbolic reasoning, and synthesizes a patch ensuring that the repaired program passes all unit tests. It applies domain-specific abstraction to simplify network data structures and exploits function summary reuse for modular symbolic analysis. We have implemented the proposed techniques in a tool called NetRep and evaluated it on 10 benchmarks adapted from real-world software-defined network controllers. The evaluation results demonstrate the effectiveness and efficiency of NetRep for repairing network programs.

1 Introduction

Emerging tools for program synthesis and repair facilitate automation of programming tasks in various domains. For example, in the domain of end-user programming, synthesis techniques allow users without any programming experience to generate scripts from examples for extracting, wrangling, and manipulating data in spreadsheets [13,40]. In computer-aided education, repair techniques are capable of providing feedback on programming assignments to novice programmers and help them improve programming skills [49,14]. In software development, synthesis and repair techniques aim to reduce the manual efforts in various tasks, including code completion [43,10], application refactoring [42], program parallelization [8], bug detection [11,41], and patch generation [11,32].

As an emerging domain, Software-Defined Networking (SDN) offers the infrastructure for monitoring network status and managing network resources based on programmable software, replacing traditional specialized hardware in communication devices. Since SDN provides an opportunity to dynamically modify the traffic handling policies on programmable routers, this technology has witnessed growing industrial adoption. However, using SDNs involves many programming tasks that are inevitably susceptible to programmer errors leading to bugs [3,23]. For example, a device with incorrect routing policies could forward a packet to undesired destinations, and a buggy firewall rule may make the entire network system vulnerable to security threats.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 353–372, 2022.
https://doi.org/10.1007/978-3-030-99527-0_19

In the SDN framework, a logically centralized control plane generates rules that are installed into data planes, which in turn decides the routing of packets throughout the network. While network verification is a well-studied field where operators can be hinted on incorrectly installed rules [3,4,22], little prior work has explored the problem of automatically repairing the corresponding bug in the control plane, especially those written in widely used general-purpose languages such as Java or Python. Existing work mostly restricts the target to control plane programs written in domain-specific languages such as Datalog [51,17].

Since networks cannot tolerate even small mistakes, and most network operators are not trained in programming skills, debugging and repair tools in this domain should prioritize accuracy and automation. This means that many existing techniques for general program repair are not suitable to this domain as they trade off accuracy for heuristics for scaling with the size of analyzed programs and number of discovered potential bugs.

Motivated by the demand for automated repair and the limitations of existing techniques, we develop a precise and scalable program repair technique for network programs. Specifically, our repair technique takes as input a network program and a set of unit tests, reveals the program location that causes the test failure, and automatically generates a patch to fix the program. In the setting of SDN, a unit test corresponds to an incorrectly installed routing rule generated by the control plane from a reported packet. Such unit tests can be discovered by a separate network verification procedure [3,4,22].

Our main idea is to use symbolic reasoning using constraints capturing the semantics of the program for accurate repair, and modular analysis to improve the efficiency. We extended the encoding techniques from prior work [21,12] to support object-oriented features in Java. We also developed a new approach to focus the analysis on one function at a time and gradually narrow down the range of faulty statements along with the specification for the expected behavior.

The proposed technique is implemented in an automatic network program repair tool called NetRep. To evaluate NetRep, we adapt 10 benchmarks from real-world faulty network programs in Floodlight that require changing up to 3 lines of code to fix and apply NetRep to repair the benchmarks automatically. The experimental results show that NetRep is able to find a repair that passes all unit tests for faulty programs up to 738 lines of code for 8 benchmarks using 2 or 3 test cases, outperforming a state-of-the-art repair tool for general Java programs. Furthermore, NetRep is efficient in terms of repair time, requiring only an average running time of 744 seconds across all benchmarks.

Contributions. We make the following main contributions in this paper:

- We present an automated program repair technique that aims to help network operators debug and fix network controller programs automatically.
- We describe a bug localization approach based on symbolic execution and constraint solving for programs with imperative object-oriented features such as virtual function calls.
- We propose novel modular analysis techniques to effectively scale up the symbolic reasoning for automatic repair.

```
@network public class MacAddr {
  private long value;
  private MacAddr(long v) { value = v; }
  public static MacAddr NONE = new MacAddr(0);
  public static MacAddr of(long v) {return new MacAddr(v);}
  ... }
public class FirewallRule {
  public MacAddr dl_dst; public boolean any_dl_dst;
  public FirewallRule() {
    dl_dst = MacAddr.NONE; any_dl_dst = true; ... }
  public boolean isSameAs(FirewallRule r) {
    if (... || any_dl_dst != r.any_dl_dst
            || (any_dl_dst == false &&
                dl_dst != r.dl_dst)) {
        return false; }
    return true; }
  ... }
```

Fig. 1: Code snippet about a bug in Floodlight.

```
public boolean test(long mac1, long mac2) {
  FirewallRule r1 = new FirewallRule();
  r1.dl_dst = MacAddr.of(mac1); r1.any_dl_dst = false;
  FirewallRule r2 = new FirewallRule();
  r2.dl_dst = MacAddr.of(mac2); r2.any_dl_dst = false;
  return r1.isSameAs(r2); }
```

Fig. 2: Unit test that reveals the bug in FirewallRule.

– We develop a tool called NetRep based on the proposed techniques and evaluate it using 10 benchmarks adapted from real-world network programs. The evaluation results demonstrate that NetRep is effective for bug localization and able to generate correct patches for realistic network programs.

2 Overview

In this section, we give a high-level overview of our repair techniques and walk through the NetRep tool using an example adapted from the Floodlight SDN controller [9].

Figure 1 shows a simplified code snippet about firewall rules in Floodlight. Specifically, the program consists of two classes – FirewallRule and MacAddr. The FirewallRule class describes rules enforced by the firewall, including information about source and destination mac addresses. The MacAddr class is an auxiliary data structure that stores the raw value of mac addresses [3].

The network program shown in Figure 1 is problematic because the `isSameAs` function compares two mac addresses using the `!=` operator rather than a negation of the `equals` functions. The `!=` operator only compares two objects based on their memory addresses, whereas the intent of the developer is to check if two mac addresses have the same raw value. The bug is revealed by the unit test in Figure 2, then confirmed and fixed by the Floodlight developers [4]. Next, let

[3] A unique 48-bit number that identifies each network device.

[4] https://github.com/floodlight/floodlight/commit/4d528e4bf5f02c59347bb9c0beb1b875ba2c821e

us illustrate how NetRep localizes this bug based on unit tests `test(1, 2) = false` and `test(1, 1) = true` and automatically synthesizes a patch to fix it.

At a high level, NetRep enters a loop that iteratively attempts to find the fault location and synthesize the patch. Since our repair technique works in a modular fashion, NetRep first selects a function F in the program and tries to repair each possible fault location at a time. If NetRep cannot synthesize a patch consistent with the provided unit tests for any potential fault location in F, it backtracks and selects the next function and repeats the same process until all possible functions are checked. We now describe the experience of running NetRep on our illustrative example.

Iteration 1. NetRep selects the constructor of FirewallRule as the target function. Fault localization determines that the fault is located at the `dl_dst = MacAddr.NONE` part of Line 10, because it is related to the equality checking in the unit test. However, it is not the fault location. NetRep tries to synthesize a patch that passes all unit tests to replace this statement, but fails.

Iteration 2. NetRep selects the same function – constructor of FirewallRule, but the fault localization switches to a different statement `any_dl_dst = true` at Line 10. Similar to Iteration 1, the synthesizer cannot generate a correct patch by replacing this statement.

Iteration 3. Since none of the statements in the constructor is the fault location, NetRep now selects a different function: `isSameAs`. The fault localization determines that `any_dl_dst = false` at Line 13 may be the fault location as it may affect the testing results. However, having tried to replace the statement with many other candidate statements, e.g., `r.any_dl_dst = false`, `any_dl_dst = true`, the synthesizer still fails to generate the correct patch.

Last iteration. Finally, after several attempts to localize the fault, NetRep identifies the fault lies in `dl_dst != r.dl_dst` at Line 14, which is indeed the reported bug location. At this time, the synthesizer manages to generate a correct patch `!dl_dst.equals(r.dl_dst)`. Replacing the original condition at Line 14 with this patch results in a program that can pass all the provided test cases, so NetRep has successfully repaired the original faulty program.

3 Preliminaries

In this section, we present the language of network programs and describe a program formalism that is used in the rest of paper. We also define the program repair problem that we want to solve.

3.1 Language of Network Programs

The language of network programs considered in this paper is summarized in Figure 3. A network program consists of a set of classes, where each class has an optional annotation @network to denote that the class can benefit from network domain-specific abstraction.

$$
\begin{array}{ll}
\textit{Prog } \mathcal{P} ::= \mathcal{C}^+ & \textit{Stmt } s ::= l := e \mid \textbf{jmp } (e)\ L \\
\textit{Class } \mathcal{C} ::= \textbf{@network? class } C\ \{a^+\ F^+\} & \quad \mid \textbf{ ret } v \mid x := \textbf{new } C \\
\textit{Func } F ::= \textbf{function } f(x_1, \ldots, x_n)\ (L : s)^+ & \quad \mid x := C.f(v_1, \ldots, v_n) \\
\textit{Expr } e ::= l \mid c \mid op(e_1, \ldots, e_n) & \quad \mid x := y.f(v_1, \ldots, v_n) \\
\textit{LValue } l ::= x \mid x.a \mid x[v] & \textit{Imm } v ::= x \mid c
\end{array}
$$

$$
x, y \in \textbf{Variable} \quad c \in \textbf{Constant} \quad L \in \textbf{LineID}
$$
$$
C \in \textbf{ClassName} \quad f, f_0 \in \textbf{FuncName} \quad a \in \textbf{FieldName}
$$

Fig. 3: Syntax of network programs.

Each class in the program consists of a list of fields and functions. Each function has a name, a parameter list, and a function body. The function body is a list of statements, where each statement is labeled with its line number. Various kinds of statements are included in our language of network programs. Specifically, assign statement $l := e$ assigns expression e to left value l. Conditional jump statement **jmp** (e) L first evaluates predicate e. If the result is true, then the control flow jumps to line L; otherwise, it performs no operation. Note that our language does not have traditional if statements or loop statements, but those statements can be expressed using conditional jumps. [5]

Return statement **ret** v exits the current function with return value v. New statement $x :=$ **new** C creates an object of class C and assigns the object address to variable x. Static call $x := C.f(v_1, \ldots, v_n)$ invokes the static function f in class C with arguments $v_1, \ldots, v_n$ and assigns the return value to variable x. Similarly, virtual call $x := y.f(v_1, \ldots, v_n)$ invokes the virtual function f on *receiver* object y with arguments $v_1, \ldots, v_n$ and assigns the return value to variable x. Different kinds of expressions are supported including constants, variable accesses, field accesses, array accesses, arithmetic operations, and logical operations. Since the semantics of network programs is similar to that of traditional programs written in object-oriented languages, we omit the formal description of semantics.

In addition, we assume each statement in the program is labeled with a globally unique line number, and line numbers are consecutive within a function.

3.2 Problem Statement

We assume a unit test t is written in the form of a pair (I, O), where I is the input and O is the expected output. Given a network program $\mathcal{P}$ and a unit test $t = (I, O)$, we say $\mathcal{P}$ passes the test t if executing $\mathcal{P}$ on input I yields the expected output O, denoted by $[\![\mathcal{P}]\!]_I = O$. Otherwise, if $[\![\mathcal{P}]\!]_I \neq O$, we say $\mathcal{P}$ fails the test t. In general, given a network program $\mathcal{P}$ and a set of unit tests $\mathcal{E}$, program $\mathcal{P}$ is *faulty* modulo $\mathcal{E}$ if there exists a test $t \in \mathcal{E}$ such that $\mathcal{P}$ fails on t.

Now let us turn the attention to the meaning of fault locations and patches.

Definition 1 (Fault location and patch). *Let $\mathcal{P}$ be a program that is faulty modulo tests $\mathcal{E}$. Line L is called the* fault location *of $\mathcal{P}$, if there exists a statement*

[5] Our repair techniques only handle bounded loops. If there are unbounded loops in the network program, we need to perform loop unrolling.

Algorithm 1 Modular Program Repair

```
1: procedure Repair(P, E)
   Input: Program P, examples E
   Output: Repaired program P' or ⊥ to indicate failure
2:     P ← Abstraction(P);
3:     V ← {L ↦ false | L ∈ Lines(P)}; P' ← ⊥;
4:     while P' = ⊥ do
5:         F ← SelectFunction(P, V);
6:         if F = ⊥ then return ⊥;
7:         V, P' ← RepairFunction(P, F, E, V);
8:     return P';
9: procedure RepairFunction(P, F, E, V)
   Input: Program P, function F, examples E, visited map V
   Output: Updated visited map V, repaired program P'
10:    P' ← ⊥;
11:    while P' = ⊥ do
12:        L ← LocalizeFault(P, F, E, V);
13:        if L ≠ ⊥ then
14:            V ← V[L ↦ true];
15:        else
16:            V ← V[L' ↦ true | TransInFunc(L', P, F)];
17:        if L = ⊥ or IsCallStmt(P, L) then return V, ⊥;
18:        P' ← SynthesizePatch(P, E, F, L);
19:    return V, P';
```

s such that replacing line L of $\mathcal{P}$ *with s yields a new program that can pass all tests in* $\mathcal{E}$*. Here, the statement s is called a* patch *to* $\mathcal{P}$*.*

Problem statement. Given a network program $\mathcal{P}$ that is faulty modulo tests $\mathcal{E}$, our goal is to find a fault location L in $\mathcal{P}$ and generate the corresponding patch s, such that for any unit test $t \in \mathcal{E}$, the patched program $\mathcal{P}'$ can always pass the test t.

4 Modular Program Repair

In this section, we present our algorithm for automatically repairing network programs from a set of unit tests.

4.1 Algorithm Overview

The top-level repair algorithm is described in Algorithm 1. The Repair procedure takes as input a faulty network program $\mathcal{P}$ and unit tests $\mathcal{E}$ and produces as output a repaired program $\mathcal{P}'$ or $\perp$ to indicate repair failure.

At a high level, the Repair procedure maintains a visited map $\mathcal{V}$ from line numbers to boolean values, representing whether each line of $\mathcal{P}$ is checked or not.

The Repair procedure first applies the domain-specific abstraction to program $\mathcal{P}$ (Line 2) and initializes the visited map $\mathcal{V}$ by setting every line in $\mathcal{P}$ as not checked (Line 3). Next, it tries to iteratively repair $\mathcal{P}$ in a modular way until it finds a program $\mathcal{P}'$ that is not faulty modulo tests $\mathcal{E}$ (Lines 4 – 8). In particular, the Repair procedure invokes SelectFunction to choose a function F as the target of repair (Line 5). If none of the functions in $\mathcal{P}$ can be repaired, it returns $\perp$ to indicate that the repair procedure failed (Line 6). Otherwise, it invokes the RepairFunction procedure (Line 7) to enter the localization-synthesis loop inside the target function F.

In addition to the program $\mathcal{P}$ and tests $\mathcal{E}$, the RepairFunction procedure takes as input a target function F and the current visited map $\mathcal{V}$. It produces as output the updated version of the visited map $\mathcal{V}$, as well as a repaired program $\mathcal{P}'$ or $\perp$ to indicate that the function F cannot be repaired. As shown in Lines 11 – 18 of Algorithm 1, RepairFunction alternatively invokes sub-procedures LocalizeFault and SynthesizePatch to repair the target function. In particular, the goal of LocalizeFault is to identify a fault location in function F. If LocalizeFault manages to find a fault location L in F, then line L is marked as visited (Line 14). Otherwise, if LocalizeFault returns $\perp$, it means function F and all functions transitively invoked in F are correct or not repairable. In this case, all lines in F and its transitive callees are marked as checked (Line 16). Furthermore, if the identified fault location L corresponds to a statement that invokes F', it means the fault location is inside F'. Thus, RepairFunction directly returns $\perp$ (Line 17) and SelectFunction will choose F' as the target function in the next iteration. On the other hand, the goal of the sub-procedure SynthesizePatch is generating a patch for function F given the fault location L. If SynthesizePatch successfully synthesizes a patch and produces a non-faulty program $\mathcal{P}'$, then the entire procedure succeeds with repaired program $\mathcal{P}'$. Otherwise, RepairFunction backtracks with a new program location and repeat the same process.

In the rest of this section, we explain fault localization, modular analysis, and patch synthesis in more detail.

4.2 Fault Localization

Next, we give a high-level description of our fault localization technique that aims to find the fault location in a given program. This corresponds to the LocalizeFault procedure in Algorithm 1. We will first show how to encode the problem on an entire program, and then explain how the analysis can be made modular to boost the performance.

At a high level, our fault localization technique uses a symbolic approach by reducing the fault localization problem into a constraint solving problem. In particular, we introduce a boolean variable for each line L, denoted by $\mathcal{B}[L]$, and encode the fault localization problem as an SMT formula, such that the value of the variable $\mathcal{B}[L]$ indicates whether line L is correct or not.

Checking faulty programs. To understand how to encode the fault localization problem, let us first explain how to encode the consistency check given a

program $\mathcal{P}$ and a test case $t = (I, O)$. Specifically, the encoded SMT formula $\Phi(t)$ consists of three components:

1. *Semantic constraints.* For each line $L_i : s_i$, we generate a formula $\Phi_i(S, S')$ to describe the semantics of the statement s_i. Specifically, given a state S that holds before statement s_i, $\Phi_i(S, S')$ is valid if S' is the state after executing s_i. There are two parts of the constraint: the memory contents that are changed, and the memory contents that are preserved. For example, in case of an assignment statement, the constraint will claim that 1) the evaluation result of the right value in state S equals to the left value in state S', and 2) all values except for the left value are the same in S and S'.
2. *Control flow integrity constraints.* In order to ensure all traces satisfying the constraint faithfully follow the control flow structure of a given program $\mathcal{P}$, we generate another set of formulae Φ_f. Specifically, we require that any line of code that is executed must have exactly one predecessor and one successor that are executed, and the branch condition in the code must be respected when picking the successor. This guarantees that there is exactly one valid execution trace corresponding to one test case,
3. *Consistency between program and test.* For the provided test case $t = (I, O)$, we also generate formula $\Phi_{in}(S_0, I)$ and $\Phi_{out}(S_n, O)$ to ensure the program behavior is consistent with the test. In particular, $\Phi_{in}(S_0, I)$ binds input I to the initial state S_0 and $\Phi_{out}(S_n, O)$ describes the connection between output O and final state S_n.

The satisfiability of formula $\Phi(t)$ indicates the result of consistency check. If $\Phi(t)$ is satisfiable, the solver generates a feasible execution trace and an assignment of all intermediate states along this trace. In this case, program $\mathcal{P}$ can pass the test t because there exists a valid trace following the control flow and every pair of adjacent states in the trace is consistent with the semantics of the corresponding statement. Otherwise, if $\Phi(t)$ is unsatisfiable, $\mathcal{P}$ fails the test t.

Now to check whether $\mathcal{P}$ against a set of unit tests $\mathcal{E}$, we can conjoin the formula $\Phi(t_j)$ for each unit test $t_j \in \mathcal{E}$ and obtain the conjunction $\Phi = \bigwedge_{t_j \in \mathcal{E}} \Phi(t_j)$. The satisfiability of formula Φ indicates whether $\mathcal{P}$ is faulty modulo tests $\mathcal{E}$ [6].

Methodology of fault localization. Let $\mathcal{P}$ be a faulty program modulo $\mathcal{E}$, we know the corresponding formula Φ for consistency check is unsatisfiable. Suppose the fault location is line L_i, one key insight is that replacing the semantic constraint $\Phi_i(S, S')$ with *true* yields a satisfiable formula. This is because *true* does not enforce any constraint between the pre-state S and post-state S', so a previously invalid trace caused by the bug at L becomes valid now.

Based on this insight, we develop a methodology to find the fault location using symbolic reasoning. Specifically, given a consistency check formula Φ, we can obtain a fault localization formula Φ' by replacing the semantic constraint $\Phi_i(S, S')$ with $\mathcal{B}[L_i] \rightarrow \Phi_i(S, S')$ for every line $L_i, i \in [1, n]$. Here, variable $\mathcal{B}[L_i]$ decides whether or not it turns the semantic constraint of L_i into *true*. Thus, $\mathcal{B}[L_i] = \mathit{false}$ indicates L_i is a fault location.

[6] The encoding is described in more detail in the extended version [46].

One hiccup here is that formula Φ' is always satisfiable and a model of Φ' can simply assign $\mathcal{B}[L_i] = \mathit{false}$ for all L_i. It means all lines in the program are fault locations, which is not useful for fault localization. To address this issue, we can add a cardinality constraint stating there are exactly K variables in map $\mathcal{B}$ that can be assigned to *false*, which forces the constraint solver to find exactly K fault locations in program $\mathcal{P}$.

Modular analysis. The method above can precisely compute a potential fault location. But an obvious shortcoming is it is hard to scale. Encoding a long program involves 1) a large number of semantic constraints, 2) many fault location choices, as well as 3) many intermediate states to be assigned.

Notice that although a program can be arbitrarily long, developers usually follow the design practice that every function is of limited size. Focusing on analyzing one function at a time and recursively search for the final fault location could be way more efficient than solving one NP-hard problem at the entire program's scale.

To facilitate modular analysis of a function, we need to summarize the behavior of its sub-modules (callee functions) and infer external specification from its higher-level module (caller function).

The encoding method introduced above treats one line of code as a constraint on its pre-state and post-state. To summarize the behavior of a callee function, we aim to turn it into a similar constraint on the pre-state and post-state for the calling statement. The inner states of this callee function should be skipped in the encoding. We can compute such summaries of the target function's callees by symbolic execution. We start with a symbolic representation of the pre-state and execute the callee function until it returns, and claim that the output state equals the post-state. In this way, we can entirely eliminate all bug location choices and inner state assignments in the callee function, as well as greatly simplifying the semantic constraint.

There are two ways to infer the specification of target function. The first way is to encode only the calling stack of the target function up until the top-level function, where we can use the test case as the specification. All function calls made by the target's caller and transitive callers that are not in the stack can be replaced by the automatically computed summary. We can also disable all fault location choices except for lines in the target function. Another way is to infer a possible pre-condition and post-condition of the target function. From the perspective of the caller, the target function is a line of code that puts an incorrect constraint on its pre-state and post-state. After the analysis, the constraint solver will infer a feasible pre-state and post-state assuming this incorrect constraint is removed. This assignment can be used as the pre-condition and post-condition, which eliminates the need to encode any caller function. Since the second approach will possibly introduce incompleteness into the analysis, we use it only to infer a specification to synthesize the final patch, and use the first one for every function's analysis.

Domain-specific abstraction. A domain-specific abstraction is essentially a function summary as discussed above. But for those repeatedly used network

classes (identified by the @network annotation), we can pre-define some more succinct abstractions based on domain knowledge to make the analysis easier. The abstraction $\mathcal{A}[F]$ of a function F is an over-approximation of F that is precise enough to characterize the behavior of F.

The abstraction is useful due to two observations. First, source code for network programs may only be partially available due to the use of high-level interface and native implementation. For example, when comparing the equality between two network addresses, the getClass function is frequently used, but its implementation depends on the runtime and is not available. To make the analysis easier, we can instead use the following abstraction for such comparison:

$$\mathcal{A}[\text{equals}] : \lambda x.\ \lambda y.\ (x.dtype = y.dtype \wedge x.value = y.value),$$

where $x.dtype$ denotes the dynamic type of the object x.

Second, network programs have complex operations that are challenging for symbolic reasoning. For instance, bit manipulations are heavily used in network data structures. While bit manipulations can improve the performance of network programs, they present significant challenges for symbolic analysis due to the encoding in the theory of bitvectors. We can give an abstraction equivalent in correctness but simpler in the behavior, e.g., using the identity function instead of a hash code computation.

4.3 Patch Synthesis

The last step of our repair algorithm is to generate a patch to fix the faulty program. This corresponds to the SynthesizePatch procedure in Algorithm 1. It can be reduced to a sketch finishing problem in program synthesis where we replace the existing faulty line with a hole.

Our general idea is to use plain enumerative search with a depth bound in the candidate patch's space, but with two significant optimizations.

First, we reduce the search space with heuristics. On one hand, we only replace the core expression in the faulty statement with a hole to focus on the most expressive part. To be specific, we consider changing the right-hand-side expressions of assignments, conditional expressions of jump statements, return values of return statements, and functions and arguments for function invocations. On the other hand, we use a limited grammar to guide the search. We parameterize all constants, variables, fields, functions, and operators over the sketch and only instantiate constructs that are in scope. For example, given a particular sketch with a hole, we only populate the variable set with all local and global variables that are in scope of the hole. Also, if the hole corresponds to the conditional expression of a if statement, we only add logical operators to the grammar.

Second, we use the local specification to guide the synthesis. Sketch completion is different from synthesizing a complete program in that the specification is defined for the entire program. We have to repeatedly waste time on executing the correct part of the program to verify a candidate patch. We use the technique described in the modular analysis section to generate a pre-condition and post-condition for only the faulty line. In this way, only the generated patch

needs to be executed to verify against the specification, which greatly saves time when the program grows larger.

5 Implementation

We have implemented the proposed repair technique in a tool called NetRep. NetRep leverages the Soot static analysis framework [26] to convert Java programs into Jimple code, which provides a succinct yet expressive set of instructions for analysis. In addition, NetRep utilizes the Rosette tool [48] to perform symbolic reasoning for fault localization and patch synthesis. While our implementation closely follows the algorithm presented in Section 4, we also conduct several optimizations important to improve the performance of NetRep.

Memories for different types. Since the conversion between bitvectors and integers imposes significant overhead on running time, NetRep divides the memory into one part for integers and another for bitvectors. In this design, NetRep automatically selects the memory chunk based on the variable types. The type checking can guarantee that no such conversion will exist.

Stack and heap. In order to reduce the number of memory operations, NetRep also divides the memory into stack and heap. As is standard, stack only stores static data and its layout is deterministic. Therefore, stacks are implemented using fixed-size vectors, and thus can be efficiently accessed for read and write operations. On the other hand, heap stores dynamic data that are usually not known at compile time, such as allocated objects. Since the heap size cannot be determined beforehand, NetRep uses an uninterpreted function $f(x)$ to represent heaps, where x is the address and $f(x)$ is the value stored at x.

String values. Since reasoning over string values is a challenging task and not always necessary for repairing network programs, we simplified the representation of strings with integer values. Specifically, NetRep maps each string literal to a unique integer and represents all string operations (e.g. concatenation) with uninterpreted functions.

Bounded program analysis. In order to improve the repair time, NetRep only performs bounded program analysis for fault localization and patch synthesis. Namely, we unroll loops and inline functions up to K times, where K is a predefined hyper-parameter. In this way, function summaries can be easily and efficiently computed using symbolic execution.

6 Evaluation

To evaluate the proposed techniques, we perform experiments that are designed to answer the following research questions:

RQ1 Is NetRep effective to repair realistic network programs?
RQ2 How efficient are the fault localization and repair techniques in NetRep?
RQ3 How helpful are modular analysis and domain-specific abstraction for repairing network programs?

ID	Module	LOC	# Funcs	# Tests	Succ	Exp	Loc Time (s)	Synth Time (s)	Total Time (s)
1	DHCP	212	17	2	Yes	Yes	40	117	157
2	Load Balancer	336	28	2	No	No	-	-	-
3	Firewall	262	13	2	Yes	Yes	893	197	1090
4	DHCP	431	32	2	Yes	Yes	95	39	134
5	Utility	809	65	2	No	No	-	-	-
6	Routing	605	44	3	Yes	Yes	271	179	450
7	Utility	454	45	2	Yes	Yes	39	46	85
8	Learning Switch	738	34	2	Yes	No	571	595	1166
9	Database	442	17	2	Yes	No	310	2139	2449
10	Link Discovery	671	46	2	Yes	No	268	158	426

Table 1: Experimental results of NetRep.

RQ4 How is NetRep compared to other repair tools for Java programs?

Benchmark collection. To obtain realistic benchmarks, we crawl the commit history of Floodlight [9], a representative open-source SDN controller in Java that supports the OpenFlow protocol and a rich set of network functions. To distinguish commits caused by bug repairs from those generated for non-repair scenarios, we identify commits based on the following criteria: 1) The commit message contains keywords about repairing bugs, e.g., "bug", "error", "fix"; 2) The commit changes no more than three lines of code.

Following these criteria, we have collected 10 commits from the Floodlight repository and adapted them into our benchmarks. Specifically, given a commit in the repository, we take the code before the commit as the faulty network program and the version after the commit as the ground-truth repaired program. The code is post-processed and the parts irrelevant to the bug of interest are removed. We also identify corresponding unit tests and modify them to directly reveal the bug as appropriate. Each benchmark in our evaluation consists of a faulty network program and its corresponding unit tests.

Experimental setup. All experiments are conducted on a computer with 4-core 2.80GHz CPU and 16GB of physical memory, running the Arch Linux Operating system. We use Racket v7.7 as the compiler and runtime system of NetRep and set a time limit of 1 hour for each benchmark.

6.1 Main Results

Our main experimental results are summarized in Table 1. The column labeled "Module" describes the network module to which the benchmark belongs. The next two columns labeled "LOC" and "# Funcs" show the number of lines of source code (in Jimple) and the number of functions, respectively. The "# Tests" column presents the number of unit tests used for fault localization and patch synthesis. Next, the "Succ" and "Exp" columns show whether NetRep can successfully repair the program and if the generated patch is exactly the same as the ground-truth. Since NetRep returns the first fix that can pass all provided test cases, the repaired programs are not necessarily the same as those

expected in the ground-truth. In this case, the table will show a "Yes" in the "Succ" column and a "No" in the "Exp" column. Finally, the last three columns in Table 1 denote the fault localization time, patch synthesis time and the total running time of NetRep.

As shown in Table 1, there is a range of 13 to 65 functions in each benchmark and the average number of functions is 34 across all benchmarks. Each benchmark has 212 – 809 lines of Jimple code, with the average being 496. NetRep succeeds in repairing 8 out of 10 benchmarks. Furthermore, for 5 benchmarks that can be successfully repaired, NetRep is able to generate exactly the same fix as ground-truth. Given that our benchmarks cover programs from a variety of modules of Floodlight, such as DHCP Server, Firewall, etc, we believe that NetRep is effective to repair realistic network programs (RQ1).

We inspected the reason why NetRep fails to repair benchmarks 2 and 5. NetRep is not able to localize the fault in benchmark 2 due to its incomplete support for unbounded data structures with dynamic allocation such as hash map. For Benchmark 5, NetRep is able to localize the fault but not able to synthesize the correct patch. This is because the expected function to be invoked has side effects with another function, which needs some improvements in the specification checking to verify.

Regarding the efficiency, NetRep can repair 8 benchmarks in an average of 744 seconds with only 2 to 3 test cases. The fault localization time ranges from 39 seconds to 893 seconds, with 50% of the benchmarks within five minutes. The patch synthesis time ranges from 39 seconds to 2139 seconds, with 60% of the benchmarks within five minutes. In summary, the evaluation results show that NetRep only takes minutes to localize bugs in a faulty program and synthesize a correct patch based on two to three unit tests (RQ2).

6.2 Ablation Study

To explore the impact of modular analysis and domain-specific abstraction on the proposed repair technique, we develop three variants of NetRep:

- NetRep-NoMod is a variant of NetRep without modular analysis. Specifically, NetRep-NoMod inlines the functions in a given program but still uses abstractions for network data structures for fault localization and patch synthesis.
- NetRep-NoAbs is a variant of NetRep without domain-specific abstraction. In particular, NetRep-NoAbs uses the original concrete implementation of network functions for symbolic reasoning. If the implementation is written in a different language, we manually translate the implementation to Java.
- NetRep-NoModAbs is a variant of NetRep without modular analysis or domain-specific abstraction. NetRep-NoModAbs simply inlines all functions in the faulty program, including those in the network data structures, and performs symbolic analysis for fault localization and patch synthesis.

To understand the impact of modular analysis and domain-specific abstraction, we run all variants on the 10 collected benchmarks. For each variant, we

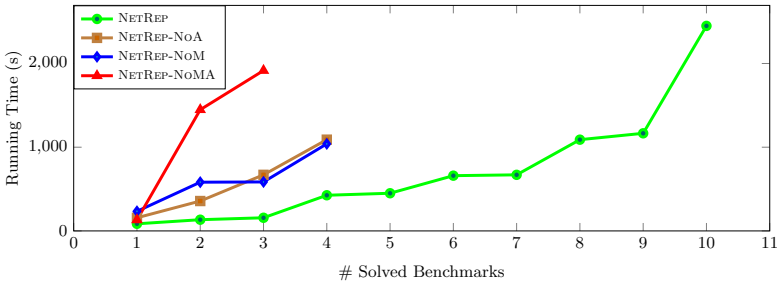


Fig. 4: Comparing NetRep against three variants.

measure the total running time (including time for fault localization and time for patch synthesis) on each benchmark, and order the results by running time in increasing order. The results for all variants are depicted in Figure 4. All lines stop at the last benchmark that the corresponding variant can solve within 1 hour time limit.

As shown in Figure 4, both NetRep-NoAbs and NetRep-NoMod can only solve 4 out of 10 benchmarks in the evaluation, with the average running time being 569 seconds and 610 seconds, respectively. NetRep-NoModAbs solves the least number of benchmarks: 3 out of 10. For the ones that it can solve, the average running time is 1165 seconds. This experiment shows that modular analysis and domain-specific abstraction are both great boost to NetRep's efficiency to repair network programs (RQ3).

6.3 Comparison with the Baseline

To understand how NetRep performs compared to other Java program repair tools, we compare NetRep against a state-of-the-art tool called Jaid [5] on our benchmarks. Specifically, Jaid takes as input a faulty Java program, a set of unit tests, and a function signature for fault localization and patch synthesis, a setting closest to NetRep among a variety of tools. Note that Jaid solves a simpler repair problem than NetRep, because it requires the user to specify a function that is potentially incorrect in the program, whereas NetRep does not need input other than the faulty program and unit tests. In order to run Jaid on our benchmarks, we adjust their formats to fit Jaid's and provide the faulty function (known from the ground truth) as input for Jaid.

Jaid will indefinitely enumerate all possible patches, rather than recommending a most correct one. We think it is successful if the expected patch can be found among the results. In practice, human assistance is needed to pick out this patch from the thousands of candidates.

As a result, Jaid is able to finish on 8 out of 10 benchmarks. The expected patches are found among 2 of them, whereas NetRep can give the expected result for 5 benchmarks on the first recommendation. For one benchmark, Jaid is unable to fix. For another one, it runs out of memory.

We argue that NetRep is better suited for automatically repairing network programs compared to Jaid. First, it only requires network operators to provide

unit test cases. As is discussed above, they can be automatically discovered by another verification or testing procedure. In comparison, Jaid requires users to have skill of programming network controllers to identify the buggy function and pick the correct patch from the results. This is beyond the ability of most network operators and starts to require an expert team. Second NetRep has higher repairing accuracy. As we discussed above, network is sensitive to small mistakes. High accuracy is crucial for a network to function correctly.

In summary, NetRep is more effective in automatically fixing bugs in network programs compared to state-of-the-art repairing tools for Java programs, especially with respect to repairing accuracy and automation (RQ4).

7 Related Work

Automated program repair. Automated program repair is an active research area that aims to automatically fix the mistakes in programs based on specifications of correctness criteria [11,28,39,18], with a variety of applications such as aiding software development [34], finding security vulnerabilities [37], and teaching novice programmers [49,14]. Different techniques have been proposed to solve the automated program repair problem, including heuristics-based techniques [16,31], semantics-based techniques [37,27], and learning-based techniques [45,30,32,47]. NetRep is a semantics-based automated repair tool. Different from prior work, NetRep is specialized to repair network programs based on modular analysis and network data structure abstractions.

Fault localization. Researchers have developed various approaches to fault localization, including spectrum-based, learning-based, and constraint-based techniques. Specifically, the spectrum-based techniques [27,1,2,7,44,6,19] perform fault localization by identifying which part of program is active during a run through execution profiles (called program spectrum). Learning-based techniques [29,53,54] typically train machine learning models to predict and rank possible fault locations. By contrast, constraint-based techniques [21,20,12] encode the semantics of problems as logical constraints and reduce the fault localization problem into constraint satisfaction problem. In spirit, NetRep uses a similar idea for fault localization. However, NetRep performs modular analysis and enables debugging programs involving object-oriented features, whereas prior work only analyzes the entire program in a C-like language. Besides, NetRep reuses the fault localization result to speedup the patch synthesis while prior work mainly focuses on the fault localization step.

Patch synthesis. Many synthesis algorithms have been developed for generating patches, including enumerative search [27], constraint-based techniques [37], statistical model [52], machine learning [15], hint from existing code [25], and so on. In terms of patch synthesis, NetRep generates a context-free grammar from the context of fault locations and performs enumerative search based on the grammar to synthesize patches. It does not require machine learning model or statistical information for ranking all possible patches. However, it is conceivable that NetRep will benefit from the guidance of such ranking techniques.

Verification and synthesis for SDN. In the networking domain, several verification tools [3,33,23,24] have been proposed based on either model checking or theorem proving. For example, VeriCon [3] performs deductive verification to verify the correctness of SDN programs specified by network-wide invariants on all admissible topologies. In addition to verification, synthesis techniques [36,35,38] have also been proposed to aid software-defined networking. NetRep aims to repair network programs automatically, which is a different problem than SDN verification or synthesis.

Repair for network programs. Our work is most related to automated repair of network programs in the SDN domain [50,51,17]. Prior work about autorepair [50,51] relies on using Datalog to capture the operational semantics of the target language to be repaired. The repair techniques work for domain-specific languages (e.g. Datalog or Ruby on Rails) with simple structure. Similarly, Hojjat et al. [17] propose a framework based on horn clause repair problem to help network operators fix faulty configurations. However, NetRep targets Java network programs with object-oriented features and more complex constructs, which cannot be handled by existing techniques.

8 Limitations and Future Work

We discuss several limitations of NetRep that we plan to improve in future work. First, NetRep repairs the faulty network program with the first correct patch that can pass all tests. A user interaction that resumes the synthesis can be introduced in case it is not intended by the user or a more formal specification.

Second, patches that require complicated changes, e.g., those involving control flow structures, are beyond NetRep's ability. They make up 44% of our collection of bug-fixing commits. We envision that the challenge can be addressed by introducing more sophisticated patch synthesis techniques such as searching over a domain-specific language for edits.

Third, in order to force symbolic execution to terminate in finite time, NetRep currently unrolls all loops in the network program, which may result in missing a potential bug. Loop invariant inference techniques can be leveraged to overcome this challenge and still guarantee termination.

9 Conclusion

In this paper, we have proposed an automated repair technique for network controller programs with unit tests as specifications. Our technique internally performs symbolic reasoning for bug localization and patch synthesis, optimized by network domain-specific abstractions and modular analysis to reduce encoding size. we have implemented a tool called NetRep and evaluated it on 10 benchmarks adapted from the Floodlight framework. The experimental results demonstrate that NetRep is effective for repairing realistic network programs with moderate change sizes.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 88–99. IEEE Computer Society (2009)
2. Abreu, R., Zoeteweij, P., van Gemund, A.J.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. pp. 89–98 (2007)
3. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: towards verifying controller programs in software-defined networks. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 282–293. ACM (2014)
4. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: A general approach to network configuration verification. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. pp. 155–168 (2017)
5. Chen, L., Pei, Y., Furia, C.A.: Contract-based program repair without the contracts. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 637–647. IEEE Computer Society (2017)
6. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.A.: Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN). pp. 595–604. IEEE Computer Society (2002)
7. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight defect localization for java. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 3586, pp. 528–550. Springer (2005)
8. Fedyukovich, G., Ahmad, M.B.S., Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 572–585. ACM (2017)
9. Floodlight: https://github.com/floodlight/floodlight (2021)
10. Galenson, J., Reames, P., Bodík, R., Hartmann, B., Sen, K.: Codehint: dynamic and interactive synthesis of code snippets. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) Proceedings of the International Conference on Software Engineering (ICSE). pp. 653–663. ACM (2014)
11. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)
12. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to c. In: International Conference on Computer Aided Verification. pp. 358–371. Springer (2006)
13. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Ball, T., Sagiv, M. (eds.) Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 317–330. ACM (2011)
14. Gulwani, S., Radicek, I., Zuleger, F.: Automated clustering and program repair for introductory programming assignments. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 465–480. ACM (2018)
15. Gupta, R., Pal, S., Kanade, A., Shevade, S.K.: Deepfix: Fixing common C language errors by deep learning. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of

the Thirty-First AAAI Conference on Artificial Intelligence. pp. 1345–1351. AAAI Press (2017)
16. Harman, M.: Automated patching techniques: the fix is in: technical perspective. Commun. ACM **53**(5), 108 (2010)
17. Hojjat, H., Rümmer, P., McClurg, J., Cerný, P., Foster, N.: Optimizing horn solvers for network repair. In: Piskac, R., Talupur, M. (eds.) Proceedings of the Formal Methods in Computer-Aided Design (FMCAD). pp. 73–80. IEEE (2016)
18. Hong, S., Lee, J., Lee, J., Oh, H.: SAVER: scalable, precise, and safe memory-error repair. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 271–283. ACM (2020)
19. Jones, J.A., Harrold, M.J., Stasko, J.T.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA. pp. 467–477. ACM (2002)
20. Jose, M., Majumdar, R.: Bug-assist: Assisting fault localization in ANSI-C programs. In: Proceedings of International Conference on Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 504–509. Springer (2011)
21. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 437–446. ACM (2011)
22. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). pp. 113–126 (2012)
23. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: Veriflow: Verifying network-wide invariants in real time. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 15–27. USENIX Association (2013)
24. Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., Clark, R.J.: Kinetic: Verifiable dynamic network control. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 59–72. USENIX Association (2015)
25. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: International Conference on Computer Aided Verification. pp. 217–233. Springer (2015)
26. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop. vol. 15 (2011)
27. Le, X.D., Chu, D., Lo, D., Goues, C.L., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the Joint Meeting on Foundations of Software Engineering, (ESEC/FSE). pp. 593–604. ACM (2017)
28. Li, G., Liu, H., Chen, X., Gunawi, H.S., Lu, S.: Dfix: automatically fixing timing bugs in distributed systems. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 994–1009. ACM (2019)
29. Li, X., Li, W., Zhang, Y., Zhang, L.: Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In: Zhang, D., Møller, A. (eds.) Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). pp. 169–180. ACM (2019)
30. Li, Y., Wang, S., Nguyen, T.N.: Dlfix: context-based code transformation learning for automated program repair. In: Proceedings of International Conference on Software Engineering (ICSE). pp. 602–614. ACM (2020)

31. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE). pp. 166–178. ACM (2015)
32. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proceedings of the Symposium on Principles of Programming Languages (POPL). pp. 298–312. ACM (2016)
33. Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 499–512. USENIX Association (2015)
34. Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., Scott, A.: Sapfix: automated end-to-end repair at scale. In: Proceedings of the International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP). pp. 269–278. IEEE / ACM (2019)
35. McClurg, J., Hojjat, H., Cerný, P.: Synchronization synthesis for network programs. In: Majumdar, R., Kuncak, V. (eds.) Proceedings of the International conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10427, pp. 301–321. Springer (2017)
36. McClurg, J., Hojjat, H., Cerný, P., Foster, N.: Efficient synthesis of network updates. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 196–207. ACM (2015)
37. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 691–701. ACM (2016)
38. Padon, O., Immerman, N., Karbyshev, A., Lahav, O., Sagiv, M., Shoham, S.: Decentralizing SDN policies. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 663–676. ACM (2015)
39. Perry, D.M., Kim, D., Samanta, R., Zhang, X.: Semcluster: clustering of imperative programming assignments based on quantitative semantic features. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 860–873. ACM (2019)
40. Polozov, O., Gulwani, S.: Flashmeta: a framework for inductive program synthesis. In: Aldrich, J., Eugster, P. (eds.) Proceedings of theACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA). pp. 107–126. ACM (2015)
41. Pradel, M., Sen, K.: Deepbugs: a learning approach to name-based bug detection. Proc. ACM Program. Lang. **2**(OOPSLA), 147:1–147:25 (2018)
42. Raychev, V., Schäfer, M., Sridharan, M., Vechev, M.T.: Refactoring with synthesis. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, (OOPSLA). pp. 339–354. ACM (2013)
43. Raychev, V., Vechev, M.T., Yahav, E.: Code completion with statistical language models. In: O'Boyle, M.F.P., Pingali, K. (eds.) Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 419–428. ACM (2014)
44. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of the IEEE International Conference on Automated Software Engineering (ASE). pp. 30–39. IEEE Computer Society (2003)

45. Sakkas, G., Endres, M., Cosman, B., Weimer, W., Jhala, R.: Type error feedback via analytic program repair. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI). pp. 16–30. ACM (2020)
46. Shi, L., Wang, Y., Alur, R., Loo, B.T.: NetRep: Automatic repair for network programs. https://arxiv.org/abs/2110.06303 (2021)
47. Sidiroglou-Douskos, S., Lahtinen, E., Long, F., Rinard, M.: Automatic error elimination by horizontal code transfer across multiple applications. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 43–54. ACM (2015)
48. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 530–541. ACM (2014)
49. Wang, K., Singh, R., Su, Z.: Search, align, and repair: data-driven feedback generation for introductory programming exercises. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). pp. 481–495. ACM (2018)
50. Wu, Y., Chen, A., Haeberlen, A., Zhou, W., Loo, B.T.: Automated network repair with meta provenance. In: Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets). pp. 26:1–26:7. ACM (2015)
51. Wu, Y., Chen, A., Haeberlen, A., Zhou, W., Loo, B.T.: Automated bug removal for software-defined networks. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 719–733. USENIX Association (2017)
52. Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 416–426. IEEE / ACM (2017)
53. Xuan, J., Monperrus, M.: Learning to combine multiple ranking metrics for fault localization. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 191–200. IEEE Computer Society (2014)
54. Zhang, Z., Lei, Y., Tan, Q., Mao, X., Zeng, P., Chang, X.: Deep learning-based fault localization with contextual information. IEICE Trans. Inf. Syst. **100-D**(12), 3027–3031 (2017)

11th Competition on Software Verification: SV-COMP 2022


Progress on Software Verification: SV-COMP 2022

Dirk Beyer ✉

LMU Munich, Munich, Germany

Abstract. The 11th edition of the Competition on Software Verification (SV-COMP 2022) provides the largest ever overview of tools for software verification. The competition is an annual comparative evaluation of fully automatic software verifiers for C and Java programs. The objective is to provide an overview of the state of the art in terms of effectiveness and efficiency of software verification, establish standards, provide a platform for exchange to developers of such tools, educate PhD students on reproducibility approaches and benchmarking, and provide computing resources to developers that do not have access to compute clusters. The competition consisted of 15 648 verification tasks for C programs and 586 verification tasks for Java programs. Each verification task consisted of a program and a property (reachability, memory safety, overflows, termination). The new category on data-race detection was introduced as demonstration category. SV-COMP 2022 had 47 participating verification systems from 33 teams from 11 countries.

Keywords: Formal Verification · Program Analysis · Competition · Software Verification · Verification Tasks · Benchmark · C Language · Java Language · SV-Benchmarks · BenchExec · CoVeriTeam

1 Introduction

This report is the 2022 edition of the series of competition reports (see footnote) that accompanies the competition, by explaining the process and rules, giving insights into some aspects of the competition (this time the focus is on trouble shooting and reproducing results on a small scale), and, most importantly, reporting the results of the comparative evaluation. The 11th Competition on Software Verification (SV-COMP, `https://sv-comp.sosy-lab.org/2022`) is the largest comparative evaluation ever in this area. The objectives of the competitions were discussed earlier (1-4 [16]) and extended over the years (5-6 [17]):

1. provide an overview of the state of the art in software-verification technology and increase visibility of the most recent software verifiers,

This report extends previous reports on SV-COMP [10, 11, 12, 13, 14, 15, 16, 17, 18].
Reproduction packages are available on Zenodo (see Table 4).

✉ `dirk.beyer@sosy-lab.org`

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 375–402, 2022.
https://doi.org/10.1007/978-3-030-99527-0_20

2. establish a repository of software-verification tasks that is publicly available for free use as standard benchmark suite for evaluating verification software,
3. establish standards that make it possible to compare different verification tools, including a property language and formats for the results,
4. accelerate the transfer of new verification technology to industrial practice by identifying the strengths of the various verifiers on a diverse set of tasks,
5. educate PhD students and others on performing reproducible benchmarking, packaging tools, and running robust and accurate research experiments, and
6. provide research teams that do not have sufficient computing resources with the opportunity to obtain experimental results on large benchmark sets.

The SV-COMP 2020 report [17] discusses the achievements of the SV-COMP competition so far with respect to these objectives.

Related Competitions. There are many competitions in the area of formal methods [9], because it is well-understood that competitions are a fair and accurate means to execute a comparative evaluation with involvement of the developing teams. We refer to a previous report [17] for a more detailed discussion and give here only the references to the most related competitions [20, 53, 67].

Quick Summary of Changes. While we try to keep the setup of the competition stable, there are always improvements and developments. For the 2022 edition, the following changes were made:

- A demonstration category on data-race detection was added. Due to several participating verification tools, this category will become a normal main category in SV-COMP 2023. The results are outlined in Sect. 5.
- New verification tasks were added, with an increase in C from 15 201 in 2021 to 15 648 in 2022 and in Java from 473 in 2021 to 586 in 2022, combined with ongoing efforts on quality improvement.

2 Organization, Definitions, Formats, and Rules

Procedure. The overall organization of the competition did not change in comparison to the earlier editions [10, 11, 12, 13, 14, 15, 16, 17, 18]. SV-COMP is an open competition (also known as comparative evaluation), where all verification tasks are known before the submission of the participating verifiers, which is necessary due to the complexity of the C language. The procedure is partitioned into the *benchmark submission* phase, the *training* phase, and the *evaluation* phase. The participants received the results of their verifier continuously via e-mail (for preruns and the final competition run), and the results were publicly announced on the competition web site after the teams inspected them.

Competition Jury. Traditionally, the competition jury consists of the chair and one member of each participating team; the team-representing members circulate every year after the candidate-submission deadline. This committee reviews

the competition contribution papers and helps the organizer with resolving any disputes that might occur (from competition report of SV-COMP 2013 [11]).

In more detail, the tasks of the jury consist of the following:

- The jury oversees the process and ensures transparency, fairness, and community involvement.
- Each jury member who participates in the competition is assigned a number of (3 or 4) submissions (papers and systems) to review.
- Participating systems are reviewed to determine whether they fulfill the requirements for verifier archives, based on the archives submitted to the repository.
- Teams and paper submissions are reviewed to verify the requirements for qualification, based on the submission data and paper in EasyChair and the results of the qualification runs.
- Some qualified competition candidates are selected to publish (in the LNCS proceedings of TACAS) a contribution paper that gives an overview of the participating system.
- The jury helps the organizer with discussing and resolving any disputes that might occur.
- Jury members adhere to the deadlines with all the duties.

The team representatives of the competition jury are listed in Table 5.

License Requirements. Starting 2018, SV-COMP required that the verifier must be publicly available for download and has a license that

(i) allows reproduction and evaluation by anybody (incl. results publication),
(ii) does not restrict the usage of the verifier output (log files, witnesses), and
(iii) allows any kind of (re-)distribution of the unmodified verifier archive.

Two exceptions were made to allow minor incompatibilities for commercial participants: The jury felt that the rule "allows any kind of (re-)distribution of the unmodified verifier archive" is too broad. The idea of the rule was to maximize the possibilities for reproduction. Starting with SV-COMP 2023, this license requirement shall be changed to "allows (re-)distribution of the unmodified verifier archive via SV-COMP repositories and archives".

Validation of Results. The validation of the verification results was done by eleven validation tools, which are listed in Table 1, including references to literature. Four new validators support the competition:

- There are two new validators for the C language: Dartagnan [new] supports result validation for violation witnesses in category *ConcurrencySafety-Main*. Symbiotic-Witch [new] supports result validation for violation witnesses in categories *ReachSafety*, *MemSafety*, and *NoOverflows*.
- For the first time, there are validators for the Java language: GWit [new] and Wit4Java [new] support result validation for violation witnesses in category *ReachSafety-Java*.

Table 1: Tools for witness-based result validation (validators) and witness linter

Validator	Reference	Representative	Affiliation
CPAchecker	[25, 26, 28]	Thomas Bunk	LMU Munich, Germany
CPA-w2t	[27]	Thomas Lemberger	LMU Munich, Germany
Dartagnan new	[89]	Hernán Ponce de León	Bundeswehr U., Germany
CProver-w2t	[27]	Michael Tautschnig	Queen Mary U. London, UK
GWit new	[68]	Falk Howar	TU Dortmund U., Germany
MetaVal	[33]	Martin Spiessl	LMU Munich, Germany
NitWit	[105]	Jana (Philipp) Berger	RWTH Aachen, Germany
Symbiotic-Witch new	[6]	Paulína Ayaziová	Masaryk U., Brno, Czechia
UAutomizer	[25, 26]	Daniel Dietsch	U. of Freiburg, Germany
Wit4Java new	[108]	Tong Wu	U. of Manchester, UK
WitnessLint		Sven Umbricht	LMU Munich, Germany

Table 2: Scoring schema for SV-COMP 2022 (unchanged from 2021 [18])

Reported result	Points	Description
Unknown	0	Failure to compute verification result
False correct	+1	Violation of property in program was correctly found and a validator confirmed the result based on a witness
False incorrect	−16	Violation reported but property holds (false alarm)
True correct	+2	Program correctly reported to satisfy property and a validator confirmed the result based on a witness
True incorrect	−32	Incorrect program reported as correct (wrong proof)

Task-Definition Format 2.0. SV-COMP 2022 used the task-definition format in version 2.0. More details can be found in the report for Test-Comp 2021 [19].

Properties. Please see the 2015 competition report [13] for the definition of the properties and the property format. All specifications used in SV-COMP 2022 are available in the directory `c/properties/` of the benchmark repository.

Categories. The (updated) category structure of SV-COMP 2022 is illustrated by Fig. 1. The categories are also listed in Tables 8, 9, and 10, and described in detail on the competition web site (`https://sv-comp.sosy-lab.org/2022/benchmarks.php`). Compared to the category structure for SV-COMP 2021, we added the sub-category *Termination-BitVectors* to category *Termination* and the sub-category *SoftwareSystems-BusyBox-ReachSafety* to category *SoftwareSystems*.

Scoring Schema and Ranking. The scoring schema of SV-COMP 2022 was the same as for SV-COMP 2021. Table 2 provides an overview and Fig. 2 visually illustrates the score assignment for the reachability property as an example. As before, the rank of a verifier was decided based on the sum of points (normalized for meta categories). In case of a tie, the rank was decided based on success run time, which is the total CPU time over all verification tasks for which the verifier reported

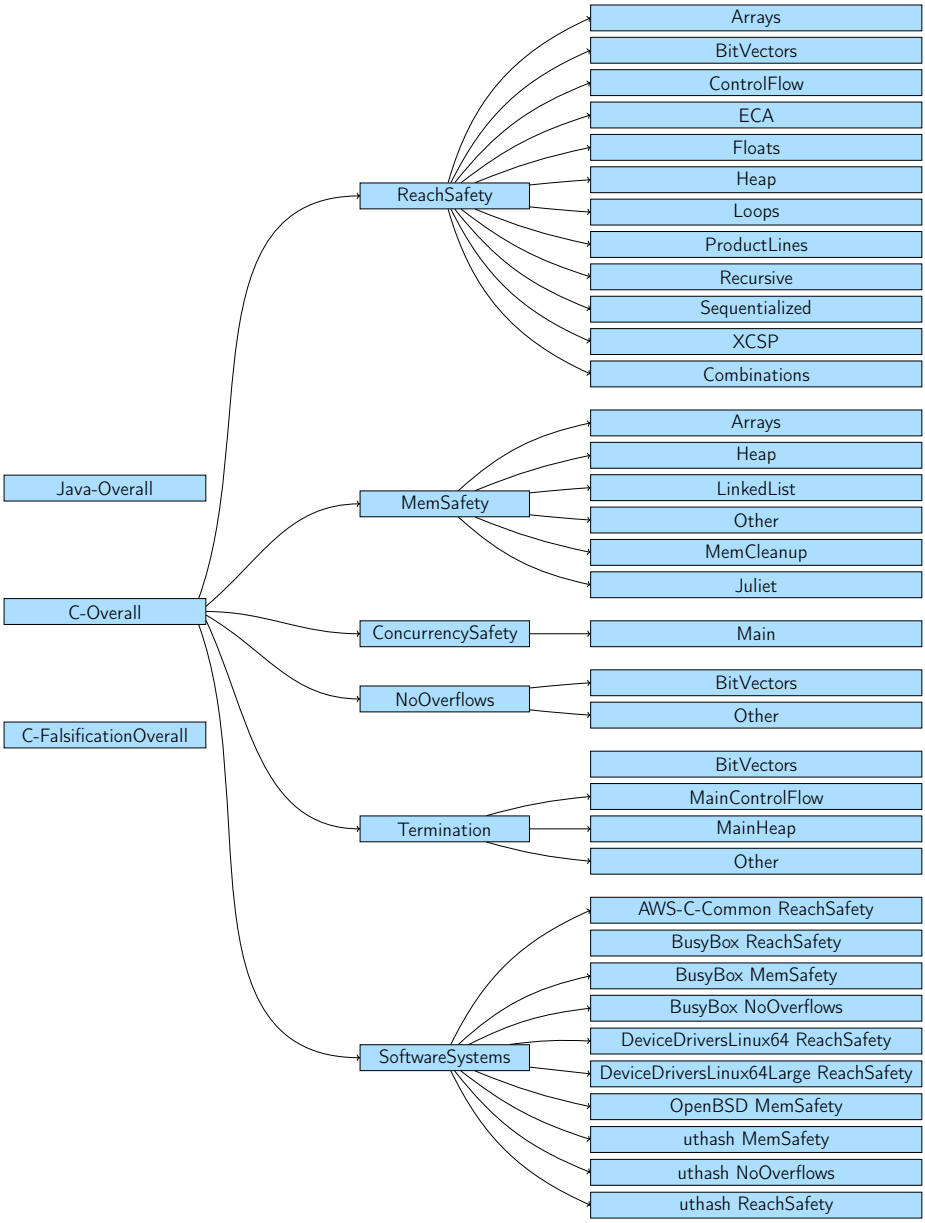


Fig. 1: Category structure for SV-COMP 2022; category *C-FalsificationOverall* contains all verification tasks of *C-Overall* without *Termination*; *Java-Overall* contains all Java verification tasks; compared to SV-COMP 2021, there is one new sub-category in *Termination* and one new sub-categories in *SoftwareSystems*

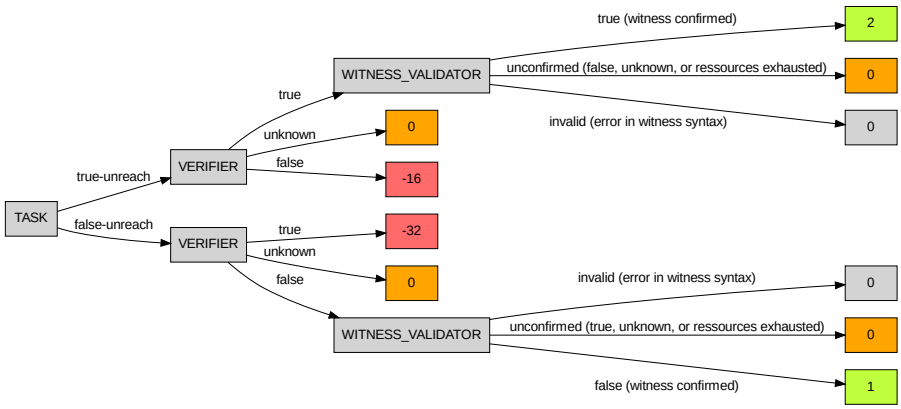


Fig. 2: Visualization of the scoring schema for the reachability property (unchanged from 2021 [18])

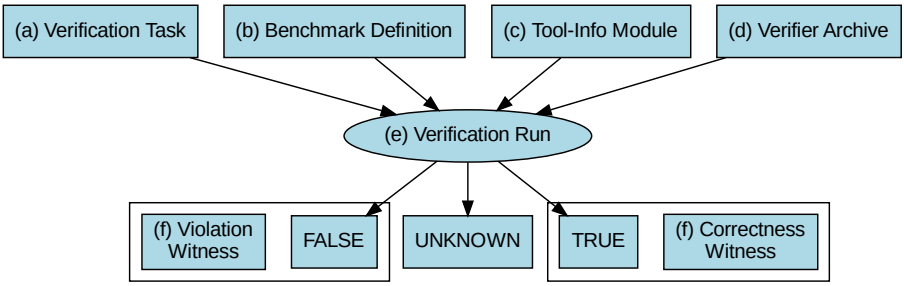


Fig. 3: Benchmarking components of SV-COMP and competition's execution flow (same as for SV-COMP 2020)

a correct verification result. *Opt-out from Categories* and *Score Normalization for Meta Categories* was done as described previously [11] (page 597).

Reproducibility. SV-COMP results must be reproducible, and consequently, all major components are maintained in public version-control repositories. The overview of the components is provided in Fig. 3, and the details are given in Table 3. We refer to the SV-COMP 2016 report [14] for a description of all components of the SV-COMP organization. There are competition artifacts at Zenodo (see Table 4) to guarantee their long-term availability and immutability.

Competition Workflow. The workflow of the competition is described in the report for Test-Comp 2021 [19] (SV-COMP and Test-Comp use a similar workflow).

Table 3: Publicly available components for reproducing SV-COMP 2022

Component	Fig. 3	Repository	Version
Verification Tasks	(a)	gitlab.com/sosy-lab/benchmarking/sv-benchmarks	svcomp22
Benchmark Definitions	(b)	gitlab.com/sosy-lab/sv-comp/bench-defs	svcomp22
Tool-Info Modules	(c)	github.com/sosy-lab/benchexec	3.10
Verifier Archives	(d)	gitlab.com/sosy-lab/sv-comp/archives-2022	svcomp22
Benchmarking	(e)	github.com/sosy-lab/benchexec	3.10
Witness Format	(f)	github.com/sosy-lab/sv-witnesses	svcomp22

Table 4: Artifacts published for SV-COMP 2022

Content	DOI	Reference
Verification Tasks	10.5281/zenodo.5831003	[22]
Competition Results	10.5281/zenodo.5831008	[21]
Verifiers and Validators	10.5281/zenodo.5959149	[24]
Verification Witnesses	10.5281/zenodo.5838498	[23]
BenchExec	10.5281/zenodo.5720267	[106]

3 Reproducing a Verification Run and Trouble-Shooting Guide

In the following we explain a few steps that are useful to reproduce individual results and for trouble shooting. It is written from the perspective of a participant.

Step 1: Make Verifier Archive Available. The first action item for a participant is to submit a merge request to the repository that contains all the verifier archives (see list of merge requests at GitLab). Typical problems include:

- The fork is not public. This means that the continuous integration (CI) pipeline results are not visible and the merge request cannot be merged.
- The shared runners are not enabled. This means that the CI pipeline cannot run and no results will be available.
- Verifier does not provide a version string (and this should not include the verifier name itself). This means that it is not possible to later determine which version of the verifier was used for the experiments. Therefore, version strings are mandatory and are checked by the CI.
- The interface between the execution (with BenchExec) and the verification tool can be checked using the procedure decribed in the BenchExec documentation.[1]

Step 2: Ensure That Verifier Works on Competition Machines. Once the CI checks passed and the archive is merged into the official competition repository, the verifier can be executed on the competition machines on a few verification

[1] https://github.com/sosy-lab/benchexec/blob/3.10/doc/tool-integration.md

tasks. The competition uses the infrastructure VerifierCloud, and remote execution in this compute cloud is possible using CoVeriTeam [29]. CoVeriTeam is a tool for constructing cooperative verification tools from existing components, and the competition is supported by this project since SV-COMP 2021. Among its many capabilities, it enables remote execution of verification runs directly on the competition machines, which was found to be a valuable service for trouble shooting. A description and example invokation for each participating verifier is available in the CoVeriTeam documentation (see file doc/competition-help.md in the CoVeriTeam repository). Competition participants are asked to execute their tool locally using CoVeriTeam and then remotely on the competition machines. Typical problems include:

- Verifiers sometimes have insufficient log output, such that it is not possible to observe what the verifier was executing. The first step towards trouble shooting is always to ensure some minimal log output.
- The verifier assumes software that is not installed yet. Each verifier states its dependencies in its documentation. For example, the verifier Cbmc specifies under `required-ubuntu-packages` that is relies on the Ubuntu packages `gcc` and `libc6-dev-i386` in file benchmark-defs/category-structure.yml in the repository with the benchmark definitions. This is easy to fix by adding the dependency in the definition file and get it installed.
- The verifier makes assumptions about the hardware of the machine, e.g., selecting a specific processing unit. This can be investigated by running the verifier in the Docker container and remotely on the competition machines.
- For the above-mentioned purpose, the competition offers a Docker image that can be used to try out if all required dependencies are available.[2]
- The competition also provides a list of installed packages, which is important for ensuring reproducibility.

Step 3: Check Prerun Results. So far, we considered executing individual verification runs in the Docker container or remotely on the competition machines. As a service to the participating teams, the competition offers training runs and provides the results to the teams. Typical checks that teams perform on the prerun results include:

- Inspect the verification results (solution to the verification task, like True, False, Unknown, etc.) and log files.
- Inspect the validation results (was the verification result confirmed by a validator) and the produced verification witnesses.
- Inspect the result of the witness linter. All witnesses should be syntactically correct according to the witness specification.
- In case the verification result does not match the expected result, investigate the verifier and the verification task; in case of problems with the verification task, report to the jury by creating a merge request with a fix or an issue for discussion in the SV-Benchmarks repository.

[2] https://gitlab.com/sosy-lab/benchmarking/competition-scripts/-/tree/svcomp22

Table 5: Competition candidates with tool references and representing jury members; new for first-time participants, ∅ for hors-concours participation

Participant	Ref.	Jury member	Affiliation
2LS	[36, 81]	Viktor Malík	BUT, Brno, Czechia
APROVE	[65, 100]	Jera Hensel	RWTH Aachen, Germany
BRICK	[37]	Lei Bu	Nanjing U., China
CBMC	[75]	Michael Tautschnig	Queen Mary U. of London, UK
COASTAL ∅	[102]	(hors concours)	–
CVT-ALGOSEL new ∅	[29, 30]	(hors concours)	–
CVT-PARPORT new ∅	[29, 30]	(hors concours)	–
CPA-BAM-BNB ∅	[3, 104]	(hors concours)	–
CPA-BAM-SMG new		Anton Vasilyev	ISP RAS, Russia
CPACHECKER	[31, 49]	Thomas Bunk	LMU Munich, Germany
CPALOCKATOR ∅	[4, 5]	(hors concours)	–
CRUX new	[52, 96]	Ryan Scott	Galois, USA
CSEQ	[47, 71]	Emerson Sales	Gran Sasso Science Institute, Italy
DARTAGNAN	[58, 88]	Hernán Ponce de León	U. Bundeswehr Munich, Germany
DEAGLE new	[62]	Fei He	Tsinghua U., China
DIVINE ∅	[8, 76]	(hors concours)	–
EBF new		Fatimah Aljaafari	U. of Manchester, UK
ESBMC-INCR ∅	[43, 46]	(hors concours)	–
ESBMC-KIND	[56, 57]	Rafael Sá Menezes	U. of Manchester, UK
FRAMA-C-SV	[34, 48]	Martin Spiessl	LMU Munich, Germany
GAZER-THETA ∅	[1, 60]	(hors concours)	–
GDART new	[84]	Falk Howar	TU Dortmund, Germany
GOBLINT	[95, 103]	Simmo Saan	U. of Tartu, Estonia
GRAVES-CPA new	[79]	Will Leeson	U. of Virginia, USA
INFER new ∅	[38, 73]	(hors concours)	–
JAVA-RANGER	[98, 99]	Soha Hussein	U. of Minnesota, USA
JAYHORN	[72, 97]	Ali Shamakhi	Tehran Inst. Adv. Studies, Iran
JBMC	[44, 45]	Peter Schrammel	U. of Sussex / Diffblue, UK
JDART	[80, 83]	Falk Howar	TU Dortmund, Germany
KORN	[55]	Gidon Ernst	LMU Munich, Germany
LART new	[77, 78]	Henrich Lauko	Masaryk U., Brno, Czechia
LAZY-CSEQ ∅	[69, 70]	(hors concours)	–
LOCKSMITH new	[90]	Vesal Vojdani	U. of Tartu, Estonia
PESCO	[93, 94]	Cedric Richter	U. of Oldenburg, Germany
PINAKA ∅	[41]	(hors concours)	–
PREDATORHP ∅	[66, 87]	(hors concours)	–
SESL new		Xie Li	Academy of Sciences, China
SMACK ∅	[61, 92]	(hors concours)	–
SPF ∅	[85, 91]	(hors concours)	–
SYMBIOTIC	[39, 40]	Marek Chalupa	Masaryk U., Brno, Czechia
THETA new	[101, 109]	Vince Molnár	BME Budapest, Hungary
UAUTOMIZER	[63, 64]	Matthias Heizmann	U. of Freiburg, Germany
UGEMCUTTER new	[74]	Dominik Klumpp	U. of Freiburg, Germany
UKOJAK	[54, 86]	Frank Schüssele	U. of Freiburg, Germany
UTAIPAN	[51, 59]	Daniel Dietsch	U. of Freiburg, Germany
VERIABS	[2, 50]	Priyanka Darke	Tata Consultancy Services, India
VERIFUZZ	[42, 82]	Raveendra Kumar M.	Tata Consultancy Services, India

Table 6: Algorithms and techniques that the participating verification systems used; new for first-time participants, ∅ for hors-concours participation

Verifier	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms	Algorithm Selection	Portfolio
2LS				✓	✓			✓	✓		✓						✓			
APROVE			✓				✓	✓		✓	✓						✓			
BRICK	✓		✓	✓				✓								✓				
CBMC				✓							✓					✓				
COASTAL∅			✓																	
CVT-ALGOSEL new∅	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓
CVT-PARPORT new∅	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓
CPA-BAM-BNB∅	✓	✓					✓				✓	✓	✓	✓						
CPA-BAM-SMG new																				
CPALOCKATOR∅	✓	✓					✓				✓	✓	✓	✓		✓				
CPACHECKER	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓
CRUX new			✓																	
CSEQ				✓							✓					✓				
DARTAGNAN				✓							✓					✓				
DEAGLE new																				
DIVINE∅			✓				✓				✓					✓			✓	✓
EBF new				✓																
ESBMC-INCR∅				✓	✓						✓					✓				
ESBMC-KIND				✓	✓			✓			✓					✓				
FRAMA-C-SV								✓												
GAZER-THETA∅	✓	✓		✓			✓				✓	✓	✓	✓						✓
GDART new			✓								✓									✓
GOBLINT								✓								✓				
GRAVES-CPA new	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓
INFER new∅								✓	✓	✓										✓
JAVA-RANGER			✓								✓									
JAYHORN	✓	✓				✓		✓					✓	✓						
JBMC				✓							✓					✓				
JDART			✓								✓									✓
KORN		✓	✓				✓													✓
LART new			✓				✓				✓									✓
LAZY-CSEQ∅				✓							✓					✓				

(continues on next page)

Verifier	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms	Algorithm Selection	Portfolio
Locksmith [new]																✓				
PeSCo	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓
Pinaka[∅]			✓	✓							✓									
PredatorHP[∅]									✓											
Sesl [new]			✓							✓										
Smack[∅]				✓							✓		✓			✓				
Spf[∅]			✓						✓							✓				
Symbiotic			✓		✓			✓	✓		✓					✓				✓
Theta [new]	✓	✓					✓				✓	✓		✓		✓			✓	✓
UAutomizer	✓	✓									✓		✓	✓	✓	✓	✓		✓	✓
UGemCutter [new]	✓	✓									✓		✓	✓	✓	✓			✓	✓
UKojak	✓	✓									✓		✓	✓						
UTaipan	✓	✓					✓	✓			✓		✓	✓	✓	✓			✓	✓
VeriAbs	✓			✓	✓		✓	✓										✓	✓	✓
VeriFuzz				✓				✓										✓		

4 Participating Verifiers

The participating verification systems are listed in Table 5. The table contains the verifier name (with hyperlink), references to papers that describe the systems, the representing jury member and the affiliation. The listing is also available on the competition web site at `https://sv-comp.sosy-lab.org/2022/systems.php`. Table 6 lists the algorithms and techniques that are used by the verification tools, and Table 7 gives an overview of commonly used solver libraries and frameworks.

Hors-Concours Participation. There are verification tools that participated in the comparative evaluation, but did not participate in the rankings. We call this kind of participation *hors concours* as these participants cannot participate in rankings and cannot "win" the competition. Those are either passive or active participants. Passive participation means that the tools are taken from previous years of the competition, in order to show progress and compare new tools against them (Coastal[∅], CPA-BAM-BnB[∅], CPALockator[∅], Divine[∅], Esbmc-incr[∅], Gazer-Theta[∅], Lazy-CSeq[∅], Pinaka[∅], PredatorHP[∅], Smack[∅], Spf[∅]). Active paritication means that there are teams actively developing the tools, but there are reasons why those tools should not occur in the rankings. For example, a

Table 7: Solver libraries and frameworks that are used as components in the participating verification systems (component is mentioned if used more than three times; new for first-time participants, ∅ for hors-concours participation

Verifier	CPAchecker	CProver	Esbmc	Jpf	Ultimate	JavaSMT	MathSAT	Cvc4	SMTInterpol	z3	MiniSAT
2ls		✓									✓
AProVE										✓	✓
Brick										✓	✓
Cbmc		✓									✓
Coastal∅				✓							
CVT-AlgoSel new∅	✓	✓	✓		✓	✓	✓				✓
CVT-ParPort new∅	✓	✓	✓		✓	✓	✓				✓
CPA-BAM-BnB∅	✓					✓	✓				
CPA-BAM-SMG new	✓					✓	✓				
CPALockator∅	✓					✓	✓				
CPAchecker	✓					✓	✓				
Crux new										✓	
CSeq		✓									✓
Dartagnan						✓					
Deagle new											
Divine∅											
Ebf new			✓				✓				
Esbmc-incr∅			✓				✓				
Esbmc-kind			✓				✓				
Frama-C-SV											
Gazer-Theta∅											
GDart new				✓							
Goblint											
Graves-CPA new	✓					✓	✓				
Infer new∅											
Java-Ranger				✓							
JayHorn											
Jbmc		✓									✓
JDart				✓							
Korn											
Lart new										✓	
Lazy-CSeq∅		✓									✓
Locksmith new											
PeSCo	✓					✓	✓				

(continues on next page)

Verifier	CPAchecker	CProver	Esbmc	Jpf	Ultimate	JavaSMT	MathSAT	Cvc4	SMTInterpol	z3	MiniSAT
Pinaka ∅											
PredatorHP ∅											
Sesl new											
Smack ∅											
Spf ∅				✓							
Symbiotic										✓	
Theta new											
UAutomizer					✓		✓	✓	✓	✓	
UGemCutter new					✓		✓	✓	✓	✓	
UKojak					✓				✓	✓	
UTaipan					✓		✓	✓	✓	✓	
VeriAbs	✓	✓								✓	✓
VeriFuzz										✓	

tool might use other tools that participate in the competition on their own, and comparing such a tool in the ranking could be considered unfair (CVT-AlgoSel new ∅, CVT-ParPort new ∅). Also, a tool might produce uncertain results and the team was not sure if the full potential of the tool can be shown in the SV-COMP experiments (Infer new ∅). Those participations are marked as 'hors concours' in Table 5 and others, and the names are annotated with a symbol (∅).

5 Results and Discussion

The results of the competition represent the the state of the art of what can be achieved with fully automatic software-verification tools on the given benchmark set. We report the effectiveness (number of verification tasks that can be solved and correctness of the results, as accumulated in the score) and the efficiency (resource consumption in terms of CPU time and CPU energy). The results are presented in the same way as in last years, such that the improvements compared to last year are easy to identify, except that due to the number of tools, we have to split the table and put the hors-concours verifiers into a second results table. The results presented in this report were inspected and approved by the participating teams.

Computing Resources. The resource limits were the same as in the previous competitions [14]: Each verification run was limited to 8 processing units (cores), 15 GB of memory, and 15 min of CPU time. Witness validation was limited to 2 processing units, 7 GB of memory, and 1.5 min of CPU time for violation witnesses and 15 min of CPU time for correctness witnesses. The machines

Table 8: Quantitative overview over all regular results; empty cells are used for opt-outs, new for first-time participants

Verifier	ReachSafety 8631 points 5400 tasks	MemSafety 5003 points 3321 tasks	ConcurrencySafety 1160 points 763 tasks	NoOverflows 685 points 454 tasks	Termination 4144 points 2293 tasks	SoftwareSystems 5898 points 3417 tasks	FalsificationOverall 5718 points 13355 tasks	Overall 25209 points 15648 tasks	JavaOverall 828 points 586 tasks
2LS	3585	810	0	428	**2178**	83	1462	7366	
APROVE					**2305**				
BRICK									
CBMC	3808	-262	460	284	1800	-198	2024	6733	
CPA-BAM-SMG new		**3101**				776			
CPACHECKER	**5572**	**3057**	498	**531**	1270	**809**	**3835**	**11904**	
CRUX new	1408			290					
CSEQ			**655**						
DARTAGNAN			481						
DEAGLE new			**757**						
EBF new			496						
ESBMC-KIND	4959	2162	-74	318	1389	633	1841	7727	
FRAMA-C-SV				213					
GOBLINT	858		106	159		340		1951	
GRAVES-CPA new	4520					**802**	2400	9218	
KORN									
LART new	3034					573			
LOCKSMITH new									
PESCO	**5080**					-273	**3683**	10515	
SESL new		345							
SYMBIOTIC	4571	**4051**	105	370	2030	**2704**	**3274**	**12249**	
THETA new	1132		-14						
UAUTOMIZER	3969	2350	493	**506**	**2552**	712	3089	**11802**	
UGEMCUTTER new			**612**						
UKOJAK	2058	1573	0	445	0	382	2056	5078	
UTAIPAN	3634	2336	535	**501**	0	486	3049	8666	
VERIABS	**6923**								
VERIFUZZ	1518	-777	-32	136	-129	0	817		
GDART new									640
JAVA-RANGER									**670**
JAYHORN									376
JBMC									**700**
JDART									**714**

Table 9: Quantitative overview over all hors-concours results; empty cells represent opt-outs, new for first-time participants, ∅ for hors-concours participation

Verifier	ReachSafety 8631 points 5400 tasks	MemSafety 5003 points 3321 tasks	ConcurrencySafety 1160 points 763 tasks	NoOverflows 685 points 454 tasks	Termination 4144 points 2293 tasks	SoftwareSystems 5898 points 3417 tasks	FalsificationOverall 5718 points 13355 tasks	Overall 25209 points 15648 tasks	JavaOverall 828 points 586 tasks
CVT-AlgoSel new ∅	5438		314						
CVT-ParPort new ∅	5904	3700	-551	553	2351	1282	1087	10704	
CPA-BAM-BnB ∅						504			
CPALockator ∅			-1154						
Divine ∅	110	99	-136	0	0	112	-1253	-248	
Esbmc-incr ∅			-74						
Gazer-Theta ∅									
Infer new ∅	-50415		-5890	-5982		-29566			
Lazy-CSeq ∅			571						
Pinaka ∅	3710			-200	1259				
PredatorHP ∅		2205							
Smack ∅						1181			
Coastal ∅									-2541
Spf ∅									430

for running the experiments are part of a compute cluster that consists of 167 machines; each verification run was executed on an otherwise completely unloaded, dedicated machine, in order to achieve precise measurements. Each machine had one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 20.04 with Linux kernel 5.4). We used BenchExec [32] to measure and control computing resources (CPU time, memory, CPU energy) and VerifierCloud to distribute, install, run, and clean-up verification runs, and to collect the results. The values for time and energy are accumulated over all cores of the CPU. To measure the CPU energy, we used CPU Energy Meter [35] (integrated in BenchExec [32]).

One complete verification execution of the competition consisted of 309 081 verification runs (each verifier on each verification task of the selected categories according to the opt-outs), consuming 937 days of CPU time and 249 kWh of CPU energy (without validation). Witness-based result validation required 1.43 million validation runs (each validator on each verification task for categories with witness validation, and for each verifier), consuming 708 days of CPU time. Each tool was executed several times, in order to make sure no installation issues occur during the execution. Including preruns, the infrastructure managed a

Table 10: Overview of the top-three verifiers for each category; new for first-time participants, measurements for CPU time and energy rounded to two significant digits ('–' indicates a missing energy value due to a configuration bug)

Rank	Verifier	Score	CPU Time (in h)	CPU Energy (in kWh)	Solved Tasks	Unconf. Tasks	False Alarms	Wrong Proofs
ReachSafety								
1	**VeriAbs**	**6923**	170	1.8	4 117	359		
2	CPAchecker	5572	130	1.5	3 245	228	4	
3	PeSCo	5080	63	0.57	3 033	314	7	
MemSafety								
1	**Symbiotic**	**4051**	2.6	0.034	2 167	1 097		
2	CPA-BAM-SMG new	3101	7.3	0.064	2 975	17		
3	CPAchecker	3057	7.8	0.069	3 119	0		
ConcurrencySafety								
1	**Deagle** new	**757**	0.50	0.0059	517	42		
2	CSeq	655	5.1	0.059	454	50		
3	UGemCutter new	612	4.9	–	445	21		
NoOverflows								
1	**CPAchecker**	**531**	1.2	0.012	366	3		
2	UAutomizer	506	2.0	0.019	356	2		
3	UTaipan	501	2.2	0.023	355	1		
Termination								
1	**UAutomizer**	**2552**	13	0.12	1 581	8		
2	AProVE	2305	38	0.43	1 114	37		
3	2ls	2178	2.9	0.025	1 163	203		
SoftwareSystems								
1	**Symbiotic**	**2704**	1.2	0.016	1 188	73		
2	CPAchecker	809	52	0.60	1 660	169	1	
3	Graves-CPA new	802	19	0.17	1 582	95	2	**3**
FalsificationOverall								
1	**CPAchecker**	**3835**	81	0.90	3 626	95	5	
2	PeSCo	3683	45	0.41	3 552	110	9	
3	Symbiotic	3274	14	0.18	2 295	1 191	3	
Overall								
1	**Symbiotic**	**12249**	34	0.44	7 430	1 529	3	
2	CPAchecker	11904	210	2.3	9 773	408	14	
3	UAutomizer	11802	170	1.7	7 948	311	2	**2**
JavaOverall								
1	**JDart**	**714**	1.2	0.012	522	0		
2	Jbmc	700	0.42	0.0039	506	0		
3	Java-Ranger	670	4.4	0.052	466	0		

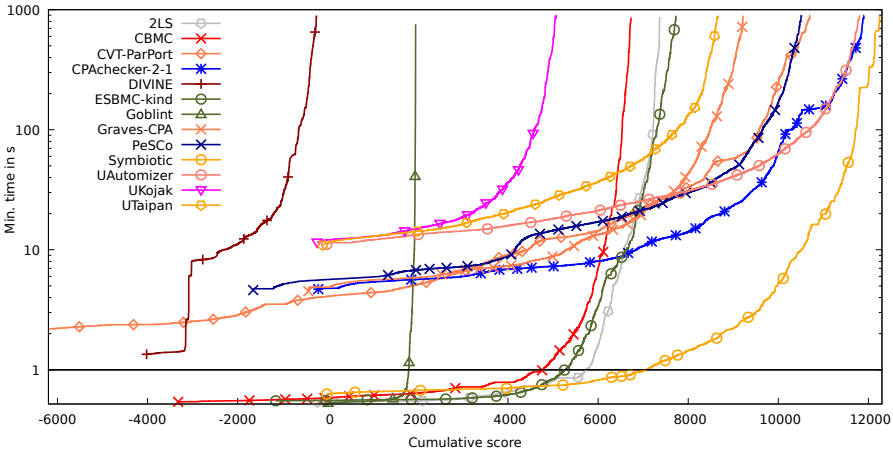


Fig. 4: Quantile functions for category *C-Overall*. Each quantile function illustrates the quantile (x-coordinate) of the scores obtained by correct verification runs below a certain run time (y-coordinate). More details were given previously [11]. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s.

total of 2.85 million verification runs consuming 19 years of CPU time, and 16.3 million validation runs consuming 11 years of CPU time.

Quantitative Results. Tables 8 and 9 present the quantitative overview of all tools and all categories. Due to the large number of tools, we need to split the presentation into two tables, one for the verifiers that participate in the rankings (Table 8), and one for the hors-concours verifiers (Table 9). The head row mentions the category, the maximal score for the category, and the number of verification tasks. The tools are listed in alphabetical order; every table row lists the scores of one verifier. We indicate the top three candidates by formatting their scores in bold face and in larger font size. An empty table cell means that the verifier opted-out from the respective main category (perhaps participating in subcategories only, restricting the evaluation to a specific topic). More information (including interactive tables, quantile plots for every category, and also the raw data in XML format) is available on the competition web site (`https://sv-comp.sosy-lab.org/2022/results`) and in the results artifact (see Table 4).

Table 10 reports the top three verifiers for each category. The run time (column 'CPU Time') and energy (column 'CPU Energy') refer to successfully solved verification tasks (column 'Solved Tasks'). We also report the number of tasks for which no witness validator was able to confirm the result (column 'Unconf. Tasks'). The columns 'False Alarms' and 'Wrong Proofs' report the number of verification tasks for which the verifier reported wrong results, i.e., reporting a counterexample when the property holds (incorrect FALSE) and claiming that the program fulfills the property although it actually contains a bug (incorrect TRUE), respectively.

Table 11: Results of verifiers in demonstration category *NoDataRace*

Verifier	Score	Correct *true*	Correct *false*	Incorrect *true*	Incorrect *false*
CSeq	39	37	61	0	6
Dartagnan	−299	47	23	13	0
Goblint	124	**62**	0	0	0
Locksmith new	34	17	0	0	0
UAutomizer	120	49	54	1	0
UGemCutter new	**151**	57	**69**	1	0
UKojak	0	0	0	0	0
UTaipan	139	56	59	1	0

Score-Based Quantile Functions for Quality Assessment. We use score-based quantile functions [11, 32] because these visualizations make it easier to understand the results of the comparative evaluation. The results archive (see Table 4) and the web site (`https://sv-comp.sosy-lab.org/2022/results`) include such a plot for each (sub-)category. As an example, we show the plot for category *C-Overall* (all verification tasks) in Fig. 4. A total of 13 verifiers participated in category *C-Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [11]). A more detailed discussion of score-based quantile plots, including examples of what insights one can obtain from the plots, is provided in previous competition reports [11, 14].

The winner of the competition, Symbiotic, not only achieves the best cumulative score (graph for Symbiotic has the longest width from $x = 0$ to its right end), but is also extremely efficient (area below the graph is very small). Verifiers whose graphs start with a negative commulative score produced wrong results. Several verifiers whose graphs start with a minimal CPU time larger than 3 s are based on Java and the time is consumed by starting the JVM.

Demo Category *NoDataRace*. SV-COMP 2022 had a new category on data-race detection and we report the results in Table 11. The benchmark set contained a total of 162 verification tasks. The category was defined as a demonstration category because it was not clear how many verifiers would participate. Eight verifiers specified the execution for this sub-category in their benchmark definition [3] and participated in this demonstration. A detailed table was generated by BenchExec's table-generator together with all other results as well and is available on the competition web site and in the artifact (see Table 4).

The results are presented as a means to show that such a category is useful; the results do not represent the full potential of the verifiers, as they were not fully tuned by their developers but handed in for demonstrating abilities only.

Alternative Rankings. The community suggested to report a couple of alternative rankings that honor different aspects of the verification process as complement to the official SV-COMP ranking. Table 12 is similar to Table 10, but contains

[3] `https://gitlab.com/sosy-lab/sv-comp/bench-defs/-/tree/svcomp22/benchmark-defs`

Table 12: Alternative rankings for catagory *Overall*; quality is given in score points (sp), CPU time in hours (h), kilo-watt-hours (kWh), wrong results in errors (E), rank measures in errors per score point (E/sp), joule per score point (J/sp), and score points (sp)

Rank	Verifier	Quality (sp)	CPU Time (h)	CPU Energy (kWh)	Solved Tasks	Wrong Results (E)	Rank Measure
Correct Verifiers							(E/sp)
1	GOBLINT	1 951	4.9	0.070	1 574	0	**0**
2	UKOJAK	5 078	66	0.71	3 988	1	.00020
3	SYMBIOTIC	12 249	34	0.44	7 430	3	.00024
worst (with pos. score)						282	.042
Green Verifiers							(J/sp)
1	GOBLINT	1 951	4.9	0.070	1 574	0	120
2	SYMBIOTIC	12 249	34	0.44	7 430	3	130
3	CBMC	6 733	25	0.27	6 479	282	140
worst (with pos. score)							690

the alternative ranking categories *Correct* and *Green Verifiers*. Column 'Quality' gives the score in score points, column 'CPU Time' the CPU usage of successful runs in hours, column 'CPU Energy' the CPU usage of successful runs in kWh, column 'Solved Tasks' the number of correct results, column 'Wrong Results' the sum of false alarms and wrong proofs in number of errors, and column 'Rank Measure' gives the measure to determine the alternative rank.

Correct Verifiers — Low Failure Rate. The right-most columns of Table 10 report that the verifiers achieve a high degree of correctness (all top three verifiers in the *C-Overall* have less than 2 ‰ wrong results). The winners of category *Java-Overall* produced not a single wrong answer. The first category in Table 12 uses a failure rate as rank measure: $\frac{\text{number of incorrect results}}{\max(\text{total score},1)}$, the number of errors per score point (E/sp). We use E as unit for number of incorrect results and sp as unit for total score. The worst result was 0.023 E/sp in SV-COMP 2021 and is now at 0.042 E/sp. GOBLINT is the best verifier regarding this measure.

Green Verifiers — Low Energy Consumption. Since a large part of the cost of verification is given by the energy consumption, it might be important to also consider the energy efficiency. The second category in Table 12 uses the energy consumption per score point as rank measure: $\frac{\text{total CPU energy}}{\max(\text{total score},1)}$, with the unit J/sp. The worst result from SV-COMP 2021 was 630 J/sp and is now at 690 J/sp. Also here, GOBLINT is the best verifier regarding this measure.

New Verifiers. To acknowledge the verification systems that participate for the first or second time in SV-COMP, Table 13 lists the new verifiers (in SV-COMP 2021 or SV-COMP 2022).

Table 13: New verifiers in SV-COMP 2021 and SV-COMP 2022; column 'Sub-categories' gives the number of executed categories (including demo category *NoDataRace*), new for first-time participants, ∅ for hors-concours participation

Verifier	Language	First Year	Sub-categories
CVT-AlgoSel new ∅	C	2022	18
CVT-ParPort new ∅	C	2022	35
CPA-BAM-SMG new	C	2022	16
Crux new	C	2022	20
Deagle new	C	2022	1
Ebf new	C	2022	1
Graves-CPA new	C	2022	35
Infer new ∅	C	2022	25
Lart new	C	2022	22
Locksmith new	C	2022	1
Sesl new	C	2022	6
Theta new	C	2022	13
UGemCutter new	C	2022	2
Frama-C-SV	C	2021	4
Gazer-Theta ∅	C	2021	9
Goblint	C	2021	25
Korn	C	2021	4
GDart new	Java	2022	1

Table 14: Confirmation rate of verification witnesses during the evaluation in SV-COMP 2022; new for first-time participants, ∅ for hors-concours participation

Result	True				False			
	Total	Confirmed		Unconf.	Total	Confirmed		Unconf.
2ls	2 394	2 388	**99.7 %**	**6**	1 648	1 363	82.7 %	285
Cbmc	3 837	3 493	91.0 %	344	3 536	2 986	84.4 %	550
CVT-ParPort new ∅	7 440	7 083	95.2 %	357	4 754	4 332	91.1 %	422
CPAchecker	6 006	5 701	94.9 %	305	4 175	4 072	97.5 %	103
Divine ∅	1 692	1 672	98.8 %	20	1 040	870	83.7 %	170
Esbmc-kind	5 542	5 483	98.9 %	59	3 034	2 556	84.2 %	478
Goblint	1 657	1 574	95.0 %	83	0	0		
Graves-CPA new	5 651	5 458	96.6 %	193	3 723	3 576	96.1 %	147
PeSCo	6 155	5 734	93.2 %	421	4 116	3 934	95.6 %	182
Symbiotic	4 878	4 798	98.4 %	80	4 081	2 632	64.5 %	1 449
UAutomizer	5 751	5 591	97.2 %	160	2 508	2 357	94.0 %	151
UKojak	2 875	2 863	99.6 %	12	1 144	1 125	**98.3 %**	**19**
UTaipan	4 567	4 513	98.8 %	54	1 719	1 576	91.7 %	143

Verifiable Witnesses. Results validation is of primary importance in the competition. All SV-COMP verifiers are required to justify the result (True or False) by producing a verification witness (except for those categories for which no result validator is available). We used ten independently developed witness-based result validators and one witness linter (see Table 1).

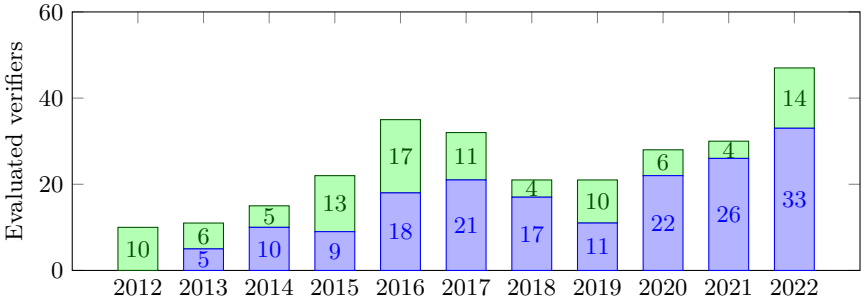


Fig. 5: Number of evaluated verifiers for each year (first-time participants on top)

Table 14 shows the confirmed versus unconfirmed results: the first column lists the verifiers of category *C-Overall*, the three columns for result TRUE reports the total, confirmed, and unconfirmed number of verification tasks for which the verifier answered with TRUE, respectively, and the three columns for result FALSE reports the total, confirmed, and unconfirmed number of verification tasks for which the verifier answered with FALSE, respectively. More information (for all verifiers) is given in the detailed tables on the competition web site and in the results artifact; all verification witnesses are also contained in the witnesses artifact (see Table 4). The verifiers 2LS and UKOJAK are the winners in terms of confirmed results for expected results TRUE and FALSE, respectively. The overall interpretation is similar to SV-COMP 2020 and 2021 [17, 18].

6 Conclusion

The 11th edition of the Competition on Software Verification (SV-COMP 2022) was the largest ever, with 47 participating verification systems (incl. 14 hors-concours and 14 new verifiers) (see Fig. 5 for the participation numbers and Table 5 for the details). The number of result validators was increased from 6 in 2021 to 11 in 2022, to validate the results (Table 1). The number of verification tasks was increased to 15 648 in the C category and to 586 in the Java category, and a new category on data-race detection was demonstrated. A new section in this report (Sect. 3) explains steps to reproduce verification results and to investigate problems during execution, and a new table tried to give an overview of the usage of common solver libraries and frameworks. The high quality standards of the TACAS conference, in particular with respect to the important principles of fairness, community support, and transparency are ensured by a competition jury in which each participating team had a member. We hope that the broad overview of verification tools stimulates their further application by an ever growing user community of formal methods.

Data-Availability Statement. The verification tasks and results of the competition are published at Zenodo, as described in Table 4. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Table 3. For easy access, the results are presented also online on the competition web site `https://sv-comp.sosy-lab.org/2022/results`.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 418257054 (Coop).

References

1. Ádám, Zs., Sallai, Gy., Hajdu, Á.: Gazer-Theta: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In: Proc. TACAS (2). pp. 433–437. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_27
2. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141 (2019). https://doi.org/10.1109/ASE.2019.00121
3. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS. pp. 355–359. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_22
4. Andrianov, P., Mutilin, V., Khoroshilov, A.: CPALockator: Thread-modular approach with projections (competition contribution). In: Proc. TACAS (2). pp. 423–427. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_25
5. Andrianov, P.S.: Analysis of correct synchronization of operating system components. Program. Comput. Softw. **46**, 712–730 (2020). https://doi.org/10.1134/S0361768820080022
6. Ayaziová, P., Chalupa, M., Strejček, J.: Symbiotic-Witch: A Klee-based violation witness checker (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
7. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT Competition 2016: Recent developments. In: Proc. AAAI. pp. 5061–5063. AAAI Press (2017)
8. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with Divine 4. In: Proc. ATVA. pp. 201–207. LNCS 10482, Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
9. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS (3). pp. 3–24. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_1
10. Beyer, D.: Competition on software verification (SV-COMP). In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_38
11. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
12. Beyer, D.: Status report on software verification (Competition summary SV-COMP 2014). In: Proc. TACAS. pp. 373–388. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_25
13. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
14. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In: Proc. TACAS. pp. 887–904. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55

15. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS. pp. 331–349. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20
16. 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
17. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
18. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24
19. Beyer, D.: Status report on software testing: Test-Comp 2021. In: Proc. FASE. pp. 341–357. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_17
20. Beyer, D.: Advances in automatic software testing: Test-Comp 2022. In: Proc. FASE. LNCS 13241, Springer (2022)
21. Beyer, D.: Results of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831008
22. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
23. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5838498
24. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
25. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
26. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
27. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
28. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_26
29. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. Springer (2022)
30. Beyer, D., Kanav, S., Richter, C.: Construction of Verifier Combinations Based on Off-the-Shelf Verifiers. In: Proc. FASE. Springer (2022)
31. 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
32. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
33. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10

34. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
35. Beyer, D., Wendler, P.: CPU Energy Meter: A tool for energy-aware algorithms engineering. In: Proc. TACAS (2). pp. 126–133. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_8
36. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9
37. Bu, L., Xie, Z., Lyu, L., Li, Y., Guo, X., Zhao, J., Li, X.: Brick: Path enumeration-based bounded reachability checking of C programs (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
38. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
39. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
40. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
41. Chaudhary, E., Joshi, S.: Pinaka: Symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS (3). pp. 234–238. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_20
42. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program-aware fuzzing (competition contribution). In: Proc. TACAS (3). pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
43. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proc. ICSE. pp. 331–340. ACM (2011). https://doi.org/10.1145/1985793.1985839
44. Cordeiro, L.C., Kesseli, P., Kröning, D., Schrammel, P., Trtík, M.: JBmc: A bounded model checking tool for verifying Java bytecode. In: Proc. CAV. pp. 183–190. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_10
45. Cordeiro, L.C., Kröning, D., Schrammel, P.: Jbmc: Bounded model checking for Java bytecode (competition contribution). In: Proc. TACAS (3). pp. 219–223. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_17
46. Cordeiro, L.C., Morse, J., Nicole, D., Fischer, B.: Context-bounded model checking with Esbmc 1.17 (competition contribution). In: Proc. TACAS. pp. 534–537. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_42
47. Coto, A., Inverso, O., Sales, E., Tuosto, E.: A prototype for data race detection in CSeq 3 (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
48. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
49. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
50. Darke, P., Agrawal, S., Venkatesh, R.: VeriAbs: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32

51. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate Taipan with symbolic interpretation and fluid abstractions (competition contribution). In: Proc. TACAS (2). pp. 418–422. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_32
52. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Proc. VSTTE. pp. 56–72. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_5
53. Dross, C., Furia, C.A., Huisman, M., Monahan, R., Müller, P.: Verifythis 2019: A program-verification competition. Int. J. Softw. Tools Technol. Transf. **23**(6), 883–893 (2021). https://doi.org/10.1007/s10009-021-00619-x
54. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Proc. VMCAI. pp. 186–201. LNCS 7148, Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_13
55. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. arXiv:2010.05812v2, arXiv (January 2020)
56. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: Esbmc v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS (3). pp. 209–213. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15
57. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (February 2017). https://doi.org/10.1007/s10009-015-0407-9
58. Gavrilenko, N., Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Proc. CAV. pp. 355–365. LNCS 11561, Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19
59. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS. pp. 128–147. LNCS 10422, Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
60. Hajdu, Á., Micskei, Z.: Efficient strategies for CEGAR-based model checking. J. Autom. Reasoning **64**(6), 1051–1091 (2020). https://doi.org/10.1007/s10817-019-09535-x
61. Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: Smack+Corral: A modular verifier (competition contribution). In: Proc. TACAS. pp. 451–454. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_42
62. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
63. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2). pp. 447–451. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
64. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
65. Hensel, J., Mensendiek, C., Giesl, J.: AProVE: Non-termination witnesses for C programs (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)

66. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Hardware and Software: Verification and Testing. pp. 202–209. LNCS 10028, Springer (2016). https://doi.org/10.1007/978-3-319-49052-6
67. Howar, F., Jasper, M., Mues, M., Schmidt, D.A., Steffen, B.: The RERS challenge: Towards controllable and scalable benchmark synthesis. Int. J. Softw. Tools Technol. Transf. **23**(6), 917–930 (2021). https://doi.org/10.1007/s10009-021-00617-z
68. Howar, F., Mues, M.: GWit (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
69. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: A lazy sequentialization tool for C (competition contribution). In: Proc. TACAS. pp. 398–401. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
70. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. ACM Trans. Program. Lang. Syst. **44**(1), 1:1–1:50 (2022). https://doi.org/10.1145/3478536
71. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proc. PPoPP. pp. 202–216. ACM (2020). https://doi.org/10.1145/3332466.3374529
72. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: Proc. CAV. pp. 352–358. LNCS 9779, Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_19
73. Kettl, M., Lemberger, T.: The static analyzer Infer in SV-COMP (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
74. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podelski, A.: Ultimate GemCutter and the axes of generalization (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
75. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
76. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_17
77. Lauko, H., Ročkai, P.: Lart: Compiled abstract execution (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
78. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. LNCS 11187, Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_17
79. Leeson, W., Dwyer, M.: Graves-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
80. Luckow, K.S., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., Raman, V.: JDart: A dynamic symbolic analysis framework. In: Proc. TACAS. pp. 442–459. LNCSS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_26
81. Malík, V., Schrammel, P., Vojnar, T.: 2ls: Heap analysis and memory safety (competition contribution). In: Proc. TACAS (2). pp. 368–372. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_22
82. Metta, R., Medicherla, R.K., Chakraborty, S.: BMC+Fuzz: Efficient and effective test generation. In: Proc. DATE. IEEE (2022)
83. Mues, M., Howar, F.: JDart: Portfolio solving, breadth-first search and SMT-Lib strings (competition contribution). In: Proc. TACAS (2). pp. 448–452. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_30

84. Mues, M., Howar, F.: GDart (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
85. Noller, Y., Păsăreanu, C.S., Le, X.B.D., Visser, W., Fromherz, A.: Symbolic Pathfinder for SV-COMP (competition contribution). In: Proc. TACAS (3). pp. 239–243. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_21
86. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: Ultimate Kojak with memory safety checks (competition contribution). In: Proc. TACAS. pp. 458–460. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44
87. Peringer, P., Šoková, V., Vojnar, T.: PredatorHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
88. Ponce-De-Leon, H., Haas, T., Meyer, R.: Dartagnan: Leveraging compiler optimizations and the price of precision (competition contribution). In: Proc. TACAS (2). pp. 428–432. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_26
89. Ponce-De-Leon, H., Haas, T., Meyer, R.: Dartagnan: Smt-based violation witness validation (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
90. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: Practical static race detection for C. ACM Trans. Program. Lang. Syst. **33**(1) (January 2011). https://doi.org/10.1145/1889997.1890000
91. Păsăreanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Autom. Software Eng. **20**(3), 391–425 (2013). https://doi.org/10.1007/s10515-013-0122-2
92. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: Proc. CAV. pp. 106–113. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_7
93. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
94. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
95. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints (competition contribution). In: Proc. TACAS (2). pp. 438–442. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28
96. Scott, R., Dockins, R., Ravitch, T., Tomb, A.: Crux: Symbolic execution meets SMT-based verification (competition contribution). Zenodo (February 2022). https://doi.org/10.5281/zenodo.6147218
97. Shamakhi, A., Hojjat, H., Rümmer, P.: Towards string support in JayHorn (competition contribution). In: Proc. TACAS (2). pp. 443–447. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_29
98. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S.A., Visser, W.: Java Ranger at SV-COMP 2020 (competition contribution). In: Proc. TACAS (2). pp. 393–397. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_27

99. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S.A., Visser, W.: Java Ranger: Statically summarizing regions for efficient symbolic execution of Java. In: Proc. ESEC/FSE. pp. 123–134. ACM (2020). https://doi.org/10.1145/3368089.3409734
100. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P., Aschermann, C.: Automatically proving termination and memory safety for programs with pointer arithmetic. J. Autom. Reasoning **58**(1), 33–65 (2017). https://doi.org/10.1007/s10817-016-9389-x
101. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: A framework for abstraction refinement-based model checking. In: Proc. FMCAD. pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257
102. Visser, W., Geldenhuys, J.: Coastal: Combining concolic and fuzzing for Java (competition contribution). In: Proc. TACAS (2). pp. 373–377. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_23
103. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
104. Volkov, A.R., Mandrykin, M.U.: Predicate abstractions memory modeling method with separation into disjoint regions. Proceedings of the Institute for System Programming (ISPRAS) **29**, 203–216 (2017). https://doi.org/10.15514/ISPRAS-2017-29(4)-13
105. Švejda, J., Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NitWit. In: Proc. TACAS. pp. 40–57. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_3
106. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.10. Zenodo (2022). https://doi.org/10.5281/zenodo.5720267
107. Wetzler, N., Heule, M.J.H., Jr., W.A.H.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: Proc. SAT. pp. 422–429. LNCS 8561, Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31
108. Wu, T., Schrammel, P., Cordeiro, L.: Wit4Java: A violation-witness validator for Java verifiers (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)
109. Ádám, Z., Bajczi, L., Dobos-Kovács, M., Hajdu, A., Molnár, V.: Theta: Portfolio of cegar-based analyses with dynamic algorithm selection (competition contribution). In: Proc. TACAS. LNCS 13244, Springer (2022)


AProVE: Non-Termination Witnesses for C Programs*

(Competition Contribution)

Jera Hensel✉, Constantin Mensendiek, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. To (dis)prove termination of C programs, AProVE uses symbolic execution to transform the program's LLVM code into an integer transition system, which is then analyzed by several backends. The transformation steps in AProVE and the tools in the backend only produce sub-proofs in their domains. Hence, we now developed new techniques to automatically combine the essence of these proofs. If non-termination is proved, then they yield an overall witness, which identifies a non-terminating path in the original C program.

1 Verification Approach and Software Architecture

To prove (non-)termination of a C program, AProVE uses the Clang compiler [7] to translate it to the intermediate representation of the LLVM framework [15]. Then AProVE symbolically executes the LLVM program and uses abstraction to obtain a finite symbolic execution graph (SEG) containing all possible program runs. We refer to [14,17] for further details on our approach to prove termination.

To prove non-termination, AProVE runs three approaches in parallel, see Fig. 1. The first two approaches transform the lassos of the SEG to integer transition systems (ITSs), which are then passed to the tools T2 [6] and LoAT [11]. If one of the tools returns a proof of non-termination, AProVE uses it to construct a non-terminating path through the C program. The path of the first succeeding approach is returned to the user while all other computations are stopped. T2's proof consists of a recurrent set characterizing those variable assignments that lead to a non-terminating ITS run. Here, AProVE uses an SMT solver to identify a corresponding concrete assignment of the variables in the ITS (which correspond to the variables in the (abstract) program states of the SEG). The third approach transforms the lassos of the SEG directly to SMT formulas which are only satisfiable if there is a non-terminating path, and in this case, we can deduce a variable assignment from the model of the formulas returned by the solver. While the first and the third approach were already available in AProVE before [13], we now extended them by the generation of non-termination witnesses. To this end, the variable assignment obtained from these approaches

* funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2)

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 403–407, 2022.
https://doi.org/10.1007/978-3-030-99527-0_21

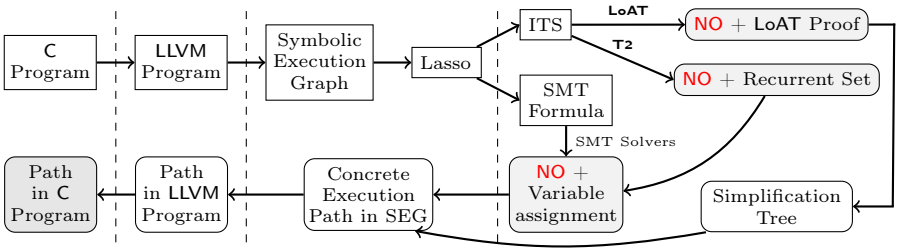


Fig. 1: AProVE's Workflow for Non-Termination Analysis

is used by AProVE to step through the corresponding lasso of the SEG in order to obtain a concrete execution path which witnesses non-termination. To ensure that the generation of the path terminates, AProVE stops as soon as a program state of the SEG is visited twice. Thus, this approach only succeeds if the first loop on the path whose body is executed several times is already the non-terminating loop. However, it does not find non-termination witnesses for programs with several loops, where the non-terminating path first leads through several iterations of other loops before it ends in a non-terminating loop.

To handle such programs as well, we now developed a novel second approach for proving non-termination which uses our tool LoAT in the backend. To understand how LoAT finds non-termination proofs, consider the function `f` in Fig. 2. The first loop decrements `x` as long as `x` is positive and increments `y` by the same amount. Afterwards, the second loop does not terminate if `y` is greater than 1. Hence, the function `f` does not terminate if the initial value of the parameter `x` is greater than 1. LoAT can detect such coherences in the corresponding ITS (Fig. 3a) generated by AProVE. To this end, LoAT uses different forms of *loop acceleration*: *Finite acceleration* combines several iterations of a looping rule into a new rule. LoAT applies this simplification to the rule r_1 representing the first loop, resulting in the new rule r_4 in Fig. 3b. In the second looping rule r_3, the guard is invariant w.r.t. the update of the variables in this rule. In such a case, LoAT applies *non-terminating acceleration*, transforming r_3 to r_5. Finally, *chaining* allows to represent the successive execution of two rules. For example, the rule r_6 is the result of chaining r_0 and r_4. The exact simplification steps performed by LoAT in this example are shown in Fig. 3c. Note that the final rule r_8 starts from the initial function

```
void f(x,y) {
  y = 0;
  while (x > 0) {
    x = x-1;
    y = y+1;
  }
  while (y > 1)
    y = y;
}
```

Fig. 2: Example C Function

$$
\begin{aligned}
&r_0\colon f(x,y) \to \ell_1(x,0)\\
&r_1\colon \ell_1(x,y) \to \ell_1(x-1,y+1) \quad [x>0]\\
&r_2\colon \ell_1(x,y) \to \ell_2(x,y) \quad [x\leq 0]\\
&r_3\colon \ell_2(x,y) \to \ell_2(x,y) \quad [y>1]
\end{aligned}
$$

Fig. 3a: Corresponding ITS

$$
\begin{aligned}
&r_4\colon \ell_1(x,y) \to \ell_1(0,y+x) \quad [x>0]\\
&r_5\colon \ell_2(x,y) \to \infty \quad [y>1]\\
&r_6\colon f(x,y) \to \ell_1(0,x) \quad [x>0]\\
&r_7\colon f(x,y) \to \ell_2(0,x) \quad [x>0]\\
&r_8\colon f(x,y) \to \infty \quad [x>1]
\end{aligned}
$$

Fig. 3b: Simplified Rules

symbol and directly goes to non-termination. Every variable assignment satisfying the respective final guard $x > 1$ results in a non-terminating run.

The simplification tree in Fig. 3c is also the starting point for our new technique to generate non-termination witnesses. AProVE constructs this tree from LoAT's proof output. Then, by processing the leaves of the simplification tree from left to right, a path through the SEG can be derived. To determine how often one has to traverse earlier loops on the path to the non-terminating loop, AProVE uses an SMT solver to find a concrete variable assignment that satisfies the final guard. In our example, the final guard $x > 1$ would be satisfied by $\{x = 2, y = 0\}$, for example. Consequently, the corresponding concrete execution path includes two iterations of the first loop before reaching the non-terminating second loop.

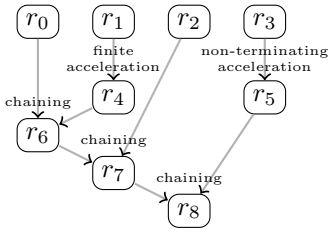


Fig. 3c: Simplification Tree

Once the path is constructed, AProVE extracts the LLVM program positions from the states, obtaining a non-terminating path through the LLVM program in form of a lasso. Using the Clang debug information output, AProVE then matches the LLVM lines to the lines in the C program. The resulting C witness can be validated by the tools CPAchecker [5] and Ultimate Automizer [12].

2 Discussion of Strengths and Weaknesses

In general, AProVE is especially powerful on programs where a precise modeling of the values of program variables and memory contents is needed to (dis)prove termination. However, on large programs containing many variables which are not relevant for termination, tools with CEGAR-based approaches are often faster. The reason is that AProVE does not implement any techniques to decide which variables are relevant for (non-)termination.

Furthermore, one of AProVE's most crucial weaknesses when proving non-termination in past editions of *SV-COMP* was to produce a meaningful witness. Therefore, in the two approaches for proving non-termination in AProVE that are based on T2 or on the direct analysis of lassos of the SEG, we added the novel techniques presented in the current paper to generate non-termination witnesses from the obtained variable assignments. Here, the problem is that when computing a concrete execution path, we cannot be sure when to stop the computation: Whenever we visit a program position repeatedly, we do not know if this position is part of the non-terminating loop of the lasso, or if it is still part of the finite path to the non-terminating loop.

In contrast, in our new approach based on LoAT, the simplification tree allows us to infer the order in which the loops of the program are traversed and this tree also contains the information which loop is the non-terminating one. Thus, this approach extends AProVE's power substantially, since it can find non-termination witnesses for programs where all non-terminating paths lead through several iterations of more than one loop. On the other hand, there are

also examples where the other two approaches outperform the approach based on LoAT, e.g., if T2 finds a non-termination proof and LoAT does not. Our observation is that especially for small programs containing only a single loop, the other approaches are often faster. This is also confirmed by our results in the *Termination* category of *SV-COMP 2022*: While in the sub-categories *MainControlFlow* and *MainHeap*, 83% of the non-termination proofs are found using T2 or the direct SMT approach, in *Termination-Other*, 95% of the non-termination proofs result from the LoAT approach. This set consists of especially large programs, which often contain more than one loop.

More information about *SV-COMP 2022* including the competition results can be found in the competition report [3].

3 Setup and Configuration

AProVE is developed in the "*Programming Languages and Verification*" group headed by J. Giesl at RWTH Aachen University. On the web site [2], AProVE can be downloaded or accessed via a web interface. Moreover, [2] also contains a list of external tools used by AProVE and a list of present and past contributors.

In *SV-COMP 2022*, AProVE only participates in the category "*Termination*". All files from the submitted archive must be extracted into one folder. AProVE is implemented in Java and needs a Java 11 Runtime Environment. Moreover, AProVE requires the Clang compiler [7] to translate C to LLVM. To analyze the resulting ITSs in the backend, AProVE uses LoAT [11] and T2 [6]. Furthermore, it applies the satisfiability checkers Z3 [8], Yices [9], and MiniSAT [10] in parallel (our archive contains all these tools). As a dependency of T2, Mono [16] (version ≥ 4.0) needs to be installed. Extending the path environment is necessary so that AProVE can find these programs. Using the wrapper script `aprove.py` in the `BenchExec` repository, AProVE can be invoked, e.g., on the benchmarks defined in `aprove.xml` in the *SV-COMP* repository. The most recent version of AProVE with the improved witness generation can be downloaded at [1].

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [3] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [4].

References

1. AProVE: https://github.com/aprove-developers/aprove-releases/releases
2. AProVE Website: https://aprove.informatik.rwth-aachen.de/
3. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS '22. LNCS (2022)
4. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022), https://doi.org/10.5281/zenodo.5959149

5. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV '11. pp. 184–190. LNCS 6806 (2011), https://doi.org/10.1007/978-3-642-22110-1_16
6. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: Temporal property verification. In: Proc. TACAS '16. pp. 387–393. LNCS 9636 (2016), https://doi.org/10.1007/978-3-662-49674-9_22
7. Clang: https://clang.llvm.org
8. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS '08. pp. 337–340. LNCS 4963 (2008), https://doi.org/10.1007/978-3-540-78800-3_24
9. Dutertre, B., de Moura, L.: System Description: Yices 1.0 (2006), https://yices.csl.sri.com/papers/yices-smtcomp06.pdf
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT '03. pp. 502–518. LNCS 2919 (2003), https://doi.org/10.1007/978-3-540-24605-3_37
11. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: Proc. FM-CAD '19. pp. 221–230 (2019), https://doi.org/10.23919/FMCAD.2019.8894271
12. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: Ultimate Automizer with array interpolation. In: Proc. TACAS '15. pp. 455–457. LNCS 9035 (2015), https://doi.org/10.1007/978-3-662-46681-0_43
13. Hensel, J., Emrich, F., Frohn, F., Ströder, T., Giesl, J.: AProVE: Proving and disproving termination of memory-manipulating C programs (competition contribution). In: Proc. TACAS '17. pp. 350–354. LNCS 10206 (2017), https://doi.org/10.1007/978-3-662-54580-5_21
14. Hensel, J., Giesl, J., Frohn, F., Ströder, T.: Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. Journal of Logical and Algebraic Methods in Programming **97**, 105–130 (2018), https://doi.org/10.1016/j.jlamp.2018.02.004
15. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO '04. pp. 75–88 (2004), https://doi.org/10.1109/CGO.2004.1281665
16. Mono: https://www.mono-project.com/
17. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P., Aschermann, C.: Automatically proving termination and memory safety for programs with pointer arithmetic. J. of Aut. Reasoning **58**(1), 33–65 (2017), https://doi.org/10.1007/s10817-016-9389-x


BRICK: Path Enumeration Based Bounded Reachability Checking of C Program (Competition Contribution)*

Lei Bu(✉), Zhunyi Xie, Lecheng Lyu, Yichao Li, Xiao Guo, Jianhua Zhao, and Xuandong Li

State Key Laboratory for Novel Software Technology, Nanjing University, China
bulei@nju.edu.cn

Abstract. BRICK is a bounded reachability checker for embedded C programs. BRICK conducts a path-oriented style checking of the bounded state space of the program, that enumerates and checks all the possible paths of the program in the threshold one by one. To alleviate the path explosion problem, BRICK locates and records unsatisfiable core path segments during the checking of each path and uses them to prune the search space. Furthermore, derivative free optimization based falsification and loop induction are introduced to handle complex program features like nonlinear path conditions and loops efficiently.

1 Verification Approach

Existing bounded software checkers usually encode the bounded state space of the program into one constraint solving problem directly. However, in this manner, when the size of the program or the bound of the checking increases, the corresponding constraint solving problem explodes quickly and becomes difficult to solve by existing SAT/SMT solvers.

To solve this problem, BRICK conducts a path-oriented style checking of the bounded state space of the program, that enumerates and checks all the possible paths in the threshold one by one [1,2]. The main merit of the approach is that, in this case, the size of the problem needs to be solved by the constraint solver is well controlled and can be easily handled. The main features of BRICK's solving are reported below:

1.1 Flexible Path Enumeration

BRICK enumerates potential paths from the control flow graph (CFG) of the given program to the user-defined step bound. Two path enumeration strategies are applied in BRICK, each with its own advantages.

* This work is supported in part by the National Key Research and Development Plan (No. 2017YFA0700604), the Leading-Edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001), and the National Natural Science Foundation of China (No.62172200, No.61632015).

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 408–412, 2022.
https://doi.org/10.1007/978-3-030-99527-0_22

First, we can simply conduct classical Depth-first-search (DFS) to enumerate program paths. The benefit of this approach is that, if the DFS stops without touching the given bound, we can get a result that the target state is not reachable in general, not only in the bounded state space.

We have also implemented a special method to encode the jump-to relation between different code blocks into an SAT formula and obtain the potential path by SAT solving. The benefit is that if the potential path is confirmed to be infeasible by following path condition solving, the infeasible path segment in the path can be located and encoded back to the SAT formula to prune all the future paths containing such infeasible segment.

1.2 Infeasible Path Segment Pool Guided State Space Pruning

BRICK conducts the lazy solving of the path by encoding the path condition of the potential path into a feasibility problem. BRICK asks a constraint solver, i.e. SMT solver (Z3 [6]), interval analysis (dReal [4]), and derivative-free optimization-based solving (Section 1.3), to solve the problem. If the path is decided to be infeasible by the solver, BRICK tries to extract the unsatisfiable core (UC) of the feasibility problem of this path, and maps the UC constraints to a infeasible path segment in the path, which will be added to infeasible paths pool. After that, all the paths that contain any infeasible path in the infeasible path pool will be reported as unreachable directly in the following path enumeration.

1.3 Derivative-free Optimization Based Constraint Falsification

We can see that constraint solving plays an important role in BRICK. However, complex path conditions, like nonlinear constraints, which widely appear in programs, are hard to be handled efficiently by the existing solvers. In BRICK, a classification model-based derivative-free-optimization (DFO) approach is used to alleviate this difficult situation by conducting a sample-feedback-learn style DFO solving [8].

More specifically, the underlying solver guesses sample solution for the feasibility problem. Then, we evaluate whether the sampled solution can satisfy the path constraint or not, and calculate the distance between the sampled solution and the correct one if the sampled one does not satisfy the path constraint. Such distance will be used as the metric of feedback in the classification-based DFO learning, to guide the solver to converge to the value that fits the path constraint. In practice, this approach works very well in nonlinear problem solving. However, this DFO-based approach can not tell the target is not reachable, if it fails to find a solution.

1.4 Induction-based Loop Handling

If the target program contains loops, the number of potential paths may explode. To alleviate this problem, we conduct an induction-based proof to try to handle the loop before we start to do the BMC.

First of all, we collect the constraints from the assertions and generate the weakest precondition respectively. Then, we conduct the normal induction-based proof to see whether such constraints are satisfied in any iteration. If no counterexamples are returned, we know that the assertions won't be violated in the loop. Furthermore, we are also working on the integration of loop invariant generation to further refine the CFG under checking.

2 Software Architecture

The architecture of BRICK is shown in Fig.1. It consists of a loop processing module, a path enumerating module, and a constraint solving module, all implemented in C++ language.

In the loop processing module, if the program contains assertion-related loop, BRICK conducts loop induction-based verification firstly. If the induction works, BRICK reports unreachable; otherwise, it builds the program CFG, and performs the following path enumeration based checking.

In the path enumerating module, BRICK employs SAT-based and DFS-based path enumerating methods to extract the program path and its corresponding path condition. The constraint solving module accepts the path condition and performs constraint solving accordingly. All the techniques used has been mentioned in Section 1 respectively. The solvers used in BRICK including SAT solver MiniSAT [3], SMT solver Z3 [6], interval analysis solver dReal [4], and our implementation of the DFO method RACOS [9].

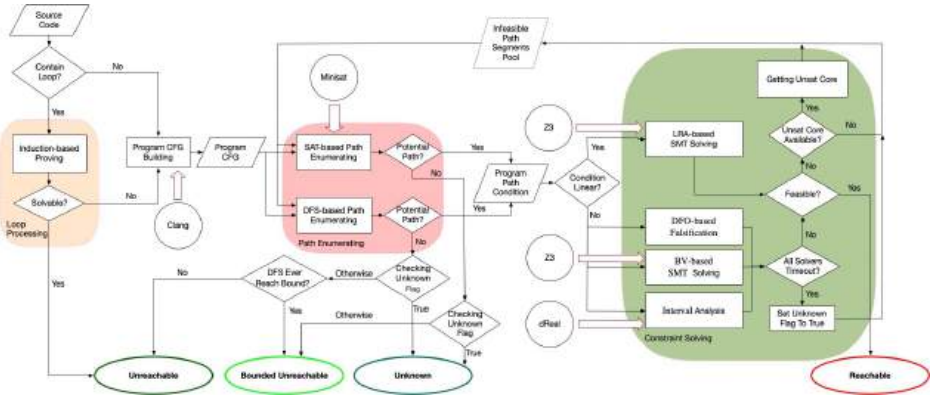


Fig. 1. Architecture of BRICK

3 Strengths and Weaknesses

Most of the bounded reachability checkers, i.e. CBMC [5], encode the bounded state space to a huge SMT formula consisting of both conjunction and disjunc-

tion of different kinds of formulas, which are difficult for the existing solvers to handle and may cause memory explosion easily. Instead, BRICK conducts the verification in a path-oriented way:

- BRICK enumerates and checks all the potential paths one by one. In this manner, the computation complexity is well controlled.
- Meanwhile, as only the ongoing path is saved in the memory and the corresponding path constraints of the path will be disjunction-free, the solving problem is much easier to handle.
- For the sake of processing capability, UC-guided backtracking and path pruning is proposed to prune the search space substantially, and DFO-based solving is conducted to handle complex nonlinear constraints efficiently.

BRICK had participated in the ReachSafety/Floats category of SV-COMP 2022 [10]. BRICK has successfully verified 439 of all the 469 tasks, ranked 1st in this sub-category. Furthermore, we can see that for these 439 solved cases, BRICK only used 1000 seconds in total. On the other hand, CoveriTeam and VeriAbs [7] which won the 2nd and 3rd place in this category spent 9300 and 18000 seconds respectively, which are 9 and 18 times higher than BRICK.

For the weakness, like all the other bounded checkers, BRICK may not be able to give proofs of correctness of a program, if it can not finish the search in the given step bound. In this case, BRICK can only report bounded true. For example, on the cases of SV-COMP 2022, besides the 439 cases which are proved by BRICK, there are also several programs that BRICK can only give a bounded result or just timeout. Therefore, for the future work, we are implementing techniques including loop summary, k-induction and so on to try to abstract the loops and give a proof of the correctness in certain cases.

4 Tool Setup and Configuration

The binary file of BRICK for Ubuntu 20.04 is available at https://github.com/brick-tool-dev/BRICK-2.0. To install the tool, please clone this repository and follow the instruction in README.md. A tailored version of BRICK took part in the ReachSafety/Floats category in SV-COMP 2022 [10]. The version [11] supports the checking of reachability of Error Function. The BenchExec wrapper script for the tool is *brick.py* and *brick.xml* is the benchmark description file.

5 Software Project and Contributors

BRICK is available under MIT License. The team of BRICK is from Software Engineering Group, Nanjing University. We would like to thank Sicun Gao for his kindly help with the usage of dReal.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [10] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [11].

References

1. L. Bu, *et al.*. BACH: Bounded Reachability Checker for Linear Hybrid Automata. In FMCAD'08, pp. 65-68.
2. D. Xie, *et al.*. SAT-LP-IIS Joint-Directed Path-Oriented Bounded Reachability Analysis of Linear Hybrid Automata. In FMSD. 45(1): 42-62, 2014.
3. N. Eénand N. Sörensson. An extensible SAT-solver. In SAT'03, 502-518.
4. S. Gao, *et al.*. dReal: An SMT solver for nonlinear theories over the reals. In CADE'13, 208-214.
5. D. Kroening, *et al.*. CBMC - C Bounded Model Checker. In TACAS'14: 389-391.
6. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In TACAS'08: 337-340.
7. P. Darke, *et al.*. VeriAbs: Verification by Abstraction and Test Generation - (Competition Contribution). In TACAS'18: 457-462.
8. L. Bu, *et al.*. Machine learning steered symbolic execution framework for complex software code. In Formal Aspects of Computing, 33:3, 301-323, 2021.
9. Y. Yu, *et al.*. Derivative-free optimization via classification. In AAAI'16, 2286-2292.
10. D. Beyer, Progress on Software Verification: SV-COMP 2022. In TACAS'22
11. D. Beyer, Verifiers and Validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo, DOI:10.5281/zenodo.5959149, 2022


A Prototype for Data Race Detection in CSeq 3*
(Competition Contribution)

Alex Coto, Omar Inverso, Emerson Sales✉, and Emilio Tuosto

Gran Sasso Science Institute, L'Aquila, Italy
{alex.coto,omar.inverso,emerson.sales,emilio.tuosto}@gssi.it

Abstract. We sketch a sequentialization-based technique for bounded detection of data races under sequential consistency, and summarise the major improvements to our verification framework over the last years.

Keywords: Bounded model checking · Context-bounded analysis · Sequentialization · Data races · Reachability · Concurrency · Threads

1 Verification Approach

Our approach is based on *lazy sequentialization* [7]. The idea is to convert the concurrent program P of interest into a non-deterministic sequential program $Q_{u,k}$ that preserves all feasible executions of P up to unwinding bound u and k rounds (or execution contexts [8]). Among different techniques [6], we choose *bounded model checking* [3] to analyse $Q_{u,k}$. In this section, we briefly overview lazy sequentialisation, and sketch a novel extension to detect data races. Further elements of novelty w.r.t. engineering of our tool are discussed in the next section.

Lazy Sequentialization. We unwind all loops and inline all functions in P, except the main function and those from which a thread is spawned, obtaining a *bounded program* P_u that preserves all feasible executions of P up to the unwinding bound u. We then transform each function of P_u into a *thread simulation function* where each *visible statement* is assigned a numerical label and a guard, and each call to a concurrency-specific function is replaced by a call to a function that models the same intended semantics; for each simulation function, we add a global variable to represent the *program counter*, initially set to zero.

A thread's execution context of P_u is simulated by invoking the corresponding thread simulation function of $Q_{u,k}$ that executes from the first statement to a non-deterministically selected label, updates the program counter, and returns. Further execution contexts are simulated by re-invoking the simulation function, where the guards ensure that the control is repositioned to the correct numerical label via a sequence of jumps, and so on. To retain consistency of the local state of the thread across different invocations of the simulation functions, static storage

* This work has been partially funded by MIUR project PRIN 2017FTXR7S IT-MATTERS and MUR project FISR2020IP_05310 MVM-Adapt.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 413–417, 2022.
https://doi.org/10.1007/978-3-030-99527-0_23

is enforced for all local variables. We drive the overall simulation of P_u from the main function of $Q_{u,k}$, by invoking the thread simulation functions appropriately.

Data Race Detection. A program contains a *data race* if it can execute two *conflicting actions* (i.e., one thread modifies a memory location and another one reads or modifies the same location), at least one of which is not atomic, and neither happens before the other [9]. Consider two threads performing the operation `v = v + 1` on a shared variable initialised to zero. Both threads try to modify the data at the memory location reserved for `v`, but the necessary sequences of memory accesses are not synchronised, and thus may interleave. If a context-switch happens between the memory read and write operations in the thread that runs first, both threads will read 0, and at the end of the execution the value of `v` will be 1. To detect such situation, we alter the encoding from P_u to $Q_{u,k}$ by (i) adding a shared array `w_addrs` that stores a pointer to the memory location targeted by a write operation for each thread, (ii) injecting additional control code at each visible statement, and (iii) splitting the modified sequentialised encoding of the visible statement into two separate sequentialised statements to allow in-between context switching. The code fragment shows the modified sequentialised encoding (no guards for simplicity, injected code greyed out) for the statement `v = v + 1` of the first thread of the program described above. We store in `w_addr` the address of the variable being written, and then assert that the other thread is not writing to the same location; in the same (simulated) execution context, we store `w_addr` in `w_addrs`, so that the assertion can be checked within the other thread too. We reset `w_addrs` right after the statement under consideration. Note the label `k+1` that allows thread pre-emption. Now, one of the threads can execute the simulated statement at label `k` and context-switch at label `k+1` while `w_addrs` still points to `v`; this makes it possible to schedule the other thread, and fail the assertion in there.

```
k:
 void *w_addr = &v;
 assert(w_addrs[1] != w_addr);
 w_addrs[0] = w_addr;
 v = v + 1;
k+1:
 w_addrs[0] = 0;
```

In the general case, handling multiple memory write accesses for a single statement requires a slightly different tracking mechanism for write addresses, or decomposition into simpler statements. Statements with read-only shared memory access are handled without updating `w_addrs`. Programs with more than two threads require multiple assertions.

2 Software Architecture

CSeq is a framework for quick development of static analysis and program transformation prototypes. For parsing the input program CSeq relies on pycparserext (`pypi.org/project/pycparserext`), an extension of pycparser (`github.com/eliben/pycparser`), which in turn is built on top of PLY (`www.dabeaz.com/ply`), a Python implementation of Lex and Yacc. All the mentioned components as well as CSeq are entirely written in Python.

We combined several groups of modules in CSeq, namely (i) program simplification, (ii) program unfolding, (iii) sequentialization, (iv) instrumentation,

and (v) backend invocation and counterexample generation. For the analysis of the sequentialised program we rely on CBMC (`www.cprover.org/cbmc`), that in turn embeds the DPLL-style MiniSat SAT solver (`minisat.se`).

CSeq 3.0 incorporates a significant number of enhancements. At an architectural level, the main element of novelty is in the modularity between the general-purpose functionalities of the framework and the specific lazy sequentialization, which opens up to the possibility of prototyping different static analysers for other applications (e.g., [11,10]) as well as improving older sequentialization-based prototypes (e.g., [4,12,13] and variations thereof). The enhancements to the framework include: Python 3 support, support for GNU C compiler extensions, a fully re-implemented symbol table, revised general-purpose modules such as constant propagation, function inlining, and loop unrolling, and a custom-built version of CBMC (not used in the competition) for SAT-solving under assumptions. For the competition we include (experimental) enhanced constant propagation, and simplified function inlining. Besides the data race checking extension, the sequentialization modules include improvements from earlier implementations [5,8,6] and for different editions of SV-COMP up to date, in particular: extended pthread API support (conditional waiting, barriers, and thread-specific data management), context-bounded analysis, and a major code overhaul.

3 Strengths and Weaknesses

The table below summarises the performance of our tool on the 764 cases of the `Concurrency` category and the 162 cases of the data race demo category.

Overall instances		764	162
Correct	safe	202	37
	unsafe	320	61
Unknown	reject	9	19
	internal error	18	17
	out of time	159	20
	out of memory	56	2
Incorrect	safe	0	0
	unsafe	0	6

Our technique excels at hunting bugs, as shown by the number of correct unsafe (incl. 17 malformed witnesses and 50 unconfirmed witnesses), but gets quickly expensive with larger bounds, hitting the resource limits. The additional context-switch points and the use of pointers for data race detection introduce further overhead. The other failures are due to limiting assumptions or glitches in the implementation. All the false positives are due to corner cases in the encoding.

4 Setup and Configuration

We competed in the `ConcurrencySafety` category and in the data race detection demo category. CSeq 3.0 is available at `https://github.com/omainv/cseq/releases`.

Installation instructions are in the `README` file within the package. A wrapper script (`lazy-cseq.py`) invokes CSeq up to three times, with the options `-l lazy` for lazy sequentialisation, `--sv-comp` to enable the required violation witnesses format, `--atomic-parameters` to assume atomic passing of function

arguments, `--nondet-condvar-wakeups` for non-deterministic spurious conditional variables wake-up calls, `--deep-propagation` for experimental constant folding and propagation, `--32` for 32-bit architectures, `--threads 100` to limit the overall number of threads, `--data-race-check` when required, and `--backend cbmc` to use CBMC 5.4 for sequential analysis.

For reachability checking, on different invocations the script adds different parameters: `-r2 -w2 -f2`, `-r4 -w3 -f5`, and `-r20 -w1 -f11`, where `r` is the number of rounds, and `f` and `w` are the unwind bounds for `for` (i.e., potentially bounded) and `while` (i.e., potentially unbounded) loops, respectively; on the last invocation `--softunwindbound` and `--unwind-for-max 10000` are also added to fully unfold `for` loops if a static bound can be found, up to the given hard bound. For data race detection, the above parameters are replaced with `-c4 -u2`, `-c10 -u10`, and `-c50 -w20 -f20` with `--unwind-for-max 100`. Note that in this case the bound is on the number of execution contexts rather than rounds (`-c` vs. `-r`), and `-u` is used as a shorthand for `-f` and `-w`.

We leave the analysis running to completion every time. When the result is `TRUE`, the scripts restarts the analysis with the next set of parameters. As soon as the script gets `FALSE`, it returns `FALSE`. Only if the analysis using the last set of parameters is finished and the result is `TRUE`, then the script returns `TRUE`.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [1] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [2].

References

1. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
2. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
3. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
4. Fischer, B., Inverso, O., Parlato, G.: Cseq: A concurrency pre-processor for sequential C verification tools. In: ASE. pp. 710–713. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693139
5. Inverso, O., Nguyen, T.L., Fischer, B., Torre, S.L., Parlato, G.: Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In: ASE. pp. 807–812. IEEE Computer Society (2015). https://doi.org/10.1109/ASE.2015.108
6. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. ACM Trans. Program. Lang. Syst. **44**(1) (dec 2021). https://doi.org/10.1145/3478536
7. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 585–602. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_39

8. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: PPoPP. pp. 202–216. ACM (2020). https://doi.org/10.1145/3332466.3374529
9. ISO/IEC: ISO/IEC 9899:2018: Information technology – Programming languages – C (Jun 2018)
10. Simic, S., Bemporad, A., Inverso, O., Tribastone, M.: Tight error analysis in fixed-point arithmetic. In: IFM. Lecture Notes in Computer Science, vol. 12546, pp. 318–336. Springer (2020). https://doi.org/10.1007/978-3-030-63461-2$_1$7
11. Simic, S., Inverso, O., Tribastone, M.: Bit-precise verification of discontinuity errors under fixed-point arithmetic. In: SEFM. Lecture Notes in Computer Science, vol. 13085, pp. 443–460. Springer (2021). https://doi.org/10.1007/978-3-030-92124-8_25
12. Tomasco, E., Inverso, O., Fischer, B., Torre, S.L., Parlato, G.: Verifying concurrent programs by memory unwinding. In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 551–565. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_52
13. Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., Torre, S.L., Parlato, G.: Lazy sequentialization for TSO and PSO via shared memory abstractions. In: FMCAD. pp. 193–200. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886679

Dartagnan: SMT-based Violation Witness Validation (Competition Contribution)

Hernán Ponce-de-León[1](✉)*, Thomas Haas[2], and Roland Meyer[2]

[1]Bundeswehr University Munich, Munich, Germany
[2]TU Braunschweig, Braunschweig, Germany
hernan.ponce@unibw.de, t.haas@tu-braunschweig.de, roland.meyer@tu-bs.de

Abstract. The validation of violation witnesses is an important step during software verification. It hides false alarms raised by verifiers from engineers, which in turn helps them concentrate on critical issues and improves the verification experience. Until the 2021 edition of the Competition on Software Verification (SV-COMP), CPAchecker was the only witness validator for the *ConcurrencySafety* category. This article describes how we extended the Dartagnan verifier to support the validation of violation witnesses. The results of the 2022 edition of the competition show that, for witnesses generated by different verifiers, Dartagnan succeeds in the validation of witnesses where CPAchecker does not. Our extension thus improves the validation possibilities for the overall competition. We discuss Dartagnan's strengths and weaknesses as a validation tool and describe possible ways to improve it in the future.

1 Introduction

Most software verification tools report witnesses to property violations. Since SV-COMP 2015, there is a common format in which witnesses are represented by automata [4]. Each edge of such an automaton is annotated with data that can be used to match program executions. A data annotation can be, e.g., *"assumption"* specifying constraints on values of variables in a given state, *"control"* specifying the outcome of a branch condition, or *"startline"* specifying a concrete line in the source code. More details about data annotations and their semantics can be found in the exchange format documentation [1].

A witness validator checks that a violation can be reproduced using the information provided by the witness. Automata-based verifiers can easily be converted into validators by analyzing the synchronized product of the program with the witness automaton. In this setting, the witness automaton guides the verifier. If none of the outgoing edges on the program state match the next edge of the witness automaton, then the verifier cannot explore the current path further. If the edge on the program state matches, then the witness automaton and the program proceed to the next state, eventually leading to a violation.

* Jury member.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 418–423, 2022.
https://doi.org/10.1007/978-3-030-99527-0_24

While this idea allows one to easily convert any automata-based verifier into a validator, not all verifiers are automata-based.

DARTAGNAN is an SMT-based verifier. In the next section, we explain how to convert it into a validator. The idea is to extract information from the witness and use it to reduce the search space explored by the backend SMT solver.

2 Validation Approach

Given a concurrent program and a specification in the form of assertions, DARTAGNAN generates an SMT formula $\varphi_{\text{VER}} = \varphi_{\text{CF}} \wedge \varphi_{\text{DF}} \wedge \varphi_{\text{SC}} \wedge \varphi_{🐞}$ which is satisfiable if and only if some assertion fails [17,16]. The formulas φ_{CF} and φ_{DF} encode (respectively) the control flow and the data flow of the program. Formula φ_{SC} encodes scheduling constraints. Finally, $\varphi_{🐞}$ expresses that at least one assertion must fail. If the formula is satisfiable, then a violation exists. The goal of DARTAGNAN (as a verifier) is to find such a violation. This amounts to finding an appropriate scheduling among the threads. Such a scheduling is encoded as a *happens-before relation* between the instructions. DARTAGNAN thus searches the space of all viable happens-before relations to find a violation or prove that none exists.

We now explain how to extend DARTAGNAN into a violation witness validator. The idea is to extract from the violation witness a formula $\varphi_{👁}$ that we conjoin to the rest of DARTAGNAN's encoding, resulting in $\varphi_{\text{VAL}} = \varphi_{\text{VER}} \wedge \varphi_{👁}$. The extra constraints in $\varphi_{👁}$ reduce the search space for the SMT solver. For the verification of concurrent programs taking inputs from the environment, there are two sources of non-determinism: the *data* coming from the input (which might influence the control flow) and the *scheduling*. The purpose of $\varphi_{👁}$ is to reduce this non-determinism. Extending the SMT encoding as described in φ_{VAL} is conceptually easy. The interesting question is *"what information from the witness shall we use?"* The less information we use, the more we move from pure validation to full verification.

While automata-based validators can use some information in a straightforward manner, this is not the case for DARTAGNAN.

1. A violation witness can contain cycles to represent infinitely many executions. However, SMT-based tools unroll cycles and perform bounded verification, thus only part of this information is helpful.
2. Since DARTAGNAN (as many other BMC tools) does not keep an explicit notion of state, using state information is not trivial.

The exchange format for violation witnesses allows for expressing information about state assumptions, the control flow, and the scheduling. We abstract out from the former two and only use scheduling information. We assume that witness automata represent a single path and that the edges contain *"startline"* data corresponding to read or write instructions[1]. Those are the only instructions

[1] Our validator accepts witnesses that do not satisfy the second assumption, but it filters out the corresponding edges.

that can affect our happens-before relation. While we do not explicitly encode the outcome of control-flow instructions, certain control-flow information is implicitly encoded based on which instructions are executed. We explain the reason behind these design decisions and assumptions, discuss its limitations, and describe how we plan to improve this in the future in Section 3. Despite these limitations, and as we show in Section 4, our validator performs well in practice.

Let (S, E) be a witness automaton with states S and edges E. For each $e \in E$, function $\texttt{e2i}(\text{e})$ returns the set of read or write instructions coming from the *"startline"* in the C file that corresponds to the given edge. Since witnesses represent single paths, they can be seen as a word over S. Let $w \in S^*$ be a witness, we define the *witness-to-formula* function which constructs $\varphi_{\textcircled{\scriptsize w}}$ as

$$\texttt{w2f}(w) = \begin{cases} \mathit{true} & \text{if } w = \epsilon \\ \texttt{w2f}(w') \wedge \bigvee\limits_{\substack{i_1 \in \texttt{e2i}((\_,s)) \\ i_2 \in \texttt{e2i}((s,\_))}} \mathsf{happens\text{-}before}(i_1, i_2) & \text{if } w = s \cdot w' \end{cases}$$

3 Strengths and Weaknesses

The main strengths of our validation approach are simplicity and modularity. The approach just requires to add a new sub-formula to the SMT encoding used for verification. The validator is modular in the sense that using more or different information from the witness does not change the validation approach. For example, adding information from the witness about the control flow just requires adding more constraints to $\varphi_{\textcircled{\scriptsize w}}$.

Our validation approach assumes that witness automata represent single paths. This is a limitation not imposed by the exchange format. However, verifiers tend to stop as soon as they find one violation and thus generate witnesses representing a single violation path. A second limitation is that we do not explicitly consider control-flow information. This might impact the performance of the validation since not all non-determinism is removed and the search space might still be large. Converting such control-flow information into SMT is simple in principle. However, since Dartagnan internally converts the C program into Boogie [15], matching conditionals with the corresponding assembly-like jumps requires some work. A second consequence of not extracting control-flow information from the witness is that we might validate witnesses that do not lead to a violation. This is because we over-approximate the paths of the program represented by the witness and thus our approximation might include the path leading to the violation even if the witness did not.

4 Validation Results

We inspected the results of SV-COMP 2022 [5] to answer the following questions

RQ1: What percentage of the witnesses can Dartagnan validate?
RQ2: What percentage can Dartagnan not validate and why?

RQ3: Can Dartagnan validate witnesses that CPAchecker cannot?
RQ4: Can CPAchecker validate witnesses that Dartagnan cannot?

From the 20 verifiers in *ConcurrencySafety*, we selected five tools implementing different verification approaches. We consider them good representatives of the whole category: *(i)* **CBMC** [13] (used as a backend by Deagle [9] and Lazy-CSeq [11]), *(ii)* **CPAchecker** [7] (used as a backend by CPA-Lockator [3] and Graves [14]), *(iii)* **EBF** [2] (combines BMC with fuzzing, a very effective technique to find bugs), *(iv)* **Dartagnan** [17] (only tool where the memory model, here sequential consistency, is taken as an input), and *(v)* **GemCutter** [12] (shares the codebase with UTaipan [8] and UAutomizer [10]).

Table 1 presents the results of the validation in SV-COMP 2022. We report the number of witnesses generated by each verifier ("Witnesses"). For each of the validators (columns "Dartagnan" and "CPAchecker"), we report the number of cases where the validation conclusively finished (i.e., it returned True or False), whether the violation was confirmed (left of "/") or not (right of "/"), and the number of correct validations by one tool where the other did not report a result (columns "Dart \ CPA" and "CPA \ Dart", respectively).

Tool	Witnesses	Dartagnan	CPAchecker	Dart \ CPA	CPA \ Dart
CBMC	305	193/0	95/0	117	19
CPAchecker	256	0/0	256/0	0	256
Dartagnan	273	245/1	35/6	204	0
EBF	290	219/0	57/0	177	15
GemCutter	299	18/237	262/1	15	28

Table 1. Results of the validation in SV-COMP 2022.

For the SMT-based verifiers CBMC and EBF, Dartagnan has 63.28% resp. 75.52% success rate in the validation (against 31.15% resp. 19.66% success rate for CPAchecker). Unfortunately, it did not validate any of the witnesses generated by CPAchecker. This was due to a bug in the witness parser that has been identified and fixed after the competition. CPAchecker validated all the witnesses that it generated as a verifier. Dartagnan validated 89.74% of its own witnesses while CPAchecker only validated 12.82%. For GemCutter, the validation success of Dartagnan is only 6.02%. This is because, due to another bug, it wrongly marked 237 witnesses as not validated. The fixed version of Dartagnan is able to validate all such cases. Despite this, from the 18 witnesses that Dartagnan validated, 15 of them were not validated by CPAchecker, thus improving the validation possibilities for the overall competition.

5 Software Project and Configuration

The project home page is https://github.com/hernanponcedeleon/Dat3M. To run Dartagnan as a validator, use the following command:

```
$ Dartagnan-SVCOMP.sh -witness <witness> <property> <program>
```

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [5] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [6].

References

1. Exchange Format for Violation Witnesses and Correctness Witnesses. https://github.com/sosy-lab/sv-witnesses.
2. Fatimah Aljaafari, Lucas C. Cordeiro, Mustafa A. Mustafa, and Rafael Menezes. EBF: A hybrid verification tool for finding software vulnerabilities in iot cryptographic protocols. *CoRR*, abs/2103.11363, 2021.
3. Pavel S. Andrianov, Vadim S. Mutilin, and Alexey V. Khoroshilov. cpalockator: Thread-modular analysis with projections - (Competition Contribution). In *TACAS (2)*, volume 12652 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 2021. doi:10.1007/978-3-030-72013-1_25.
4. Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *TACAS*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015. doi:10.1007/978-3-662-46681-0_31.
5. Dirk Beyer. Progress on software verification: SV-COMP 2022. In *TACAS (2)*. Springer, 2022.
6. Dirk Beyer. Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo, 2022. doi:10.5281/zenodo.5959149.
7. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. doi:10.1007/978-3-642-22110-1_16.
8. Daniel Dietsch, Matthias Heizmann, Alexander Nutz, Claus Schätzle, and Frank Schüssele. Ultimate Taipan with symbolic interpretation and fluid abstractions - (Competition Contribution). In *TACAS (2)*, volume 12079 of *Lecture Notes in Computer Science*, pages 418–422. Springer, 2020. doi:10.1007/978-3-030-45237-7_32.
9. Fei He, Zhihang Sun, and Hongyu Fan. Deagle: An SMT-based verifier for multithreaded programs (Competition Contribution). In *TACAS (2)*. Springer, 2022.
10. Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate Automizer and the search for perfect interpolants - (Competition Contribution). In *TACAS (2)*, volume 10806 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2018. doi:10.1007/978-3-319-89963-3_30.
11. Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-CSeq: A lazy sequentialization tool for C - (Competition Contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 398–401. Springer, 2014. doi:10.1007/978-3-642-36742-7_46.
12. Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. Ultimate GemCutter and the axes of generalization (Competition Contribution). In *TACAS (2)*. Springer, 2022.

13. Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (Competition Contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014. doi:10.1007/978-3-642-54862-8_26.
14. William Leeson and Matthew Dwyer. GraVeS: Graph-based verifier selector (Competition Contribution). In *TACAS (2)*. Springer, 2022.
15. K. Rustan M. Leino. This is Boogie 2. 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.
16. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In *SAS*, volume 10422 of *LNCS*, pages 299–320. Springer, 2017. doi:10.1007/978-3-319-66706-5_15.
17. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (Competition Contribution). In *TACAS (2)*, volume 12079 of *LNCS*, pages 378–382. Springer, 2020. doi:10.1007/978-3-030-45237-7_24.

Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution) *

Fei He[1,2,3](✉), Zhihang Sun[1,2,3], and Hongyu Fan[1,2,3]

[1] School of Software, Tsinghua University, Beijing, China
[2] Key Laboratory for Information System Security, MoE, Beijing, China
[3] Beijing National Research Center for Information Science and Technology, Beijing, China

Abstract. `Deagle` is an SMT-based multi-threaded program verification tool. It is built on top of `CBMC` (front-end) and `MiniSAT` (back-end). The basic idea of `Deagle` is to integrate into the SMT solver an ordering consistency theory that handles ordering relations over the shared variable accesses in the program. The front-end encodes the input program into an extended propositional formula that contains ordering constraints. The back-end is reinforced with a solver for the ordering consistency theory. This paper presents the basic idea, architecture, installation, and usage of `Deagle`.

Keywords: Program verification · Satisfiability modulo theories · Concurrency.

1 Verification Approach

Given a multi-threaded program, the thread communication behaviors can be modeled using the *happens-before* relations over memory access (read/write) events [1]. There are various kinds of happens-before relations: program order (PO), read-from order (RF), write serialization order (WS), and from-read order (FR). A *happens-before ordering formula* (abbreviated as *ordering formula*) is a logical formula that involves only memory access events and happens-before relations.

`Deagle` is an SMT-based multi-threaded program verifier, which consists of

- a front-end that encodes the intra-threaded behaviors (e.g., the control and data flow per thread) into propositional formulas, and the inter-threaded behaviors (i.e., the communication between threads) into *ordering formulas*;
- a back-end that extends `MiniSAT` with an *ordering consistency theory* solver [8] by following the DPLL(T) framework [7], and is able to solve propositional formulas and ordering formulas mixed together.

* This work was supported in part by the National Key Research and Development Program of China (No. 2018YFB1308601) and the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 424–428, 2022.
https://doi.org/10.1007/978-3-030-99527-0_25

Compared with [8]: The theory solver in [8] uses a *from-read axiom* to derive *FR* orders. Besides the *from-read axiom*, `Deagle` also implements a *write-serialization axiom* [11], with which *WS* orders can also be derived. In return, the front-end of `Deagle` need not encode both *FR* and *WS* orders explicitly.

2 Software Architecture

`Deagle` is developed on top of `CBMC` [9] and `MiniSAT` [6] using C++. Additionally, for ease of usage and debugging, `Deagle` reuses some modules developed in `Yogar-CBMC` [10,11]. `Deagle` is not a strategy selection-based verifier. `Deagle` runs the following procedures successively to verify a given C program:

Preprocessing (from `Yogar-CBMC`) For each global structure variable in the C program, the preprocessing procedure unfolds it by creating a fresh variable for each member. Note that arrays need no preprocessing; `CBMC` is able to handle each array as an entity.

Parsing and Goto-Program Generation (originally in `CBMC`) `CBMC` employs `Flex` and `Bison` to transform the preprocessed C program into an *abstract syntax tree* (*AST*). Then `CBMC` builds a *goto program*, where all branching statements and loop statements are represented with (conditional) goto statements.

Library Function Modeling (extended from `CBMC`) `CBMC` models each multithreading-related library function (e.g., *pthread_cond_wait*). For example, mutex m contains a Boolean variable *m_locked* indicating whether m is locked; *pthread_mutex_lock*(&m) assumes *m_locked* to be originally $false$ and sets *m_locked* to *true*. Based on `CBMC`, we extend `Deagle` to support the modeling of more library functions.

Unwinding We employ *bounded model checking* (*BMC*) [3,4,5] to handle loops. If the program contains loops, we determine an *unwinding limit* and unwind the program to a loop-free bounded program:

- If the maximal loop time of the program can be determined through static analysis, e.g.,
$$for\ (i = 0; i < 10; i++)$$
we set the unwinding limit to this maximal loop time;
- If the maximal loop time depends on non-determinism. e.g.,
$$for\ (i = 0; i < n; i++)$$
where n is attained from the function *__VERIFIER_nondet_int*, we report UNKNOWN since such loops cannot be fully unwound.
- Otherwise, we set the unwinding limit to 2.

Formula Generation (extended from `CBMC`) After unwinding, the loop-free program is represented in the *static single assignment* (*SSA*) form, where each thread is a chain of assignments. These assignments can be directly modeled into first-order logic formulas (for ease of solving, we further convert them into propositional logic formulas). Additionally, an assignment may contain global memory access events; we model program orders and read-from orders (please refer to [8] for more information) of these events into the formulas.

Constraint Solving (extended from `MiniSAT`) We develop an ordering consistency theory solver and integrate it into the DPLL(T) framework [8]. For efficiency, we extend `MiniSAT`, an SAT-based solver, to run our theory solver exclusively. Please refer to [8] for the detailed algorithms of our decision procedure.

Witness Generation (adapted from `Yogar-CBMC`) If the back-end solver returns *satisfiable* (i.e., finds a counterexample violating the property), our ordering consistency theory solver reports a sequence (total order) of these events, which can be used for generating the witness of the counterexample.

3 Strengths and Weaknesses

Compared to the traditional method [1] which explicitly converts ordering formulas into propositional formulas, `Deagle` employs a dedicated theory solver to handle ordering formulas, which improves both time and space efficiency. Ignoring some tasks in *goblint-regression* that require unwinding 10000 times, `Deagle` reports TIMEOUT in only 9 tasks and OUT OF MEMORY in only 7 tasks – fewer than most `ConcurrencySafety` competitors.

In most *weaver* tasks (117 out of 169), the number of loop iterations is nondeterministic. As is mentioned in previous section, `Deagle` reports UNKNOWN. Since such tasks are common in real-world programs, we are exploring an approach to dealing with such programs in the future work.

4 Tool Setup and Configuration

The source code of `Deagle 1.3` (the submitted version in `SV-COMP 2022` [2]) is publicly accessible [4]. Please refer to README for more installation instructions. In `SV-COMP 2022`, `Deagle` participates in `ConcurrencySafety` category and only checks property `Unreach-Call` [5]. By setting parameters

$$--32--no-unwinding-assertions--closure$$

one can reproduce `Deagle`'s results of `SV-COMP 2022`.

[4] `Deagle` repository: https://github.com/thufv/Deagle

[5] The benchmark definition of `Deagle`: https://gitlab.com/sosy-lab/sv-comp/bench-defs/-/blob/main/benchmark-defs/deagle.xml

4.1 Parameter Definition

`Deagle` inherits lots of parameters from `CBMC`. Due to the page limit, we only describe parameters related to the competition or newly added in `Deagle`:

* $--32/--64$: sets the width of integers to 32/64.
* $--no-unwinding-assertions$: does not generate *unwinding assertions* into the formula. Assuming a loop is unwound n times, its unwinding assertion asserts the loop condition to be $false$ after n iterations. Since unwinding assertions can lead to false counterexamples, we disable the generation of unwinding assertions.
* $--closure/--icd$ (new in `Deagle`): uses our proposed approach. Once the parameter $--closure$ is enabled, `Deagle` employs a transitive closure-based theory solver (recommended). If $--icd$ is enabled, `Deagle` employs an incremental cycle detection-based solver. In `SV-COMP 2022` [2], `Deagle` solves all tasks with the parameter $--closure$.

5 Software Project

`Deagle` is developed by Fei He, Zhihang Sun, and Hongyu Fan from the Formal Verification Lab[6] in Tsinghua University. `Deagle` is licensed under GPLv3. Since `Deagle` is developed over `CBMC` and `MiniSAT`, and reuses some modules from `Yogar-CBMC`, it also contains copyright of those tools.

6 Acknowledgement

We appreciate `SV-COMP` hosts for holding the competition and giving advice on participating. We are also grateful to developers, maintainers, and contributors of `CBMC`, `MiniSAT`, and `Yogar-CBMC`, on which `Deagle` is based.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 141–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.5555/2958031.2958083
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
3. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proceedings of the 36th Annual ACM/IEEE Design Automation Conference. p. 317–320. DAC '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/309847.309942

[6] homepage: https://thufv.github.io/team

4. 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). https://doi.org/10.1007/3-540-49059-0_14
5. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (Jul 2001). https://doi.org/10.1023/A:1011276507260
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing. pp. 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/3-540-49059-0_14
7. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: CAV (2004). https://doi.org/10.1007/978-3-540-27813-9_14
8. He, F., Sun, Z., Fan, H.: Satisfiability modulo ordering consistency theory for multi-threaded program verification. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 1264–1279. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454108
9. Kroening, D., Tautschnig, M.: CBMC–C bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
10. Yin, L., Dong, W., Liu, W., Li, Y., Wang, J.: Yogar-CBMC: CBMC with scheduling constraint based abstraction refinement. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 422–426. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_25
11. Yin, L., Dong, W., Liu, W., Wang, J.: Scheduling constraint based abstraction refinement for multi-threaded program verification. IEEE Transactions on Software Engineering **PP** (08 2017). https://doi.org/10.1109/TSE.2018.2864122


The Static Analyzer Frama-C in SV-COMP (Competition Contribution)

Dirk Beyer ✉ and Martin Spiessl

LMU Munich, Munich, Germany

Abstract. Frama-C is a well-known platform for source-code analysis of programs written in C. It can be extended via its plug-in architecture by various analysis backends and features an extensive annotation language called ACSL. So far it was hard to compare Frama-C to other software verifiers. Our competition participation contributes an adapter named Frama-C-SV, which makes it possible to evaluate Frama-C against other software verifiers. The adapter transforms standard verification tasks (from the well-known SV-Benchmarks collection) in a way that can be understood by Frama-C and produces a verification witness as output. While Frama-C provides many different analyses, we focus on the Evolved Value Analysis (EVA), which uses a combination of different domains to over-approximate the behavior of the analyzed program.

Keywords: Software verification · Program analysis · Formal methods · Competition on Software Verification · Comparative Evaluation · SV-COMP · Frama-C

1 Approach

This competition contribution is based on Frama-C [12], a program-analysis platform for C programs. The purpose of the participation in the comparative evaluation SV-COMP is to show the strengths of Frama-C when applied to the problem of verifying C programs from the SV-Benchmarks [4] collection of verification tasks.

2 Architecture

Although Frama-C has a large configuration space, it does not support standard specifications as used in SV-COMP, and it does not produce verification witnesses as default. In order to overcome this obstacle we implemented an adapter for Frama-C using input and output transformers, and the adaption architecture is illustrated in Fig. 1. In the following, we describe the artifacts and actors of the participating verifier: in Sect. 2.1 we describe all the components that are developed as part of the adapter, while in Sect. 2.2 we describe in more detail how the used EVA analysis of Frama-C works.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 429–434, 2022.
https://doi.org/10.1007/978-3-030-99527-0_26

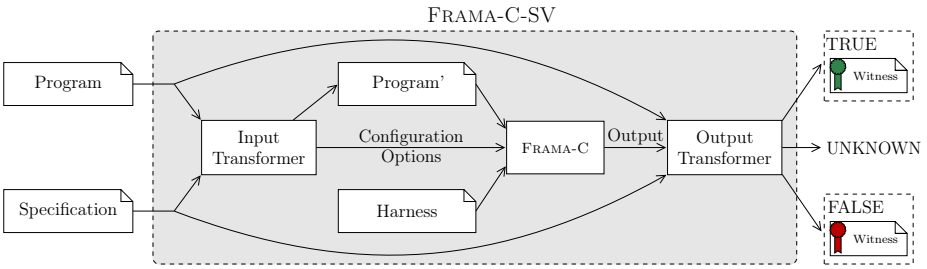


Fig. 1: Architecture of Frama-C-SV: the inputs and outputs of Frama-C are translated to interface with the established standards as used by SV-COMP; the components that are necessary to adapt Frama-C for comparison with other verifiers amount to 678 lines of code mostly written in Python

2.1 Frama-C-SV

Input Transformer. The input transformer takes the program p and specification s and creates a new program p' in which the specification s has been expressed as Frama-C-specific annotations. Frama-C uses ACSL [1] as language to specify annotations. The input transformer also selects configuration parameters for Frama-C that are best suited for the verification task. Currently we encode reachability tasks into signed integer overflows by adding an artificial overflow to the body of the function `reach_error`. This works well in practice and is also sound, since if there were any other overflows, the task would contain undefined behavior and would not be a valid reachability task in the first place.

Configuration Options. Depending on the input program and specification, we can choose different options that are passed to Frama-C. In essence, this acts like an algorithm selection [14] and, e.g., allows us to choose a different configuration of Frama-C depending on the specified property.

Harness. Some programs in the SV-Benchmarks collection use specific functions to model non-determinism. We provide implementations for those functions (`__VERIFIER_*`) in a separate C program such that the semantics of those functions can be understood by Frama-C. This separate C program is passed to Frama-C together with the transformed program p'.

Output Transformer. The output of Frama-C needs to be interpreted regarding the original specification, and depending on the outcome, a verification witness needs to be generated. Thus, we need an output transformer for (a) providing a verdict for the verification task and (b) providing a verification witness. Regarding (a), the output transformer interprets the CSV report that can be generated by Frama-C to determine whether the program was proven to be safe (verdict `TRUE`), whether a specification violation occurred (verdict `FALSE`), or whether no such statement can be made (verdict `UNKNOWN`). We also generate a minimal correctness or violation witness for the verdicts `TRUE` and `FALSE`, respectively.

The witness automata consist of only one node, which for violation witnesses is marked as violation node. In the future we plan to augment these witnesses with information such as invariants that have been found by Frama-C.

2.2 Frama-C

One of the strengths of Frama-C is its modular architecture [10], which allows a configuration of the best possible analysis backends for a certain verification problem. We choose the plug-in EVA [9], which is well suited for an automatic analysis. Other plug-ins such as the Weakest-Preconditions (WP) plug-in require hints from the user in order to be effective. In the following we will briefly describe the most important aspects of the EVA analysis configuration that we use. For a more detailed description, we refer the reader to the relevant literature [7, 8, 9].

Frama-C provides a meta-option called `-eva-precision` for the EVA plug-in with possible values ranging from 0 to 11. With higher values for this option more precise domains and thresholds are used, at the cost of increased computation time. We currently use the maximum value of 11 in order to make the best use of the 900 s CPU time limit. In the future we might want to iteratively increase this value starting at lower precisions.

Domains. The EVA analysis always uses the domain *cvalue*, which tracks values of variables either as constant values, sets, or intervals of possible values (including modular congruence constraints). For pointer addresses, these are either tracked as addresses with offsets or as so-called garbled mix, which overapproximates the set of possible memory locations. In addition, depending on the precision level, various other domains are used that we describe in the following. The domain *symbolic-locations* tracks a map of symbolic locations to values, which is, e.g., helpful for analyzing expressions containing array accesses such as `a[i]<a[j]`. The *equality* domain tracks equalities of C expressions found in the code, whereas the *gauges* domain tracks relations between variables in a loop with the goal to discover linear inequality invariants [16]. Lastly the *octagon* domain tracks certain linear constraints between pairs of variables [13]. As we use the highest precision level, all of these domains are used in our contribution.

Precision of the State-Space Exploration. Apart from the domains, the precision of state-space exploration in Frama-C is affected by various options. We will describe some of these in the following; a complete list of affected settings and values is always printed by Frama-C when the option *eva-precision* is specified by the user. Option *slevel* (set to 5 000) determines how many separate states are kept before new states will be joined into existing ones. Option *ilevel* (set to 256) determines how many different values are tracked per variable before overapproximating the value range. Option *plevel* (set to 2 000) affects the size up to which arrays are tracked. The option *auto-loop-unroll* (set to 1 024) will determine up to which bound a loop is considered for unrolling.

3 Strengths and Weaknesses

The competition contribution shows the strengths of Frama-C in checking C programs for overflows and also —in the currently supported sub-categories [1]— for reachability. Here we are able to show that our results are comparable and often surpass those of other tools based on abstract interpretation [11] such as Goblint [15]. While the EVA analysis of Frama-C that we use is based on abstract interpretation, the precision options described in Sect. 2.2 allow for a more precise state-space exploration, which behaves more like model checking. More details about the results can be found in the competition report [2] and artifact [3].

The approach that we describe in this paper creates a compatibility layer between the abilities used by Frama-C and the standards used in the SV-Benchmarks collection. While still a work in progress, we have shown that it is possible to bridge this gap while preserving overall soundness. It is also interesting to consider the results on verification tasks from the SV-Benchmarks collections for a tool that did not participate before.

Although our approach is sound in general, we are likely not showcasing the full potential of Frama-C. One aspect to consider here is the large configuration space, which means there might be ways to verify more tasks with a better heuristic for selecting the configuration options. The other aspect is that Frama-C also provides different plug-ins such as the WP plug-in, which requires more (manual) annotations, but can also potentially solve more tasks than the more automatic EVA plug-in.

4 Software Project and Contributors

The software project Frama-C is developed at https://git.frama-c.com/pub/frama-c/ and our adapter Frama-C-SV is developed at https://gitlab.com/sosy-lab/software/frama-c-sv, both being released under open-source licenses. The exact version of the adapter that participated in SV-COMP 2022 is also archived in the competition's tool-archive repository [2] [6]. Frama-C was funded by the European Commission in program Horizon 2020. The adapter Frama-C-SV was funded by the DFG. We thank the Frama-C authors [3] for their contribution to the software-verification community.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [2] and available on the competition web site. This includes the verification tasks [4], competition results [3], verification witnesses [5], scripts, and instructions for reproduction. The version of Frama-C-SV as used in the competition is archived together with other participating tools [6].

Funding Statement. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

[1] We opted out of subcategories with unsound results caused by Frama-C making assumptions that are different from the conventions of SV-COMP.

[2] https://gitlab.com/sosy-lab/sv-comp/archives-2022/blob/svcomp22/2022/frama-c-sv.zip

[3] https://frama-c.com/html/authors.html

References

1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
3. Beyer, D.: Results of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831008
4. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
5. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5838498
6. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
7. Blazy, S., Bühler, D., Yakobowski, B.: Structuring abstract interpreters through state and value abstractions. In: Proc. VMCAI. pp. 112–130. LNCS 10145, Springer (2017). https://doi.org/10.1007/978-3-319-52234-0_7
8. Bühler, D.: Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C. Ph.D. thesis, University of Rennes 1, France (2017), available at https://tel.archives-ouvertes.fr/tel-01664726
9. Bühler, D., Cuoq, P., Yakobowski, B., Lemerre, M., Maroneze, A., Perelle, V., Prevosto, V.: Eva: The Evolved Value Analysis plug-in (2020), available at https://frama-c.com/download/frama-c-eva-manual.pdf
10. Correnson, L., Cuoq, P., Kirchner, F., Maroneze, A., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C user manual (2020), available at https://frama-c.com/download/frama-c-user-manual.pdf
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL. pp. 238–252. ACM (1977)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
13. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
14. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
15. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints (competition contribution). In: Proc. TACAS (2). pp. 438–442. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28
16. Venet, A.: The gauge domain: Scalable analysis of linear inequality invariants. In: Proc. CAV. pp. 139–154. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_15


GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution)*

Malte Mues (✉)[1] and Falk Howar[1,2]

[1] TU Dortmund University, Dortmund, Germany
{malte.mues, falk.howar}@tu-dortmund.de
[2] Fraunhofer ISST, Dortmund Germanny

Abstract. GDart is an ensemble of tools allowing dynamic symbolic execution of JVM programs. The dynamic symbolic execution engine is decomposed into three different components: a symbolic decision engine (DSE), a concolic executor (SPouT), and a SMT solver backend allowing meta-strategy solving of SMT problems (JConstraints). The symbolic decision component is loosely coupled with the executor by a newly introduced communication protocol. At SV-COMP 2022, GDart solved 471 of 586 tasks finding more correct false results (302) than correct true results (169). It scored fourth place.

Keywords: Dynamic Symbolic Execution · Software Verification

1 Verification Approach

This paper presents the GDart ensemble tool, a dynamic symbolic execution engine for the JVM. Dynamic symbolic execution is a well-established technique for software testing (cf. DART [6]) and there have been already two contestants to SV-COMP 2021 using this technique (cf. JDart [7, 9] and COASTAL[3]). It is a search algorithm for systematic exploration of a program's state space for a property violation which either stops after exhausting the resource limits, exploring the complete symbolic state space, or encountering an error. The end of the search is fully configurable in GDart.

In SV-COMP 2022 [3], a dynamic symbolic execution tool (JDart (714 Points)) wins the Java track for the first time beating JBMC (700 Points) [4], a bounded model checker for Java, and Java Ranger (670 Points) [11], a symbolic execution engine extended by veritesting [1] for Java. JDart's result underlines the potential of dynamic symbolic execution for the verification of Java programs in general. The concrete implementation of JDart is closely coupled to the Java PathFinder VM (JPF-VM) [12] running the complete analysis within one virtual machine. The advantage of the JPF-VM is that it runs

* This work has been partially founded by an Amazon Research Award
[3] https://github.com/DeepseaPlatform/coastal

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 435–439, 2022.
https://doi.org/10.1007/978-3-030-99527-0_27

as a guest JVM on top of a host JVM. The analysis might mock parts of the guest JVM and use the host JVM for running side computation required to compute results used in the mock. The downside of the JPF-VM is its research tool status and that it is costly to maintain it given JAVA's fast pace in releasing new features.

COASTAL demonstrated for the first time what a loosely coupled architecture between the symbolic exploration engine and a concolic execution engine might look like. It instruments the bytecode with ASM[4], a java bytecode manipulation framework, to obtain symbolic traces. This makes the analysis independent of the JPF-VM. The downside is that bytecode manipulation offers less flexibility than hooking directly into the JVM.

2 Software Architecture

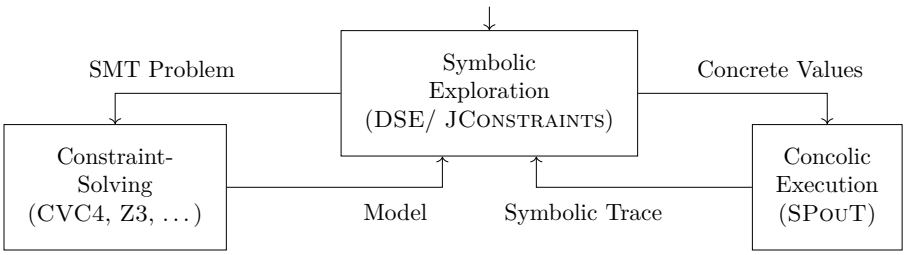


Fig. 1: GDart's ensemble architecture and the interplay between the components.

GDart takes the strengths of JDart's mocking flexibility and combines it with COASTAL's modular design. Figure 1 demonstrates the architecture of the GDart ensemble tool. The main analysis component is the symbolic explorer. It orchestrates the concolic executor and requests solutions for SMT problems from the constraint solvers powering the symbolic exploration.

Symbolic Exploration. We name the symbolic explorer "DSE" component as it does symbolic exploration and starts the concolic executor, the two main steps in applying dynamic symbolic execution. It manages the constraint tree and guides its exploration. Both steps together are the main tasks of a dynamic symbolic execution engine. To explore a path, it computes a set of concrete values that drives the concolic executor down the path of interest and seeds the executor with these values. After the termination of the executor, it parses the obtained symbolic trace and integrates it into the symbolic tree. Next, it constructs from the symbolic tree a SMT problem that describes the next path to explore and starts a constraint solver to get a model suitable to drive the execution down this path or an unsatisfiable verdict implying that the path is unreachable. The

[4] https://asm.ow2.io

search behavior of GDart is configured in the DSE. Once the search terminates, DSE generates a verification witness from the constraint tree.

Concolic Executor. One of the core contributions of GDart is the new concolic executor SPouT implemented as part of the Espresso guest language running on top of the GraalVM [13][5]. The GraalVM is an industrial-grade JVM maintained by Oracle allowing to use most of the architectural benefits the JPF-VM offered apart from state tracking. But concolic execution does not require JPF-VM's state tracing feature. SPouT can be seeded with concrete values to drive down the execution along a concrete path. In addition, it can introduce new symbolic variables for previously unknown inputs. During execution, it records manipulation and constraints checks on symbolic variables and reports a symbolic execution trace together with the concrete execution result on termination of the path exploration. Decisions on the symbolic variables are encoded in the SMT-Lib format. As SPouT maintains the two VM layers, it allows mocking of behavior in the Espresso VM running the analysis and implements a substitute executed on the host GraalVM during concolic execution the same way JDart does for mocking the environment if needed. The feature is also used for intercepting invocations of the string library in Java and encoding them symbolically.

Constraint Solving. The third component is constraint solving. DSE uses the JConstraints library to model SMT-Lib constraints internally and interact with the solver. GDart is backed by CVC4 [2] and Z3 [5]. We combine these two SMT solvers in a portfolio approach according to the CvcSeqEval strategy presented in our previous work [8].

3 Strengths and Weaknesses

GDart is the fourth place with 640 points behind JDart (714 points), JBMC (700 points), and Java Ranger (670 points). Dynamic symbolic execution tools tend to be stronger in finding property violations than confirming the absence of property violations on the SV-COMP benchmark. This is partially by design as some of the problems (e.g., those problems in the jayhorn-recursive subgroup) aim for testing the handling of tremendously large and hard to explore state spaces. GDart disproves the property in 302 cases and confirms it in 169 cases. In total, GDart answered 471 of 586 tasks correctly and none incorrect. These are 40 more correct false proved tasks than Java Ranger found (262 correct false tasks out of 466 solved tasks). In total GDart solved five more tasks than Java Ranger and 35 less than JBMC.

In direct comparison with GDart, JDart solved 192 (+23) correct true tasks and 330 (+28) correct false tasks. Three factors are contributing to the gap between GDart and JDart: the performance overhead of spinning up one JVM per executor run (We do not have the exact number, but spinning up a JVM

[5] https://www.graalvm.org

costs at least 500 ms per JVM affecting especially tasks with huge exploration trees.), technical maturity of the implementation as JDart is around for more time, and a value tracing heuristic built into JDart for tracking numerical values origin from a serialized string representation not built into GDart. The performance overhead for spinning up multiple JVMs is the only drawback that is influenced by the modular design of GDart and will not go away in the future. JDart's time per task after archiving 600 points is close to five seconds CPU time in the score-based quantile plots for CPU time while GDart's time per task reaches close to 50 seconds CPU time for the same score.

The weakness of dynamic symbolic execution is state space explosion which also affects GDart. Slowing down each executor run by spinning up new VMs is a disadvantage given the resource constraints of SV-COMP. On the bright side, with more relaxed resource limits it is possible to run the execution runs in parallel to the symbolic exploration of the constraints tree as future work for the DSE component allowing parallel breadth-first search on multi-core machines. At the moment all paths are explored sequentially.

4 Tool Setup

GDart is run with various configuration options hard-coded into the SV-COMP run scripts. More precisely, we enabled witness generation, used the described solver strategy in the constraint backend, chose a breadth-first search on the constraint tree, and used the same bounded solving as JDart. The search is configured to terminate on the first hit assertion error.

5 Software Project

The components are currently all developed at TU Dortmund by the group led by Falk Howar. DSE[6] is available under the Apache 2.0 license, JConstraints[7] as well, and SPouT[8] is available under the GPL v2 license. We also provide the run scripts for SV-COMP on GitHub[9].

6 Data Availability Statement

The GDart archive used for SV-COMP 2022 is available at Zenoodo [10].

References

1. Avgerinos, T., Rebert, A., Cha, S.K., Brumley, D.: Enhancing symbolic execution with veritesting. In: Proc. ICSE. pp. 1083–1094 (2014). https://doi.org/10.1145/2568225.2568293

[6] https://github.com/tudo-aqua/dse
[7] https://github.com/tudo-aqua/jconstraints
[8] https://github.com/tudo-aqua/spout
[9] https://github.com/tudo-aqua/gdart-svcomp

2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. CAV. pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
4. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Proc. TACAS. pp. 219–223. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_17
5. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM (2005). https://doi.org/10.1007/978-3-642-19237-1_4
7. Luckow, K., Dimjaevi, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamari, Z., Raman, V.: JDart: A dynamic symbolic analysis framework. In: TACAS 2016 (2016). https://doi.org/10.1007/978-3-662-49674-9_26
8. Mues, M., Howar, F.: Data-driven design and evaluation of SMT meta-solving strategies: Balancing performance, accuracy, and cost. In: Proc. ASE. pp. 179–190 (2021). https://doi.org/10.1109/ASE51524.2021.9678881
9. Mues, M., Howar, F.: JDart: Portfolio solving, breadth-first search and smt-lib strings. In: Proc. TACAS (2021). https://doi.org/10.1007/978-3-030-72013-1_30
10. Mues, M., Howar, F.: Gdart artifact for sv-comp 2022 (Feb 2022). https://doi.org/10.5281/zenodo.5957294
11. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W.: Java Ranger: Statically summarizing regions for efficient symbolic execution of Java. In: Proc. ESEC/FSE 2020. pp. 123–134 (2020). https://doi.org/10.1145/3368089.3409734
12. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (Apr 2003). https://doi.org/10.1023/A:1022920129859
13. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Proc. SPLASH. pp. 187–204 (2013)

Graves-CPA: A Graph-Attention Verifier Selector (Competition Contribution)

Will Leeson(✉) and Matthew B. Dwyer

University of Virginia, Charlottesville VA 22903, USA
{will-leeson,matthewbdwyer}@virginia.edu

Abstract. Graves-CPA is a verification tool which uses algorithm selection to decide an ordering of underlying verifiers to most effectively verify a given program. Graves-CPA represents programs using an amalgam of traditional program graph representations and uses state-of-the-art graph neural network techniques to dynamically decide how to run a set of verification techniques. The Graves technique is implementation agnostic, but it's competition submission, Graves-CPA, is built using several CPAchecker configurations as its underlying verifiers.

Keywords: Software Verification · Graph Attention Networks · Graph Neural Networks · Algorithm Selection

1 Verification Approach

Graves-CPA is an algorithm selector for software verification based on graph neural network techniques. As the tool PeSCo [14] has shown, dynamic ordering of verification techniques can result in faster and more accurate verification. Computing an ordering on techniques dynamically will incur some runtime, but an effective ordering will oftentimes make this overhead insignificant in comparison to the time saved by using a more appropriate technique. Like most algorithm selectors, Graves-CPA uses machine learning to make its selections. However, it uses graph neural networks (GNNs) so it can represent programs using traditional program abstractions, such as abstract syntax trees (ASTs). Graves-CPA uses a variant of GNNs called Graph Attention Networks (GATs) [16]. GATs use a learned attention mechanism which is trained to learn the importance of edges in a given graph.

GNNs are an emerging field in machine learning. Traditional neural networks accept input vectors, which have a fixed size and a natural ordering on elements, but graphs, in general, have neither. GNNs avoid these issues by operating on individual nodes in the graph, instead of the graph as a whole [15]. Typically, the input to a GNN is the current representation of a node v and a collation of the representations of its neighboring nodes. The output is then a new representation for v. This process is repeated independently for all nodes in the graph. Thus, the number of nodes in the graph and order in which they are processed is irrelevant.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 440–445, 2022.
https://doi.org/10.1007/978-3-030-99527-0_28

The Graves technique is tool agnostic [11], meaning it can be trained to select from any set of verifiers. Our competition contribution selects an ordering from the techniques utilized by CPAchecker [3], similar to PeSCo in previous competitions.

To form its selection, Graves-CPA produces a graph representation of a given program, G, which is based on its AST with control flow, data flow, and function call and return edges added between the tree's nodes. The AST's nodes and edges ensure the semantics of the statements in the program are maintained. Control flow edges maintain the branching and order of execution between these statements. Data flow edges explicitly relate the definitions, uses, and interactions of values in the program. G is passed to a GNN, consisting of a series of GATs, which outputs a graph feature vector This feature vector is finally passed to a fully connected neural network which decides the sequence in which Graves-CPA's suite of verification techniques are run.

2 System Architecture

2.1 Graph Generation

To generate a graph from a program, Graves-CPA relies on the AST produced by the C compiler Clang [10]. Using a visitor pattern [9], Graves-CPA walks the AST to generate data flow edges and the edges of the program's Interprocedural Control Flow Graph (ICFG). Function call and return edges in the ICFG are those which can be determined purely syntactically. Using the ICFG and data flow edges, Graves-CPA produces additional data flow edges using the worklist reaching definition algorithm [1]. We limit the number of iterations of the reaching definition algorithm, making our data edges an under-approximation of possible data flow edges. Once this graph is generated, it is parsed into a list of nodes and several edge sets. Nodes represent the AST token which corresponds to them using a one-hot encoding. These nodes and edges are used as input to the GNN.

2.2 Prediction

To form a prediction, Graves-CPA uses a GNN, visualized in Figure 1, which consists of 2 GAT layers, a jumping knowledge layer [17], and an attention-based pooling layer [12]. The GAT layers are crucial to our technique. When propagating data through the graph, the attention mechanisms in each layer weights edges so information important to predictions is more prominent than superfluous data.

The jumping knowledge layer concatenates intermediate graph representations, denoted by A, B, and C, allowing the model to learn from each representation. The attention-based pooling layer calculates an attention value for each node in the graph. All nodes are weighted by their respective attention values and then summed together to form a graph feature vector. The combination of

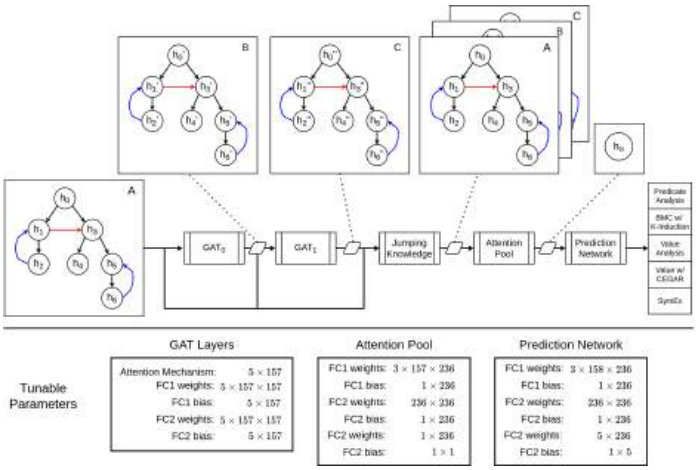


Fig. 1. GRAVES' uses a GNN comprised of 2 GAT layers, a Jumping Knowledge layer, and attention pooling layer. These layers produce a graph feature vector which a 3 layer prediction network uses to order verifiers for sequential execution. An in depth description of this architecture can be found in Leeson et al. [11].

GAT layers and the attention-based pool allows the network to weigh the importance of both edges and nodes when forming the graph feature vector. This feature vector is fed to a three layer neural network which decides the sequence of tool execution.

GRAVES-CPA was trained using data collected from running 5 configurations of the CPAchecker framework on the verification tasks from SV-COMP 2021. Labels for each configuration come from the SV-COMP score the configuration would receive for a given program minus a time penalty. Similar to CPAchecker's competition contribution, these configurations are symbolic execution [6], value analysis [7], value analysis with CEGAR [7], predicate analysis [5], and bounded model checking with k-induction [4]. To prevent GRAVES-CPA from overfitting to the SV-COMP benchmarks, we train on a subset of the dataset, only utilizing 20% of it. Like previous iterations of PeSCo, the network is trained to rank the configurations in the order in which they should be executed.

GRAVES-CPA uses the machine learning libraries PyTorch [13] and PyTorch-Geometric [8], an extension of PyTorch for graphs and other irregularly shaped data, to implement its machine learning components. GRAVES-CPA is implemented using a combination of Python, C++, and Java.

2.3 Execution

Using the ordering produced by the previous step, CPAchecker is run in a sequential fashion with each verification configuration. If a technique goes past a given time limit or fails to produce a result, the next technique is executed.

3 Strengths and Weaknesses

Graves-CPA operates on program graphs which are an abstraction of the program. Its underlying model uses this abstraction to learn what software patterns a particular verification technique excels at handling. This allows Graves-CPA to produce a dynamic ordering which should run techniques more equipped to the given problem first, reducing run time. In [11], the authors perform a qualitative study which suggests the network learns to rank verification techniques using program features an expert would use to decide between techniques.

In SV-COMP 2022 [2], there were 4,548 problems both Graves-CPA and CPA-checker reported the correct result. Graves-CPA's dynamic selection of CPA-checker's static configuration ordering allowed it to solve these problems 37 hours faster. Further, Graves-CPA was able to solve 142 problems that CPAchecker could not, due to resource constraints or other issues.

Machine learning relies on the fact that training data is representative of the real world. If this is not the case, the model can easily make poor predictions. These poor decisions can be seen in competition in the 559 instances where Graves-CPA chooses an ordering that doesn't produce the correct result, but CPAchecker does. In most of these instances, Graves-CPA runs out of resources or incorrectly predicts the remaining techniques will not produce a correct result.

4 Tool Setup and Configuration

Graves-CPA is built on the PeSCo codebase, which in turn is built on the CPAchecker codebase, and participates in the ReachSafety and Overall categories. It can be downloaded as a fork: https://github.com/will-leeson/cpachecker. Graves-CPA requires cmake, LLVM, either make or ninja, and ant (a CPAchecker dependency) to be built and the python libraries PyTorch and PyTorch-Geometric to be executed. To build the project, simply run the shell script `setup.sh` and add our graph generation tool, `graph-builder`, to your path. Now, you may verify a program with Graves-CPA using the command:

```
scripts/cpa.sh -svcomp22-graves -spec [prop.prp] [file.c]
```

5 Software Project and Contributions

Graves-CPA is an open source project developed by the authors at the University of Virginia. We would like to thank the team behind the PeSCo and CPAChecker tools for allowing us to build on their work.

Acknowledgements

We would like to thank Hongning Wang for his advice on graph neural networks and prediction systems. This material is based in part upon work supported by the U.S. Army Research Office under grant number W911NF-19-1-0054 and by the DARPA ARCOS program under contract FA8750-20-C-0507.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, principles, techniques. Addison wesley **7**(8), 9 (1986)
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
3. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification. pp. 144–159. Springer International Publishing, Cham (2018)
4. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: International Conference on Computer Aided Verification. pp. 622–640. Springer (2015)
5. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Formal Methods in Computer Aided Design. pp. 189–197. IEEE (2010)
6. Beyer, D., Lemberger, T.: Cpa-symexec: efficient symbolic execution in cpachecker. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 900–903 (2018)
7. Beyer, D., Löwe, S.: Explicit-state software model checking based on cegar and interpolation. In: International Conference on Fundamental Approaches to Software Engineering. pp. 146–162. Springer (2013)
8. Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)
9. Johnson, R., Vlissides, J.: Design patterns. Elements of Reusable Object-Oriented Software Addison-Wesley, Reading (1995)
10. Lattner, C.: Clang: a c language family frontend for llvm, https://clang.llvm.org/
11. Leeson, W., Dwyer, M.B.: Algorithm selection for software verification using graph attention networks (2022), https://arxiv.org/abs/2201.11711
12. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.: Gated graph sequence neural networks (2017)
13. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), https://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
14. Richter, C., Wehrheim, H.: Pesco: Predicting sequential combinations of verifiers. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 229–233. Springer (2019)
15. Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE transactions on neural networks **20**(1), 61–80 (2008)
16. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
17. Xu, K., Li, C., Tian, Y., Sonobe, T., ichi Kawarabayashi, K., Jegelka, S.: Representation learning on graphs with jumping knowledge networks (2018)

GWIT: A Witness Validator for Java based on GraalVM (Competition Contribution)*

Falk Howar (✉)[1,2] and Malte Mues[1]

[1] TU Dortmund University, Dortmund, Germany
{falk.howar, malte.mues}@tu-dortmund.de
[2] Fraunhofer ISST, Dortmund Germanny

Abstract. GWIT is a validator for violation witnesses produced by Java verifiers in the SV-COMP software verification competition. GWIT weaves assumptions documented in a witness into the source code of a program, effectively restricting the part of the program that is explored by a program analysis. It then uses the GDart tool (dynamic symbolic execution) to search for reachable errors in the modified program.

1 Introduction

Software verification tools, like any other software, can contain bugs. Given their intended use, i.e., proving the absence of errors in programs, however, bugs in verification tools are particularly problematic. On the other hand, verification tools can generate certificates for computed verdicts (e.g., counterexamples) that can be used to validate verification results. In the SV-COMP competition on software verification *violation witnesses* and *correctness witnesses*, based on annotated abstract control-flow automata have been established as a standardized representation of such certificates [1, 2]. Participating verifiers are expected to produce witnesses for verdicts and *witness validators* are used for confirming verdicts based on these witnesses.

In this paper, we present GWIT (as in "**G**uess **W**hat **I**'m **T**hinking" or as in **G**Dart-based **wit**ness validator), a validator of violation witnesses for Java programs, based on the GDart tool ensemble [6]. GWIT validates violation witnesses by weaving the assumptions documented in a witness into the original program under analysis and checks the restricted program with dynamic symbolic execution.

2 Witness Validation in GWIT

We illustrate the operation of GWIT for the small example shown in Figure 1: In the program, a String value is created nondeterministically before asserting that the value of this String value should not be "whoopsy". This program contains a reachable error: in case the value "whoopsy" is returned by the call to `Verifier.nondetString()`, an assertion violation will be triggered.

* This work has been partially founded by an Amazon Research Award

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 446–450, 2022.
https://doi.org/10.1007/978-3-030-99527-0_29

```
1  public static void main(String[] args) {
2    String s = Verifier.nondetString();
3    assert !s.equals("whoopsy")
4  }
```

Fig. 1: Small program with reachable error.

Java verifiers will generate a violation witness in such a case. In SV-COMP, witnesses are produced in a standardized format, conceptually based on control-flow automata and technically realized as models in the *GraphML* format [2]. Figure 2 shows an excerpt of such a witness for the above example. The witness makes an assumption on the state of the program when executing line 2 of the example program, namely that variable `s` has value "whoopsy". As discussed, execution paths on which this assumption holds, will lead to an error.

GWIT weaves the assumptions from the witness into the original program, restricting the number of program paths that have to be explored for finding the error. Figure 3 shows the result for our example: a call to `Witness.assume(...)` is generated from the assumption from the witness in Figure 2. The assume method wraps potentially many calls to the `Verifier.assume(...)` method, enabling multiple assumptions on the same line of code (e.g., due to execution of that line in a loop). The `counters` array keeps statistic on assumptions per line. The `Verifier.assume(...)` method is used by GDart to stop analysis on paths that violate the corresponding assumption.

Figure 4, finally, shows the effect of weaving the witness into the code on the obtained constraints-trees. In the left of the figure, the tree computed by GDart for the original program is shown. The tree has two satisfiable paths, branching on the condition of the assert statement. The right of the figure shows the tree for the modified program. This tree contains a node for the assumption, one path that is not executed after the violation of the assumption, one path that is not feasible after the assumption for the assert statement, and one path leading to an error (i.e., assertion violation). In this small example, the tree for the modified program is more complex than the tree for the original program, but it has fewer complete execution paths. In more complex programs, assumptions will typically remove multiple execution paths, making the validation task significantly easier than the original verification task.

```
<edge source="n0" target="n1">
  <data key="originfile">Main.java</data>
  <data key="startline">2</data>
  <data key="threadId">0</data>
  <data key="assumption">s.equals("whoopsy")</data>
  <data key="assumption.scope">...</data>
</edge>
```

Fig. 2: Excerpt of violation witness produced by GDart or JBMC.

```
1  static int[] counters = new int[] { 0 };
2  public static void assume(int id, boolean ... assumptions) {
3    int idx = counters[id];
4    counters[id]++;
5    Verifier.assume(assumptions[idx]);
6  }
7
8  public static void main(String[] args) {
9    String s = Verifier.nondetString();
10   Witness.assume(0, s.equals("whoopsy"));
11   assert !s.equals("whoopsy")
12 }
```

Fig. 3: Program with assumption from witness weaved into the code.

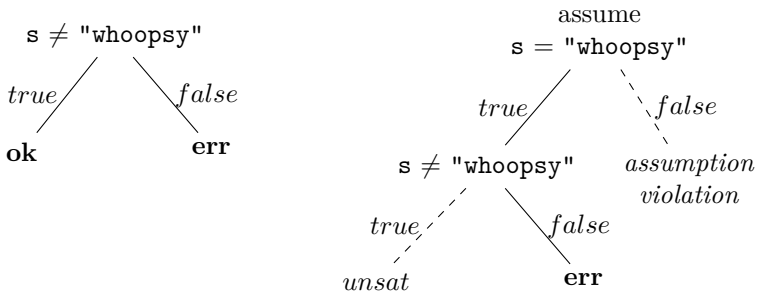


Fig. 4: Constraints-tree for original program (left) and modified program (right).

3 Performance and Limitations

While the approach of GWIT is sound for violation witnesses, the current implementation still has limitations, validating roughly half of the witnesses provided by verifiers.

Soundness. GWIT is sound: weaving a witness into the code adds additional decision nodes to the constraints-tree. In the sub-tree rooted at such a new node, some paths become unsatisfiable and will not be explored. Every complete path ψ in the modified tree has an equivalent path ϕ in the original constraints-tree such that $\psi \implies \phi$. If an error is reached in the modified tree, it is also reachable in the original program.

Performance. For programs with few decisions, the modified program may actually be more complex than the original program, but GDart does only explore more paths than in the original program in cases where the initial value along some path does not satisfy an assumption. Comparing the CPU times of GDart used as a verifier and used through GWIT, using almost identical configuration

options (only difference: GWIT does not produce witnesses), complexity is reduced for most benchmark instances that do not fail due to syntactic errors during weaving (see below).

Two extreme examples are the `BellmanFord-FunSat02` for which weaving a witness with 13 assumptions increases CPU time more than twice, leading to a timeout during validation and the `nanoxml_eqchk/prop2` instance for which the CPU time required for validation is less than 14% of the CPU time needed for the original verification task.

Overall, GWIT successfully validates 301 of 614 witnesses provided by GDart and JBMC [3] (the only Java verifiers that currently produce witnesses). In 286 cases, validation failed with inconclusive verdicts due to currently unsupported features of witness. In 15 cases, incorrect weaving (see below) prevented validation of witnesses. For 12 witnesses, validation attempts exhaust resource limits.

Limitations. First, GWIT currently only supports violation witnesses. In principle, it should be possible to validate verification witnesses by weaving assertions into the program code, but it is not obvious that such an approach makes the validation of witnesses a simpler problem than the original verification task. Second, since weaving witnesses is done on the source code, it only works correctly on proper blocks, delimited with braces, and with one statement per line. While this does not affect soundness, it makes the validation of witnesses impossible in some cases.

4 Tool Setup

GWIT is shipped as a git repository with sub-projects delivering all required components. Checking out the repository and initializing all sub-projects pulls in all required source code. For building the SPouT component, the mx build system[3] maintained by the GraalVM [7] team is required. Other components are built with maven. Once all build systems are available, the `./compile-all.sh` script builds GWIT. The `./run-gwit.sh` is used to validate witnesses, taking the witness file and source folders of a benchmark instance as parameters. GWIT currently does not expose any other configuration parameters.

5 Software Project

The GWIT tool is available on GitHub[4]. GWIT's scripts are licensed under the Apache 2.0 license. The sub-project bring their own license as follows: DSE[5] is available under the Apache 2.0 license, JConstraints[6] [4] as well, and SPouT[7] is available under the GPL v2 license. The components of GWIT and GWIT itself are currently developed at TU Dortmund by the group led by Falk Howar.

[3] https://github.com/graalvm/mx
[4] https://github.com/tudo-aqua/gwit
[5] https://github.com/tudo-aqua/dse
[6] https://github.com/tudo-aqua/jconstraints
[7] https://github.com/tudo-aqua/spout

6 Data Availability Statement

The GWIT archive used for SV-COMP 2022 is available at Zenoodo [5].

References

1. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. p. 326337. FSE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2950290.2950351
2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. p. 721733. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786867
3. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Proc. TACAS. pp. 219–223. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_17
4. Howar, F., Jabbour, F., Mues, M.: JConstraints: A library for working with logic expressions in Java. In: Models, Mindsets, Meta: The What, the How, and the Why Not?, pp. 310–325. Springer (2019). https://doi.org/10.1007/978-3-030-22348-9_19
5. Howar, F., Mues, M.: Gwit artifact for sv-comp 2022 (Feb 2022). https://doi.org/10.5281/zenodo.5956885
6. Mues, M., Howar, F.: GDart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In: Proc. TACAS (2). Springer (2022)
7. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Proc. SPLASH. pp. 187–204 (2013)

The Static Analyzer Infer in SV-COMP
(Competition Contribution)

Matthias Kettl(✉) and Thomas Lemberger

LMU Munich, Germany

Abstract. We present Infer-sv, a wrapper that adapts Infer for SV-COMP. Infer is a static-analysis tool for C and other languages, developed by Facebook and used by multiple large companies. It is strongly aimed at industry and the internal use at Facebook. Despite its popularity, there are no reported numbers on its precision and efficiency. With Infer-sv, we take a first step towards an objective comparison of Infer with other SV-COMP participants from academia and industry.

1 Facebook Infer

Infer [6] is a compositional and incremental static-analysis tool developed at Facebook. Infer supports a wide array of analyses; this includes memory safety, buffer overruns, performance constraints and different reachability analyses for C, C++, Objective C, Java, C#, and .Net. For memory analysis, Infer uses bi-abduction [7] with separation logic [14]. Infer supports the integration of new abstract domains through the abstract-interpretation framework Infer:AI. Infer analyzes programs compositionally (building method summaries) and incrementally (only analyzing changed program parts). In contrast to most other tools that participate in SV-COMP, Infer is not an academic verifier. Instead, it is aimed at practical use during software development. This has direct implications on the development focus: When Infer is told to incrementally analyze software, it outputs only newly discovered bugs and does not re-report bugs found in previous analyses. This allows developers to ignore warnings not deemed relevant and reduces the cognitive burden on developers due to false alarms. Multiple large companies use Infer—among others: Amazon Web Services, Facebook, Microsoft, Mozilla, and Spotify. At the time of this writing, Infer has more than 12 000 stars on GitHub and was forked over 1 500 times. Despite its popularity, there are no reported numbers on Infer's precision and soundness. With the participation of Infer in the C language track of SV-COMP '22, we hope to take a first step towards an objective comparison of Infer with other verifiers.

The following other commercial verifiers participate in SV-COMP '22: 2ls [16], Cbmc [10], Crux [1], Frama-C [5], VeriAbs [12], and VeriFuzz [9].

[1] https://crux.galois.com/

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 451–456, 2022.
https://doi.org/10.1007/978-3-030-99527-0_30

2 Infer in SV-COMP

2.1 Infer-SV

Verification. We provide the wrapper Infer-sv to adapt Infer to the SV-COMP specification format for program properties. Infer-sv parses the property to analyze, adjusts the program under analysis for Infer, runs Infer with fitting analyses, and reports a verification verdict based on the feedback produced by Infer. Infer-sv supports the following SV-COMP program properties:

no-overflow. The aim is to check for arithmetic overflows on signed-integer types. Infer-sv runs Infer's buffer-overrun analysis [2] to detect these.

unreach-call. The aim is to check for reachable calls to function `reach_error`. Infer provides a function-call reachability analysis [3], but this analysis proved very imprecise. To mitigate this, Infer-sv performs a program transformation [4]: It replaces each call to function `reach_error` with an overflow-provoking statement `int __reach_error_x = 0x7fffffff + 1`. No task with property unreach-call contains a signed-integer overflow, so the original reachability property holds if and only if any of the introduced overflows is reachable. Infer-sv runs Infer's buffer-overrun analysis on the transformed program to check this.

valid-memsafety. The aim is to check for invalid pointer dereferences, invalid frees of memory, and memory leaks. To analyze memory safety, Infer-sv uses two analyses: bi-abduction [5] and Infer:Pulse [6]. SV-COMP requires verifiers to report the concrete type of violation detected: *valid-deref*, *valid-memtrack*, or *valid-free*. Infer-sv analyzes the error codes reported by Infer to determine the exact violation found. If Infer reports multiple fitting warnings, we take the first.

Witnesses. SV-COMP requires participants to report GraphML verification-result witnesses [3, 4] in tandem with each result, and these witnesses must be successfully validated by at least one participating witness validator. Natively, Infer does not support the generation of GraphML witnesses. To mitigate this, Infer-sv creates generic witnesses: When reporting a violation, it generates a violation witness [4] that represents all possible program paths. When reporting a program safe, it generates a correctness witness [3] that only contains the trivial invariant 'true'. These witnesses do not helpfully guide towards a violation or proof, but are valid according to the SV-COMP rules.

Participation. Infer-sv participates hors concours in the categories ReachSafety, ConcurrencySafety, NoOverflows, and SoftwareSystems. Because of missing support, we exclude Infer-sv from categories aimed at float handling, as well as category MemSafety-MemCleanup.

[2] https://fbinfer.com/docs/checker-bufferoverrun
[3] https://fbinfer.com/docs/checker-annotation-reachability
[4] https://github.com/facebook/infer/issues/763
[5] https://fbinfer.com/docs/checker-biabduction
[6] https://fbinfer.com/docs/checker-pulse

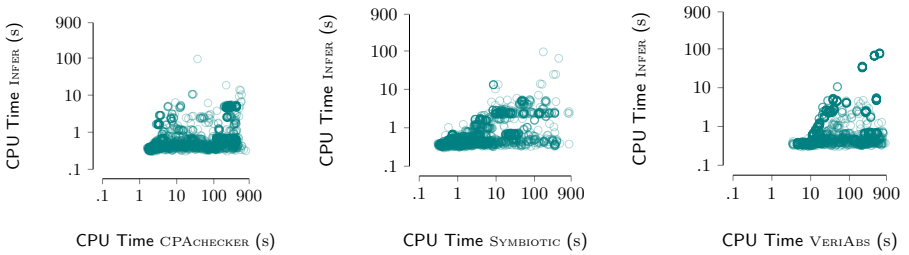


Fig. 1: Comparison of the run time (in CPU time seconds) of three SV-COMP '22 medalists and Infer, across all tasks correctly solved by the respective pair

```
1  int main() {
2    if (0) {
3      int x = 0x7fffffff + 1;
4    }
5  }
```

(a) Infer correctly reports safety

```
1  void reach_error() {
2    int x = 0x7fffffff + 1;
3  }
4  int main() {
5    if (0) {
6      reach_error();
7    }
8  }
```

(b) Infer incorrectly reports an alarm

```
1  int main() {
2    int x = 0x7fffffff;
3    int y = -1;
4    while (x > 0) {
5      x = x - 2*y;
6    }
7  }
```

(c) Infer correctly reports an alarm

```
1  int main() {
2    int x = 0x7fffffff;
3    int y = -1;
4    while (x > 0) {
5      x = x - 2*y;
6      y = y + 2;
7    }
8  }
```

(d) Infer incorrectly reports safety

Fig. 2: Examples of Infer's inconsistent results

2.2 Strengths of Infer

Infer scales well [6]. This shows in the SV-COMP results: For 6 000 out of 8 000 tasks with a verification verdict, Infer finishes the analysis in less than one second of CPU time. The remaining 2 000 tasks each take less than 100 s of CPU time. This means that Infer stays significantly below the time limit of 900 s per task. Figure 1 compares the run time of Infer (in CPU-time seconds) to the best SV-COMP '22 tools in the categories that Infer participated in: CPAchecker [11], Symbiotic [8], and VeriAbs [12]. Each plot shows the run time for all tasks that are correctly solved by both Infer and the respective other verifier (independent of result validation). It is visible that Infer (y-axis) is significantly faster than the other tools (x-axis) for almost all tasks. This speed makes Infer integrate well in continuous-integration development systems [13, 15].

2.3 Weaknesses of Infer

Infer demonstrates low analysis precision. Figures 2a and 2b illustrate a low precision across function calls (intraprocedural analysis): Both programs contain an unreachable, signed integer overflow. The only difference is the indirection in Fig. 2b due to the additional function call. Infer correctly reports Fig. 2a safe, but incorrectly reports an alarm for Fig. 2b. We assume that the intraprocedural analysis of Infer does not check whether `reach_error` is reachable from the program entry. Infer-sv mitigates this issue for property *unreach-call* through the mentioned program transformation, but this imprecision still leads Infer to report wrong alarms across all program properties.

Infer can also show imprecision within a single function. Consider Figs. 2c and 2d: The only change between Fig. 2c and Fig. 2d is the addition of a statement in line 6, `y = y + 2`. This has no influence on the integer overflow in line 5, so both programs contain an overflow. Infer correctly reports the overflow for Fig. 2c, but wrongly reports Fig. 2d safe.

These imprecisions strongly reflect in the SV-COMP results of Infer, leading to many incorrect proofs and alarms.

3 Usage

Infer-sv requires Python 3.6 or later. Script `setup.sh` downloads and extracts version 1.1.0 of Infer. From the tool's directory, Infer-sv can be run with the following command:

```
./infer-wrapper.py \
    --data-model {ILP32 or LP64} \
    --property path/to/property.prp \
    --program path/to/program.c \
```

Setting the data model is optional. Infer-sv will print the recognized property and the command line it uses to call Infer. Infer-sv prints the full output of Infer, including all warnings, and the final verification verdict on the last line. The verification verdict can be `true`, `false`, `unknown` or `error`.

4 Conclusion

The participation of Infer in SV-COMP allows an objective comparison with other verifiers for C. This shows that the selected analyses of Infer are very efficient, but suffer from strong imprecision on the considered benchmark tasks.

Contributors. Infer [7] is developed by Facebook and the open-source community under the MIT license, and Infer-sv [8] is developed under the Apache 2.0 license at the Software and Computational Systems Lab at LMU Munich, led by Dirk Beyer.

[7] https://github.com/facebook/infer
[8] https://gitlab.com/sosy-lab/software/infer-sv

Funding Statement. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 418257054 (Coop).

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [1] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [2].

References

1. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
2. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
3. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
4. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
5. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Proc. TACAS (2). Springer (2022)
6. 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
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
8. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: Symbiotic 9: Parallelism and invariants (competition contribution). In: Proc. TACAS (2). Springer (2022)
9. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing - (competition contribution). In: Proc. TACAS, part 3. pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
10. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
11. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
12. Darke, P., Agrawal, S., Venkatesh, R.: VeriAbs: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32
13. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019). https://doi.org/10.1145/3338112

14. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Proc. TACAS. LNCS, vol. 3920, pp. 287–302. Springer (2006). https://doi.org/10.1007/11691372_19
15. Harman, M., O'Hearn, P.W.: From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In: Proc. SCAM. pp. 1–23. IEEE (2018). https://doi.org/10.1109/SCAM.2018.00009
16. Malík, V., Schrammel, P., Vojnar, T.: 2LS: Heap analysis and memory safety (competition contribution). In: Proc. TACAS (2). pp. 368–372. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_22


LART: Compiled Abstract Execution*
(Competition Contribution)

Henrich Lauko ✉ ** and Petr Ročkai

Faculty of Informatics, Masaryk University, Brno, Czech Republic
xlauko@mail.muni.cz

Abstract. LART – LLVM abstraction and refinement tool – originates from the DIVINE model-checker [5,7], in which it was employed as an abstraction toolchain for the LLVM interpreter. In this contribution, we present a stand-alone tool that does not need a verification backend but performs the verification natively. The core idea is to instrument abstract semantics directly into the program and compile it into a native binary that performs program analysis. This approach provides a performance gain of native execution over the interpreted analysis and allows compiler optimizations to be employed on abstracted code, further extending the analysis efficiency. Compilation-based abstraction introduces new challenges solved by LART, like domain interaction of concrete and abstract values simulation of nondeterministic runtime or constraint propagation.

Keywords: Abstract interpretation · Compilation-based abstraction · LLVM · LART · DIVINE · Formal verification · Symbolic execution.

1 Verification Approach and Software Architecture

As it is with many tasks in computer science, one can approach them in multiple ways, and verification is not an exception. In general, tools approach program analysis using an interpretation, giving them complete control over a program state and program execution but paying the cost for performance. Our tool LART challenges the task utilizing the toolset from the opposite side of the spectrum – compilation – using a technique of so-called *compilation-based abstraction*. The main idea of this approach is to compile nondeterministic execution directly into the executable and perform reachability analysis by its native execution. This approach is most similar to one presented in symcc [6]. Symcc performs a compilation of symbolic execution into the native binary. In contrast, we present a more general approach that allows arbitrary abstraction. Spin model checker [4] also provides a mode where the model is compiled together with a verifier to a single executable.

During the compilation, LART performs LLVM-to-LLVM transformation to augment instructions that can manipulate with nondeterministic values. This is

* This work has been partially supported by Red Hat, Inc.
** Jury member representing LART at SV-COMP 2022.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 457–461, 2022.
https://doi.org/10.1007/978-3-030-99527-0_31

a purely syntactic abstraction of a program, e.g., `add` instruction is replaced by call to `__lart_add`. Additionally, LART provides a set of semantic libraries (abstract domains) to give meaning to abstract instruction. Each abstract domain defines the native representation of abstract values, implements abstract instructions and transformations to and from concrete values and other domains. The tool provides multiple domains that allow analyses with various precisions, e.g., interval analysis, nullity analysis, or symbolic analysis. Finally, to allow native execution, domains are present in static libraries linked to instrumented programs under test.

In comparison to concrete programs, abstracted programs also exhibit nondeterministic control flow. To explore all possible execution paths, LART provides a configurable runtime library. The overall architecture of compilation-based abstraction is depicted in figure Figure 1.

The configuration used in the competition contribution employs an iterative deepening search of program paths. At each branching point of a program, the execution forks to explore all possibilities. Finally, the main process of the analysis gathers results from explored paths and notifies the user if an error is reachable. This approach eventually suffers from potential infinite loops and path explosion problem. However, it is sufficient for bug hunting or even verification in the case of employed overapproximative abstraction, which widens the effect of infinite loops. Also, in many simple cases, a compiler can summarize the effects of program loops, minimizing the impact of path explosion.

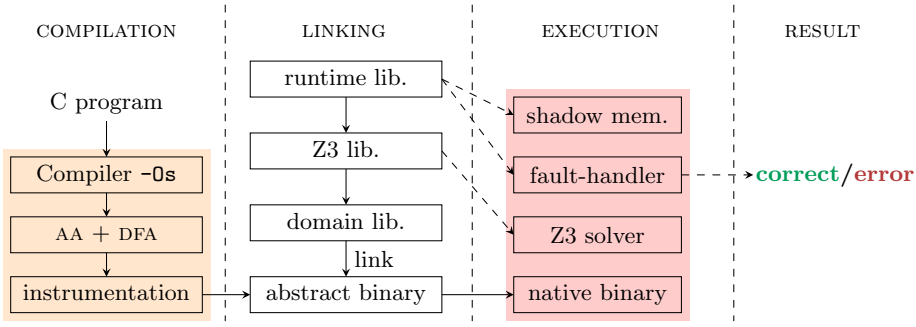


Fig. 1. LART architecture overview.

In order to obtain a performant result, we strive to minimize the amount of syntactic abstraction. Instrumentation achieves this by combining forward dataflow analysis and Andersen alias analysis [1], tainting only those instructions that might encounter nondeterministic values, and abstracting only the tainted instructions. This analysis is entirely overapproximative and detects all possible candidates for abstraction quickly. The actual abstract computation is resolved later during execution.

However, we don't want to perform expensive abstract computation when tainted instructions do not obtain nondeterministic operands. This might occur when a C function at one point receives concrete arguments and at another call

site some abstract arguments. In the former case, we would like to perform it fully concretely. While in the latter, we want to execute only the necessary amount of tainted instructions abstractly. Therefore, LART synthesizes simple dispatch routines that pick a concrete or abstract instruction depending on the operands. The dispatch routine also handles the possibility of mixing concrete and abstract operands – lifting concrete values to an abstract domain if necessary. We require that all operands of abstract instructions are in the same domain. See an example of dispatch in Figure 2.

```
__lart_value __lart_dispatch_add(__lart_value a, __lart_value b) {
    if (is_abstract(a) || is_abstract(b)) {
        if (!is_abstract(a))
            a.abstract = lift(a.concrete);
        else if (!is_abstract(b))
            b.abstract = lift(b.concrete);
        return domain::add(a.abstract, b.abstract);
    }

    return a.concrete + b.concrete;
}
```

Fig. 2. Syntactically abstracted values in LART are represented in union type of an abstract or concrete type (`__lart_value`). The dispatch routine lifts operands to an abstract domain and resolves in which domain the instruction should be executed. Since the abstraction dispatch is purely syntactic, it can be inlined to abstracted source code and further optimized. This gives the compiler a possibility to optimize repeated checks in dispatch routines.

The runtime for native execution takes care of multiple responsibilities. First of all, it implements an execution fork when a branch is conditioned by the abstract value that results in both possibilities, e.g., when a branch is conditioned on symbolic term $x < 5$, both outcomes are possible. Furthermore, the runtime takes care of memory management of abstraction. To not disrupt the original program's memory layout, LART keeps all abstract data in a shadow memory. Therefore the union values presented in Figure 2 are split into two separately addressed regions – concrete program memory and abstract shadow memory. The information on whether variables hold an abstract value is also kept in the shadow memory.

2 Strengths and Weaknesses

The main strength of the compilation-based abstraction is in the utilization of native runtime and compiler optimizations on abstracted code. From theory, the native execution should consistently outperform the same interpreted analysis. However, it comes at the cost of a more complex source transformation that is harder to relate to its origin. Furthermore, the overapproximative nature of the syntactic analysis produces unnecessary execution of dispatch functions when not needed. In contrast, an interpreter can compute in specific domain without additional dispatches. Another advantage of the approach is a reusable result of syntactic abstraction that can be linked with various domains to perform analysis concurrently without repeated LLVM instrumentation.

The best comparison of LART is with the DIVINE model-checker, which uses LART's transformation and domain libraries internally, but instead of compiling to native executable, it interprets abstracted LLVM IR. Results from the competition support the hypothesis that the compilation-based approach of LART outperforms DIVINE in all reachability subcategories, except one where longer times are caused by different state space exploration order.

Given the simplistic runtime, abstracted binaries produced by LART lack further analysis optimizations and verification capabilities. Presently, the exploration algorithm only supports reachability analysis of single-threaded programs. However, we plan to support memory safety and overflow checking using sanitizers like approach.

Another goal of LART's compilation-based approach is to provide a reusable abstraction component for verification tools. The proof of this concept is shown on DIVINE and now on the native mode that can be analyzed by standard programmer toolset, like debuggers or sanitizers.

3 Tool Setup and Configuration

The verifier archive can be found on the SV-COMP 2022 [2] page under the name LART. In case the binary distribution does not work on your system, we also provide a source distribution and build instructions at https://github.com/xlauko/lart/tree/svcomp-2022. It is sufficient to run LART using compiler wrapper script as follows: `lartcc <domain> testcase.c -o abstract` and then execute the `abstract` binary to perform the analysis.

For SV-COMP contribution, LART wrapper handles additional settings and setup of workflow presented in Figure 1. The wrapper sets LART options based on the property file and the benchmark. In particular, LART enables symbolic mode if any nondeterminism is found, and it sets which errors should be reported based on the property file. It also generates witness files. More details can be found on the aforementioned distribution page. Due to support limitation LART participates only in ReachSafety and DeviceDrivers categories.

4 Software Project and Contributors

The project home page is https://github.com/xlauko/lart. The LART is open source software distributed under the MIT license. Active contributors to the tool are listed as authors of this paper.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [2] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [3].

References

1. Andersen, L.O.: Program analysis and specialization for the C programming language. Ph.D. thesis, Citeseer (1994)
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
3. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
4. Holzmann, G., Najm, E., Serhrouchni, A.: Spin model checking: An introduction. STTT **2**, 321–327 (03 2000). https://doi.org/10.1007/s100090050039
5. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. Theoretical Aspects of Computing – ICTAC 2018 (2018). https://doi.org/10.1007/978-3-030-02508-3_17
6. Poeplau, S., Francillon, A.: Symbolic execution with SymCC: Don't interpret, compile! In: 29th USENIX Security Symposium (USENIX Security 20). pp. 181–198. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau
7. Ročkai, P., Štill, V., Černá, I., Barnat, J.: DiVM: Model Checking with LLVM and Graph Memory. Journal of Systems and Software **143**, 1–13 (2018). https://doi.org/10.1016/j.jss.2018.04.026

Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding*

(Competition Contribution)

Marek Chalupa✉, Vincent Mihalkovič, Anna Řechtáčková, Lukáš Zaoral, and Jan Strejček

Masaryk University, Brno, Czech Republic

Abstract. The development of Symbiotic 9 focused mainly on two components. One is the symbolic executor Slowbeast, which newly supports *backward symbolic execution* including its extension called *loop folding*. This technique can infer inductive invariants from backward symbolic execution states. Thanks to these invariants, Symbiotic 9 is able to produce non-trivial correctness witnesses, which is a feature that is missing in previous versions of Symbiotic. We have also extended forward symbolic execution in Slowbeast with a basic support for parallel programs. The second component with significant improvements is the instrumentation module. In particular, we have extended the static analysis of accesses to arrays with features designed for programs that manipulate C strings.

Symbiotic 9 is the *Overall* winner of SV-COMP 2022. Moreover, it won also the categories *MemSafety* and *SoftwareSystems*, and placed third in *FalsificationOverall*.

1 Verification Approach

Symbiotic 9 combines fast static analyses with code instrumentation and program slicing [13] to speed up the code verification. In the SV-COMP configuration of Symbiotic 9, the code verification is performed by symbolic executors, namely by Slowbeast [8] and our fork of Klee [4].

As Symbiotic works internally with llvm [10], it first compiles the given C program into llvm bitcode. The following steps depend on the verified property.

Verification of the Property `unreach-call` For this property, Symbiotic 9 directly slices the llvm bitcode to remove instructions that have no influence on the reachability of error calls and then run Klee with the time limit of 333 seconds. Klee is very efficient and often decides the task within this time limit. If Klee fails to decide, we parse its output and proceed according to the case of the failure. If Klee failed because the program contains threads, we

* This work has been supported by the Czech Science Foundation grant GA19-24397S.
✉ Jury member and the corresponding author: `chalupa@fi.muni.cz`

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 462–467, 2022.
https://doi.org/10.1007/978-3-030-99527-0_32

Table 1. The comparison of supported features of Klee (our fork and the upstream) and Slowbeast (SV-COMP 2022 and SV-COMP 2021 versions). The marks ✓/✓/✗ mean supported/partially supported/unsupported.

	Klee upstream	Klee our fork	Slowbeast SV-COMP 2021	Slowbeast SV-COMP 2022
Backward SE	✗	✗	✗	✓
Loop folding	✗	✗	✓	✓
Invariant generation	✗	✗	✓	✓
Symbolic floats	✗	✗	✓	✓
Symbolic pointers	✓	✓	✗	✓
Symbolic-sized allocations	✗	✓	✗	✓
Symbolic addresses	✗	✓	✗	✓
Parallel programs	✗	✗	✗	✓
Incremental solving	✗	✗	✓	✓
Caching solver calls	✓	✓	✗	✗
Lazy memory	✗	✗	✓	✓

run Slowbeast with forward *symbolic execution (SE)* and the threads support turned on. If Klee failed for any other reason, we run Slowbeast with *backward symbolic execution with loop folding (BSELF)* [8] described later. If BSELF also fails (the current implementation supports only selected program features), we run Slowbeast with forward symbolic execution.

Note that running forward symbolic execution first with Klee and then with Slowbeast if Klee fails does make a good sense as Klee and Slowbeast support a different set of features. The main differences between these tools (and the upstream Klee and the version of Slowbeast used in Symbiotic 8) are summarized in Table 1. Row *symbolic addresses* indicates whether tools model the non-determinism in the placement of allocated objects (this is useful, e.g., when comparing addresses of such objects). Row *incremental solving* indicates whether tools can associate the state of an SMT solver to every symbolic execution state and incrementally add constraints instead of always solving formulas from the scratch. Row *caching solver calls* indicates whether tools can remember results of solver calls and use them later to quickly decide some other solver calls. Finally, row *lazy memory* indicates if the tool can create memory objects on-demand when first accessing them, without their previous allocation (it assumes that the accesses to memory are valid). This feature is crucial when we want execute a program by parts, without starting from the entry point. The meaning of the remaining rows should be clear or is explained later.

If an error is found by either tool, it is replayed on the unsliced code. If the replay succeeds, we generate a violation witness. If no error is found and the analysis was complete, we generate a correctness witness. If the program correctness was proved by Slowbeast with BSELF, we generate a witness containing the computed invariants, otherwise we generate a trivial correctness witness as we have no invariants at hand. In all other cases, Symbiotic 9 answers *unknown*.

Verification of Other Properties For verification of other properties than `unreach-call`, Symbiotic 9 uses the same workflow as Symbiotic 8 [7]. In brief, the instrumentation module marks program instructions that can potentially violate the considered property. The module employs suitable fast static analyses to identify these instructions (e.g., when checking the property `no-overflow`, it uses a range analysis to discover the instructions that may perform a signed integer overflow). The bitcode with marked instructions is sliced such that the arguments and the reachability of these instructions are preserved. The sliced bitcode is passed to Klee. If it discovers a property violation and then replays it on the unsliced code, we produce a violation witness. If Klee completes its analysis without any property violation found, we produce a trivial correctness witness. In all other cases, Symbiotic 9 returns *unknown*.

Backward Symbolic Execution with Loop Folding (BSELF) [8] Slowbeast newly implements *backward symbolic execution (BSE)* [9], which explores the program backward from target locations towards the initial location and incrementally computes weakest preconditions for the explored program paths. BSE is a valuable technique on its own as it precisely corresponds to k-induction on control-flow paths [8]. *Loop folding* is a technique that aims to infer inductive invariants during BSE. Roughly speaking, when BSE starts from an error location and reaches a loop header, loop folding creates an initial *invariant candidate* that is disjoint with the current weakest precondition (i.e., the states that can reach the error location). If the invariant candidate is actually an invariant, we know that the error location is not reachable via the explored path. Otherwise, a pre-image of the invariant candidate along a loop path is computed, over-approximated, and added to the candidate. This process is repeated until an invariant is found or until it fails for some reason, e.g., when it discovers that the error location is actually reachable. Loop folding can infer complex disjunctive invariants and since it uses the error states, it is also property-driven.

String Analysis and Other Improvements The second major improvement in Symbiotic 9 is in the instrumentation for the property `valid-memsafety`. We have improved the analysis for the identification of out-of-bounds array accesses.

In Symbiotic 8, this analysis only determined whether an array access done via the index variable is in bounds [14]. The analysis in Symbiotic 9 also handles more general patterns where the array contains a concrete value (0 in the case of C strings) and the index pointer is incremented by one until it points to this concrete value, and where the pointer is incremented a fixed number of times.

Further, we have extended the forward symbolic execution in Slowbeast to handle parallel programs. For now, the symbolic execution is highly inefficient as it examines each interleaving of globally visible events. We plan to implement some reductions in the future. Slowbeast has been also extended to generate witnesses as this functionality was missing. Notably, it can generate non-trivial correctness witnesses using the invariants computed by BSELF. Previous versions of Symbiotic generate only trivial correctness witnesses.

Slicing has been also improved. It now applies a fast and coarse slicing before the main slicing. The coarse slicing detects all basic blocks from which no *slicing criterion* (i.e., an instruction whose reachability and arguments should be preserved) is syntactically reachable and replaces them by calls to `abort`.

2 Strengths and Weaknesses

Forward symbolic execution is unable to fully analyze unbounded loops or infinite execution paths. Hence, unless program slicing removes the unbounded computation from the program, forward symbolic execution cannot verify it. However, backward symbolic execution and BSELF can fully analyze at least some unbounded programs [8]. Still, both these methods are computationally complex as the number of paths they must search may be enormous and their exploration may involve many non-trivial calls to the SMT solver. Therefore, these methods do not scale to real-world programs.

A strong aspect of Symbiotic is the very interplay of fast static analyses in the instrumentation, program slicing, and forward and backward symbolic execution. Fast static analyses are able to deem correct many parts of the code (with respect to the verified property). These parts of the code are then usually removed by slicing and only the possibly unsafe parts of the program (and their dependencies) get into a symbolic executor. In this sense, Symbiotic does incremental or conditional [3] verification.

Results of Symbiotic 9 in SV-COMP 2022 In SV-COMP 2022 [1], Symbiotic 9 won categories *MemSafety*, *SoftwareSystems*, and *Overall*, and got the 3rd place in *FalsificationOverall*. Moreover, it produced 1529 correct answers that were not confirmed, which is the highest number in SV-COMP 2022. 1073 unconfirmed answers are in *MemSafety-Juliet*, where we produced some incorrect witnesses due to a bug. Another 258 unconfirmed answers are in *Termination*. Symbiotic 9 produced only 3 incorrect answers caused by a bug in the replay mode of Slowbeast.

3 Software Project and Contributors

All components of Symbiotic 9 use llvm 10 [10]. Slicer and instrumentation module are written in C++ and extensively use the library DG [5]. Klee is implemented in C++ and Slowbeast [12] is written in Python. Both symbolic executors use Z3 [11] as the SMT solver. Control scripts are written in Python.

Symbiotic 9 and all its components and external libraries are available under open-source licenses that comply with SV-COMP's policy for the reproduction of results. Symbiotic 9 participated in all categories of SV-COMP 2022 except the categories with Java programs.

Symbiotic 9 has been developed by Marek Chalupa, Vincent Mihalkovič, Anna Řechtáčková, and Lukáš Zaoral under the supervision of Jan Strejček.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [1] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of Symbiotic used in the competition is archived together with other participating tools [2] and also in its own artifact [6] at Zenodo.

References

1. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
2. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
3. Beyer, D., Jakobs, M.: Fred: Conditional model checking via reducers and folders. In: SEFM 2020. LNCS, vol. 12310, pp. 113–132. Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_7
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
5. Chalupa, M.: DG: analysis and slicing of LLVM bitcode. In: ATVA 2020. LNCS, vol. 12302, pp. 557–563. Springer (2020), https://doi.org/10.1007/978-3-030-59152-6_33
6. Chalupa, M.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (artifact). Zenodo (2022). https://doi.org/10.5281/zenodo.5947909
7. Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., Strejček, J.: Symbiotic 8: Beyond symbolic execution - (competition contribution). In: TACAS 2021. LNCS, vol. 12652, pp. 453–457. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_31
8. Chalupa, M., Strejček, J.: Backward symbolic execution with loop folding. In: SAS 2021. LNCS, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
9. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: PLDI 2009. pp. 363–374. ACM (2009). https://doi.org/10.1145/1542476.1542517
10. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004. pp. 75–88. IEEE Computer Society (2004), https://doi.org/10.1109/CGO.2004.1281665
11. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
12. Slowbeast repository. https://gitlab.com/mchalupa/slowbeast (2021)
13. Weiser, M.: Program slicing. In: Proceedings of ICSE. pp. 439–449. IEEE (1981)
14. Řechtáčková, A.: Improving out-of-bound access checking in Symbiotic (2020), https://is.muni.cz/th/tmq7m/, bachelor thesis, accessed 2022-02-02

Symbiotic-Witch: A Klee-Based Violation Witness Checker*
(Competition Contribution)

Paulína Ayaziová, Marek Chalupa, and Jan Strejček ✉

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xayaziov,chalupa,strejcek}@fi.muni.cz

Abstract. Symbiotic-Witch is a new tool for checking violation witnesses in the GraphML-based format used at SV-COMP since 2015. Roughly speaking, Symbiotic-Witch symbolically executes a given program with Klee and simultaneously tracks the set of nodes the witness automaton can be in. Moreover, it reads the return values of nondeterministic functions specified in the witness and uses them to prune the symbolic execution. The violation witness is confirmed if the symbolic execution reaches an error and the current set of witness nodes contains a matching violation node.
Symbiotic-Witch currently supports violation witnesses of *reachability safety*, *memory safety*, *memory cleanup*, and *overflow* properties.

1 Verification Approach

We present a new checker of violation witnesses called Symbiotic-Witch. The checker first loads a given violation witness in the GraphML format [5] and a given program. Then it performs symbolic execution [11] of the program and simultaneously tracks the progress of the execution in the witness automaton. More precisely, every state of symbolic execution is accompanied by the set of witness automaton nodes that can be reached under the executed program path. If the symbolic execution detects a violation of the considered property and the tracked set of witness automata nodes contains a violation node, the witness is confirmed.

Note that the original description of the witness format [5] does not provide any formal semantics of the format. We interpret it in the way that if an edge in a witness automaton matches an executed program instructions, then we can follow the edge but we can also stay in its starting node. Hence, if we have the set of witness automaton nodes reached under a certain program path, then prolongation of this path can add some nodes to this set, but it never removes any node from the set. A brief reading of an upcoming detailed description of the format [4] reveals that it can be the case that an edge matching an executed program instruction has to be taken. If this is indeed the case, we will adjust

* This work has been supported by the Czech Science Foundation grant GA19-24397S.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 468–473, 2022.
https://doi.org/10.1007/978-3-030-99527-0_33

our tool, but the current implementation and the following texts consider the former semantic.

Before Symbiotic-Witch starts the symbolic execution, we remove from the witness automaton all nodes that are not on any path from the entry node to a violation node. In general, witness automata are related to program executions using node and edge attributes. Symbiotic-Witch currently supports only some attributes of witness edges to map a program execution to a given witness automaton. Namely, it uses the line number of executed instructions, the information whether *true* or *false* branch is taken, and the information about entering a function or returning from a function. Additionally, if the witness automaton contains a single path from the entry node to a violation node and there is some information about return values of the `__VERIFIER_nondet_*` functions on this path, then we use these values in the symbolic execution of the program. Return values not provided in the witness are treated as symbolic values.

A more precise description of the approach can be found in the bachelor's thesis of P. Ayaziová [1].

2 Software Architecture

The approach has been implemented in a tool called Symbiotic-Witch, which is basically a modification of the symbolic executor Klee [8]. More precisely, it is derived from the clone of Klee used in Symbiotic, which employs the SMT solver Z3 [13] and supports symbolic pointers, memory blocks of symbolic sizes etc. For parsing of witnesses in the GraphML format, we use the library RapidXML.

As Klee executes programs in llvm [12], a given C program has to be translated to llvm first. We use Clang for this translation as explained in Section 4.

The current version of Symbiotic-Witch runs on llvm version 10.0.0.

3 Strengths and Weaknesses

Existing violation witness checkers (excluding Dartagnan [10] designed for concurrent programs) can be roughly divided into two categories.

– CPA-witness2test [6], FShell-witness2test [6], and Nitwit [14] perform one program execution based on the information in the witness. If this execution violates the specification, the witness is confirmed. This approach is very efficient for witnesses fully describing one program execution that violates the property. However, if a witness describes more program executions and only some of them violate the property, these tools can easily miss the violating executions. In particular, if a witness does not specify some return value of a `__VERIFIER_nondet_*` function, FShell-witness2test uses the default value 0, Nitwit picks a random value, and CPA-witness2test fails the witness confirmation.

– CPAchecker [5], UltimateAutomizer [5], and MetaVal [7] create a product of a given witness automaton and the original program and analyze it. As a result, some execution paths of the original program can be analyzed repeatedly for different paths in the witness automaton. To suppress this effect, these checkers usually ignore the possibility to stay in a witness automaton node whenever there is a matching transition leaving the node. Unfortunately, a valid witness can be unconfirmed due to this strategy.

We believe that our approach to checking violation witnesses removes all mentioned disadvantages. Symbolic execution allows us to efficiently examine many program executions corresponding to a given witness automaton, and program executions are not analyzed repeatedly. The approach can easily handle witnesses based on return values from the `__VERIFIER_nondet_*` functions as well as those based on description of branching.

There is only one principal case when a valid witness is not confirmed by Symbiotic-Witch (ignoring the cases when Symbiotic-Witch simply runs out of resources). This case can arise when Symbiotic-Witch uses the information about return values of `__VERIFIER_nondet_*` functions stored in the witness. Symbiotic-Witch uses the information immediately when the symbolic execution calls such a function and there is a matching edge in the witness with a return value that has not been used yet (i.e., the starting node of the edge is in the set of tracked witness nodes and the target node is not). This "eager approach" usually works very well, especially for witnesses containing return values for all calls of `__VERIFIER_nondet_*` functions. However, there can be witnesses where some return values are missing and a particular contained return value should not be used for the first matching call of the `__VERIFIER_nondet_*` function. Such witnesses can be valid, but Symbiotic-Witch can fail to confirm them. As far as we know, such witnesses do not appear in SV-COMP and other witness checkers would probably fail to confirm them as well.

On the negative side, our approach inherits the disadvantages and limitations of symbolic execution and Klee. In particular, it can suffer the *path explosion problem* on witnesses that do not provide return values of `__VERIFIER_nondet_*` functions. Further, Symbiotic-Witch does not support parallel programs as Klee does not support them.

Our current approach is suitable for cases when a witness can be checked based on a finite program execution. That is why our tool supports violation witnesses of safety properties. Table 1 shows the numbers of violation witnesses confirmed in SV-COMP 2022 [2] by individual witness checkers in the categories supported by Symbiotic-Witch.

We believe that symbolic execution can be also used for checking *termination* violation witnesses and for checking correctness witnesses. We plan to extend Symbiotic-Witch in these directions. We also plan to add a witness *refinement mode* [5] already provided by CPAchecker and UltimateAutomizer. In this mode, when a witness is confirmed, Symbiotic-Witch would produce another witness describing a single program execution (by specifying return values for all calls of `__VERIFIER_nondet_*` functions) that exhibits the property violation.

Table 1. The numbers of confirmed witnesses in relevant SV-COMP 2022 categories

	ReachSafety	MemSafety	NoOverflows	SoftwareSystems
number of witnesses	26 797	16 984	2 808	2 102
CPAchecker	14 908	**12 594**	2 334	**621**
CPA-witness2test	8 628	231	887	6
FShell-witness2test	14 168	954	1 436	33
MetaVal	0	116	1 982	0
Nitwit	**15 507**	-	-	0
Symbiotic-Witch	11 176	8 394	**2 609**	179
UltimateAutomizer	8 592	4 197	2 468	26

4 Tool Setup and Configuration

For the use in SV-COMP 2022, we have integrated our witness checker (originally called Witch-Klee) with Symbiotic [9], which takes care of translation of a given C program into llvm using Clang and then slightly modifies the llvm program to improve the efficiency of witness checking.

The archive with Symbiotic-Witch can be downloaded from SV-COMP archives. The witness checking process is invoked by

```
./symbiotic [-prp <prop>] [-32] -witness-check <wit.graphml> <prog.c>
```

where `<wit.graphml>` is a violation witness to be checked and `<prog.c>` is the corresponding program. By default, the tool considers *reachability safety* property and 64-bit architecture. The considered property can be changed by the `-prp` option and `<prop>` instantiated to `memsafety` or `memcleanup` or `no-overflow`. The 32-bit architecture is set by `-32`.

Our witness checker can be also downloaded directly from its repository mentioned below. The version used in SV-COMP 2022 is marked with the tag *SV-COMP22*. It can be executed without Symbiotic via a shell script as

```
./witch.sh <prog.c> <wit.graphml>
```

which calls Clang to translate `<prog.c>` to llvm and then passes the llvm program and the witness `<wit.graphml>` to the witness checker.

5 Software Project and Contributors

Symbiotic-Witch has been developed at Faculty of Informatics, Masaryk University by Paulína Ayaziová under the guidance of Marek Chalupa and Jan Strejček. The tool is available under the MIT license and all used tools and libraries (llvm, Klee, Z3, RapidXML, Symbiotic) are also available under open-source licenses that comply with SV-COMP's policy for the reproduction of results. The source code of our witness checker can be found at:

https://github.com/ayazip/witch-klee

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [2] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of Symbiotic-Witch used in the competition is archived together with other participating tools [3].

References

1. Ayaziová, P.: Klee-based error witness checker. Bachelor's thesis, Masaryk University (2021), https://is.muni.cz/th/rnv19/?lang=en
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
3. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
4. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. (2022), to appear.
5. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015. pp. 721–733. ACM (2015), https://doi.org/10.1145/2786805.2786867
6. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses - execution-based validation of verification results. In: Dubois, C., Wolff, B. (eds.) Tests and Proofs - 12th International Conference, TAP@STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10889, pp. 3–23. Springer (2018), https://doi.org/10.1007/978-3-319-92994-1_1
7. Beyer, D., Spiessl, M.: Metaval: Witness validation via verification. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 165–177. Springer (2020), https://doi.org/10.1007/978-3-030-53291-8_10
8. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
9. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: Symbiotic 9: Parallelism and invariants (competition contribution). In: Proc. TACAS (2). Springer (2022)
10. Haas, T., Meyer, R., de León, H.P.: Dartagnan: SMT-based violation witness validation (competition contribution). In: Proc. TACAS (2). Springer (2022)
11. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976), https://doi.org/10.1145/360248.360252
12. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004. pp. 75–88. IEEE Computer Society (2004), https://doi.org/10.1109/CGO.2004.1281665
13. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24

14. Švejda, J., Berger, P., Katoen, J.: Interpretation-based violation witness validation for C: NITWIT. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 40–57. Springer (2020), https://doi.org/10.1007/978-3-030-45190-5_3

Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (Competition Contribution)

Zsófia Ádám[1], Levente Bajczi[1], Mihály Dobos-Kovács[1], Ákos Hajdu[2], and Vince Molnár[1] *(✉)

[1] Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
molnarv@mit.bme.hu
[2] Meta Platforms Inc., London, United Kingdom

Abstract. Theta is a model checking framework based on abstraction refinement algorithms. In SV-COMP 2022, we introduce: 1) reasoning at the source-level via a direct translation from C programs; 2) support for concurrent programs with interleaving semantics; 3) mitigation for non-progressing refinement loops; 4) support for SMT-LIB-compliant solvers. We combine all of the aforementioned techniques into a portfolio with dynamic algorithm selection.

1 Verification Approach and Software Architecture

Theta [10] is a generic and configurable model checking framework written in Java 11. A simplified version of the architecture (focusing on software verification aspects) can be seen in Figure 1.

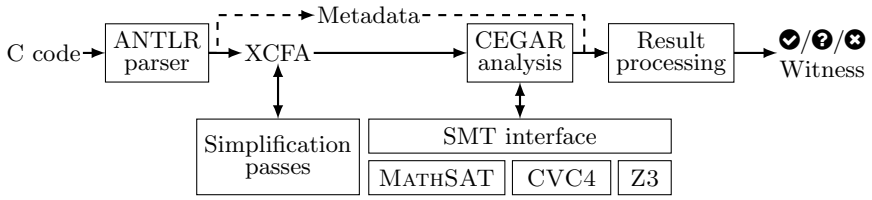


Fig. 1. Architecture of Theta.

The input is a C program that is first translated to extended control-flow automata (XCFA). Previously, Theta used LLVM [3], which had various advantages, but its static single assignment (SSA) form proved overall disadvantageous for abstraction-based algorithms. This year we use a new, direct translation (no

* Jury member representing Theta at SV-COMP 2022.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 474–478, 2022.
https://doi.org/10.1007/978-3-030-99527-0_34

intermediate language and SSA form) via an ANTLR parser. Furthermore, the CFA being "extended" refers to the fact that since this year we support concurrent programs by an analysis with interleaving semantics. After parsing we apply various passes to the XCFA (e.g., large-block encoding or partial order reduction). The core of THETA is a CEGAR-based analysis framework, targeting *reachability properties* via predicate and explicit analyses [8], along with interpolation and Newton-based refinements [7]. This year, THETA added generic support for SMT solvers (including interpolation) via the SMT-LIB interface. At SV-COMP'22 we use CVC4 [4], MATHSAT [6], and Z3 [9], where the latter is used via the Java API from before. Finally, a verdict (safe, unsafe, unknown) and a witness is produced corresponding to the C program (using metadata from the translation).

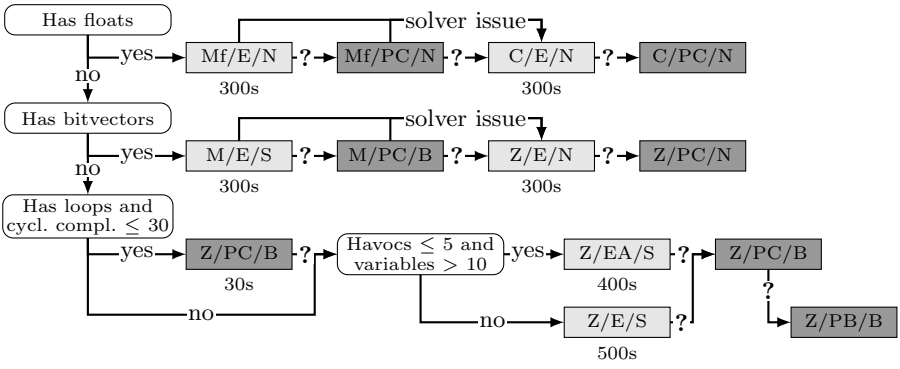


Fig. 2. Overview of the dynamic portfolio of THETA.

Verification portfolio. Based on preliminary experiments and domain knowledge, we manually constructed a dynamic algorithm selection portfolio [1] for SV-COMP'22, illustrated by Figure 2. Rounded white boxes correspond to decision points. We start by branching on the arithmetic (floats, bitvectors, integers). Under integers, there are further decision points based on the cyclomatic complexity and the number of havocs and variables. Grey boxes represent configurations, defining the *solver*/*domain*/*refinement* in this order. Lighter and darker grey represents explicit and predicate domains respectively. Internal timeouts are written below the boxes. An unspecified timeout means that the configuration can use all the remaining time. The solver can be CVC4 (C) [4], MATHSAT (M), MATHSAT with floats (Mf) [6] or Z3 (Z) [9]. Abstract domains are explicit values (E), explicit values with all variables tracked (EA), Cartesian predicate abstraction (PC) or Boolean predicate abstraction (PB) [8]. Finally, refinement can be Newton with weakest preconditions (N) [7], sequence interpolation (S) or backward binary interpolation (B) [8]. Arrows marked with a question mark (?) indicate an inconclusive result, that can happen due to timeouts or unknown re-

sults. Furthermore, this year's portfolio also includes a novel dynamic (run-time) check for refinement progress between iterations that can shut down potential infinite loops (by treating them as unknown result) [1]. Note also that for solver issues (e.g., exceptions from the solver) we have different paths in some cases.

2 Strengths and Weaknesses

Theta currently targets *ReachSafety* and *ConcurrencySafety* with limited support for structs, arrays and pointers, and no support for dynamic memory allocation, mutexes and recursion. Due to this, Theta fails for most tasks in *ProductLines*, *Recursive*, *Heap* and *Arrays*. Out of the 6163 tasks, roughly 2/3 can be translated and there are 888 confirmed correct (541 safe, 347 unsafe), 116 unconfirmed correct, and only 15 incorrect (11 false positive, 4 false negative) results [5]. Note that almost all unsupported cases are detected and reported as an error, and we only have a few incorrect results due to subtle issues.

The main strength of the tool is the combination of algorithm selection (pick algorithm based on input) and portfolios (try multiple algorithms until one succeeds). Out of the 1004 correct results, 315 could not be solved by the first configuration that the portfolio tries: dynamic checks intervened for 181 internal timeouts, 72 solver issues (e.g. wrong models), 19 non-progressing refinements, and 74 other (unknown) faults before the eventual success.

Having a diverse portfolio also paid off. Bitvector and float arithmetic tasks were either solved by explicit analyses (with a mixture of interpolation- and Newton-based refinements) before even trying predicate configurations, or if explicit analyses failed, predicate configurations were unsuccessful too. The integer arithmetic required a more diverse configuration set: Predicate abstraction solved roughly 48% of the tasks (45% Cartesian, 3% Boolean) and explicit analysis solved 52% (33% with empty precision, 19% with all variables tracked).

The SMT-LIB support provided a great improvement: previously we only had Z3, which still dominates the integer cases. However, all of the bitvector tasks were solved by MathSAT, making Z3 an unused backup. With floats, roughly half of the tasks were solved by MathSAT, while the other half needed CVC4 as backup. Since floats are reduced to bitvectors, we did not rely on Z3 based on poor performance in our preliminary experiments.

The most successful subcategories are *BitVectors*, *ControlFlow*, *Loops*, *XCSP* (38-45% correct), mostly because they use features of C that our frontend supports well. We plan to mitigate the high number of timeouts in the future with approximations (e.g. mixing integers and bitvectors), and further analyses (e.g., inferring loop invariants). We also have a significant amount of unconfirmed results: we believe this can be improved by generating more compact witnesses.

This year Theta added support for sequential concurrency via a preprocessing step: it yields an encoding where exploring all interleavings preserve inter-thread behaviors. The analyses treat consecutive non-global memory accesses as one atomic block, reducing the exploration of unnecessary total orders. A drawback of using preprocessing for partial order reduction instead of an on-line

algorithm is the superfluous exploration of *certain* total orders, e.g., all interleavings of independent global memory accesses will also be explored. This is because such accesses *might* overlap with non-independent memory accesses at other times, and the preprocessing step is not aware of such details.

Using a wrapper, Theta integrates concurrency seamlessly with the existing framework (abstract domains, refinements), except the error location-based search [8] (used for non-concurrent cases) because the required distance metric is not well defined in concurrent programs. Instead, we opted to use a breadth-first search, which had outperformed depth-first strategies in preliminary tests. We theorize that this is due to bugs being reachable within the first few instructions most of the time, but only via a specific total order. The performance for concurrent programs is still limited though, and we plan to integrate a declarative approach in the future, which could be used for weakly-ordered programs as well.

3 Tool Setup and Configuration

The competition contribution is based on Theta 3.0.0-svcomp22-v1.[3] Additionally, Theta uses CVC4 v1.9, MathSAT v5.6.6 and Z3 v4.5.0. The project's repository contains build instructions, but an archive can be found at the SV-COMP repository[4] and Zenodo [2]. with pre-built binaries for Ubuntu 20.04 (LTS). The toolchain requires packages `openjdk-11-jre-headless`, `libgomp1` and `libmpfr-dev` to be installed. The entry point of the toolchain is the script `theta/theta-start.sh`, which takes the verification task (C program) as its only mandatory input and runs the portfolio. As additional arguments we use `--portfolio COMPLEX --witness-only --loglevel RESULT`. Additional arguments are described in the readme included with the binaries.

4 Software Project

Theta is maintained by the Critical Systems Research Group[5] of the Budapest University of Technology and Economics with various contributors. The project is available open-source on GitHub[3] under an Apache 2.0 license.

Data Availability. The version of Theta used in this paper is available at [2].

Acknowledgment and Funding. The authors would like to thank Tamás Tóth, Milán Mondok, István Majzik, Zoltán Micskei and András Vörös for their contributions to the project; and the competition organizers, especially Dirk Beyer for their help during the preparation for SV-COMP. The research contributions of the authors from the Budapest Univ. of Tech. and Econ. were funded by the EC and NKFIH through the Arrowhead Tools project (EU grant No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003), and by the UNKP-21-2 New National Excellence Program of ITM from the NRDI Fund.

[3] https://github.com/ftsrg/theta/releases/tag/svcomp22-v1
[4] https://gitlab.com/sosy-lab/sv-comp/archives-2022/-/blob/main/2022/theta.zip
[5] https://ftsrg.mit.bme.hu

References

1. Ádám, Zs.: Efficient techniques for formal verification of C programs. Bachelor's thesis, Budapest University of Technology and Economics (2021)
2. Ádam, Zs., Levente, B., Dobos-Kovács, M., Hajdu, Á., Molnár, V.: Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (competition contribution): Tool archive (data set) (2022). https://doi.org/10.5281/zenodo.5956737
3. Ádám, Zs., Sallai, Gy., Hajdu, Á.: Gazer-Theta: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In: TACAS 2021, LNCS, vol. 12652, pp. 435–439. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_27
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV 2011, LNCS, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
6. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS 2013, LNCS, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
7. Dobos-Kovács, M., Hajdu, Á., Vörös, A.: Bitvector support in the Theta formal verification framework. In: Proceedings of the 2nd Workshop on Validation and Verification of Future Cyber-Physical Systems (2021), in press.
8. Hajdu, Á., Micskei, Z.: Efficient strategies for CEGAR-based model checking. Journal of Automated Reasoning **64**(6), 1051–1091 (2020). https://doi.org/10.1007/s10817-019-09535-x
9. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008, LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
10. Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., Majzik, I.: Theta: a framework for abstraction refinement-based model checking. In: FMCAD 2017. pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257

Ultimate GemCutter and the Axes of Generalization
(Competition Contribution)

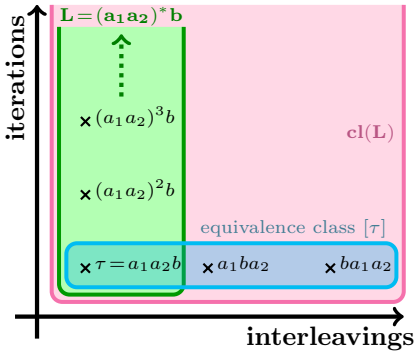
Dominik Klumpp*[1]✉, Daniel Dietsch[1], Matthias Heizmann[1], Frank Schüssele[1], Marcel Ebbinghaus[1], Azadeh Farzan[2], and Andreas Podelski[1]

[1] University of Freiburg, Freiburg im Breisgau, Germany
klumpp@informatik.uni-freiburg.de
[2] University of Toronto, Toronto, Canada

Abstract. Ultimate GemCutter verifies concurrent programs using the CEGAR paradigm, by generalizing from spurious counterexample traces to larger sets of correct traces. We integrate classical CEGAR generalization with orthogonal generalization across interleavings. Thereby, we are able to prove correctness of programs otherwise out-of-reach for interpolation-based verification. The competition results show significant advantages over other concurrency approaches in the Ultimate family.

1 Verification Approach

Ultimate GemCutter is a verification tool for concurrent programs based on the CEGAR paradigm: It *(1)* picks a trace from the set of all program interleavings (a possible *"counterexample"*), *(2)* proves correctness of this trace (the counterexample is *"spurious"*), and *(3)* generalizes the proof to conclude that a larger (usually infinite) set of traces is correct. Classically, CEGAR focuses on generalization across traces with varying numbers of loop iterations, by finding inductive loop invariants. GemCutter proposes additional generalization along an orthogonal axis: *across interleavings*.



Concurrent programs contain many redundant interleavings of actions from different threads, i.e., interleavings with the same (input/output-) behaviour. A naïve application of CEGAR requires explicit proofs of correctness for all these interleavings. Intermediate states during execution of redundant interleavings differ, and different interleavings often require different correctness proofs. GemCutter addresses this as illustrated in the figure on the right: We prove correctness of a trace τ, here $\tau = a_1a_2b$, where a_1, a_2 are actions of the first thread,

* Jury Member: Dominik Klumpp

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 479–483, 2022.
https://doi.org/10.1007/978-3-030-99527-0_35

and b is an action of the second thread. The proof of correctness is generated using Craig interpolation or similar techniques. We generalize this proof into a Floyd-Hoare automaton [8] to show that a regular language L (green area in the figure above) of traces is correct. The new contribution is the subsequent generalization step: If a trace τ_1 differs from a (correct) trace τ_2 in L only by the ordering of *independent* statements, these traces are *(Mazurkiewicz-) equivalent* [3]. We conclude that τ_1 is also correct. Hence, the set of all such traces, denoted $cl(L)$ (pink area), contains only correct traces. If the set of all program interleavings P is a subset of $cl(L)$, we conclude that the program is correct.

To soundly make this conclusion, we need a suitable notion of *independence* between statements, which guarantees that the order of execution of two independent statements does not matter for program correctness. An intuitive sufficient condition is that neither statement writes to a memory location read or written by the other statement. If we cannot establish this condition syntactically, we use an SMT solver to check if executing the statements in either order is guaranteed to give the same result. We use information from the Floyd-Hoare automaton to refine this check in the style of *conditional independence* [5]. Such information can for instance express (but is not limited to) non-aliasing of pointers.

However, the inclusion $P \subseteq cl(L)$ is in general undecidable [3]; $cl(L)$ may not be regular. We reverse our viewpoint to provide a sufficient condition that can be effectively checked: Rather than adding all equivalent traces to L – thus obtaining $cl(L)$ –, we instead remove all but one trace of each equivalence class from P – yielding a *reduction* P' of P (formally, $cl(P') = P$). We use the *sleep set* technique [5] to remove transitions from an automaton for P to get an automaton that recognizes one such reduction P'. We then check whether the (regular) reduction P' is included in the (regular) language L. If this inclusion $P' \subseteq L$ holds, it implies that $P \subseteq cl(L)$ also holds, and the program is correct. If the inclusion does not (yet) hold, GemCutter picks another program trace and repeats the process, iteratively building up the language L of correct traces by taking the union of the Floyd-Hoare automata computed in all iterations.

A key feature of the reduction-based approach is that the generalization along the iteration and interleaving axes is combined not just additively, but multiplicatively: In the geometrical intuition of the figure above, we do not just take the union of L (green area) with the equivalence class $[\tau]$ of τ (blue area), but consider all traces in $cl(L)$ (the pink area which is spanned by both). Further, we heuristically try to pick a set of representatives in a way that harmonizes with CEGAR generalization, i.e., a reduction P' with simple loop invariants. To this end, we prefer representatives with context-switches at all loop boundaries. Ideally, each thread performs one complete loop iteration and then hands control over to the next thread (the last thread hands back control to the first thread). Consider the example program on the right, with the postcondition $x = y$. Here, a proof for the

```
// Thread 1:
int x = 0;
for (int i = 0; i < N; ++i) {
  x += A[i];
}
```

```
// Thread 2:
int y = 0;
for (int j = 0; j < N; ++j) {
  y += A[j];
}
```

set of all interleavings P, or some inopportunely chosen reduction, needs invariants that capture the fact that $x = \sum_{k=0}^{i} A[k]$, and similar for y. Such invariants are usually not found by Craig interpolation. However, the loop invariant $i = j \wedge x = y$ suffices for the reduction that places context-switches at all loop boundaries. The general idea is that for this kind of reduction, the proof often needs to summarize only the effect of a single loop iteration rather than unboundedly many iterations (which may require quantifiers or non-linear arithmetic). Similar observations were first made by Farzan and Vandikas [4].

GemCutter furthermore aims to improve efficiency of the proof check, i.e., the check whether a reduction P' is a subset of the set of proven traces L. The state explosion problem of concurrent programs makes the computation of an automaton recognizing a reduction P' as well as the subsequent inclusion check prohibitively expensive. To address this, we implemented a form of *persistent set reduction* [5], which allows us to compute a more compact automaton recognizing P'. This results in a more time- and memory-efficient inclusion check.

Reductions that interact harmoniously with CEGAR generalization do not always allow for an efficient proof check, nor vice versa. In the ConcurrencySafety category, where correctness proofs may become complicated, we prioritize generalization by computing reductions that typically allow for simpler proofs (described above), even though proof checking for such reductions is often more expensive. By contrast, in the NoDataRace category we found proof assertions to be usually quite simple (often only expressing non-aliasing of pointers), so we prioritize faster proof checks (and postpone context-switches as far as possible).

Implementation GemCutter uses the libraries and the front-end of the Ultimate framework, and extends Ultimate with a new CEGAR loop implementation and new algorithms operating on finite automata. We represent programs P, reductions P' and sets of proven traces L as finite automata. Ultimate constructs Floyd-Hoare automata (for L) only on-demand [7]. Due to the state explosion problem, GemCutter extends this approach to the program and the reduction. The necessary parts of the automata are constructed just-in-time during traversal by automata algorithms. Various techniques are implemented as instances of a few generic interfaces (on-demand automata, and visitors that monitor and guide automaton traversal) for flexibility: Radically different algorithms can be created by configuring, exchanging and stacking interface implementations. The following techniques and optimizations (all used in SV-COMP) can be combined with each other independently: **(i)** sleep set reduction; **(ii)** persistent set reduction; **(iii)** discovery and pruning of states that cannot reach accepting states; **(iv)** guidance towards representatives of a specific form, e.g. with context-switches at loop boundaries; and **(v)** inclusion check between automata.

2 Strengths and Weaknesses

The main advantage over other concurrency approaches in Ultimate (in Automizer and Taipan) lies in the generalization across interleavings: Automizer

and Taipan typically require more complex proofs possibly out-of-reach for Craig interpolation and similar techniques. GemCutter performs significantly better, winning 3rd place in the ConcurrencySafety category (behind the bounded model checkers Deagle [6] and CSeq [10]) and 1st place in the NoDataRace demo category. For details, refer to the competition report [1].

Since our proof check decides a stronger condition ($P' \subseteq L$), it might miss some cases in which the proof is actually sufficient, i.e., $P \subseteq cl(L)$ holds. This is because P' and L might contain different representatives for the same equivalence class of interleavings. This weakness cannot be resolved completely due to the undecidability of the inclusion $P \subseteq cl(L)$. It can however be attenuated by considering other choices of representatives (other than preferring context-switches at loop boundaries) and exploring the effect. This choice is currently given as an input parameter; an approach that heuristically chooses a reduction based on the program structure might perform better. Our notion of independence between statements is currently ignorant of the specification being verified. We hope to extend our approach to take this into account. Finally, our approach (and implementation) can be easily extended with other reduction methods that correspond to more aggressive generalization along the interleaving axis.

Our approach only verifies programs with a bounded number of threads. GemCutter runs out of time or memory if it is unable to establish such an upper bound, e.g. for many benchmarks in `pthread-ext/` or `goblint-regression/`.

3 Architecture, Setup, Configuration, and Project

GemCutter is part of the program analysis framework Ultimate[3], written in Java and licensed under LGPLv3[4]. GemCutter version 0.2.2-839c364b requires Java 11 and Python 3.6. Its Linux version, binaries of the required SMT solvers[5], and a Python wrapper script were submitted as a `.zip` archive. GemCutter is invoked with

```
./Ultimate.py --spec <p> --file <f> --architecture <a> --full-output
```

where `<p>` is an SV-COMP property file, `<f>` is an input C file, `<a>` is the architecture (`32bit` or `64bit`), and `--full-output` enables verbose output to `stdout`. A violation witness may be written to the file `witness.graphml`. The benchmarking tool BenchExec [2] supports GemCutter through the tool-info module `ultimategemcutter.py`[6]. GemCutter participates in the ConcurrencySafety and NoDataRace categories, as declared in its SV-COMP benchmark definition file `ugemcutter.xml`[7].

Data Availability Our `.zip` archive is available online[8] and on Zenodo [9].

[3] ultimate.informatik.uni-freiburg.de and github.com/ultimate-pa/ultimate

[4] www.gnu.org/licenses/lgpl-3.0.en.html

[5] Z3 (github.com/Z3Prover/z3), CVC4 (cvc4.github.io) and Mathsat (mathsat.fbk.eu)

[6] github.com/sosy-lab/benchexec/blob/main/benchexec/tools/ultimategemcutter.py

[7] gitlab.com/sosy-lab/sv-comp/bench-defs/-/blob/main/benchmark-defs/ugemcutter.xml

[8] gitlab.com/sosy-lab/sv-comp/archives-2022/-/blob/main/2022/ugemcutter.zip and git.io/JM69B

References

1. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
2. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
3. Diekert, V., Rozenberg, G. (eds.): The Book of Traces. World Scientific (1995). https://doi.org/10.1142/2563
4. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: CAV (1). Lecture Notes in Computer Science, vol. 11561, pp. 200–218. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_11
5. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7
6. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). Springer (2022)
7. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Nutz, A., Musa, B., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Automizer with an on-demand construction of Floyd-Hoare automata - (competition contribution). In: TACAS (2). Lecture Notes in Computer Science, vol. 10206, pp. 394–398 (2017). https://doi.org/10.1007/978-3-662-54580-5_30
8. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: SAS. Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
9. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podelski, A.: Ultimate GemCutter SV-COMP 2022 Competition Contribution (Nov 2021). https://doi.org/10.5281/zenodo.5956945
10. Sales, E., Coto, A., Inverso, O., Tuosto, E.: A prototype for data race detection in CSeq 3 (competition contribution). In: Proc. TACAS (2). Springer (2022)

Wit4Java: A Violation-Witness Validator for Java Verifiers (Competition Contribution)

Tong Wu[1], Peter Schrammel[2], and Lucas C. Cordeiro[1](✉)

[1] University of Manchester, Manchester, United Kingdom
[2] University of Sussex, Brighton, and Diffblue Ltd, Oxford, United Kingdom
lucas.cordeiro@manchester.ac.uk

Abstract. We describe and evaluate a violation-witness validator for Java verifiers called Wit4Java. It takes a Java program with a safety property and the respective violation-witness output by a Java verifier to generate a new Java program whose execution deterministically violates the property. We extract the value of the program variables from the counterexample represented by the violation-witness and feed this information back into the original program. In addition, we have two implementations for instantiating source programs by injecting counterexamples. Experimental results show that Wit4Java can correctly validate the violation-witnesses produced by JBMC and GDart in a few seconds.

Keywords: Witness Validation · Software Verification · Java Bytecode.

1 Overview

Witness validation is the process of checking whether the same results can be reproduced independently according to the given program, specification, verification result, and the generated witness, improving the trust level of the software verifiers [2].

Here, we describe and evaluate a new violation-witness validator for Java programs called Wit4Java. We take an approach similar to Rocha et al. [5] and Beyer et al. [1] for C programs and apply it to Java programs. As a result, we implement Wit4Java as a Python script that creates a new Java program or a unit test case using Mockito with the program variable values extracted from the counterexample. As input, Wit4Java uses the violation-witness in the GraphML format to extract the value of the non-deterministic variables in Java programs. Lastly, Wit4Java runs the new created program using the Java Virtual Machine (JVM) to check the *assert* statements.

There are some validators for C programs in the literature [6,12]. For example, NitWit is an interpretation-based witness validator that can execute each statement step-by-step without compiling the entire program [12]. The concept of MetaVal is to generate a new program based on the input and then use any checker to check for specifications [6]. CPA-witness2test and FShell-witness2test are execution-based validators for C programs that can process the witness in

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 484–489, 2022.
https://doi.org/10.1007/978-3-030-99527-0_36

GraphML format and generate a test harness that drives the program to the specification violation [1]. Rocha et al. focus on the counterexample produced by ESBMC [4] while CPA-witness2test and FShell-witness2test can process GraphML files. However, witness validation for SV-COMP's Java track [7] is still at an early stage. GWIT is another validator that uses assumptions to prune the search space for dynamic symbolic execution, limiting the analysis to paths where a given assumption holds [10,11].

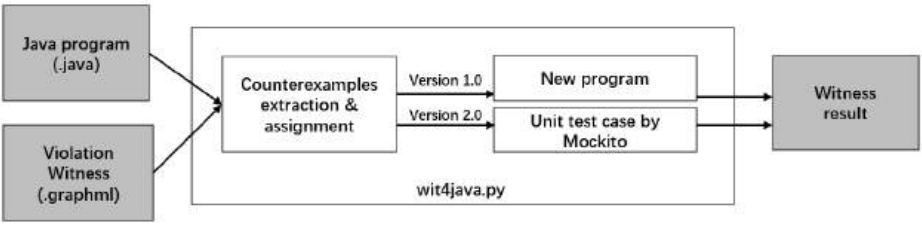


Fig. 1. Wit4Java Architecture. The grey boxes represent the inputs and outputs, and the white boxes represent the validation process.

2 Validation Approach

The architecture of Wit4Java is illustrated in Fig. 1. First, Wit4Java takes the Java program and the witness as input. Then, it uses the Python package NetworkX to read the graph content of the witness and extracts the counterexample values of the variables corresponding to the source program from the violation-witness and saves them. After that, it generates new programs that contain the witness's assumptions. Finally, the validation process is performed by the JVM (using the `-ea` option) to check whether the execution of the generated program exhibits the detected assertion failure.

There are two implementations (Wit4Java 1.0 and Wit4Java 2.0) to extract and use counterexamples. The first version is to save them as tuples (*linenum*, *counterexample*). Then it reads the source program and replaces the variables of the program statements with counterexamples if the line number and variable in the program match the tuple, thus generating a new created Java program. In comparison, the second version records the data types and values of the counterexamples and saves them sequentially into two lists. Moreover, only the assumptions made in the witness for the non-deterministic variables (as determined by Verifier.nondet) are recorded. Then, it builds a unit test case and employs the Mockito framework to mock the Verifier.nondet calls in the source program to make them return deterministic counterexample values from the lists. This makes the execution of the source program follow the path described in the witness and eventually reach the violated property.

Listing 1.1. Analyzed program

```
int v1 = Verifier.nondetInt();
int v2 = Verifier.nondetInt();
assert v1 == v2;
```

Listing 1.2. Output of Wit4Java 1.0

```
int v1 = 1;
int v2 = 0;
assert v1 == v2;
```

We show examples for both implementations in Listings 1.1 to 1.4. Wit4Java 1.0 (the naive version) saves the counterexamples in witness in line number order. It directly replaces the variable values in the source program, thus generating a new program (cf. Listing 1.2). Wit4Java 2.0 (We name it the Mockito version) generates a test case that returns the counterexample value when the mocked function is called (cf. Listing 1.4).

Listing 1.3. Violation witness

```
<edge source="203.167" target="207.186">
      <data key="originfile">
      Main.java
      </data>
      <data key="startline">
      13
      </data>
      <data key="assumption">
      v1 = 1;
      </data>
</edge>
 <edge source="207.186" target="252.201">
      <data key="originfile">
      Main.java
      </data>
      <data key="startline">
      14
      </data>
      <data key="assumption">
      v2 = 0;
      </data>
 </edge>
```

Listing 1.4. Output of Wit4Java 2.0

```
List_type = [int, int];
List_value = [1, 0];
Mockito.mockStatic(Verifier.class);
int n = List_type.length;
OngoingStubbing <Integer>
  stubbing_int = Mockito.
    when(Verifier.nondetInt());
for (int i = 0; i < n; i++) {
 if ("int".equals(List_type[i])) {
   stubbing_int = stubbing_int.
     thenReturn(Integer.
       parseInt(List_value[i]));
 }
}
Main.main(new String[0]);
```

3 Discussion of Strengths and Weaknesses

Fig. 2 on the left compares the validation results of the two validation tools Wit4Java and GWIT. The former is based on version 1.0 (naive version). The latter is based on violation-witnesses produced by GDart. The results indicate that Wit4Java has successfully validated 140 out of 302 witnesses, while GWIT correctly validates 150 results. Version 2.0 handles counterexamples with different values for each iteration within a loop better than version 1.0. This is because version 1.0 skips the counterexamples before the last iteration. However, version 2.0 can fully use the counterexamples generated by each iteration. Fig. 2 on the right compares the validation results of the two versions of Wit4Java, which shows that version 2.0 (Mockito version) has a better validation ability (168 out of 302), thereby outperforming both version 1.0 and GWIT. However, the tool can only handle witnesses with concrete counterexamples. There are two main reasons why Wit4Java shows the result *unknown*: JBMC [3,8] produces an empty witness, or the witness does not contain a counterexample for a non-deterministic value. Besides, the validation for strings is not supported yet, which occurs in almost half of witnesses because JBMC does not yet output counterexample values for strings. Thus we were not able to test it. Generally, there are not enough

witnesses of high quality for testing the witness validator yet because JBMC sometimes correctly terminates without producing a witness in SV-COMP. The witness support in the Java verifiers requires further development work so that they are able to produce complete violation witnesses whenever they terminate with verdict false.

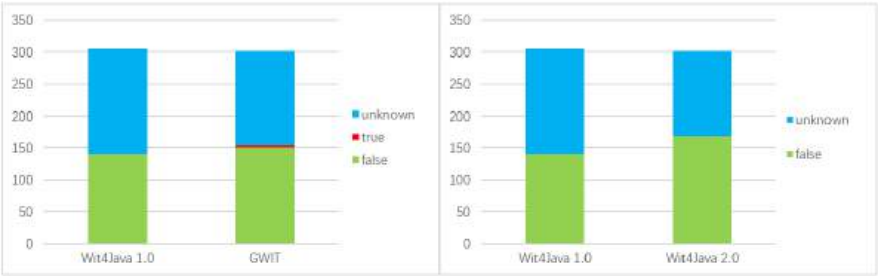


Fig. 2. Validation results based on 302 witnesses. The x-axis represents the names of the two tools and the y-axis represents the number of witnesses. A green "false" indicates a confirmed correct result.

4 Tool Setup and Configuration

The competition submission is based on Wit4Java version 1.0 (naive version).[3] For the competition [9], Wit4Java is called by executing the script wit4java.py. It reads .java source files and corresponding witnesses in the given benchmark directories. The answer would be *false* if the assertion failure was found. As an example, we can validate the witness by executing the following command:

./wit4java.py -witness <path-to-sv-witnesses>/witness.graphml <path-to-sv-benchmarks>/java/jbmc-regression/return2

where witness.graphml indicates the witness to be validated, and return2 indicates the benchmark name. The Benchexec tool info module is called wit4java.py and the benchmark definition file wit4java-validate-violation-witnesses.xml. NetworkX should be installed separately in the SV-COMP machines. If a validation task does not find a property violation, it will return *unknown*.

5 Software Project and Contributors

Tong Wu maintains Wit4Java. It is publicly available under a BSD-style license. The source code is available at https://github.com/Anthonysdu/wit4java, and instructions for running the tool are given in the README file.

[3] https://github.com/Anthonysdu/wit4java

Acknowledgment

The work in this paper is partially funded by the EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

References

1. Beyer et al. “Tests from Witnesses - Execution-Based Validation of Verification Results”. In: Tests and Proofs - 12th International Conference, TAP@STAF. Vol. 10889. Lecture Notes in Computer Science. 2018, pp. 3–23. https://doi.org/10.1007/978-3-319-92994-1_1.
2. Beyer et al. “Witness validation and stepwise testification across software veri- fiers”. In: ESEC/FSE. 2015, pp. 721–733. https://doi.org/10.1145/2786805.2786867.
3. Cordeiro et al. “JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode”. In: CAV. Vol. 10981. LNCS. 2018, pp. 183–190. https://doi.org/10.1007/978-3-319-96145-3_10.
4. Gadelha et al. “ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference - (Competition Contribution)”. In: TACAS: vol. 11429. LNCS. 2019, pp. 209–213. https://doi.org/10.1007/978-3-030-17502-3_15.
5. Rocha et al. “Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples”. In: IFM. Vol. 7321. LNCS. 2012, pp. 128–142. https://doi.org/10.1007/978-3-642-30729-4_10.
6. Dirk Beyer and Martin Spiessl. “MetaVal: Witness Validation via Verification”. In: CAV Part II. Vol. 12225. LNCS. 2020, pp. 165–177. https://doi.org/10.1007/978-3-030-53291-8_10.
7. Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. “Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP)”. In: ACM SIGSOFT Softw. Eng. Notes 43.4 (2018), p. 56. https://doi.org/10.1145/3282517.3282529.
8. Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. “JBMC: Bounded Model Checking for Java Bytecode - (Competition Contribution)”. In: TACAS. Vol. 11429. LNCS. 2019, pp. 219–223. https://doi.org/10.1007/978-3-030-17502-3_17.
9. D. Beyer. “Progress on Software Verification: SV-COMP 2022”. In: Proc. TACAS. Springer, 2022.
10. Falk Howar and Malte Mues. “GWIT (Competition Contribution)”. In: Proc. TACAS (2). Springer, 2022.
11. Falk Howar and Malte Mues. Tudo-Aqua/gwit. https://github.com/tudo-aqua/gwit.
12. Jan Svejda, Philipp Berger, and Joost-Pieter Katoen. “Interpretation-Based Violation Witness Validation for C: NITWIT”. In: TACAS. Vol. 12078. LNCS. 2020, pp. 40–57. https://doi.org/10.1007/978-3-030-45190-5_3.

Author Index