LNCS 14572

ARCoSS

Bernd Finkbeiner
Laura Kovács (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

30th International Conference, TACAS 2024
Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part III

3 Part III

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Springer

OPEN ACCESS

Lecture Notes in Computer Science 14572

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

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

Bernd Finkbeiner · Laura Kovács
Editors

Tools and Algorithms for the Construction and Analysis of Systems

30th International Conference, TACAS 2024
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part III

Springer

Editors

Bernd Finkbeiner
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Laura Kovács
TU Wien
Vienna, Austria

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-57255-5 ISBN 978-3-031-57256-2 (eBook)
https://doi.org/10.1007/978-3-031-57256-2

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

Paper in this product is recyclable.

ETAPS Foreword

Welcome to the 27th ETAPS! ETAPS 2024 took place in Luxembourg City, the beautiful capital of Luxembourg.

ETAPS 2024 is the 27th 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. Organising these conferences in a coherent, highly synchronized conference programme enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences. On the weekend before the main conference, numerous satellite workshops took place that attracted many researchers from all over the globe.

ETAPS 2024 received 352 submissions in total, 117 of which were accepted, yielding an overall acceptance rate of 33%. 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 2024 featured the unifying invited speakers Sandrine Blazy (University of Rennes, France) and Lars Birkedal (Aarhus University, Denmark), and the invited speakers Ruzica Piskac (Yale University, USA) for TACAS and Jérôme Leroux (Laboratoire Bordelais de Recherche en Informatique, France) for FoSSaCS. Invited tutorials were provided by Tamar Sharon (Radboud University, the Netherlands) on computer ethics and David Monniaux (Verimag, France) on abstract interpretation.

As part of the programme we had the first ETAPS industry day. The goal of this day was to bring industrial practitioners into the heart of the research community and to catalyze the interaction between industry and academia. The day was organized by Nikolai Kosmatov (Thales Research and Technology, France) and Andrzej Wąsowski (IT University of Copenhagen, Denmark).

ETAPS 2024 was organized by the SnT - Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg. The University of Luxembourg was founded in 2003. The university is one of the best and most international young universities with 6,000 students from 130 countries and 1,500 academics from all over the globe. The local organisation team consisted of Peter Y.A. Ryan (general chair), Peter B. Roenne (organisation chair), Maxime Cordy and Renzo Gaston Degiovanni (workshop chairs), Magali Martin and Isana Nascimento (event manager), Marjan Skrobot (publicity chair), and Afonso Arriaga (local proceedings chair). This team also organised the online edition of ETAPS 2021, and now we are happy that they agreed to also organise a physical edition of ETAPS.

ETAPS 2024 is further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology).

The 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 Marieke Huisman (Twente, chair), Andrzej Wąsowski (Copenhagen), Thomas Noll (Aachen), Jan Kofroň (Prague), Barbara König (Duisburg), Arnd Hartmanns (Twente), Caterina Urban (Inria), Jan Křetínský (Munich), Elizabeth Polgreen (Edinburgh), and Lenore Zuck (Chicago).

Other members of the steering committee are: Maurice ter Beek (Pisa), Dirk Beyer (Munich), Artur Boronat (Leicester), Luís Caires (Lisboa), Ana Cavalcanti (York), Ferruccio Damiani (Torino), Bernd Finkbeiner (Saarland), Gordon Fraser (Passau), Arie Gurfinkel (Waterloo), Reiner Hähnle (Darmstadt), Reiko Heckel (Leicester), Marijn Heule (Pittsburgh), Joost-Pieter Katoen (Aachen and Twente), Delia Kesner (Paris), Naoki Kobayashi (Tokyo), Fabrice Kordon (Paris), Laura Kovács (Vienna), Mark Lawford (Hamilton), Tiziana Margaria (Limerick), Claudio Menghi (Hamilton and Bergamo), Andrzej Murawski (Oxford), Laure Petrucci (Paris), Peter Y.A. Ryan (Luxembourg), Don Sannella (Edinburgh), Viktor Vafeiadis (Kaiserslautern), Stephanie Weirich (Pennsylvania), Anton Wijs (Eindhoven), and James Worrell (Oxford).

I would like to take this opportunity to thank all authors, keynote speakers, attendees, organizers of the satellite workshops, and Springer Nature for their support. ETAPS 2024 was also generously supported by a RESCOM grant from the Luxembourg National Research Foundation (project 18015543). I hope you all enjoyed ETAPS 2024.

Finally, a big thanks to both Peters, Magali and Isana and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

April 2024

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

This three-volume proceedings contains the papers presented at the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024). TACAS 2024 was part of the 27th European Joint Conferences on Theory and Practice of Software (ETAPS 2024), which was held between April 6–11, 2024, in Luxembourg City, Luxembourg.

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 systems. TACAS 2024 interleaves and integrates various disciplines, including formal verification of software and hardware systems, static analysis, probabilistic programming, program synthesis, concurrency, testing, simulations, verification of machine learning/autonomous systems, Cyber-Physical Systems, SAT/SMT solving, automated and interactive theorem proving, and proof checking.

There were four submission categories for TACAS 2024:

1. **Regular research papers** identifying and justifying a principled advance to the theoretical foundations for the construction and analysis of systems.
2. **Case study papers** describing the application of techniques developed by the community to a single problem or a set of problems of practical importance, preferably in a real-world setting.
3. **Regular tool papers** presenting a novel tool or a new version of an existing tool built using novel algorithmic and engineering techniques.
4. **Tool demonstration papers** demonstrating a new tool or application of an existing tool on a significant case-study.

Regular research, case study, and regular tool paper submissions were restricted to 16 pages, whereas tool demonstration papers to 6 pages, excluding the bibliography and appendices.

TACAS 2024 received 159 submissions, consisting of 114 regular research papers, 10 case study papers, 28 regular tool papers, and 7 tool demonstration papers. Each submission was assigned for review to at least three Program Committee (PC) members, who made use of subreviewers. Regular research papers were reviewed in double-blind mode, whereas case study, regular tool, and tool-demonstration papers were reviewed using a single-blind reviewing process.

Similarly to previous years, it was possible to submit an artifact alongside a paper. Artifact submission was mandatory for regular tool and tool demo papers, and voluntary for regular research and case study papers at TACAS 2024. 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. Artifact evaluation at TACAS 2024 consisted of two rounds. The first round implemented the mandatory artifact evaluation of regular tool and tool demonstration papers; this 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 of artifact evaluation carried out the voluntary artifact evaluation of regular research and case study papers, and took place after paper acceptance notifications were sent out; authors of accepted regular research and case study papers were able to update and revise their respective artifacts before artifact evaluation started. In both rounds, the AEC provided 3 reviews per artifact and anonymously communicated with the authors to resolve apparent technical issues. In total, 104 artifacts were submitted and the AEC evaluated a total of 62 artifacts regarding their availability, functionality, and/or reusability. Papers with an artifact that were successfully evaluated include one or more badges on the first page, certifying the respective properties.

Selected papers were requested to provide a rebuttal in case a PC review gave rise to questions. Using the review reports and rebuttals, the PC had a thorough discussion on each paper. For regular tool and tool demonstration papers, the PC also discussed the corresponding artifact, using the AEC recommendations. As a result, the PC decided to accept 53 papers, out of which there were 35 regular research papers, 11 regular tool papers, 3 case study papers, and 4 tool demonstration papers. This corresponds to an overall acceptance rate of 33%. Each accepted paper at TACAS 2024 had either all positive reviews and/or a “championing” PC member who argued in favor of accepting the paper. All accepted papers at TACAS 2024 had a positive average review score.

TACAS 2024 also hosted SV-COMP 2024, the 13th International Competition on Software Verification. This event to compare tools evaluated 59 software systems for automatic verification of C and Java programs and 17 software systems for witness validation. The TACAS 2024 proceedings contains a competition report by the SV-Comp chair and organizer. From the 46 actively participating teams, the SV-Comp jury selected 16 short papers that describe the participating verification and validation systems. These 16 short papers are also published in the proceedings and were reviewed by a separate program committee (jury); each of these short papers was assessed by at least four jury members. Two sessions in the TACAS 2024 program were reserved for the presentation of the results: (1) a presentation session with a report by the competition chair and summaries by the developer teams of participating tools, and (2) an open community meeting in the second session.

We would like to thank everyone who helped to make TACAS 2024 successful. We thank the authors for submitting their papers to TACAS 2024. The PC members and additional reviewers did an excellent job in reviewing papers: they provided detailed reports and engaged in the PC discussions. We thank the TACAS steering committee, and especially its chair, Joost-Pieter Katoen, for his valuable advice. We are grateful to the ETAPS steering committee, and in particular its chair, Marieke Huisman, for supporting our changes and suggestions on the TACAS 2024 review process and final program. We also acknowledge the invaluable support provided by the EasyChair developers. Lastly, we would like to thank the overall organization team of ETAPS 2024.

April 2024

Bernd Finkbeiner
Laura Kovács
PC Chairs

Hadar Frenkel
Michael Rawson
AEC Chairs

Dirk Beyer
SV-Comp Chair

Organization

Program Committee Chairs

Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Laura Kovács	TU Wien, Austria

Program Committee

Alessandro Abate	University of Oxford, UK
Erika Ábrahám	RWTH Aachen University, Germany
S. Akshay	IIT Bombay, India
Elvira Albert	Universidad Complutense de Madrid, Spain
Leonardo Alt	Ethereum Foundation
Suguman Bansal	Georgia Institute of Technology, USA
Nikolaj Bjørner	Microsoft Research, USA
Ahmed Bouajjani	IRIF, Université Paris Cité, France
Claudia Cauli	Amazon Web Services, UK
Rance Cleaveland	University of Maryland, USA
Mila Dalla Preda	University of Verona, Italy
Rayna Dimitrova	CISPA Helmholtz Center for Information Security, Germany
Madalina Erascu	West University of Timişoara, Romania
Javier Esparza	Technical University of Munich, Germany
Carlo A. Furia	USI - Università della Svizzera Italiana, Switzerland
Alberto Griggio	Fondazione Bruno Kessler, Italy
Arie Gurfinkel	University of Waterloo, Canada
Holger Hermanns	Saarland University, Germany
Marijn Heule	Carnegie Mellon University, USA
Hossein Hojjat	Tehran Institute for Advanced Studies, Iran
Nils Jansen	Ruhr-University Bochum, Germany and Radboud University, Netherlands
Sebastian Junges	Radboud University, Netherlands
Amir Kafshdar Goharshady	Hong Kong University of Science and Technology, China
Benjamin Lucien Kaminski	Saarland University, Germany and University College London, UK
Guy Katz	The Hebrew University of Jerusalem, Israel
Gergely Kovásznai	Eszterházy Károly University, Eger, Hungary
Tamás Kozsik	Eötvös Loránd University, Budapest, Hungary
Anthony Widjaja Lin	TU Kaiserslautern, Germany
Dorel Lucanu	Alexandru Ioan Cuza University, Romania

Filip Maric	University of Belgrade, Serbia
Laura Nenzi	University of Trieste, Italy
Aina Niemetz	Stanford University, USA
Elizabeth Polgreen	University of Edinburgh, UK
Kristin Yvonne Rozier	Iowa State University, USA
Cesar Sanchez	IMDEA Software Institute, Spain
Mark Santolucito	Barnard College, USA
Anne-Kathrin Schmuck	Max-Planck-Institute for Software Systems, Germany
Sharon Shoham	Tel Aviv University, Israel
Mihaela Sighireanu	University Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, France
Martin Suda	Czech Technical University in Prague, Czech Republic
Silvia Lizeth Tapia Tarifa	University of Oslo, Norway
Caterina Urban	Inria & ENS—PSL, France
Yakir Vizel	Technion, Israel
Tomas Vojnar	Brno University of Technology, Czech Republic
Georg Weissenbacher	TU Wien, Austria
Sarah Winkler	Free University of Bozen-Bolzano, Italy
Ningning Xie	University of Toronto and Google Brain, Canada

Artifact Evaluation Committee Chairs

Hadar Frenkel	CISPA Helmholtz Center for Information Security, Germany
Michael Rawson	TU Wien, Austria

Artifact Evaluation Committee

Tripti Agarwal	University of Utah, USA
Guy Amir	The Hebrew University of Jerusalem, Israel
Ahmed Bhayat	The University of Manchester, UK
Martin Blicha	University of Lugano, Switzerland
Alexander Bork	RWTH Aachen University, Germany
Lea Salome Brugger	ETH Zürich, Switzerland
Marco Campion	Inria & École Normale Supérieure—Université PSL, France
David Cerna	Czech Academy of Sciences Institute of Computer Science, Czech Republic
Kevin Cheang	Amazon Web Services, USA
Md Solimul Chowdhury	Carnegie Mellon University, USA
Vlad Craciun	BitDefender, UAIC, Romania
Jip J. Dekker	Monash University, Australia
Rafael Dewes	CISPA Helmholtz Center for Information Security, Germany
Oyendrila Dobe	Michigan State University, USA
Clemens Eisenhofer	TU Wien, Austria

Yizhak Elboher	The Hebrew University of Jerusalem, Israel
Raya Elsaleh	The Hebrew University of Jerusalem, Israel
Ferhat Erata	Yale University, USA
Zafer Esen	Uppsala University, Sweden
Aoyang Fang	Chinese University of Hong Kong, Shenzhen, China
Pritam Gharat	Microsoft Research, India
R. Govind	Uppsala University, Sweden
Thomas Hader	TU Wien, Austria
Philippe Heim	CISPA Helmholtz Center for Information Security, Germany
Maximilian Heisinger	Johannes Kepler University Linz, Austria
Alejandro Hernández-Cerezo	Complutense University of Madrid, Spain
Singh Hitarth	Hong Kong University of Science and Technology, China
Petra Hozzová	TU Wien, Austria
Jingmei Hu	Amazon, USA
Tobias John	University of Oslo, Norway
Martin Jonáš	Masaryk University, Czech Republic
Aniruddha Joshi	UC Berkeley, USA
Cezary Kaliszyk	University of Innsbruck, Austria
Elad Kinsbruner	Technion – Israel Institute of Technology, Israel
Åsmund Aqissiaq Arild Kløvstad	University of Oslo, Norway
Paul Kobialka	University of Oslo, Norway
Kerim Kochekov	Hong Kong University of Science and Technology, China
Satoshi Kura	National Institute of Informatics, Japan
Lorenz Leutgeb	Max Planck Institute for Informatics, Germany
Marco Lewis	Newcastle University, UK
Jing Liu	University of California, Irvine, USA
Yonghui Liu	Monash University, Australia
Ioan Vlad Luca	West University of Timişoara, Romania
Kaushik Mallik	Institute of Science and Technology Austria, Austria
Denis Mazzucato	École Normale Supérieure, France
Baoluo Meng	GE Global Research, USA
Niklas Metzger	CISPA Helmholtz Center for Information Security, Germany
Srinidhi Nagendra	Chennai Mathematical Institute, India
Jens Otten	University of Oslo, Norway
Jiří Pavela	FIT VUT, Czech Republic
Bartosz Piotrowski	IDEAS NCBR, Poland
Sumanth Prabhu	TRDDC, India
Jyoti Prakash	University of Passau, Germany
Siddharth Priya	University of Waterloo, Canada
Felipe R. Monteiro	Amazon Web Services, USA

Idan Refaeli	Hebrew University of Jerusalem, Israel
Simon Robillard	Université de Montpellier, France
Clara Rodríguez-Núñez	Complutense University of Madrid, Spain
Hans-Jörg Schurr	University of Iowa, USA
Tobias Seufert	University of Freiburg, Germany
Akshatha Shenoy	Tata Consultancy Services, India
Boris Shminke	Independent Researcher
Julian Siber	CISPA Helmholtz Center for Information Security, Germany
Cristian Simionescu	Alexandru Ioan Cuza University, Romania
Abhishek Kr Singh	Tel Aviv University, Israel
Alexander Steen	University of Greifswald, Germany
Geoff Sutcliffe	University of Miami, USA
Joseph Tafese	University of Waterloo, Canada
Jinhao Tan	University of Hong Kong, China
Abhishek Tiwari	University of Passau, Germany
Divyesh Unadkat	Synopsys, India
Lena Verscht	Saarland University and RWTH Aachen University, Germany
Christoph Wernhard	University of Potsdam, Germany
Haoze Wu	Stanford University, USA
Jiong Yang	National University of Singapore, Singapore
Yi Zhou	Carnegie Mellon University, USA

SV-COMP Program Committee and Jury

(more info: https://sv-comp.sosy-lab.org/2024/committee.php, sorted by tool name)

Dirk Beyer (Chair)	LMU Munich, Germany
Viktor Malík	Brno University of Technology, Czech Republic
Zhenbang Chen	National University of Defense Technology, China
Lei Bu	Nanjing University, China
Marek Chalupa	ISTA, Austria
Levente Bajczi	Budapest University of Technology and Economics, Hungary
Daniel Baier	LMU Munich, Germany
Thomas Lemberger	LMU Munich, Germany
Po-Chun Chien	LMU Munich, Germany
Hernán Ponce de León	Huawei Dresden Research Center, Germany
Fei He	Tsinghua University, China
Fatimah Aljaafari	University of Manchester, UK
Franz Brauße	University of Manchester, UK
Martin Spiessl	LMU Munich, Germany
Falk Howar	TU Dortmund, Germany
Simmo Saan	University of Tartu, Estonia

Hassan Mousavi	University of Tehran, Tehran Institute for Advanced Studies, Iran
Peter Schrammel	University of Sussex and Diffblue, UK
Zaiyu Cheng	University of Manchester, UK
Gidon Ernst	LMU Munich, Germany
Raphaël Monat	Inria and University of Lille, France
Jana (Philipp) Berger	RWTH Aachen, Germany
Veronika Šoková	Brno University of Technology, Czech Republic
Ravindra Metta	TCS, India
Vesal Vojdani	University of Tartu, Estonia
Nils Loose	University of Luebeck, Germany
Paulína Ayaziová	Masaryk University, Brno, Czech Republic
Martin Jonáš	Masaryk University, Brno, Czech Republic
Matthias Heizmann	University of Freiburg, Germany
Dominik Klumpp	University of Freiburg, Germany
Frank Schüssele	University of Freiburg, Germany
Daniel Dietsch	University of Freiburg, Germany
Priyanka Darke	Tata Consultancy Services, India
Marian Lingsch-Rosenfeld	LMU Munich, Germany

TACAS Steering Committee

Dirk Beyer	LMU Munich, Germany
Rance Cleaveland	University of Maryland, USA
Dana Fisman	Ben-Gurion University, Israel
Holger Hermanns	Universität des Saarlandes, Germany
Joost-Pieter Katoen (Chair)	RWTH Aachen, Germany and Universiteit Twente, Netherlands
Kim G. Larsen	Aalborg University, Denmark
Corina Păsăreanu	NASA Ames, USA

Additional Reviewers

Parosh Aziz Abdulla
Guy Amir
Andrei Arusoaie
Shaun Azzopardi
Thom Badings
Milan Banković
Chinmayi Prabhu Baramashetru
Sebastien Bardin
Ludovico Battista
Anna Becchi
Lena Becker
Sidi Mohamed Beillahi
Yoav Ben Shimon
Csaba Biró
León Bohn
Alberto Bombardelli
Wael-Amine Boutglay
Eline Bovy
Matías Brizzio
Gianpiero Cabodi
Francesca Cairoli
Marco Campion
Marco Carbone
Martin Ceresa
Kevin Cheang
Md Solimul Chowdhury

Alessandro Cimatti
Stefan Ciobaca
Cayden Codel
Jesús Correas
Arthur Correnson
Florin Craciun
Philipp Czerner
Tomáš Dacík
Luis Miguel Danielsson
Alessandro De Palma
Aldric Degorre
Rafael Dewes
Antonio Di Stasio
Denisa Diaconescu
Crystal Chang Din
Clemens Dubslaff
Serge Durand
Alec Edwards
Neta Elad
Yizhak Elboher
Raya Elsaleh
Constantin Enea
Zafer Esen
Soroush Farokhnia
Csaba Fazekas
Jan Fiedor
Emmanuel Fleury
James Fox
Felix Freiberger
Eden Frenkel
Florian Frohn
Maris Galesloot
Samir Genaim
Blaise Genest
Pamina Georgiou
Debarghya Ghoshdastidar
Adwait Godbole
Miguel Gomez-Zamalloa
Pablo Gordillo
Felipe Gorostiaga
R. Govind
Orna Grumberg
Roland Guttenberg
Serge Haddad
Philippe Heim
Martin Helfrich
Alejandro Hernández-Cerezo
Ivan Homoliak
Dániel Horpácsi
Karel Horák
Tzu-Han Hsu
Attila Házy
Miguel Isabel
Omri Isac
Radoslav Ivanov
Predrag Janicic
Chris Johannsen
Eduard Kamburjan
Ambrus Kaposi
Joost-Pieter Katoen
Lutz Klinkenberg
Paul Kobialka
Wietze Koops
Katherine Kosaian
David Kozák
Merlijn Krale
Valentin Krasotin
Loes Kruger
Gabor Kusper
Maximilian Alexander Köhl
Faezeh Labbaf
Nham Le
Matthieu Lemerre
Ondrej Lengal
Dániel Lukács
Michael Luttenberger
Viktor Malík
Alessio Mansutti
Niccolò Marastoni
Oliver Markgraf
Enrique Martin-Martin
Ruben Martins
Denis Mazzucato
Tobias Meggendorfer
Roland Meyer
Marcel Moosbrugger
Federico Mora
Alexander Nadel
Satya Prakash Nayak
Tobias Nießen
Andres Noetzli
Mohammed Nsaif

Robin Ohs
Emanuel Onica
Michele Pasqua
Andrea Pferscher
Zoltan Porkolab
Kostiantyn Potomkin
Mathias Preiner
Siddharth Priya
Tim Quatmann
Peter Rakyta
Omer Rappoport
Jakob Rath
Rodrigo Raya
Adrian Rebola Pardo
Gianluca Redondi
Joseph Reeves
Luke Rickard
Andoni Rodriguez
Clara Rodríguez-Núñez
Adam Rogalewicz
Enrique Román Calvo
Guillermo Román-Díez
Vlad Rusu
Krishna S.
Irmak Saglam
Matteo Sammartino
Raimundo Saona Urmeneta
Gaia Saveri
Andre Schidler
Christoph Schmidl
Andreas Schmidt
Yannik Schnitzer
Philipp Schröer
Stefan Schwoon
Traian Florin Serbanuta
Daqian Shao
Xujie Si
Mate Soos
Martin Steffen
Gregory Stock
Sana Stojanović-Đurđević
Bernardo Subercaseaux
Marnix Suilen
Mantas Šimkus
Máté Tejfel
Simon Thompson
Hazem Torfah
Dmitriy Traytel
Marck van der Vegt
Sarat Varanasi
Sarat Chandra Varanasi
Ennio Visconti
Sebastian Wolff
Yechuan Xia
Mitsuharu Yamamoto
Raz Yerushalmi
Emre Yolcu
Pian Yu
Hanwei Zhang
Zhiwei Zhang
Shufang Zhu
Djordje Zikelic
Zoltán Zimborás
Dominic Zimmer

Contents – Part III

Neural Networks

Testing and Verification

Games

Concurrency

13th Competition on Software Verification—SV-Comp 2024

Neural Networks

Provable Preimage Under-Approximation for Neural Networks

Xiyue Zhang(✉), Benjie Wang, and Marta Kwiatkowska

Department of Computer Science, University of Oxford, Oxford, UK
{xiyue.zhang,benjie.wang,marta.kwiatkowska}@cs.ox.ac.uk

Abstract. Neural network verification mainly focuses on local robustness properties, which can be checked by bounding the image (set of outputs) of a given input set. However, often it is important to know whether a given property holds globally for the input domain, and if not then for what proportion of the input the property is true. To analyze such properties requires computing *preimage* abstractions of neural networks. In this work, we propose an efficient anytime algorithm for generating symbolic under-approximations of the preimage of any polyhedron output set for neural networks. Our algorithm combines a novel technique for cheaply computing polytope preimage under-approximations using linear relaxation, with a carefully-designed refinement procedure that iteratively partitions the input region into subregions using input and ReLU splitting in order to improve the approximation. Empirically, we validate the efficacy of our method across a range of domains, including a high-dimensional MNIST classification task beyond the reach of existing preimage computation methods. Finally, as use cases, we showcase the application to quantitative verification and robustness analysis. We present a sound and complete algorithm for the former, which exploits our disjoint union of polytopes representation to provide formal guarantees. For the latter, we find that our method can provide useful quantitative information even when standard verifiers cannot verify a robustness property.

1 Introduction

Despite the remarkable empirical success of neural networks, guaranteeing their correctness, especially when using them as decision-making components in safety-critical autonomous systems [7, 13, 43], is an important and challenging task. Towards this aim, various approaches have been developed for the verification of neural networks, with extensive effort devoted to local robustness verification [20, 22, 44, 11, 35, 32, 40, 41, 36]. While local robustness verification focuses on deciding the absence of adversarial examples within an ϵ-perturbation neighbourhood, an alternative approach for neural network analysis is to construct the preimage of its predictions [27, 15]. Given a set of outputs, the preimage is defined as the set of all inputs mapped by the neural network to that output set. By characterizing the preimage symbolically in an abstract representation, e.g.,

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 3–23, 2024.
https://doi.org/10.1007/978-3-031-57256-2_1

polyhedra, one can perform more complex analysis for a wider class of properties beyond local robustness, such as computing the *proportion* of inputs satisfying a property (quantitative verification) even if standard robustness verification fails.

Exact preimage generation [27] is intractable, taking time exponential in the number of neurons in a network; thus approximations are necessary. Unfortunately, existing methods are limited in their applicability. The inverse abstraction method in [15] bypasses the intractability of exact preimage generation by leveraging symbolic interpolants [14, 2] for abstraction of neural network layers. However, due to the complexity of interpolation, the time to compute the abstraction also scales exponentially with the number of neurons in hidden layers. A concurrent work [23] proposed an input bounding algorithm targeting backward reachability analysis for control policies and out-of-distribution (OOD) detection in low-dimensional domains. Their method produces a preimage *over-approximation*, which cannot be used for quantitative verification. Therefore, more efficient and flexible computation methods for (symbolic abstraction of) preimages of neural networks are needed.

The main contribution of this paper is a scalable method for preimage approximation, which can be used for a variety of robustness analysis tasks. More specifically, we propose an efficient *anytime* algorithm for generating symbolic under-approximations of the preimage of piecewise linear neural networks as a union of disjoint polytopes. The algorithm computes a sound preimage under-approximation leveraging linear relaxation based perturbation analysis (LiRPA) [40, 41, 32], applied backwards from a polyhedron output set. It iteratively refines the preimage approximation by adding input and/or intermediate (ReLU) splitting (hyper)planes to partition the input region into disjoint subregions, which can be approximated independently in parallel in a divide-and-conquer approach. The refinement scheme uses a novel differential objective to optimize the quality (volume) of the polytope subregions. We also show that our method can be generalized to generate preimage over-approximations. We illustrate the application of our method to quantitative verification, input bounding for control tasks, and robustness analysis against adversarial and patch attacks. Finally, we conduct an empirical analysis on a range of control and computer vision tasks, showing significant gains in efficiency compared to exact preimage generation methods and scalability to high-input-dimensional tasks compared to existing preimage approximation methods.

For space reasons, proofs and additional technical details have been moved to Appendix of the full version of the paper [45].

2 Preliminaries

We use $f : \mathbb{R}^d \to \mathbb{R}^m$ to denote a feedforward neural network. For layer i, we use $\mathbf{W}^{(i)}$ to denote the weight matrix, $\mathbf{b}^{(i)}$ the bias, $h^{(i)}$ the pre-activation neurons, and $a^{(i)}$ the post-activation neurons, such that we have $h^{(i)} = \mathbf{W}^{(i)} a^{(i-1)} + \mathbf{b}^{(i)}$. In this paper, we focus on ReLU neural networks with $a^{(i)}(x) = ReLU(h^{(i)}(x))$,

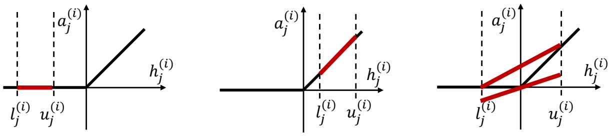


Fig. 1: Linear bounding functions for inactive, active, unstable ReLU neurons.

where $\mathrm{ReLU}(h) := \max(h, 0)$ is applied element-wise. However, our method can be generalized to other activation functions bounded by linear relaxation [44].

Linear Relaxation of Neural Networks. Nonlinear activation functions lead to the NP-completeness of the neural network verification problem [22]. To address such intractability, linear relaxation is often used to transform the nonconvex constraints into linear programs. As shown in Figure 1, given *concrete* lower and upper bounds $\mathbf{l}^{(i)} \leq h^{(i)}(x) \leq \mathbf{u}^{(i)}$ on the pre-activation values of layer i, there are three cases to consider. In the *inactive* ($u_j^{(i)} \leq 0$) and *active* ($l_j^{(i)} \geq 0$) cases, the post-activation neurons $a_j^{(i)}(x)$ are linear functions $a_j^{(i)}(x) = 0$ and $a_j^{(i)}(x) = h_j^{(i)}(x)$ respectively. In the *unstable* case, $a_j^{(i)}(x)$ can be bounded by $\alpha_j^{(i)} h_j^{(i)}(x) \leq a_j^{(i)}(x) \leq -\frac{u_j^{(i)} l_j^{(i)}}{u_j^{(i)} - l_j^{(i)}} + \frac{u_j^{(i)}}{u_j^{(i)} - l_j^{(i)}} h_j^{(i)}(x)$, where $\alpha_j^{(i)}$ is a configurable parameter that produces a valid lower bound for any value in $[0, 1]$. Linear bounds can also be obtained for other non-piecewise linear activation functions [44].

Linear relaxation can be used to compute linear upper and lower bounds of the form $\underline{\mathbf{A}}x + \underline{\mathbf{b}} \leq f(x) \leq \overline{\mathbf{A}}x + \overline{\mathbf{b}}$ on the output of a neural network, for a given bounded input region $\mathcal{C}$. These methods are known as linear relaxation based perturbation analysis (LiRPA) algorithms [40, 41, 32]. In particular, *backward-mode* LiRPA computes linear bounds on f by propagating linear bounding functions backward from the output, layer-by-layer, to the input layer.

Polytope Representations. Given an Euclidean space $\mathbb{R}^d$, a polyhedron T is defined to be the intersection of a set of half spaces. More formally, suppose we have a set of linear constraints defined by $\psi_i(x) := c_i^T x + d_i \geq 0$ for $i = 1, ...K$, where $c_i \in \mathbb{R}^d, d_i \in \mathbb{R}$ are constants, and $x = x_1, ..., x_d$ is a set of variables. Then a polyhedron is defined as $T = \{x \in \mathbb{R}^d | \bigwedge_{i=1}^K \psi_i(x)\}$, where T consists of all values of x satisfying the first-order logic (FOL) formula $\alpha(x) := \bigwedge_{i=1}^K \psi_i(x)$. We use the term polytope to refer to a bounded polyhedron, that is, a polyhedron T such that $\exists R \in \mathbb{R}^{>0} : \forall x_1, x_2 \in T, \|x_1 - x_2\|_2 \leq R$ holds. The abstract domain of polyhedra [32, 6, 8] has been widely used for the verification of neural networks and computer programs. An important type of polytope is the hyperrectangle (box), which is a polytope defined by a closed and bounded interval $[\underline{x_i}, \overline{x_i}]$ for each dimension, where $\underline{x_i}, \overline{x_i} \in \mathbb{Q}$. More formally, using the linear constraints $\phi_i := (x_i \geq \underline{x_i}) \wedge (x_i \leq \overline{x_i})$ for each dimension, the hyperrectangle takes the form $\mathcal{C} = \{x \in \mathbb{R}^d | x \models \bigwedge_{i=1}^d \phi_i\}$.

3 Problem Formulation

3.1 Preimage Approximation

In this work, we are interested in the problem of computing preimages for neural networks. Given a subset $O \subset \mathbb{R}^m$ of the codomain, the preimage of a function $f : \mathbb{R}^d \to \mathbb{R}^m$ is defined to be the set of all inputs $x \in \mathbb{R}^d$ that are mapped to an element of O by f. For neural networks in particular, the input is typically restricted to some bounded input region $\mathcal{C} \subset \mathbb{R}^d$. In this work, we restrict the output set O to be a polyhedron, and the input set $\mathcal{C}$ to be an axis-aligned hyperrectangle region $\mathcal{C} \subset \mathbb{R}^d$, as these are commonly used in neural network verification. We now define the notion of a restricted preimage:

Definition 1 (Restricted Preimage). *Given a neural network $f : \mathbb{R}^d \to \mathbb{R}^m$, and an input set $\mathcal{C} \subset \mathbb{R}^d$, the restricted preimage of an output set $O \subset \mathbb{R}^m$ is defined to be the set $f_{\mathcal{C}}^{-1}(O) := \{x \in \mathbb{R}^d | f(x) \in O \wedge x \in \mathcal{C}\}$.*

Example 1. To illustrate our problem formulation and approach, we introduce a vehicle parking task [3] as a running example. In this task, there are four parking lots, located in each quadrant of a 2 × 2 grid $[0, 2]^2$, and a neural network with two hidden layers of 10 ReLU neurons $f : \mathbb{R}^2 \to \mathbb{R}^4$ is trained to classify which parking lot an input point belongs to. To analyze the behaviour of the neural network in the input region $[0, 1] \times [0, 1]$ corresponding to parking lot 1, we set $\mathcal{C} = \{x \in \mathbb{R}^2 | (0 \leq x_1 \leq 1) \wedge (0 \leq x_2 \leq 1)\}$. Then the restricted preimage $f_{\mathcal{C}}^{-1}(O)$ of the set $O = \{\boldsymbol{y} \in \mathbb{R}^4 | \bigwedge_{i \in \{2,3,4\}} y_1 - y_i \geq 0\}$ is the subspace of the region $[0, 1] \times [0, 1]$ that is *labelled* as parking lot 1 by the network.

We focus on *provable* approximations of the preimage. Given a first-order formula A, α is an *under-approximation* (resp. *over-approximation*) of A if it holds that $\forall x.\alpha(x) \implies A(x)$ (resp. $\forall x.A(x) \implies \alpha(x)$). In our context, the restricted preimage is defined by the formula $A(x) = (f(x) \in O) \wedge (x \in \mathcal{C})$, and we restrict to approximations α that take the form of a disjoint union of polytopes (DUP). The goal of our method is to generate a DUP approximation $\mathcal{T}$ that is as tight as possible; that is, to maximize the volume $\text{vol}(\mathcal{T})$ of an under-approximation, or minimize the volume $\text{vol}(\mathcal{T})$ of an over-approximation.

Definition 2 (Disjoint Union of Polytopes). *A disjoint union of polytopes (DUP) is a FOL formula α of the form $\alpha(x) := \bigvee_{i=1}^{D} \alpha_i(x)$, where each α_i is a polytope formula (conjunction of a finite set of linear half-space constraints), with the property that $\alpha_i \wedge \alpha_j$ is unsatisfiable for any $i \neq j$.*

3.2 Quantitative Properties

One of the most important verification problems for neural networks is that of proving guarantees on the output of a network for a given input set [18, 19, 30]. This is often expressed as a property of the form (I, O) such that $\forall x \in I \implies f(x) \in O$. We can generalize this to *quantitative* properties:

Definition 3 (Quantitative Property). *Given a neural network $f : \mathbb{R}^d \to \mathbb{R}^m$, a measurable input set with non-zero measure (volume) $I \subseteq \mathbb{R}^d$, a measurable output set $O \subseteq \mathbb{R}^m$, and a rational proportion $p \in [0,1]$ we say that the neural network satisfies the property (I, O, p) if $\frac{\text{vol}(f_I^{-1}(O))}{\text{vol}(I)} \geq p$.* [1]

Neural network verification algorithms [25] can be divided into two categories: sound, which always return correct results, and complete, guaranteed to reach a conclusion on any verification query. We now define soundness and completeness of verification algorithms for quantitative properties.

Definition 4 (Soundness). *A verification algorithm QV is sound if, whenever QV outputs* True*, the property (I, O, p) holds.*

Definition 5 (Completeness). *A verification algorithm QV is complete if (i) QV never returns* Unknown*, and (ii) whenever QV outputs* False*, the property (I, O, p) does not hold.*

If the property (I, O) holds, then the quantitative property $(I, O, 1)$ holds, while quantitative properties for $0 \leq p < 1$ provide more information when (I, O) does not hold. Most neural network verification methods produce approximations of the *image* of I in the output space, which cannot be used to verify quantitative properties. Preimage *over-approximations* include false regions, thereby not applicable for quantitative verification. In contrast, preimage *under-approximations* provide a lower bound on the volume of the preimage, allowing us to soundly verify quantitative properties.

4 Methodology

Overview. In this section we present the main components of our methodology. Firstly, in Section 4.1, we show how to cheaply and soundly under-approximate the (restricted) preimage with a single polytope, using linear relaxation methods (Algorithm 2). Secondly, in Section 4.2, we propose a novel differentiable objective to optimize the quality (volume) of the polytope under-approximation. Thirdly, in Section 4.3, we propose a refinement scheme that improves the approximation by partitioning a (sub)region into subregions with splitting planes, with each subregion then being under-approximated more accurately. The main contribution of this paper (Algorithm 1) integrates these three components and is described in Section 4.4. Finally, in Section 4.5, we apply our method to quantitative verification (Algorithm 3) and prove its soundness and completeness.

4.1 Polytope Under-Approximation via Linear Relaxation

We first show how to adapt linear relaxation techniques to efficiently generate valid under-approximations to the restricted preimage for a given input region $\mathcal{C}$.

[1] In particular, the restricted preimage of a polyhedron under a neural network is Lebesgue measurable since polyhedra (intersection of a finite set of half-spaces) are Borel measurable and NNs are continuous functions.

```
Algorithm 1: Preimage Approximation
Input: Neural network $f$, Input region $\mathcal{C}$, Output region $O$, Volume threshold
       $v$, Maximum iterations $R$, Boolean $SplitOnInput$
Output: Disjoint union of polytopes $\mathcal{T}$
$T \leftarrow$ GenUnderApprox($\mathcal{C}$, $O$) ;              // Initial preimage polytope
$\widehat{\text{vol}}_T, \widehat{\text{vol}}_{f_{\mathcal{C}}^{-1}(O)} \leftarrow$ EstimateVol($T$), EstimateVol($f_{\mathcal{C}}^{-1}(O)$) ;
Dom $\leftarrow \{(\mathcal{C}, T, \widehat{\text{vol}}_{f_{\mathcal{C}}^{-1}(O)} - \widehat{\text{vol}}_T)\}$ ;                  // Priority queue
// $\mathcal{T}_{\text{Dom}}$ is the union of polytopes in Dom
while EstimateVol($\mathcal{T}_{\text{Dom}}$) $< v$ and Iterations $\leq R$ do
    $\mathcal{C}_{\text{sub}}, T$, Priority $\leftarrow$ Pop(Dom) ;    // Subregion with highest priority
    if $SplitOnInput$ then
        $id \leftarrow$ SelectInputFeature(Feature$_I$) ;      // Feature$_I$ is the set of
          input features/dimensions
    else
        $id \leftarrow$ SelectReLUNode(Node$_Z$);   // Node$_Z$ is the set of unstable
          ReLU nodes
    [$\mathcal{C}^l_{sub}$,$\mathcal{C}^u_{sub}$] $\leftarrow$ SplitOnNode($\mathcal{C}_{sub}$, $id$);     // Split on the selected node
    [$T^l$,$T^u$] $\leftarrow$ GenUnderApprox([$\mathcal{C}^l_{sub}$,$\mathcal{C}^u_{sub}$], $O$) ;       // Generate preimage
    [$\widehat{\text{vol}}_{T^l}, \widehat{\text{vol}}_{T^u}$] $\leftarrow$ EstimateVol([$T^l, T^u$]);
    $\widehat{\text{vol}}_{f^{-1}_{\mathcal{C}^l_{sub}}(O)}, \widehat{\text{vol}}_{f^{-1}_{\mathcal{C}^u_{sub}}(O)} \leftarrow$ EstimateVol($f^{-1}_{\mathcal{C}^l_{sub}}(O)$), EstimateVol($f^{-1}_{\mathcal{C}^u_{sub}}(O)$) ;
    Dom $\leftarrow$ Dom $\cup \{(\mathcal{C}^l_{sub}, T^l, \widehat{\text{vol}}_{f^{-1}_{\mathcal{C}^l_{sub}}(O)} - \widehat{\text{vol}}_{T^l})\} \cup$
      $\{(\mathcal{C}^u_{sub}, T^u, \widehat{\text{vol}}_{f^{-1}_{\mathcal{C}^u_{sub}}(O)} - \widehat{\text{vol}}_{T^u})\}$;              // Disjoint polytope
return $\mathcal{T}_{\text{Dom}}$
```

Recall that LiRPA methods enable us to obtain linear lower and upper bounds on the output of a neural network f, that is, $\underline{\mathbf{A}}x + \underline{\mathbf{b}} \leq f(x) \leq \overline{\mathbf{A}}x + \overline{\mathbf{b}}$, where the linear coefficients depend on the input region $\mathcal{C}$.

Now, suppose that we are interested in computing an under-approximation to the restricted preimage, given the input hyperrectangle $\mathcal{C} = \{x \in \mathbb{R}^d | x \models \bigwedge_{i=1}^d \phi_i\}$, and the output polytope specified using the half-space constraints $\psi_i(y) = (c_i^T y + d_i \geq 0)$ for $i = 1, ..., K$ over the output space. Given a constraint ψ_i, we append an additional linear layer at the end of the network f, which maps $y \mapsto c_i^T y + d_i$, such that the function $g_i : \mathbb{R}^d \to \mathbb{R}$ represented by the new network is $g_i(x) = c_i^T f(x) + d_i$. Then, applying LiRPA bounding to each g_i, we obtain lower bounds $\underline{g_i}(x) = \underline{a}_i^T x + \underline{b}_i$ for each i, such that $\underline{g_i}(x) \geq 0 \implies g_i(x) \geq 0$ for $x \in \mathcal{C}$. Notice that, for each $i = 1, ..., K$, $\underline{a}_i^T x + \underline{b}_i \geq 0$ is a half-space constraint in the input space. We conjoin these constraints, along with the restriction to the input region $\mathcal{C}$, to obtain a polytope $T_{\mathcal{C}}(O) := \{x | \bigwedge_{i=1}^K (\underline{g_i}(x) \geq 0) \wedge \bigwedge_{i=1}^d \phi_i(x)\}$.

Proposition 1. *$T_{\mathcal{C}}(O)$ is an under-approximation to the restricted preimage $f_{\mathcal{C}}^{-1}(O)$.*

Algorithm 2: GenUnderApprox

```
  Input: List of subregions $\mathcal{C}$, Output set $O$, number of samples $N$
  Output: List of polytopes $\mathbf{T}$
1 $\mathbf{T} = []$;
2 for subregion $\mathcal{C}_{sub} \in \mathcal{C}$ // Parallel over subregions
3 do
4     $[\underline{g_1}(x, \boldsymbol{\alpha}_1), ..., \underline{g_K}(x, \boldsymbol{\alpha}_K)] \leftarrow$ LinearLowerBound($\mathcal{C}_{sub}, O$);
5     $x_1, ..., x_N \leftarrow$ Sample($\mathcal{C}_{sub}, N$);
6     Loss($\boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_K$) $\leftarrow -\sum_{j=1,...,N} \sigma(-\mathrm{LSE}(-\underline{g_1}(x_j, \boldsymbol{\alpha}_1), ..., -\underline{g_K}(x_j, \boldsymbol{\alpha}_K))$;
7     $\boldsymbol{\alpha}_1^*, ..., \boldsymbol{\alpha}_K^* \leftarrow$ Optimize(Loss($\boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_K$));
8     $\mathbf{T}$ = Append($\mathbf{T}$, $[\underline{g_1}(x, \boldsymbol{\alpha}_1^*) \geq 0, ..., \underline{g_K}(x, \boldsymbol{\alpha}_K^*) \geq 0, x \in \mathcal{C}_{sub}]$)
9 return $\boldsymbol{T}$
```

Example 2. Returning to Example 1, the output constraints (for $i = 2, 3, 4$) are given by $\psi_i = (y_1 - y_i \geq 0) = (c_i^T y + d_i \geq 0)$, where $c_i := e_1 - e_i$ (where e_i is the i^{th} standard basis vector) and $d_i := 0$. Applying LiRPA bounding, we obtain the linear lower bounds $\underline{g_2}(x) = -3.79x_1 + x_2 + 2.65 \geq 0; \underline{g_3}(x) = 0.34x_1 - x_2 - 0.60 \geq 0; \underline{g_4}(x) = -1.11x_1 - x_2 + 1.99 \geq 0$ for each constraint. The intersection of these constraints, shown in Figure 2a, represents the region where any input is guaranteed to satisfy the output constraints.

We generate the linear bounds in parallel over the output polyhedron constraints $i = 1, ..., K$ using the *backward mode* LiRPA [44], and store the resulting input polytope $T_{\mathcal{C}}(O)$ as a list of constraints. This highly efficient procedure is used as a sub-routine `LinearLowerBound` when generating a preimage under-approximation as a polytope union using Algorithm 2 (Line 4).

4.2 Local Optimization

One of the key components behind the effectiveness of LiRPA-based bounds is the ability to efficiently improve the tightness of the bounding function by optimizing the relaxation parameters $\boldsymbol{\alpha}$, via projected gradient descent. In the context of local robustness verification, the goal is to optimize the concrete lower or upper bounds over the (sub)region $\mathcal{C}$ [40], i.e., $\min_{x \in \mathcal{C}} \underline{\mathbf{A}}(\boldsymbol{\alpha})x + \underline{\mathbf{b}}(\boldsymbol{\alpha})$, where we explicitly note the dependence of the linear coefficients on $\boldsymbol{\alpha}$. In our case, we are instead interested in optimizing $\boldsymbol{\alpha}$ to refine the polytope under-approximation, that is, increase its volume. Unfortunately, computing the volume of a polytope exactly is a computationally expensive task, and requires specialized tools [12] that do not permit easy optimization with respect to the $\boldsymbol{\alpha}$ parameters.

To address this challenge, we propose to use statistical estimation. In particular, we sample N points $x_1, ..., x_N$ uniformly from the input domain $\mathcal{C}$ then employ Monte Carlo estimation for the volume of the polytope approximation:

$$\widehat{\mathrm{vol}}(T_{\mathcal{C},\boldsymbol{\alpha}}(O)) = \frac{\sum_{i=1}^{N} \mathbb{1}_{x_i \in T_{\mathcal{C},\boldsymbol{\alpha}}(O)}}{N} \times \mathrm{vol}(\mathcal{C}) \tag{1}$$

where we highlight the dependence of $T_{\mathcal{C}}(O) = \{x | \bigwedge_{i=1}^{K} \underline{g_i}(x, \boldsymbol{\alpha}_i) \geq 0 \wedge \bigwedge_{i=1}^{d} \phi_i(x)\}$ on $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, ..., \boldsymbol{\alpha}_K)$, and $\boldsymbol{\alpha}_i$ are the α-parameters for the linear relaxation of the neural network g_i corresponding to the i^{th} half-space constraint in O. However, this is still non-differentiable w.r.t. $\boldsymbol{\alpha}$ due to the identity function. We now show how to derive a differentiable relaxation which is amenable to gradient-based optimization:

$$\begin{aligned}
\widehat{\text{vol}}(T_{\mathcal{C},\boldsymbol{\alpha}}(O)) = \frac{\text{vol}(\mathcal{C})}{N} \sum_{j=1}^{N} \mathbb{1}_{x_j \in T_{\mathcal{C},\boldsymbol{\alpha}}(O)} &= \frac{\text{vol}(\mathcal{C})}{N} \sum_{j=1}^{N} \mathbb{1}_{\min_{i=1,\dots K} \underline{g_i}(x_j, \boldsymbol{\alpha}_i) \geq 0} \\
&\approx \frac{\text{vol}(\mathcal{C})}{N} \sum_{j=1}^{N} \sigma\left(\min_{i=1,\dots K} \underline{g_i}(x_j, \boldsymbol{\alpha}_i)\right) \\
&\approx \frac{\text{vol}(\mathcal{C})}{N} \sum_{j=1}^{N} \sigma\left(-\text{LSE}(-\underline{g_1}(x_j, \boldsymbol{\alpha}_1), ..., -\underline{g_K}(x_j, \boldsymbol{\alpha}_K))\right)
\end{aligned}$$

The second equality follows from the definition of the polytope $T_{\mathcal{C},\boldsymbol{\alpha}}(O)$; namely that a point is in the polytope if it satisfies $\underline{g_i}(x_j, \boldsymbol{\alpha}_i) \geq 0$ for all $i = 1, ..., K$, or equivalently, $\min_{i=1,\dots K} \underline{g_i}(x_j, \boldsymbol{\alpha}_i) \geq 0$. After this, we approximate the identity function using a sigmoid relaxation, where $\sigma(y) := \frac{1}{1+e^{-y}}$, as is commonly done in machine learning to define classification losses. Finally, we approximate the minimum over specifications using the log-sum-exp (LSE) function. The log-sum-exp function is defined by $LSE(y_1, ..., y_K) := \log(\sum_{i=1,...,K} e^{y_i})$, and is a differentiable approximation to the maximum function; we employ it to approximate the minimization by adding the appropriate sign changes. The final expression is now a differentiable function of $\boldsymbol{\alpha}$. We employ this as the loss function in Algorithm 2 (Line 6) for generating a polytope approximation, and optimize volume using projected gradient descent.

Example 3. We revisit the vehicle parking problem in Example 1. Figure 2a and 2b show the computed under-approximations before and after local optimization. We can see that the bounding planes for all three specifications are optimized, which effectively improves the approximation quality.

4.3 Global Branching and Refinement

As LiRPA performs crude linear relaxation, the resulting bounds can be quite loose even with $\boldsymbol{\alpha}$-optimization, meaning that the polytope approximation $T_{\mathcal{C}}(O)$ is unlikely to constitute a tight under-approximation to the preimage. To address this challenge, we employ a divide-and-conquer approach that iteratively refines our under-approximation of the preimage. Starting from the initial region $\mathcal{C}$ represented at the root, our method generates a tree by iteratively partitioning a subregion $\mathcal{C}_{sub}$ represented at a leaf node into two smaller subregions $\mathcal{C}_{sub}^{l}, \mathcal{C}_{sub}^{u}$, which are then attached as children to that leaf node. In this way, the subregions represented by all leaves of the tree are disjoint, such that their union is the initial region $\mathcal{C}$.

For each leaf subregion $\mathcal{C}_{sub}$ we compute, using LiRPA bounds (Line 4, Algorithm 2), an associated polytope that under-approximates the preimage in $\mathcal{C}_{sub}$. Thus, irrespective of the number of refinements performed, the union of the polytopes corresponding to all leaves forms an *anytime* DUP under-approximation $\mathcal{T}$ to the preimage in the original region $\mathcal{C}$. The process of refining the subregions continues until an appropriate termination criterion is met.

Unfortunately, even with a moderate number of input dimensions or unstable ReLU nodes, naïvely splitting along all input- or ReLU-planes quickly becomes computationally infeasible. For example, splitting a d-dimensional hyperrectangle using bisections along each dimension results in 2^d subdomains to approximate. It thus becomes crucial to identify the subregion splits that have the most impact on the quality of the under-approximation. Another important aspect is how to prioritize which leaf subregion to split. We describe these in turn.

Subregion Selection. Searching through all leaf subregions at each iteration is computationally too expensive. Thus, we propose a subregion selection strategy that prioritizes splitting subregions according to (an estimate of) the difference in volume between the exact preimage $f^{-1}_{\mathcal{C}_{sub}}(O)$ and the (already computed) polytope approximation $T_{\mathcal{C}_{sub}}(O)$ on that subdomain, that is:

$$\text{Priority}(\mathcal{C}_{sub}) = \text{vol}(f^{-1}_{\mathcal{C}_{sub}}(O)) - \text{vol}(T_{\mathcal{C}_{sub}}(O)) \tag{2}$$

which measures the gap between the polytope under-approximation and the optimal approximation, namely, the preimage itself.

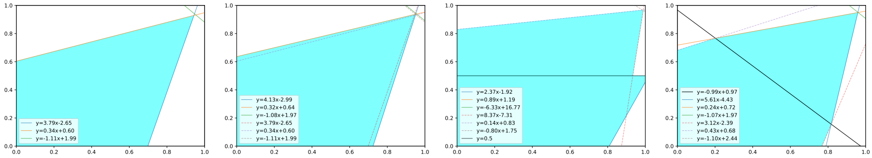
Suppose that a particular leaf subdomain attains the maximum of this metric among all leaves, and we partition it into two subregions $\mathcal{C}^l_{sub}, \mathcal{C}^u_{sub}$, which we approximate with polytopes $T_{\mathcal{C}^l_{sub}}(O), T_{\mathcal{C}^u_{sub}}(O)$. As tighter intermediate concrete bounds, and thus linear bounding functions, can be computed on the partitioned subregions, the polytope approximation on each subregion will be refined compared with the single polytope restricted to that subregion.

Proposition 2. *Given any subregion $\mathcal{C}_{sub}$ with polytope approximation $T_{\mathcal{C}_{sub}}(O)$, and its children $\mathcal{C}^l_{sub}, \mathcal{C}^u_{sub}$ with polytope approximations $T_{\mathcal{C}^l_{sub}}(O), T_{\mathcal{C}^u_{sub}}(O)$ respectively, it holds that:*

$$T_{\mathcal{C}^l_{sub}}(O) \cup T_{\mathcal{C}^u_{sub}}(O) \supseteq T_{\mathcal{C}_{sub}}(O) \tag{3}$$

Corollary 1. *In each refinement iteration, the volume of the polytope approximation $\mathcal{T}_{Dom}$ does not decrease.*

Since computing the volumes in Equation 2 is intractable, we sample N points $x_1, ..., x_N$ uniformly from the subdomain $\mathcal{C}_{sub}$ and employ Monte Carlo estimation to estimate the volume for both the preimage and the polytope approximation using the same set of samples, i.e., $\widehat{\text{vol}}(f^{-1}_{\mathcal{C}_{sub}}(O)) = \text{vol}(\mathcal{C}_{sub}) \times \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{x_i \in f^{-1}_{\mathcal{C}_{sub}}(O)}$, and $\widehat{\text{vol}}(T_{\mathcal{C}_{sub}}(O)) = \text{vol}(\mathcal{C}_{sub}) \times \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{x_i \in T_{\mathcal{C}_{sub}}(O)}$. We



(a) No optimization (b) After optimization (c) Input split (d) ReLU split

Fig. 2: Refinement and optimization for preimage approximation.

stress that volume estimation is only used to prioritize subregion selection, and does not affect the soundness of our method.

Input Splitting. Given a subregion (hyperrectangle) defined by lower and upper bounds $x_i \in [\underline{x}_i, \overline{x}_i]$ for all dimensions $i = 1, ..., d$, input splitting partitions it into two subregions by cutting along some feature i. This splitting procedure will produce two subregions which are similar to the original subregion, but have updated bounds $[\underline{x}_i, \frac{\underline{x}_i + \overline{x}_i}{2}], [\frac{\underline{x}_i + \overline{x}_i}{2}, \overline{x}_i]$ for feature i instead. In order to determine which feature/dimension to split on, we propose a greedy strategy. Specifically, for each feature, we generate a pair of polytopes for the two subregions resulting from the split, and choose the feature that results in the greatest total volume of the polytope pair. In practice, another commonly-adopted splitting heuristic is to select the dimension with the longest edge [10], that is, to select feature i with the largest range: $\arg\max_i(\overline{x}_i - \underline{x}_i)$. However, this method falls short in per-iteration approximation volume improvement compared to our greedy strategy.

Example 4. We revisit the vehicle parking problem in Example 1. Figure 2b shows the polytope under-approximation computed on the input region $\mathcal{C}$ before refinement, where each solid line represents the bounding plane for each output specification ($y_1 - y_i \geq 0$). Figure 2c depicts the refined approximation by splitting the input region along the vertical axis, where the solid and dashed lines represent the bounding planes for the two resulting subregions. It can be seen that the total volume of the under-approximation has improved significantly.

Intermediate ReLU Splitting. Refinement through splitting on input features is adequate for low-dimensional input problems such as reinforcement learning agents. However, it may be infeasible to generate sufficiently fine subregions for high-dimensional domains. We thus propose an algorithm for ReLU neural networks that uses intermediate ReLU splitting for preimage refinement. After determining a subregion for refinement, we partition the subregion based upon the pre-activation value of an intermediate unstable neuron $z_j^{(i)} = 0$. As a result, the original subregion $\mathcal{C}_{sub}$ is split into two new subregions $\mathcal{C}^+_{z_j^{(i)}} = \{x \in \mathcal{C}_{sub} \mid z_j^{(i)} = h_j^{(i)}(x) \geq 0\}$ and $\mathcal{C}^-_{z_j^{(i)}} = \{x \in \mathcal{C}_{sub} \mid z_j^{(i)} = h_j^{(i)}(x) < 0\}$.[2]

[2] To obtain a polytope under-approximation, we can utilize linear lower/upper bounds on $h_j^{(i)}(x)$ as an approximation to the subregion boundary.

In this procedure, the order of splitting unstable ReLU neurons can greatly influence the refinement quality and efficiency. Existing heuristic methods of ReLU prioritization select ReLU nodes that lead to greater improvement in the final bound (maximum or minimum value) of the neuron network on the input domain [10], i.e. $\min_{x \in \mathcal{C}} \underline{f}(x)$. However, these ReLU prioritization methods are not effective for preimage analysis, because our objective is instead to refine the overall preimage approximation. We thus propose a heuristic method to prioritize unstable ReLU nodes for preimage refinement. Specifically, we compute (an estimate of) the volume difference between the split subregions $|\text{vol}(\mathcal{C}^+_{z_j^{(i)}}) - \text{vol}(\mathcal{C}^-_{z_j^{(i)}})|$, using a single forward pass for a set of sampled datapoints from the input domain; note that this is bounded above by the total subregion volume $\text{vol}(\mathcal{C}_{sub})$. We then propose to select the ReLU node that minimizes this difference. Intuitively, this choice results in balanced subdomains after splitting.

Another advantage of ReLU splitting is that we can replace the unstable neuron bound $\underline{c}h_j^{(i)}(x) + \underline{d} \leq a_j^{(i)}(x) \leq \overline{c}h_j^{(i)}(x) + \overline{d}$ with the exact linear function $a_j^{(i)}(x) = h_j^{(i)}(x)$ and $a_j^{(i)}(x) = 0$, respectively, as shown in Figure 1 (unstable to stable). This can then tighten the linear bounds for the other neurons, thus tightening the under-approximation on each subdomain.

Example 5. We now apply our algorithm with ReLU splitting to the vehicle parking problem in Example 1. Figure 2d shows the refined preimage polytope by adding the splitting plane (black solid line) along the direction of a selected unstable ReLU node. Compared with Figure 2b, we can see that the volume of the approximation is improved.

Remark 1 (Preimage Over-approximation). While Algorithms 1 and 2 focus on preimage under-approximations, they can be easily configured to generate over-approximations with two key modifications. Firstly, we generate polytope over-approximations by using LiRPA to propagate a linear *upper* bound $\overline{g_i}(x) = \overline{a}_i^T x + \overline{b}_i$ for each output constraint, such that $g_i(x) \geq 0 \implies \overline{g_i}(x) \geq 0$ for $x \in \mathcal{C}$. Secondly, the refinement and optimization objective is to *minimize* the volume of the over-approximation instead of maximizing the volume as in the case of under-approximation.

4.4 Overall Algorithm

Our overall preimage approximation method is summarized in Algorithm 1. It takes as input a neural network f, input region $\mathcal{C}$, output region O, target polytope volume threshold v (a proxy for approximation precision), termination iteration number R, and a Boolean indicating whether to use input or ReLU splitting, and returns a disjoint polytope union $\mathcal{T}$ representing an underapproximation to the preimage.

The algorithm initiates and maintains a priority queue of (sub)regions according to Equation 2. The *initialization* step (Lines 1-3) generates an initial polytope approximation on the whole region using Algorithm 2 (Sections 4.1,

```
Algorithm 3: Quantitative Verification
  Input: Neural network $f$, Property $(I, O, p)$, Maximum iterations $R$
  Output: Verification Result $\in$ {True, False, Unknown}
  $\mathrm{vol}(I) \leftarrow \mathrm{ExactVolume}(I)$;
  $\mathcal{C} \leftarrow \mathrm{OuterBox}(I)$ ;                // For general polytopes $I$
  $\mathcal{T} \leftarrow \mathrm{InitialRun}(f, \mathcal{C}, O)$;
  while Iterations $\leq R$ do
      $\mathcal{T} \leftarrow \mathrm{Refine}(f, \mathcal{T}, \mathcal{C}, O)$;
      if $\mathrm{EstimateVolume}(\mathcal{T}) \geq p \times \mathrm{vol}(I)$ then
          if $\mathrm{ExactVolume}(\mathcal{T}) \geq p \times \mathrm{vol}(I)$ then
              return True
      if AllReLUSplit then
          return False
  return Unknown
```

4.2). Then, the *preimage refinement* loop (Lines 4-14) partitions a subregion in each iteration, with the preimage restricted to the child subregions then being re-approximated (Line 10-11). In each iteration, we choose the region to split (Line 5) and the splitting plane to cut on (Line 7 for input split and Line 9 for ReLU split), as detailed in Section 4.3. The preimage under-approximation is then updated by computing the priorities for each subregion (by approximating volumes) (Lines 12-14). The loop terminates and the approximation returned when the target volume threshold v or maximum iteration limit R is reached.

4.5 Quantitative Verification

We now show how to use our efficient preimage under-approximation method (Algorithm 1) to verify a given quantitative property (I, O, p), where O is a polyhedron, I a polytope and p the desired proportion value, summarized in Algorithm 3. To simplify assume that I is a hyperrectangle, so that we can take $\mathcal{C} = I$ (in view of space constraints the case of general polytopes is discussed in Appendix of [45]). We utilize Algorithm 1 by setting the volume threshold to $p \times \mathrm{vol}(I)$, such that we have $\frac{\widehat{\mathrm{vol}}(\mathcal{T})}{\mathrm{vol}(I)} \geq p$ if the algorithm terminates before reaching the maximum number of iterations. However, the Monte Carlo estimates of volume cannot provide a sound guarantee that $\frac{\mathrm{vol}(\mathcal{T})}{\mathrm{vol}(I)} \geq p$. To resolve this problem, we propose to run exact volume computation [5] only when the Monte Carlo estimate reaches the threshold. If the exact volume $\mathrm{vol}(\mathcal{T}) \geq p \times \mathrm{vol}(I)$, then the property is verified. Otherwise, we continue running the preimage refinement.

In Algorithm 3, `InitialRun` generates an initial approximation to the preimage as in Lines 1-3 of Algorithm 1, and `Refine` performs one iteration of approximation refinement (Lines 5-14). Termination occurs when we have verified or falsified the quantitative property, or when the maximum number of iterations has been exceeded.

Proposition 3. *Algorithm 3 is sound for quantitative verification with input splitting.*

Proposition 4. *Algorithm 3 is sound and complete for quantitative verification on piecewise linear neural networks with ReLU splitting.*

5 Experiments

We have implemented our approach as a prototype tool [3] for preimage approximation for polyhedron-type output sets/specifications. In this section, we perform experimental evaluation of the proposed approach on a set of benchmark tasks and demonstrate its effectiveness in approximation generation and its application to quantitative analysis of neural networks.

5.1 Benchmark and Evaluation Metric

We evaluate our preimage analysis approach on a benchmark of reinforcement learning and image classification tasks. Besides the vehicle parking task [3] shown in the running example, we use the following (trained) benchmarks: (1) aircraft collision avoidance system (VCAS) [21] with 9 feed-forward neural networks (FNNs); (2) neural network controllers from VNN-COMP 2022 [1] for three reinforcement learning tasks (Cartpole, Lunarlander, and Dubinsrejoin) [9]; and (3) the neural network from VNN-COMP 2022 for MNIST classification. Details of the models and additional experiments can be found in Appendix of [45].

Evaluation Metric To evaluate the quality of the preimage approximation, we define the *coverage ratio* to be the ratio of volume covered to the volume of the exact preimage, i.e., $\mathrm{cov}(\mathcal{T}, f_{\mathcal{C}}^{-1}(O)) := \frac{\mathrm{vol}(\mathcal{T})}{\mathrm{vol}(f_{\mathcal{C}}^{-1}(O))}$. Note that this is a normalized measure for assessing the quality of the approximation, as shown in Algorithm 3 when comparing with target coverage proportion p for termination of the refinement loop, and not as a measure for formal verification guarantees. In practice, we estimate $\mathrm{vol}(f_{\mathcal{C}}^{-1}(O))$ as $\widehat{\mathrm{vol}}(f_{\mathcal{C}}^{-1}(O)) = \mathrm{vol}(\mathcal{C}) \times \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{f(x_i)\in O}$, where $x_1, ...x_N$ are samples from $\mathcal{C}$. In Algorithm 1, the target volume (stopping criterion) is set as $v = r \times \widehat{\mathrm{vol}}(f_{\mathcal{C}}^{-1}(O)$, where r is the *target coverage ratio.*

5.2 Evaluation

Effectiveness in Preimage Approximation with Input Split We apply Algorithm 1 with input splitting to the input bounding problem for low-dimensional reinforcement learning tasks to evaluate its effectiveness. For comparison, we also run the exact preimage (Exact) [27] and preimage over-approximation (Invprop) [23, 24] methods.

Vehicle Parking & VCAS. Table 1 presents experimental results on the vehicle parking and VCAS tasks. In the table, we show the number of polytopes (#Poly)

[3] The source code is at `https://github.com/Zhang-Xiyue/PreimageApproxForNNs`.

Table 1: Performance comparison in preimage generation.

Models	Exact		Invprop		Our		
	#Poly	**Time**	**Time**	**Cov(%)**	**#Poly**	**Time**	**Cov(%)**
Vehicle (FNN 1 × 20)	10	3110.979	2.642	92.1	4	1.175	95.7
VCAS (FNN 1 × 21)	131	6363.272	-	-	12	11.281	91.0

Table 2: Performance of preimage approximation for reinforcement learning tasks.

Task	Property	Config	#Poly	Cov(%)	Time
Cartpole (FNN 2 × 64)	$\{y \in \mathbb{R}^2 \mid y_1 \geq y_2\}$	$\dot{\theta} \in [-2, -1]$	8	82.0	8.933
		$\dot{\theta} \in [-2, -0.5]$	17	75.5	14.527
		$\dot{\theta} \in [-2, 0]$	32	76.5	27.344
Lunarlander (FNN 2 × 64)	$\{y \in \mathbb{R}^4 \mid \wedge_{i \in \{1,3,4\}}\ y_2 \geq y_i\}$	$\dot{v} \in [-0.5, 0]$	38	75.5	34.311
		$\dot{v} \in [-1, 0]$	71	75.1	63.333
		$\dot{v} \in [-2, 0]$	159	75.0	134.929
Dubinsrejoin (FNN 2 × 256)	$\{y \in \mathbb{R}^8 \mid \wedge_{i \in [2,4]}\ y_1 \geq y_i \bigwedge \wedge_{i \in [6,8]}\ y_5 \geq y_i\}$	$x_v \in [-0.1, 0.1]$	26	75.8	34.558
		$x_v \in [-0.2, 0.2]$	61	75.4	78.437
		$x_v \in [-0.3, 0.3]$	1002	57.6	1267.272

in the preimage, computation time (Time(s)), and the approximate coverage ratio (Cov(%)) when the preimage approximation algorithm terminates with target coverage 90%. Compared with the exact method, our approach yields *orders-of-magnitude* improvement in efficiency. It can also characterize the preimage with much fewer (and also disjoint) polytopes (average reduction of 91.1% for VCAS).

The Invprop method [23] cannot be directly applied as it computes preimage over-approximations. We adapt it to produce an under-approximation by computing over-approximations for the complement of each output constraint; the resulting approximation is then the complement of a union of polytopes, rather than a DUP. On the 2D vehicle parking task, we find that the results (see Table 1) are comparable with ours in time and approximation coverage. Their implementation currently only supports two-dimensional input tasks [24]. While their algorithm, which employs input splitting, can in theory be extended to higher-dimensional tasks, a significant unaddressed technical challenge is in how to choose the input splits effectively in high dimensions. This is confounded by the fact that, to generate an under-approximation, we need separate runs of their algorithm for each output constraint. In contrast, our method naturally incorporates a principled splitting and refinement strategy, and can also effectively employ ReLU splitting for further scalability, as we will show below. Our method can also be configured to generate over-approximations (Section 4.3, Remark 1).

Neural Network Controllers. In this experiment, we consider preimage under-approximation for neural network controllers in reinforcement learning tasks. Note that [27] (Exact) is unable to deal with neural networks of these sizes and

Table 3: Refinement with ReLU split for MNIST (FNN 6 × 100)

L_∞ **attack**	**#Poly**	**Cov(%)**	**Time**	**Patch attack**	**#Poly**	**Cov(%)**	**Time**
0.05	2	100.0	3.107	3 × 3(center)	1	100.0	2.611
0.07	247	75.2	121.661	4 × 4(center)	678	38.2	455.988
0.08	522	75.1	305.867	6 × 6(corner)	2	100.0	9.065
0.09	733	16.5	507.116	7 × 7(corner)	7	84.2	10.128

[23, 24] (Invprop) does not support these higher-dimensional input domains. Table 2 summarizes the experimental results. We evaluate Algorithm 1 with input split on a range of tasks/properties and configurations of the input region (e.g., angular velocity $\dot{\theta}$ for Cartpole). Empirically, for the same coverage ratio, our method requires a number of polytopes and time roughly linear in the input region size, with the exception of Dubinsrejoin, where the larger number of output constraints and larger network size contribute to greater relaxation error.

MNIST Preimage Approximation with ReLU Split Next, we evaluate the scalability of Algorithm 1 with ReLU splitting by applying it to MNIST image classifiers. To our knowledge, this is the first time preimage computation has been attempted for this challenging, high-dimensional task.

Table 3 summarizes the evaluation results for two types of image attacks: l_∞ and patch attack. For L_∞ attacks, bounded perturbation noise is applied to all image pixels. The patch attack applies only to a smaller patch area but allows arbitrary perturbations covering the whole valid range $[0, 1]$. The task is then to produce a DUP under-approximation of the perturbation region that is guaranteed to be classified correctly. For L_∞ attack, our approach generates a preimage approximation that achieves the targeted coverage of 75% for noise up to 0.08. Notice that, from e.g. 0.05 to 0.07, the volume of the input region increases by tens of orders of magnitude due to the high dimensionality. The fact that the number of polytopes and computation time remains manageable is due to the effectiveness of ReLU splitting. Interestingly, for the patch attack, we observe that the number of polytopes required increases sharply when increasing the patch size at the center of the image, while this is not the case for patches in the corners of the image. We hypothesize this is due to the greater influence of central pixels on the neural network output, and correspondingly a greater number of unstable neurons over the input perturbation space.

Comparison with Robustness Verifiers We now illustrate empirically the utility of preimage computation in robustness analysis compared to robustness verifiers. Table 4 shows comparison results with α,β-CROWN, winner of the VNN competition [1]. We set the tasks according to the problem instances from VNN-COMP 2022 for local robustness verification (localized perturbation regions). For Cartpole, α,β-CROWN can provide a verification guarantee (yes/no or safe/unsafe) for both of the problem instances. However, in the case where the robustness property does not hold, our method explicitly generates a preimage approximation in the form of a disjoint polytope union (where correct classi-

Table 4: Comparison with a robustness verifier.

Task	**α, β-CROWN**		**Our**		
	Result	Time	Cov(%)	#Poly	Time
Cartpole ($\dot{\theta} \in [-1.642, -1.546]$)	yes	3.349	100.0	1	1.137
Cartpole ($\dot{\theta} \in [-1.642, 0]$)	no	6.927	94.9	2	3.632
MNIST (L_∞ 0.026)	yes	3.415	100.0	1	2.649
MNIST (L_∞ 0.04)	unknown	267.139	100.0	2	3.019

fication is guaranteed), and covers 94.9% of the exact preimage. For MNIST, while the smaller perturbation region is successfully verified, α, β-CROWN with tightened intermediate bounds by MIP solvers returns unknown with a timeout of 300s for the larger region. In comparison, our algorithm provides a concrete union of polytopes where the input is guaranteed to be correctly classified, which we find covers 100% of the input region (up to sampling error). Note also (Table 3) that our algorithm can produce non-trivial under-approximations for input regions far larger than α, β-CROWN can verify.

Quantitative Verification We now demonstrate the application of our preimage generation framework to quantitative verification of the property (I, O, p); that is, to check whether $f(x) \in O$ for at least proportion p of input values $x \in I$. This leverages the disjointness of our approximation, such that we can exactly compute the volume covered by exactly computing the volume of each polytope.

Vehicle Parking. We consider the quantitative property with input set $I = \{x \in \mathbb{R}^2 \mid x \in [0,1]^2\}$, output set $O = \{y \in \mathbb{R}^4 | \bigwedge_{i=2}^{4} y_1 - \ y_i \geq 0\}$, and quantitative proportion $p = 0.95$. We use Algorithm 3 to verify this property, with iteration limit 1000. The computed under-approximation is a union of two polytopes, which takes 0.942s to reach the target coverage. We then compute the exact volume ratio of the under-approximation against the input region. The final quantitative proportion reached by our under-approximation is 95.2%, verifying the quantitative property.

Aircraft Collision Avoidance. In this example, we consider the VCAS system and a scenario where the two aircraft have negative relative altitude from intruder to ownship ($h \in [-8000, 0]$), the ownship aircraft has a positive climbing rate $\dot{h}_A \in [0, 100]$ and the intruder has a stable negative climbing rate $\dot{h}_B = -30$, and time to the loss of horizontal separation is $t \in [0, 40]$, which defines the input region I. For this scenario, the correct advisory is “Clear Of Conflict” (COC). We apply Algorithm 3 to verify the quantitative property where $O = \{y \in \mathbb{R}^9 | \bigwedge_{i=2}^{9} y_1 - y_i \geq 0\}$ and the proportion $p = 0.9$, with an iteration limit of 1000. The under-approximation computed is a union of 6 polytopes, which takes 5.620s to reach the target coverage. The exact quantitative proportion reached by the generated under-approximation is 90.8%, which verifies the quantitative property.

6 Related Work

Our paper is related to a series of works on robustness verification of neural networks. To address the scalability issues with *complete* verifiers [20, 22, 35] based on constraint solving, convex relaxation [31] has been used for developing highly efficient *incomplete* verification methods [44, 39, 32, 40]. Later works employed the branch-and-bound (BaB) framework [11, 10] to achieve completeness, using incomplete methods for the bounding procedure [41, 36, 17]. In this work, we adapt convex relaxation for efficient preimage approximation. Further, our divide-and-conquer procedure is analogous to BaB, but focuses on maximizing covered volume rather than maximizing a function value. There are also works that have sought to define a weaker notion of local robustness known as *statistical robustness* [37, 26], which requires that a proportion of points under some perturbation distribution around an input point are classified in the same way. Verification of statistical robustness is typically achieved by sampling and statistical guarantees [37, 4, 34, 42]. In this paper, we apply our symbolic approximation approach to quantitative analysis of neural networks, while providing *exact quantitative* rather than *statistical* guarantees [38].

Another line of related works considers deriving exact or approximate abstractions of neural networks, which are applied for explanation [33], verification [16, 29], reachability analysis [28], and preimage approximation [15, 23]. [15] leverages symbolic interpolants [2] for preimage approximations, facing exponential complexity in the number of hidden neurons. Concurrently, [23] employs Lagrangian dual optimization for preimage over-approximations. Our anytime algorithm, which combines convex relaxation with principled splitting strategies for refinement, is applicable for both under- and over-approximations. Their work may benefit from our splitting strategies to scale to higher dimensions.

7 Conclusion

We present an efficient and flexible algorithm for preimage under-approximation of neural networks. Our *anytime* method derives from the observation that linear relaxation can be used to efficiently produce under-approximations, in conjunction with custom-designed strategies for iteratively decomposing the problem to rapidly improve the approximation quality. Unlike previous approaches, it is designed for, and scales to, both low and high-dimensional problems. Experimental evaluation on a range of benchmark tasks shows significant advantage in runtime efficiency and scalability, and the utility of our method for important applications in quantitative verification and robustness analysis.

Acknowledgments This project received funding from the ERC under the European Union's Horizon 2020 research and innovation programme (FUN2MODEL, grant agreement No. 834115) and ELSA: European Lighthouse on Secure and Safe AI project (grant agreement No. 101070617 under UK guarantee). This work was done in part while Benjie Wang was visiting the Simons Institute for the Theory of Computing.

References

1. VnnComp 2022. https://github.com/ChristopherBrix/vnncomp2022_benchmarks, accessed: 2022-09-30
2. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: Computer Aided Verification - 25th International Conference, CAV 2013, Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 313–329. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_22
3. Ayala, D., Wolfson, O., Xu, B., DasGupta, B., Lin, J.: Parking slot assignment games. In: 19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS, Proceedings. pp. 299–308. ACM (2011)
4. Baluta, T., Chua, Z.L., Meel, K.S., Saxena, P.: Scalable quantitative verification for deep neural networks. In: Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings. p. 248–249. ICSE '21, IEEE Press (2021)
5. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The quickhull algorithm for convex hulls. ACM Trans. Math. Softw. pp. 469–483 (1996). https://doi.org/10.1145/235815.235821
6. Benoy, P.M.: Polyhedral domains for abstract interpretation in logic programming. Ph.D. thesis, University of Kent, UK (2002)
7. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., et al.: End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316 (2016)
8. Boutonnet, R., Halbwachs, N.: Disjunctive relational abstract interpretation for interprocedural program analysis. In: Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Proceedings. Lecture Notes in Computer Science, vol. 11388, pp. 136–159. Springer (2019). https://doi.org/10.1007/978-3-030-11245-5_7
9. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. CoRR (2016), http://arxiv.org/abs/1606.01540
10. Bunel, R., Lu, J., Turkaslan, I., Torr, P.H., Kohli, P., Kumar, M.P.: Branch and bound for piecewise linear neural network verification. Journal of Machine Learning Research pp. 1–39 (2020)
11. Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS. pp. 4795–4804 (2018)
12. Chevallier, A., Cazals, F., Fearnhead, P.: Efficient computation of the the volume of a polytope in high-dimensions using piecewise deterministic markov processes. In: International Conference on Artificial Intelligence and Statistics, AISTATS 2022, 28-30 March 2022, Virtual Event. Proceedings of Machine Learning Research, vol. 151, pp. 10146–10160. PMLR (2022)
13. Codevilla, F., Müller, M., López, A.M., Koltun, V., Dosovitskiy, A.: End-to-end driving via conditional imitation learning. In: Proceedings of the 2018 IEEE International Conference on Robotics and Automation. pp. 1–9. IEEE, Brisbane, Australia (2018). https://doi.org/10.1109/ICRA.2018.8460487
14. Craig, W.: Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. The Journal of Symbolic Logic pp. 269–285 (1957)
15. Dathathri, S., Gao, S., Murray, R.M.: Inverse abstraction of neural networks using symbolic interpolation. In: The Thirty-Third AAAI Conference

on Artificial Intelligence, AAAI 2019. pp. 3437–3444. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33013437

16. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Proceedings, Part I 32. pp. 43–65. Springer (2020)
17. Ferrari, C., Müller, M.N., Jovanovic, N., Vechev, M.T.: Complete verification via multi-neuron relaxation guided branch-and-bound. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net (2022)
18. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE symposium on security and privacy (SP). pp. 3–18. IEEE (2018)
19. Gopinath, D., Converse, H., Păsăreanu, C.S., Taly, A.: Property inference for deep neural networks. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. p. 797–809. ASE '19, IEEE Press (2020). https://doi.org/10.1109/ASE.2019.00079
20. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Computer Aided Verification - 29th International Conference, CAV 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10426, pp. 3–29. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_1
21. Julian, K.D., Kochenderfer, M.J.: A reachability method for verifying dynamical systems with deep neural network controllers. CoRR (2019), http://arxiv.org/abs/1903.00520
22. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: Computer Aided Verification: 29th International Conference, CAV 2017, Proceedings, Part I 30. pp. 97–117. Springer (2017)
23. Kotha, S., Brix, C., Kolter, Z., Dvijotham, K., Zhang, H.: Provably bounding neural network preimages. Accepted to NeurIPS 2023, CoRR (2023). https://doi.org/10.48550/arXiv.2302.01404
24. Kotha, S., Brix, C., Kolter, Z., Dvijotham, K., Zhang, H.: INVPROP for provably bounding neural network preimages. https://github.com/kothasuhas/verify-input (accessed October, 2023)
25. Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M.J., et al.: Algorithms for verifying deep neural networks. Foundations and Trends in Optimization pp. 244–404 (2021)
26. Mangal, R., Nori, A.V., Orso, A.: Robustness of neural networks: a probabilistic and practical approach. In: Sarma, A., Murta, L. (eds.) Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019. pp. 93–96. IEEE / ACM (2019)
27. Matoba, K., Fleuret, F.: Exact preimages of neural network aircraft collision avoidance systems. In: Proceedings of the Machine Learning for Engineering Modeling, Simulation, and Design Workshop at Neural Information Processing Systems 2020 (2020)
28. Prabhakar, P., Afzal, Z.R.: Abstraction based output range analysis for neural networks. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019. pp. 15762–15772 (2019)

29. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Computer Aided Verification: 22nd International Conference, CAV 2010, Proceedings 22. pp. 243–257. Springer (2010)
30. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018. pp. 2651–2659. ijcai.org (2018)
31. Salman, H., Yang, G., Zhang, H., Hsieh, C., Zhang, P.: A convex relaxation barrier to tight robustness verification of neural networks. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019. pp. 9832–9842 (2019)
32. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages pp. 1–30 (2019)
33. Sotoudeh, M., Thakur, A.V.: Syrenn: A tool for analyzing deep neural networks. In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Proceedings, Part II 27. pp. 281–302. Springer (2021)
34. Tit, K., Furon, T., Rousset, M.: Efficient statistical assessment of neural network corruption robustness. In: Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual. pp. 9253–9263 (2021)
35. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: 7th International Conference on Learning Representations, ICLR 2019. OpenReview.net (2019)
36. Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C., Kolter, J.Z.: Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In: Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual. pp. 29909–29921 (2021)
37. Webb, S., Rainforth, T., Teh, Y.W., Kumar, M.P.: A statistical approach to assessing neural network robustness. In: 7th International Conference on Learning Representations, ICLR 2019. OpenReview.net (2019)
38. Wicker, M., Laurenti, L., Patane, A., Kwiatkowska, M.: Probabilistic safety for bayesian neural networks. In: In Proc. 36th Conference on Uncertainty in Artificial Intelligence (UAI-2020). PMLR (2020)
39. Wong, E., Kolter, Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: International conference on machine learning. pp. 5286–5295. PMLR (2018)
40. Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.: Automatic perturbation analysis for scalable certified robustness and beyond. In: Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020)
41. Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S., Lin, X., Hsieh, C.: Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event. OpenReview.net (2021)

42. Yang, P., Li, R., Li, J., Huang, C., Wang, J., Sun, J., Xue, B., Zhang, L.: Improving neural network verification through spurious region guided refinement. In: Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12651, pp. 389–408. Springer (2021)
43. Yun, S., Choi, J., Yoo, Y., Yun, K., Choi, J.Y.: Action-decision networks for visual tracking with deep reinforcement learning. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017. pp. 1349–1358. IEEE Computer Society (2017)
44. Zhang, H., Weng, T., Chen, P., Hsieh, C., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018. pp. 4944–4953 (2018)
45. Zhang, X., Wang, B., Kwiatkowska, M.: Provable preimage under-approximation for neural networks. arXiv preprint arXiv:2305.03686 (2023)

Training for Verification: Increasing Neuron Stability to Scale DNN Verification

Dong Xu[1](✉), Nusrat Jahan Mozumder[1], Hai Duong[2], and Matthew B. Dwyer[1]

[1] University of Virginia, Charlottesville, VA 22904, USA
{dx3yy,nm8tm,matthewbdwyer}@virginia.edu
[2] George Mason University, Fairfax, VA 22030, USA
hduong22@gmu.edu

Abstract. With the growing use of deep neural networks(DNN) in mission and safety-critical applications, there is an increasing interest in DNN verification. Unfortunately, increasingly complex network structures, non-linear behavior, and high-dimensional input spaces combine to make DNN verification computationally challenging. Despite tremendous advances, DNN verifiers are still challenged to scale to large verification problems. In this work, we explore how the number of stable neurons under the precondition of a specification gives rise to verification complexity. We examine prior work on the problem, adapt it, and develop several novel approaches to increase stability. We demonstrate that neuron stability can be increased substantially without compromising model accuracy and this yields a multi-fold improvement in DNN verifier performance.

Keywords: neural network verification · neuron stability · pruning

1 Introduction

In recent years, there has been significant research on adapting formal verification to target deep neural network(DNN) model behavior. Approaches have been developed that incorporate a diverse range of algorithmic approaches including reachability [19,27,39–42,45,51,52], optimization [5,12,15,30,31,34,44,50], and search [1,7,9,21,26,46,47,49,58]. These techniques aim to verify the validity of a network's behavior for a wide range of inputs, e.g., perturbations of test samples that capture models of noise or malicious manipulation.

DNN verification is challenging due to the high input dimension of models, the ever-growing complexity of network layers, the inherent non-linearity of learned function approximations, and the algorithmically complex methods required to formulate the verification problem [25]. Several approaches [4,14,16,38] have been proposed to address the scalability issue, but as the results of recent DNN verifier competitions show scalability remains a challenge [2,22,32].

Stable neurons exhibit linear behavior and thereby have the potential to reduce DNN verification costs. Several researchers have explored how DNNs

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 24–44, 2024.
https://doi.org/10.1007/978-3-031-57256-2_2

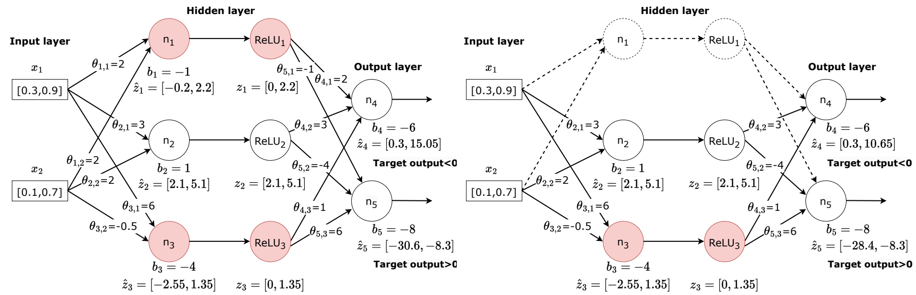
can be defined to increase the number of stable neurons and thereby facilitate verification. For example, one can incorporate a loss term that uses an estimate of neuron stability to train a network that can be verified more efficiently [53]. Another training time approach identifies neurons that are likely to be stable and active and replaces them with linear functions [10], while this approach requires customization of the verifier to show performance improvement.

Whereas prior work studied individual methods for increasing neuron stability in combination with individual verifiers, in this paper we conduct a broad exploratory study considering 18 different stabilizers paired with 3 state-of-the-art verifiers across DNNs for different datasets and comprising different architectures. We use three algorithmic approaches to increase stability: RS Loss [53] incorporates a stability-oriented loss term, Bias Shaping is a novel training time method that only modifies bias parameters to increase stability, and Stable Pruning is a novel approach that adapts structural DNN pruning [43] to increase stability. These are paired with *stability estimation* algorithms that operate at training time to guide them towards increasing stability. We develop 4 estimators based on prior work: **NIP** [53], **SIP** [46,47], **ALR** [56], and **ALRo** [57], and 2 novel estimators **SDD** and **SAD**.

Neuron instability can be a source of verification complexity for the two primary algorithmic approaches to DNN verification: abstraction-based methods and constraint-based methods. Abstraction-based verifiers [3, 17, 40, 42, 48] overapproximate neuron behavior, but when the approximation is too coarse – due to unstable neurons – the approximations must be refined which can slow down verification. Constraint-based verifiers [13, 23, 24, 44] are challenged by the disjunctive nature of constraints that encode unstable neurons. Orthogonal to these approaches, branch and bound techniques [9, 17, 48] are also sensitive to neuron stability since they need to generate sub-problems for each of the active phases of unstable neurons. In our exploratory study, we evaluate the performance of verifiers that span several of these algorithmic approaches and that also constitute the state-of-the-art based on their performance in the most recent VNN-COMP [32]. This allows us to assess the extent to which increasing neuron stability can improve the state-of-the-art.

In § 5 we report the findings of a study spanning 18 stable training algorithms, 3 state-of-the-art verifiers, 3 network architectures, and a large number of challenging property specifications. Our primary finding is that stable training can significantly increase the number of verifications problem solved – by as much as 5-fold – and significantly speed up verification – by as much as a factor of 14 – without compromising test accuracy or training time. Moreover, we find that if one is willing to tolerate a modest loss in test accuracy, then even greater improvement in verifier performance can be achieved.

The contributions of the work lie in a comprehensive evaluation of the potential for optimizing DNN verifier performance by increasing the number of stable neurons. More specifically, (1) we adapt RS Loss with different stability estimators and evaluate its performance across multiple verifiers and benchmarks; (2) we propose two novel approaches (Bias Shaping and Stable Pruning) to



(a) Original network (b) Stable Pruning

Fig. 1. Illustration of the applying Stable Pruning to verifying that a small original network outputs a pair of values where the first is negative and the second positive for inputs $\boldsymbol{x} \in [0.3, 0.9] \times [0.1, 0.7]$. Unstable neurons are shown in red and pruned neurons and their edges are dashed.

increase neuron stability and evaluate their performance across multiple verifiers and benchmarks; (3) we integrate these state-of-the-art neuron stabilizers into an open-source framework that supports experimentation with stability optimization by the DNN verification research community; and (4) show empirically that the performance of state-of-the-art verifiers can be significantly enhanced using stable training methods. These contributions set the stage for further work on *training for verification* that aim to further characterize the best stable training strategy for a given verifier and verification problem.

2 Overview

The popularity of the rectified linear unit (ReLU) activation function, $z = \max(\hat{z}, 0)$, which allows for more efficient training and inference [20, 29], has led verification researchers to target networks using them. In this section, we illustrate how ReLU leads to exponential verification costs and how training can mitigate that cost.

For a DNN with ReLU activation functions, $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$, comprised of k neurons, an inference, $\mathcal{N}(\boldsymbol{x})$, results in each neuron being either *active*, when $z = \max(\hat{z}, 0) = \hat{z}$, or *inactive*, when $z = \max(\hat{z}, 0) = 0$. The status of each neuron in a network during inference defines an activation pattern, $ap(\boldsymbol{x})$ – a Boolean vector of length k. Verifying a set of inputs, $\phi_{\boldsymbol{x}} \subseteq \mathbb{R}^n$, involves symbolically reasoning about the set of activation patterns, and the associated neuron outputs, for each $\boldsymbol{x} \in \phi_{\boldsymbol{x}}$. In the worst case, there are 2^k possible activation patterns which lead to the exponential complexity of ReLU verification [23].

For a given set of inputs, $\phi_{\boldsymbol{x}}$, a neuron, n_i, is *stable* and active if $\forall \boldsymbol{x} \in \phi_{\boldsymbol{x}} : ap(\boldsymbol{x})[i]$, and stable and inactive if $\forall \boldsymbol{x} \in \phi_{\boldsymbol{x}} : \neg ap(\boldsymbol{x})[i]$. A neuron's stability is dependent on the computation performed by its cone of influence [6] taking into account both $\phi_{\boldsymbol{x}}$ and the behavior of neurons on which n_i depends. In Fig. 1a, consider verification of a local robustness property centered at $\boldsymbol{x} = (0.6, 0.4)$ with a radius of $\epsilon = 0.3$ – so $\phi_{\boldsymbol{x}} = [0.3, 0.9] \times [0.1, 0.7]$. For such inputs, a single neuron, n_2, is stable – its pre-activation values are all positive, $\hat{z}_2 = [2.1, 5.1]$.

In §4, we define a set of techniques that aim to estimate which neurons are unstable during training and then bias the training process to stabilize them. Fig. 1b shows the application of one pair of those techniques to the original network and property. More specifically, the **NIP** estimator propagates interval approximations of neuron pre-activation values to estimate whether they are stable and then the Stable Pruning technique removes neurons that are stable and inactive. During training this method estimates the pre-activation value for n_1 to be $\hat{z}_1 = [-0.2, 2.2]$ which is nearly stable. Stable Pruning ranks neurons based on the distance they need to be shifted to be stable; for $\hat{z}_1$ that distance is 0.2. We adapt the iterative pruning approach of DropNet [43] to use this ranking. The intuition is that when a neuron is nearly stable it can be removed and in subsequent training, the parameters of the remaining neurons will adapt to compensate and preserve accuracy [18]. As illustrated in Fig. 1b, the number of unstable neurons is halved which can reduce verification costs.

3 Background & Related Work

Deep Neural Networks (DNN) are trained to accurately approximate a target function, $f : \mathbb{R}^n \to \mathbb{R}^m$. A network, $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$, is comprised of a sequence of L hidden layers, $l_1, \ldots, l_L$, along with an input layer, $l_{in} = l_0$, and output layer, $l_{out} = l_{L+1}$(e.g. (a) in Fig. 1) Hidden layers are comprised of a set of *neurons* that accumulate a weighted sum of their inputs from the prior layer and then apply an *activation function* to determine how to non-linearly scale that sum to compute the output from the layer. Different activation functions have been explored in the literature, including: Rectified Linear Units (ReLU), Sigmoid, and Tanh.

Given a neural network architecture, $\mathcal{N}(\cdot)$, the network is *trained* to define *weight* values, denoted θ, and *bias* values, denoted b, that are associated with each neuron's input. A trained network defines for input $\boldsymbol{x}$, the output $\mathcal{N}(\boldsymbol{x}; \theta, b)$; when it is clear from the context we drop θ, b and write $\mathcal{N}(\boldsymbol{x})$.

Specifying DNN Properties Given a network $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$, a property, ϕ, defines a set of constraints over the inputs, $\phi_{\boldsymbol{x}}$, and an associated set of constraints over the outputs, ϕ_y. Verification of $\mathcal{N} \models \phi$ seeks to prove: $\forall \boldsymbol{x} \in \mathbb{R}^n : \phi_{\boldsymbol{x}}(\boldsymbol{x}) \Rightarrow \phi_y(\mathcal{N}(\boldsymbol{x}))$.

Recent work has demonstrated that a general class of specifications, where $\phi_{\boldsymbol{x}}$ and $\phi_{\boldsymbol{y}}$ are defined as half-space polytopes, can be reduced to local robustness specifications [35, 36]. This means that the essential complexity of DNN verification is present when verifying simpler local robustness specifications, which state that $\forall \boldsymbol{x} \in c \pm \epsilon : \phi_y(\mathcal{N}(\boldsymbol{x}))$, for some constant input(centerpoint), c, and radius, ϵ, around it. Consequently, in §5, we explore the performance of verifiers on local robustness specifications.

Verifying DNN Properties The inherent complexity of the DNN verification problem arises from the non-linear expressive power of DNNs – so it is generally unavoidable. We explain the source of this complexity below for a network with L fully-connected layers, each with M neurons.

Let $\hat{z}_{i,j}$ denote the value computed for the input of neuron j in hidden layer i prior to the application of the activation function – the pre-activation value – and $z_{i,j}$ the post-activation value. For a ReLU activation function, $z_{i,j} = \max(\hat{z}_{i,j}, 0)$. The input to layer i is computed as the weighted sum of the output of the prior layer, using the learned weights θ, and bias b. The semantics of $\mathcal{N}(\boldsymbol{x}; \theta, b)$ is given by the constraints as shown in Eq. (1)

$$\bigwedge_{i\in[1,L],j\in[1,M]} \left(\hat{z}_{i,j} = \sum_{k\in[1,M]} (\theta_{i,j,k} \cdot z_{i-1,j}) + b_{i,j} \wedge z_{i,j} = \max(\hat{z}_{i,j}, 0) \right) \tag{1}$$

with additional constraints relating the $z_{L,j}$ to the output layer, l_{out}, and $\boldsymbol{x} = z_0$.

Computing $\mathcal{N}(\boldsymbol{x})$ for a single input value, $\boldsymbol{x}$, results in a pattern of ReLU activations in which each neuron is either *active*, $\max(\hat{z}_{i,j}, 0) = \hat{z}_{i,j}$, or *inactive*, $\max(\hat{z}_{i,j}, 0) = 0$. However, a property specification, ϕ, constrains l_{in} to define a set of input values, e.g., as in the case of local robustness $\boldsymbol{x} \in c \pm \epsilon$. Through Eq. (1), this may give rise to constraints on $\hat{z}_{i,j}$ that define values for which the neuron is both active, $\hat{z}_{i,j} \geq 0$, and inactive, $\hat{z}_{i,j} < 0$. When the set of pre-activation values spans 0 in this way, we say that neuron $n_{i,j}$ is *unstable.*

Unstable neurons require that verification approaches reason about the disjunctions present in Eq. (1). In the worst case, if all neurons are unstable, then there are 2^{L*M} different ways of resolving the disjunctions. More generally, for a property, ϕ, only a subset of neurons will be unstable, $U_\phi \subseteq L \times M$, and, as we discuss in §4, controlling the size of this subset is a means of reducing the cost of DNN verification.

Several approaches have been introduced to verify a DNN behavior in recent years [28]. One class of verifiers, including α,β-CROWN [48], NNENUM [3], ERAN [40], and MN-BAB [17] overapproximate ReLU behavior which allows them to efficiently calculate an overapproximation of Eq. (1), which we denote $\overline{\mathcal{N}}$. When $\overline{\mathcal{N}} \not\models \phi$ some techniques, like ERAN, simply return *unknown*, but others, like NNENUM, α,β-CROWN or MN-BAB, perform a case split on unstable neurons to refine the over-approximation. Another class of verifiers, including MARABOU [24] and PLANET [13], explore the space of case-splits to formulate separate constraint queries that constitute verification conditions. Here again, the number of possible case-splits leads to exponential complexity.

RS Loss [53] is a regularization technique that induces neuron stability in the training process. The RS LOSS, L_R is blended with the regular training loss L_T to yield a weighted sum as the optimization target, $L = L_T + w_R \times L_R$, where w_R is the hyperparameter to control the degree of stabilization. The RS LOSS term L_R is formulated as $L_R = \sum_{i=1}^{n} -\textsc{Tanh}(1 + \underline{\hat{z}_i} \times \overline{\hat{z}_i})$ where $\underline{\hat{z}}$ and $\overline{\hat{z}}$ are the lower and upper bounds of the pre-activation values. NRS Loss [59] is a variant of RS LOSS that regularizes the pre-batch normalization (BN) bounds instead of pre-activation bounds. Whereas RS Loss indirectly biases the network toward neuron stability, in §4 we introduce Bias Shaping which directly manipulates neuron bias towards the same goal.

DropNet [43] is a structured model compression method to generate sparse and reduced neural networks based on the *lottery ticket hypothesis* [18]. According to the hypothesis, a dense network contains a sub-network that can match the test accuracy of the base network if trained in isolation. DropNet iteratively prunes a predefined percentage of less important neurons by setting their weights to zero. Although the iteration process is resource expensive, the flatness of the error landscape at the end of training limits the fraction of weights that can be pruned, hence sharp pruning at once reduces the network accuracy [33].

While the initial purpose of pruning was preserving network accuracy only, recent studies have revealed that pruning can significantly increase a network's robustness and scale robustness verification [59]. The removal of non-linearity from the insignificant neurons by converting them to linear functions has been proposed in literature [10]. However, the existence of linear activation functions in a network can sometimes result in unnecessary computational costs, as the networks are supposed to work on complex data and linear functions are incapable of handling the complexity. Also, special treatments are required to handle these non-standard architectures in network inference and verification. Thus, we propose to use iterative pruning to remove the redundant non-linearity from the network using the pre-activation values of the ReLU function during mini-batch training. In §4, we present a variant of DropNet named Stable Pruning that uses stability measures to determine how neurons should be pruned.

4 Approach

This section presents the two novel neuron stabilization methods: Bias Shaping and Stable Pruning, as well as six different stability estimators. Alg. 1 shows the general training iterations for a neural network with stabilizers(pairs of stabilization method, $\mathcal{A}$, and stability estimator, $\mathcal{B}$). The conventional neural network training process of a mini-batch is shown in Line 2. Stabilizers are applied at every sth mini-batch (line 3). Line 4 determines each neuron's stability estimation by calculating their boundaries, $\hat{Z}$, using different estimators described in §4.1. Lastly, Line 5 applies the main stabilization algorithms, e.g. Bias Shaping (Alg. 2) and Stable Pruning (Alg. 3).

Alg. 1: Training with Stabilizers

```
input  : neural network $\mathcal{N}$, data loader $D$,
         stabilization method $\mathcal{A}$, stability
         estimator $\mathcal{B}$, ratio $i$, and step $s$
output : stabilized network $\mathcal{N}'$
1 for $j$, $(X, Y)$ in $D$ do
2     Train_Mini-Batch($\mathcal{N}, X, Y$)
3     if $j \equiv 0 \pmod{s}$ then
4         $\hat{Z} \leftarrow$ Estimate_Stability($\mathcal{B}$, $\mathcal{N}$)
5         $\mathcal{N}' \leftarrow$ Stablize($\mathcal{A}$, $\mathcal{N}$, $\hat{Z}$, $i$)
6 return $\mathcal{N}'$
```

4.1 Neuron Stability Estimation

The neural network training process is performed on the data samples, while the verification process seeks to prove certain properties on an effectively unbounded set of inputs. Hence, there exists a gap between the two stages since a neuron that is stable on the training dataset is not guaranteed to also be stable based on the set of values described by the precondition of the verification problem. Guiding the training process to produce neural networks with more stable neurons in the verification stage requires reducing this gap. This is achieved by estimating neuron stability over a broader set of values representative of those encountered during the verification process and then stabilizing the unstable neurons.

We identify two general categories of neuron stability estimators that can be calculated during the training phase: **Sampled**[**S**] and **Reachability**[**R**] estimators. The sampled estimators consider a finite set of sampled data gathered directly or inferred from the training dataset. The reachability estimators operate on set propagations that generalize the training dataset. The six neuron stability estimators are defined as follows:

$$\mathcal{B}(D) = \{x | x = \beta(x') \land x' \sim D\}$$

where $\beta \in \{\mathbf{SDD}, \mathbf{SAD}, \mathbf{NIP}, \mathbf{SIP}, \mathbf{ALR}, \mathbf{ALRo}\}$ and D is the network training dataset distribution. The **SDD** (Sampled Dataset Distribution[**S**]) estimator uses the training mini-batch samples directly and takes advantage of the training process's forward propagations to determine whether neurons are stable. The **SAD** (Sampled Adjacent Distribution[**S**]) estimator samples from the robustness radii of the training mini-batch and runs extra forward propagations on the adjacent examples to determine the stability of neurons. The **NIP** (Naive Interval Propagation[**R**]) [53] estimator generates a set of intervals based on the mini-batch samples and the given robustness radii. However, instead of propagating exact samples, it propagates the intervals through the network. The **SIP** (Symbolic Interval Propagation[**R**]) [46, 47] extends **NIP** by using symbolic intervals instead of concrete intervals when propagating through the network. The symbolic intervals are concretized whenever neuron stability needs to be evaluated. The **ALR** and **ALRo** (Auto_LiRPA[**R**]) [56, 57] estimators further improve **SIP** by applying more precise but computationally expensive over-approximation constraints and parameterizing upper and lower bounds of hidden neurons to optimize objectives with respect to the property of interest. **ALRo** applies the α optimization [57] when compared to the base approach. Note that although many of these approaches were developed for other uses, the integration of them to induce stable neurons during training is novel.

4.2 Bias Shaping

To increase the number of stable neurons in the neural network, we adapt training to ensure the same polarity of lower and upper bounds of neuron pre-activation values. In Eq. (1), the pre-activations of the current ReLU function are controlled by the parameters of the neural network and the post-activations of the previous layer. The weighted-sum term depends on the weights, bias, and the post-activations of the previous layer. The pre-activation values can be easily manipulated by changing the bias term. We refer to this as Bias Shaping, as described in Alg. 2.

```
Alg. 2: Bias Shaping
input  : neural network $\mathcal{N}$, stability
         estimation boundaries $\hat{Z}$,
         ratio $i$
output : stabilized network $\mathcal{N}'$
1  $\underline{\hat{Z}}, \overline{\hat{Z}} \leftarrow$ Get_Bounds($\hat{Z}$)
2  $N_u \leftarrow \{n_i$ in $\mathcal{N}$ where $\underline{\hat{z}}_i < 0 \wedge \overline{\hat{z}}_i > 0\}$
3  $Z_u \leftarrow \{$Min$(-\underline{\hat{z}_{n_i}}, \overline{\hat{z}_{n_i}})$ where $n_i \in N_u\}$
4  $\gamma \leftarrow$ Sort$(Z_u)[|\hat{Z}| \times i]$
5  for $n_i$ in $N_u$ do
6      if $(\overline{\hat{z}_i} < \gamma) \wedge (\overline{\hat{z}_i} < -\underline{\hat{z}_i})$ then
7          $n_i.b \leftarrow n_i.b - \overline{\hat{z}_i}$
8      else if $|\underline{\hat{z}_i}| < \gamma$ then
9          $n_i.b \leftarrow n_i.b - \underline{\hat{z}_i}$
10 $\mathcal{N}' \leftarrow$ Load_Parameters($N_u$)
11 return $\mathcal{N}'$
```

Instead of using just the native pre-activation of the mini-batch samples, the stability estimators are applied to further close the gap between neuron stability during training and verification. Alg. 2 takes the set of stability estimations for all neurons, $\hat{Z} = [\hat{z}_1, \hat{z}_2, ..., \hat{z}_m]$, the neural network $\mathcal{N}$ with m neurons $(n_1, n_2, ..., n_m)$, and the ratio i as inputs. Line 1 calculated the lower and upper bounds of the estimation $\hat{Z}$. Using those bounds, the algorithm first finds the unstable neurons of the input network (line 2). Next, those neurons are ranked based on their distance to zero(lines 5 - 9), and the smallest subset of neurons will be selected for shaping if their distances are less than an adaptive threshold γ (lines 3, 4). Note that the number of selections is controlled by a parameter i – a percentage of neurons would be shaped at a time. Each neuron's bias term of the subset is modified by (a) shifting left by the value of the upper bound if the upper bound is closer to zero (line 7); or (b) shifting right by the absolute value of lower bound if the lower bound is closer to zero (line 9). As a result, the stabilized network is created by loading the new parameters at line 10.

4.3 Stable Pruning

Inspired by the DropNet [43] approach, we developed a new pruning method to reduce unstable neurons, named Stable Pruning as shown in Alg. 3. It uses iterative structured pruning to modify the global weight matrix by selectively masking neurons. Its novel criteria target specifically unstable neurons for masking. Stable Pruning sets weight and bias to zero to softly "remove" the neuron from the network, allowing back-propagation to recover accuracy loss by the harsh parameter modifications.

Given the stability estimation $\hat{z}$ for a neuron, $\underline{\hat{z}}$ and $\overline{\hat{z}}$ denote the lower and upper bounds respectively. When lower bound $\underline{\hat{z}}$ is greater than 0, although the neuron is stable-active, it cannot be pruned without changing the network's behavior, as the ReLU function is treated as an identity function. When $\overline{\hat{z}}$ is less than 0, the ReLU function is treated as a zero-function, and this neuron can be removed safely (line 3). In order to prune unstable neurons with minimal effects on network behavior, STABLE PRUNING ranks the unstable neurons by the distance between $\bar{\hat{z}}$ and 0, from smallest to largest (line 4), and a subset of neurons (also controlled by the ratio parameter, i) will be selected for pruning if their distances are less than an adaptive threshold γ (line 5). Initially, all neurons are enabled in the mask, m, (line 1) and those that fall below the threshold are updated to be removed from the network (line 6). Finally, the stabilized network is generated by applying the pruning mask on the network (line 7).

4.4 Implementation

Alg. 3: STABLE PRUNING

```
input  : neural network $\mathcal{N}$, stability
         estimation boundaries $\hat{Z}$,
         ratio $i$
output : stabilized network $\mathcal{N}'$
1 $m = \{1\}^{|n|}$
2 $\bar{\hat{Z}} \leftarrow$ GET_UPPER_BOUND($\hat{Z}$)
3 $m[\bar{\hat{Z}} \leq 0] \leftarrow 0$
4 $Z'_u = \text{sort}(\bar{\hat{Z}} > 0)$
5 $\gamma = Z_u[|Z'_u| \times i]$
6 $m[\hat{Z} < \gamma] \leftarrow 0$
7 $\mathcal{N}' \leftarrow \mathcal{N} \bigodot m$
8 return $\mathcal{N}'$
```

We implemented all of the above techniques, including: **SDD**, **SAD**, **NIP**, **SIP**, **ALR**, **ALRo**, RS LOSS (§3), BIAS SHAPING (§4.2), and STABLE PRUNING (§4.3), into the OCTOPUS framework. OCTOPUS allows training neural networks with stabilizer methods and stability estimators, including their free combinations. It can be easily applied to different datasets and network architectures and presents a rich hyper-parameter space that can be tuned by hand or algorithmically, e.g., by search methods. RS LOSS [53] is reimplemented to support all the additional neuron stability estimators. The **SIP** estimator uses the Symbolic Interval Analysis Library developed in [46], and the **ALR** and **ALRo** estimators integrate the Auto_LiRPA Library [57]. OCTOPUS also allows combinations of various neuron stabilizers and estimators, i.e., training with multiple stabilizers sequentially or simultaneously. The framework is built for ease of extension to adopt new techniques and is available at both FigShare [54] and GitHub.[3]

5 Evaluation

We explore two research questions to understand how stabilizers can be beneficial for DNN verification:

RQ1. How effective are the stabilizers in increasing the proportion of stable neurons?

RQ2. How effective are stabilizers in enhancing DNN verification performance?

[3] OCTOPUS GitHub link: `https://github.com/edwardxu0/octopus`

Tab. 1. Experimental parameter space

Parameters	Choices
Architectures	**M2**: MNIST_FC2(FC(256)×2), **M6**: MNIST_FC6(FC(256)×6)) **C3**: CIFAR2020(Conv(32,5,2), Conv(128,4,2), FC(250))
Verifiers	α,β-CROWN, MN-BAB, NNEnum
Properties	[0,1,. . . ,9]
Epsilon Radii	**M2**, **M6**:[12e-3, 14e-3, 16e-3, 18e-3, 20e-3] **C3**:[18e-4, 20e-4,22e-4, 24e-4, 26e-4]
Stabilization Methods	Baseline, **B**ias Shaping, **R**S Loss, Stable **P**runing
Stability Estimators	**SDD**, **SAD**, **NIP**, **SIP**, **ALR**, **ALRo**
Seeds	[0,1,2,3,4]

5.1 Study Design

To answer these questions, we design a broad study considering different neural network architectures, specifications, and verifiers. Tab. 1 shows the full experimental parameter space we consider across the research questions.

The annual VNN-COMP DNN verification competition [2, 22, 32] provides a range of benchmarks with standard network and property formats to evaluate state-of-the-art verifiers. These benchmarks cover a variety of network architectures and activation functions. This *architectural variety* evaluates verifiers' applicability across a range of network graph operations, e.g. ResNets with skip connections, max-pooling layers, non-linear activation, and domain-specific networks. Benchmarks also vary in *scale* with some having large numbers of layers, neurons, and parameters under the assumption that this will yield challenging benchmarks.

We conducted an exploratory study of the VNN-COMP 2022 benchmarks and found that 1156 of 1288 (89%) could be solved within 30 seconds. Nearly all of the solved problems were proven (UNSAT) with coarse over-approximation or falsified (SAT) with adversarial attacks. Such benchmarks do not exhibit the exponential complexity that is inherent in DNN verification [23]. To address this limitation, we designed a set of benchmarks that are better suited to assessing DNN verification algorithm performance.

Selecting Networks A retrospective analysis of VNN-COMP benchmarks determined that small weakly-regularized networks exhibit exponential complexity and medium-sized with large numbers of neurons are hard to scale for precise methods, such as branch and bound [8]. Of course, large weekly-regularized networks with large numbers of neurons are even harder, but it was found that these incur significant memory requirements which makes experimentation challenging, e.g., due to hardware limitations. Based on this analysis, we focus on three small and medium-sized networks with traditional network architectures selected from the VNN-COMP 2022 benchmarks, since these proved capable of forcing verifier algorithms to cope with exponential complexity.

Selecting Properties Rather than focusing on a variety of structurally distinct property specifications, we exploit the fact that general reachability proper-

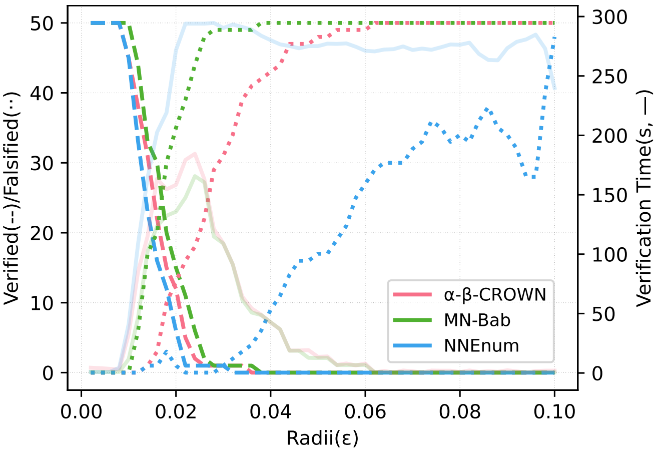


Fig. 2. Solved problems and verification time vs. epsilon radii

ties can be reduced to local robustness properties [37]. This allows us to vary the verification problem difficulty by controlling the robustness property's epsilon-radius. Conceptually, we know that verification problems with sufficiently small (large) radii will be verified (falsified) – a radius of 0 is trivially verified and a radius comprising the full input domain requires that a network produce a constant output. Verifier developers have incorporated techniques, like applying adversarial attacks and using coarse overapproximations, to quickly handle such cases [3, 48]. To sidestep these verification fast paths and exercise the core verification algorithms in our study, we select epsilon values for properties as follows.

For each network, we conducted a preliminary study with varying radii to assess the difficulty of the verification problems. Fig. 2 shows the results for **M2** on 50 different center-points with the three verifiers. The dashed lines show the number of verified problems and the dotted lines the number of falsified problems (left y-axis). We observe the trend that small epsilon leads to uniformly verified problems and large epsilon to uniformly falsified problems. Moreover, one can observe low verification times (right y-axis) in these extreme epsilon regimes, due to the fast path optimizations.

Our strategy for selecting harder verification properties is to choose a sample of radii around the point where the number of verified and falsified problems crossover, e.g., 0.018 in this plot for MN-BAB. We choose the crossover point of the best verifier who solved the most problems to design the radii shown in Tab. 1. This leads to a balance in verification ground truth between SAT and UNSAT answers, and these more challenging problems force the underlying algorithms to more precisely model network behavior, e.g., splitting of unstable neurons into branch and bound cases.

Selecting Verifiers Unlike other research that focuses on improving the performance of a single verifier with a single customized pruning techniques [10, 53, 59], our goal is to explore how the space of stabilization strategies impact a range of verification approaches. Towards this goal, we select the three best-performing verifiers from VNN-COMP 2022 [32] that were available: α,β-CROWN, MN-BAB, and NNEnum [4]. Improving the performance of these verifiers will extend the state-of-the-art in scalable DNN verification.

Network Training Stabilizers are incorporated into training, so we use a baseline(Baseline) trained without any stabilizers using the Adam optimizer with a 10^{-3} learning rate and 0.99 decay for 20 epochs. All stabilizers are customizable with hyperparameters, as described in §3 and §4. We use the well-tuned parameter for RS Loss introduced in [53], and perform a binary search of the parameter space for Bias Shaping and Stable Pruning. To elaborate, RS Loss uses always-active scheduling with 10^{-4} weight parameter; Bias Shaping uses interval scheduling activated every 5/25/50 mini-batches and adjusts 2%/5%/5% of unstable neurons each time it is applied for **M2/M6/C2** architectures respectively; Stable Pruning undertakes an interval scheduling that is activated for every 5/50/50 mini-batches with a pruning ratio of 2%/5%/5% respectively. The resulting neural networks with the largest test accuracy of the last five epochs are selected for verification. To account for stochasticity in training, we train each network 5 times and report the mean data for each.

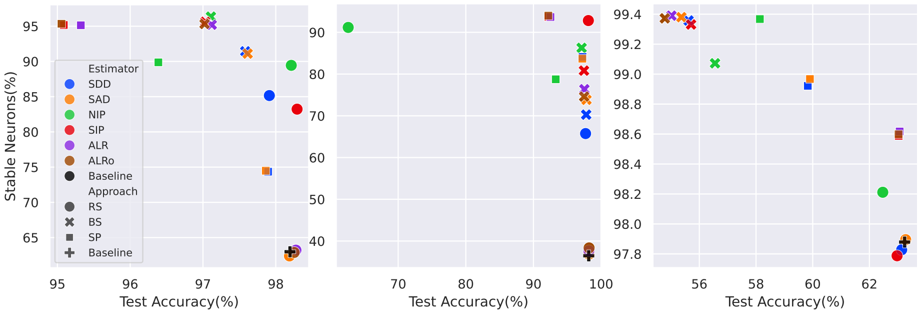
These choices for the space of experiments yield a total of 1,215 training tasks and 36,450 verification tasks. Each training task is run with one GTX 1080 Ti GPU with 11G VRAM. Each verification task is run with 8GB of memory on one core of the Intel Xeon Gold 6130 CPU @ 2.10GHz with a timeout of 300 seconds. The total CPU time spent on training and verification across our experiments is 1858 and 1052 hours, respectively.

5.2 RQ1: Stabilizing Neurons

Stabilizers aim to linearize a portion of the behavior encoded by ReLU activation across the set of computations activated for a property precondition. In this experiment, we directly measure this by recording the percentage of neurons that are stable during verification. We also record model test accuracy to understand the trade-offs of the stabilization methods and stability estimators. Existing verifiers do not record the number of stable neurons, so we modified an open-source DNN verifier, NeuralSAT [11], to record the number of stable neurons computed during verification.

Fig. 3 presents the average test accuracy and the average number of stable neurons computed across the five training seeds for the three architectures across the stabilizers in the benchmark as described in §5.1. The black ✚ sign indicates the Baseline (Baseline), the ● sign represents RS Loss (RS), ✖ means the Bias Shaping (BS) method, and ■ is Stable Pruning (SP). Six different colors denote the different stability estimators. Across all three architectures, most techniques can increase the number of stable neurons, but some of the techniques

[4] Verinet performed well in the competition, but it required a custom solver that is not freely available.



(a) **M2** (b) **M6** (c) **C3**

Fig. 3. Stable neurons(%) vs. test accuracy(%) per model

lead to a loss in test accuracy. For the **M2** architecture, RS Loss with NIP can significantly increase the number of stable neurons by more than 26 percentage points without compromising accuracy. For **M6**, RS Loss yields an even greater increase of 55 percentage points but in combination with the SIP estimator. For the Convolutional **C3** network, a very high percentage of neurons are already stable so only marginal improvement can be achieved. Here the Stable Pruning method performs best while preserving accuracy, but it only yields a percentage point increase. For all of the architectures, if one is willing to sacrifice a degree of accuracy then further increases in stability can be achieved. For example, for **M2** bias shaping can achieve an additional 7 percentage point increase in stable neurons at the cost of just over 1 percentage point in test accuracy.

Incorporating stabilization in training can increase training time. Fig. 4 shows the average training time for **M2** normalized to the Baseline. The trend for **M6** is similar to the other architectures. The clear outlier in terms of cost is the **ALRo** estimator when used with RS Loss, which incurs more than a 5-fold increase in training time. This overhead even prevents RS Loss from practically training with **ALR** and **ALRo** on the **C3** architecture. The overhead of most of the other estimators is negligible, including those that yielded significant increases in stable neurons.

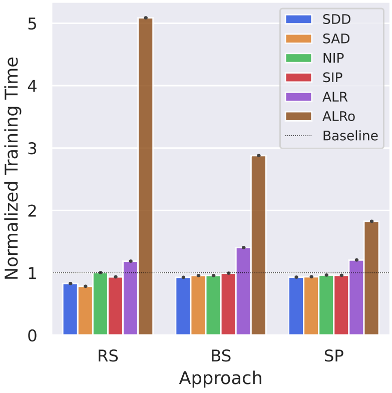
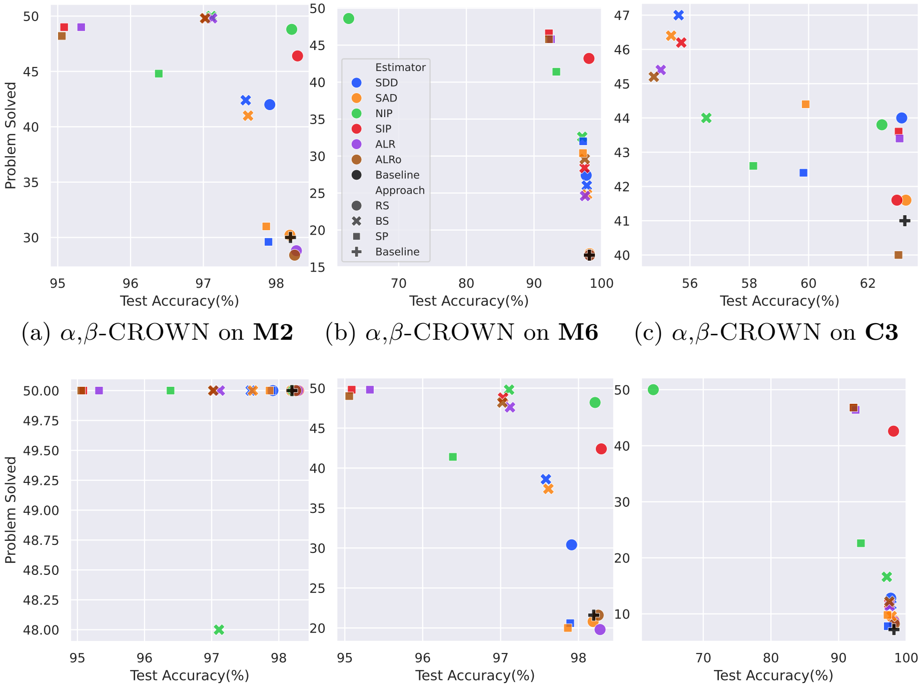


Fig. 4. Normalized training time

RQ1 Findings Across the study there are combinations of stabilization methods and stability estimators that are capable of increasing the number of stable neurons, in many cases substantially, without compromising test accuracy or training time.



(a) α,β-CROWN on **M2** (b) α,β-CROWN on **M6** (c) α,β-CROWN on **C3**

(d) MN-BAB on **M2** (e) NNEnum on **M2** (f) NNEnum on **M6**

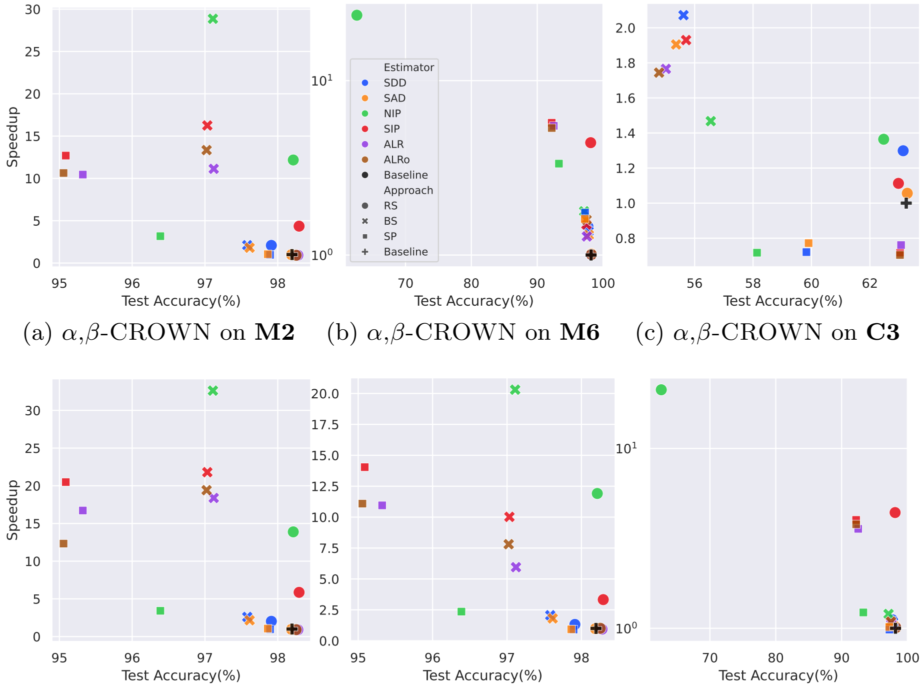
Fig. 5. Solved verification problems vs. test accuracy(%)

5.3 RQ2: Enhancing Verification

RQ1 demonstrates the ability of stabilizers to increase the number of stable neurons across a space of verification problems. This question explores whether those increases lead to improvements in verifier performance. To assess the generalization of the stabilizers to variations of DNN properties, we verify 50 local robustness properties per trained network, pairing 10 center points with each of the 5 epsilon radii. We run the three selected state-of-the-art verifiers on each problem.

We measure two metrics to assess verification performance: (1) the number of problems, i.e., the network, center-point, and radii combination, each verifier can solve, i.e., produce either an SAT or UNSAT result, and (2) the time taken to solve those problems. Note that our metrics exclude runs that produce errors, exceed a 300-second timeout, or an 8GB memory bound. These metrics are standard for assessing verifier performance and while sometimes they are aggregated, as in PAR2 [55], we keep them separate here to explore them independently.

Fig. 5 shows six plots of the number of verification problems solved versus test accuracy across the three architectures using three of the verifiers. The trends in



(a) α,β-CROWN on **M2** (b) α,β-CROWN on **M6** (c) α,β-CROWN on **C3**

(d) MN-BAB on **M2** (e) NNENUM on **M2** (f) NNENUM on **M6**

Fig. 6. Verification time speedup vs. test accuracy(%)

these plots are largely consistent with the findings of RQ1 - when more neurons are stable the verifiers are more effective in solving problems. RS LOSS, with different estimators, increases the number of problems solved by factors up to 5.92 for these verifier network combinations without sacrificing test accuracy. As in RQ1, further performance improvements are possible by sacrificing accuracy. For example, on **M2** α,β-CROWN can improve by a factor of 1.67 using BIAS SHAPING with a reduction of 1 percentage point in accuracy.

The trends shown here are consistent with the performance of α,β-CROWN and NNENUM across the study, but MN-BAB exhibited different performance. For **M2** and **M6**, the baseline technique was able to solve all 50 problems so there is no opportunity for improvement, while almost all the stabilizers can maintain the 50 problems solved. Note that the implementation of MN-BAB just doesn't support the **C3** architecture. While the number of problems does not change for MN-BAB with stabilization as we discuss below its runtime is reduced.

Fig. 6 plots the verification time speedup over BASELINE against test accuracy for 6 verifier network pairs. We observe a similar trend to what was observed for the number of neurons stabilized and the number of verification problems solved

– stabilization can speed up verification without compromising test accuracy. For MN-BAB on **M2** while the number of problems solved did not change, using RS Loss with **NIP** yielded a factor of 14 speedup. For **M6** we see a speedup of up to a factor of 5 with NNEnum and for **C3** more modest speedups for α,β-CROWN. The MN-BAB plot also shows, as observed above, that further speedups – greater than 30 fold – can be achieved if one compromises accuracy by about 1 percentage point.

RQ2 Findings Stabilizing neurons during training can substantially increase the number of problems solved and reduce the time required to solve them by state-of-the-art DNN verifiers without compromising test accuracy. Further improvement in verifier performance can be achieved with a small sacrifice in test accuracy.

5.4 Discussion

The data show a significant degree of variability in the effectiveness of particular stable training approaches with verifiers and verification problems. Broadly speaking RS Loss seems to perform well when one is unwilling to sacrifice test accuracy, but the best estimator varies depending on the verifier and problem – with **SDD**, **NIP**, and **SIP** yielding the best performance. For the large Convolutional network, Stable Pruning also performs well without compromising test accuracy. We believe this to be consistent with the broader results from the field of structured pruning [18,43], where it has been found that large networks tend to be over-parameterized and can thus accommodate significant pruning without compromising accuracy. While the study shows that many of the methods can yield benefits, we believe that it also demonstrates that certain stabilization approaches, e.g., RS Loss with **ALRo**, are too costly for use in practice. Further study should focus on how to select the best stable training approach, and its hyperparameters, to yield the best improvement for a given verifier and class of verification problems. We believe it will be fruitful to develop such *training for verification* approaches in concert with algorithmic and engineering improvements to verification algorithms.

5.5 Threats to Validity

The chief threats to internal validity relate to whether the collection of test accuracy, stable neurons, verification problems solved, and verification time were accurate. We tested the accuracy of all stabilizer-trained networks, cross-checked problem solutions across verifiers, and thoroughly tested our instrumentation of NeuralSAT for recording neuron stability. Regarding external validity, while our study was scoped to manage experimental costs, it spanned: 3 verifiers, 3 network architectures, 50 property specifications, and 5 seeds. We used fixed sets of training and stabilizer parameters per neural network architecture, which potentially underestimated the benefit that might be observed by customizing parameters. While broadening the study further would be a valuable direction for future work, the scope of the study is sufficient to support the finding that stabilizers can enhance DNN verification across a breadth of contexts.

6 Conclusion

Verifying neural networks is a challenging task due to their high computational complexity. In this work, we propose two novel approaches Bias Shaping and Stable Pruning, to enhance the scalability of DNN verifiers by inducing more stable neurons during the training process. In addition, we designed six neuron stability estimators to drive stability-oriented training. Across a significant study, we found that focusing on stability yields a viable method to achieve training for verification that can significantly improve the ability to solve problems and speed up state-of-the-art verifiers.

Besides the promising results, we identified more opportunities when working on this project. In the future, we plan to (1) extend our methods to real-world large neural network architectures; (2) explore automatic ways to tune hyperparameters that lead to better performance; (3) further enhance the stabilizers' performance while minimizing accuracy trade-offs; (4) study the applicability of stabilizer combinations; and lastly (5) study the verification algorithms to understand how to customize stabilizers to benefit the most.

Acknowledgment

This material is based in part upon work supported by National Science Foundation awards 1900676, 2019239, 2129824, 2217071, and 2312487.

References

1. Bak, S.: Execution-guided overapproximation (ego) for improving scalability of neural network verification. In: International Workshop on Verification of Neural Networks (2020)
2. Bak, S., Liu, C., Johnson, T.: The second international verification of neural networks competition (vnn-comp 2021): Summary and results. arXiv preprint arXiv:2109.00498 (2021)
3. Bak, S., Tran, H.D., Hobbs, K., Johnson, T.T.: Improved geometric path enumeration for verifying relu neural networks. In: International Conference on Computer Aided Verification. pp. 66–96. Springer (2020)
4. Baluta, T., Chua, Z.L., Meel, K.S., Saxena, P.: Scalable quantitative verification for deep neural networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 312–323. IEEE (2021)
5. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A.: Measuring neural net robustness with constraints. Advances in neural information processing systems **29** (2016)
6. Biere, A., Clarke, E., Raimi, R., Zhu, Y.: Verifying safety properties of a powerpc-microprocessor using symbolic model checking without bdds. In: Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11. pp. 60–71. Springer (1999)

7. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient verification of relu-based neural networks via dependency analysis. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34(04), pp. 3291–3299 (2020)
8. Brix, C., Müller, M.N., Bak, S., Johnson, T.T., Liu, C.: First three years of the international verification of neural networks competition (vnn-comp). International Journal on Software Tools for Technology Transfer pp. 1–11 (2023)
9. Bunel, R., Mudigonda, P., Turkaslan, I., Torr, P., Lu, J., Kohli, P.: Branch and bound for piecewise linear neural network verification. Journal of Machine Learning Research **21**(2020) (2020)
10. Chen, T., Zhang, H., Zhang, Z., Chang, S., Liu, S., Chen, P.Y., Wang, Z.: Linearity grafting: Relaxed neuron pruning helps certifiable robustness. In: International Conference on Machine Learning. pp. 3760–3772. PMLR (2022)
11. Duong, H., Li, L., Nguyen, T., Dwyer, M.: A dpll (t) framework for verifying deep neural networks. arXiv preprint arXiv:2307.10266 (2023)
12. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: UAI. vol. 1(2), p. 3 (2018)
13. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: International Symposium on Automated Technology for Verification and Analysis. pp. 269–286. Springer (2017)
14. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: International Conference on Computer Aided Verification. pp. 43–65. Springer (2020)
15. Fazlyab, M., Morari, M., Pappas, G.J.: Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming. IEEE Transactions on Automatic Control (2020)
16. Feng, C., Chen, Z., Hong, W., Yu, H., Dong, W., Wang, J.: Boosting the robustness verification of dnn by identifying the achilles's heel. arXiv preprint arXiv:1811.07108 (2018)
17. Ferrari, C., Müller, M.N., Jovanovic, N., Vechev, M.T.: Complete verification via multi-neuron relaxation guided branch-and-bound. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net (2022), `https://openreview.net/forum?id=l_amHf1oaK`
18. Frankle, J., Carbin, M.: The lottery ticket hypothesis: Finding sparse, trainable neural networks. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019), `https://openreview.net/forum?id=rJl-b3RcF7`
19. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 3–18. IEEE (2018)
20. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. pp. 315–323. JMLR Workshop and Conference Proceedings (2011)
21. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: International conference on computer aided verification. pp. 3–29. Springer (2017)
22. Johnson, T.T., Liu, C.: Vnn-comp2020 report, `https://www.overleaf.com/read/rbcfnbyhymmy`
23. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)

24. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification. pp. 443–452. Springer (2019)
25. Khedher, M.I., Ibn-Khedher, H., Hadji, M.: Dynamic and scalable deep neural network verification algorithm. In: ICAART (2). pp. 1122–1130 (2021)
26. Khedr, H., Ferlez, J., Shoukry, Y.: Effective formal verification of neural networks using the geometry of linear regions. arXiv preprint arXiv:2006.10864 (2020)
27. Li, J., Liu, J., Yang, P., Chen, L., Huang, X., Zhang, L.: Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In: International Static Analysis Symposium. pp. 296–319. Springer (2019)
28. Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M.J., et al.: Algorithms for verifying deep neural networks. Foundations and Trends® in Optimization **4**(3-4), 244–404 (2021)
29. Livni, R., Shalev-Shwartz, S., Shamir, O.: On the computational efficiency of training neural networks. Advances in neural information processing systems **27** (2014)
30. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks. arXiv preprint arXiv:1706.07351 (2017)
31. Lu, J., Kumar, M.P.: Neural network branching for neural network verification. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net (2020), `https://openreview.net/forum?id=B1evfa4tPB`
32. Müller, M.N., Brix, C., Bak, S., Liu, C., Johnson, T.T.: The third international verification of neural networks competition (vnn-comp 2022): summary and results. arXiv preprint arXiv:2212.10376 (2022)
33. Paul, M., Chen, F., Larsen, B.W., Frankle, J., Ganguli, S., Dziugaite, G.K.: Unmasking the lottery ticket hypothesis: What's encoded in a winning ticket's mask? In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023), `https://openreview.net/pdf?id=xSsW2Am-ukZ`
34. Raghunathan, A., Steinhardt, J., Liang, P.: Certified defenses against adversarial examples. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net (2018), `https://openreview.net/forum?id=Bys4ob-Rb`
35. Shriver, D., Elbaum, S., Dwyer, M.: Artifact: Reducing dnn properties to enable falsification with adversarial attacks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 162–163 (2021). https://doi.org/10.1109/ICSE-Companion52605.2021.00068
36. Shriver, D., Elbaum, S., Dwyer, M.B.: Dnnv: A framework for deep neural network verification. In: International Conference on Computer Aided Verification. pp. 137–150. Springer (2021)
37. Shriver, D., Elbaum, S., Dwyer, M.B.: Reducing dnn properties to enable falsification with adversarial attacks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 275–287. IEEE (2021)
38. Shriver, D., Xu, D., Elbaum, S., Dwyer, M.B.: Refactoring neural networks for verification. arXiv preprint arXiv:1908.08026 (2019)
39. Singh, G., Ganvir, R., Püschel, M., Vechev, M.: Beyond the single neuron convex barrier for neural network certification. Advances in Neural Information Processing Systems **32**, 15098–15109 (2019)
40. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. NeurIPS **1**(4), 6 (2018)

41. Singh, G., Gehr, T., Püschel, M., Vechev, M.: Boosting robustness certification of neural networks. In: International Conference on Learning Representations (2018)
42. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
43. Tan, C.M.J., Motani, M.: Dropnet: Reducing neural network complexity via iterative pruning. In: International Conference on Machine Learning. pp. 9356–9366. PMLR (2020)
44. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019), `https://openreview.net/forum?id=HyGIdiRqtm`
45. Tran, H.D., Yang, X., Lopez, D.M., Musau, P., Nguyen, L.V., Xiang, W., Bak, S., Johnson, T.T.: Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: International Conference on Computer Aided Verification. pp. 3–17. Springer (2020)
46. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. Advances in neural information processing systems **31** (2018)
47. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 1599–1614 (2018)
48. Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.J., Kolter, J.Z.: Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. Advances in Neural Information Processing Systems **34**, 29909–29921 (2021)
49. Weng, L., Zhang, H., Chen, H., Song, Z., Hsieh, C.J., Daniel, L., Boning, D., Dhillon, I.: Towards fast computation of certified robustness for relu networks. In: International Conference on Machine Learning. pp. 5276–5285. PMLR (2018)
50. Wong, E., Kolter, Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: International Conference on Machine Learning. pp. 5286–5295. PMLR (2018)
51. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE transactions on neural networks and learning systems **29**(11), 5777–5783 (2018)
52. Xiang, W., Tran, H.D., Rosenfeld, J.A., Johnson, T.T.: Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In: 2018 Annual American Control Conference (ACC). pp. 1574–1579. IEEE (2018)
53. Xiao, K.Y., Tjeng, V., Shafiullah, N.M.M., Madry, A.: Training for faster adversarial robustness verification via inducing relu stability. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019), `https://openreview.net/forum?id=BJfIVjAcKm`
54. Xu, D., Mozumder, N.J., Duong, H., Dwyer, M.B.: The OCTOPUS Framework $\models$ Training for Verification: Increasing Neuron Stability to Scale DNN Verification (1 2024). https://doi.org/10.6084/m9.figshare.24916248.v3
55. Xu, D., Shriver, D., Dwyer, M.B., Elbaum, S.: Systematic generation of diverse benchmarks for dnn verification. In: International Conference on Computer Aided Verification. pp. 97–121. Springer (2020)
56. Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.W., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.J.: Automatic perturbation analysis for scalable certified robustness and beyond. Advances in Neural Information Processing Systems **33** (2020)

57. Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S., Lin, X., Hsieh, C.J.: Fast and Complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In: International Conference on Learning Representations (2021), `https://openreview.net/forum?id=nVZtXBI6LNn`
58. Zhang, H., Weng, T., Chen, P., Hsieh, C., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 4944–4953 (2018), `https://proceedings.neurips.cc/paper/2018/hash/d04863f100d59b3eb688a11f95b0ae60-Abstract.html`
59. Zhangheng, L., Chen, T., Li, L., Li, B., Wang, Z.: Can pruning improve certified robustness of neural networks? Transactions on Machine Learning Research (2022)

NeuroSynt: A Neuro-symbolic Portfolio Solver for Reactive Synthesis

Matthias Cosler[1(✉)], Christopher Hahn[2], Ayham Omar[1], and Frederik Schmitt[1]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{matthias.cosler,ayham.omar,frederik.schmitt}@cispa.de
[2] X, the moonshot factory, Mountain View, USA**
chrishahn@google.com

Abstract. We introduce NeuroSynt, a neuro-symbolic portfolio solver framework for reactive synthesis. At the core of the solver lies a seamless integration of neural and symbolic approaches to solving the reactive synthesis problem. To ensure soundness, the neural engine is coupled with model checkers verifying the predictions of the underlying neural models. The open-source implementation of NeuroSynt provides an integration framework for reactive synthesis in which new neural and state-of-the-art symbolic approaches can be seamlessly integrated. Extensive experiments demonstrate its efficacy in handling challenging specifications, enhancing the state-of-the-art reactive synthesis solvers, with NeuroSynt contributing novel solves in the current SYNTCOMP benchmarks.

1 Introduction

The reactive synthesis problem [16] seeks to automatically construct an *implementation* from a system's *specification*. Rather than delving into the intricate nuances of *how* a system computes, hardware designers can describe *what* the system should achieve and leave implementation details to the synthesis engine. We introduce NeuroSynt, a portfolio solver for reactive synthesis that combines the efficiency and scalability of neural approaches with the soundness and completeness of symbolic solvers.

The reactive synthesis problem has seen significant progress in recent years [12,29,30,50] with active tooling development [1,11,25,27,46,39], and an annual competition (SYNTCOMP [36]). However, applications beyond the competition to an industrial scale are still limited. The advent of machine learning, empowered by the advancements in deep learning architecture and hardware accelerators, has the potential to drastically increase performance in reactive synthesis. While deep learning approaches offer efficiency, they lack soundness and completeness guarantees, which are essential to the reactive synthesis problem.

We address this challenge by introducing NeuroSynt, a portfolio solver framework for reactive synthesis that aims to bridge the gap between soundness, completeness, and practical efficiency through the combination of state-of-the-art

** Work done while being at Stanford University.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 45–67, 2024.
https://doi.org/10.1007/978-3-031-57256-2_3

symbolic solver, model-checker, and deep learning techniques. The integrated neural solver computes candidate implementations while model-checking tools verify the candidate solutions to ensure soundness. To ensure completeness, the neural solver is backed up by several state-of-the-art symbolic solvers running in parallel.

In particular, our main contribution is the design and open-source implementation of the extensible and efficient portfolio solver. `NeuroSynt`'s design prioritizes extensibility: Its modular architecture facilitates the seamless integration of new models, algorithms, or optimization techniques. This adaptability ensures that `NeuroSynt` remains relevant amidst evolving methodologies, providing researchers with a unified platform to experiment, validate, and advance their innovations in the reactive synthesis domain.

Additionally, we contribute an advanced neural solver for reactive synthesis (based on [57]) that handles larger and more complex specifications, improving its performance on real-world instances from SYNTCOMP.

Our results show that deep learning methods can indeed increase the performance of reactive synthesis tools. `NeuroSynt` provides smaller solutions faster while maintaining soundness and completeness. Our portfolio solver enhances the performance of the state-of-the-art Strix [46] by 31 samples on the SYNTCOMP 2022 benchmark, and the bounded synthesis tool BoSy [27] by 152 samples. Notably, a virtual best solver (VBS) that combines the neural solver with all tools in the SYNTCOMP 2022 competition solves an additional 20 instances that a VBS without the neural solver could not solve.

2 Background

Reactive Synthesis. The reactive synthesis problem is a well-known algorithmic challenge, that dates back to Church [16,15] as the problem of automatically constructing an *implementation* from a system's *specification*. With the decidability findings in 1969 [10] (using games) and 1972 [54] (using automata), a long history of work on reactive synthesis was initiated. After the introduction of temporal logics in 1977 [51], the complexity for LTL reactive synthesis was found to be 2-EXPTIME complete [52] but undecidable for distributed systems [53]. Since then, many different approaches have been developed (e.g., [12,29,30,50]) and implemented in tools (e.g. [1,11,25,27,38,39,46,55]). Moreover, an annual competition, the Reactive Synthesis Competition (SYNTCOMP [36]), associated with the International Conference on Computer Aided Verification (CAV) is organized to track the improvement of algorithms and tooling.

Linear-time Temporal Logic (LTL). LTL extends propositional logic by introducing temporal operators $\mathcal{U}$ (until) and $\bigcirc$ (next). Several additional operators can be derived: $\diamondsuit \varphi \equiv \mathit{true}\, \mathcal{U} \varphi$ and $\square \varphi \equiv \neg \diamondsuit \neg \varphi$. $\diamondsuit \varphi$ is interpreted as φ will *eventually* hold in the future and $\square \varphi$ as φ holds *globally*. Operators can be nested, e.g. $\square \diamondsuit \varphi$ states that φ has to occur infinitely often. Linear-time Temporal Logic (LTL) [51] is the prototypical temporal logic for expressing requirements of reactive systems. For example, the following formula describes an arbiter: Given two processes and a shared resource, the formula $\square(r_0 \rightarrow \diamondsuit g_0) \wedge \square(r_1 \rightarrow \diamondsuit g_1) \wedge \square\neg(g_0 \wedge g_1)$ describes that whenever a process requests (r) access to a shared resource, it will eventually be granted (g). Formally, the reactive synthesis problem for LTL is defined over the notion of a strategy as follows: An LTL formula φ over atomic propositions $AP = I \,\dot{\cup}\, O$ is realizable if there exists a strategy $f : (2^I)^* \rightarrow (2^O)$ that satisfies φ. We show the formal syntax and semantics of LTL and the definition of a strategy in the full version [20].

And-Inverter Graphs. And-Inverter Graphs are directed acyclic graphs that represent reactive systems using three fundamental building blocks: the AND gate, the inverter (NOT gate), and latches, which can store a single bit for one timestep. The graph's edges define the connections between gates, indicating how signals propagate through the circuit. And-Inverter Graphs, especially the AIGER format [8,9], are widely used in formal verification and reactive synthesis. The AIGER format follows a well-defined specification. The first line contains header information: the maximal variable id, the number of inputs, outputs, latches, and AND gates in the circuit. The circuit's components are following in this order: inputs, latches, outputs, AND-gates, with each component in one line. Each input, AND-gate, and latch defines an even number (variable id) to which other gates and outputs can refer to establish connections between gates. NOT gates are implicitly encoded by the odd version of each number. True and False are encoded by the numbers `1` and `0`.

Deep Learning in Formal Methods. Deep Learning methods have been successfully applied to various domains in formal methods. Applications of deep learning methods in symbolic reasoning include SAT/SMT solving [4,13,58,59], temporal logics such as generating satisfying traces [33], reactive synthesis and repair [21,42,57], as well as generating symbolic reasoning problems in temporal logics and symbolic mathematics [41]. Mathematical reasoning problems, including integration and differential equations, have been approached with transformers [43] and through code generation with Large Language Models (LLMs)[22]. Mathematical reasoning has also been tackled through automatic proof generation [44]. More general applications of deep learning to theorem proving are guiding the proof search with clause selection for CNF formulas [45] and tactic and premise selection/prediction for Coq and HOL light [5,6,34,48]. In contrast to proof guidance, LLMs can be used for end-to-end generation and repair of proofs in Isabelle/HOL [31]. LLMs have recently also enabled a step towards autoformalization of unstructured natural language for theorem proving [37,64] and temporal logic [19]. Further, deep learning has had a considerable impact on program verification and synthesis, i.e., for termination analysis [3,32], creating loop invariants [49,56,61] and program synthesis/induction [2,18,26,28].

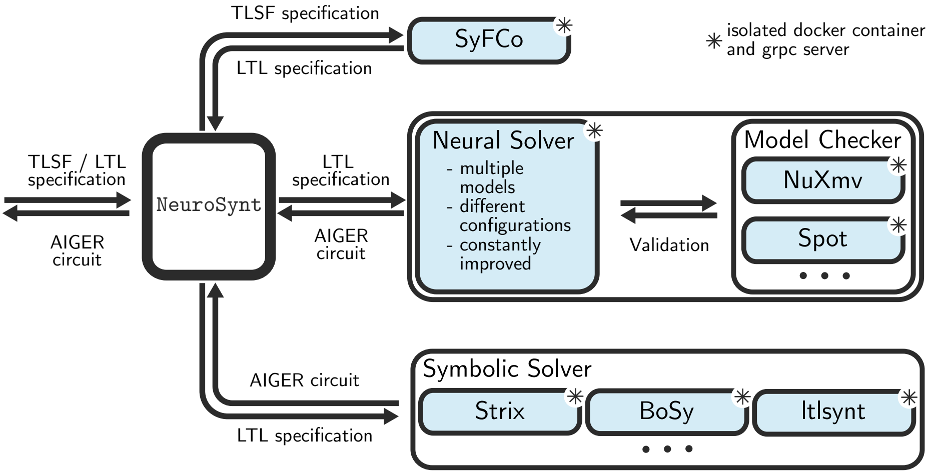


Fig. 1. An overview of NeuroSynt.

3 The Neuro-symbolic Portfolio Solver NeuroSynt

The portfolio solver provides a unified approach to neural and symbolic methods for reactive synthesis. For a seamless integration of the neural method, NeuroSynt relies on model checking (for soundness) and is backed up by symbolic synthesis tools (for completeness).

3.1 Overview

We provide an overview of NeuroSynt in the following. Figure 1 shows the system design of NeuroSynt. With a single call, a sample is 1) translated from TLSF [35], the standardized input format for reactive synthesis, to LTL assumptions and guarantees. 2) Fed into the neural solver described in Section 4 with candidate solutions being verified by a model-checker. This is a feasible approach since LTL model checking is computationally significantly easier than reactive synthesis (PSPACE [62] vs. 2-EXPTIME [52]). 3) A symbolic solver is queried simultaneously with the neural solver. The final result is an implementation in the form of an AIGER [8] circuit, which is either a verified candidate circuit of the neural solver or the circuit returned by the symbolic solver. Depending on the specification's realizability, the circuit either represents the system implementation (proving realizability) or the environment behavior (proving unrealizability).

All components, neural solver, symbolic solver, and model-checker, are isolated Docker containers. All communication channels between components are defined through a standardized API. Therefore, extending, maintaining, and updating tools are uncoupled from NeuroSynt's implementation. Currently integrated are solvers based on the Python library ML2[3], including the neural solver

[3] https://github.com/reactive-systems/ml2

described in Section 4, nuXmv [14], NuSMV [17], Spot [23], Strix [46], and BoSy [27]. We use SyFCo [35] to convert from TLSF to assumptions and guarantees in LTL.

3.2 Usage

Since NeuroSynt comprises multiple tools that operate in conjunction during each execution, users must specify arguments to tailor the behavior of these tools. We categorize these arguments into tool-specific and general arguments to simplify this process. General arguments are unrelated to any specific tool and are passed with the execution command in the command-line interface.

For tool-specific arguments, we use the YAML format [7] to create configuration files that encompass the neural engine arguments, the chosen tool for model checking, and symbolic synthesis tasks, along with their respective arguments. These configuration files facilitate reproducibility and provide a structured way to manage tool-specific settings.

Depending on user choice, `NeuroSynt` can either wait for all tools to finish/timeout and report all results or return the fastest solution. We allow the standardized input format TLSF [35] and simple assume-guarantee structured files in LTL.

`NeuroSynt` offers two primary execution commands: *benchmark* for solving a dataset of samples and *synthesize* for processing individual samples. For *benchmark*, all results are saved in a CSV file, which can be further analyzed. In all other cases, the result is printed to the command line. First, we indicate whether the specification was found to be REALIZABLE or UNREALIZABLE, after which we print the system in AIGER format [8].

We refer to the full version [20] for more usage instructions and examples.

3.3 Implementation and Extensibility

The central design goal of `NeuroSynt` is to provide interfaces that are easy to implement when adding and integrating new components. We first describe the communication interfaces between components. Secondly, we detail some of the messages, and lastly, we describe the options to extend the portfolio solver.

Each solver or model-checker is isolated in a Docker container and communicates with other components through gRPC interfaces. gRPC is a high-performance open-source framework initially developed by Google for building remote procedure call (RPC) APIs. Protocol buffers (protobuf) are used as the interface definition language, ensuring programming-language-agnostic interfaces.

In Figure 2, we show the communication through gRPC APIs for the run of `NeuroSynt` with one specification. In the first step, each tool is initialized using setup messages, ensuring the components' successful connection. After setup, a synthesis problem call is sent to the symbolic and neural solver in parallel. Both solvers eventually report with a synthesis solution. Before responding, the neural

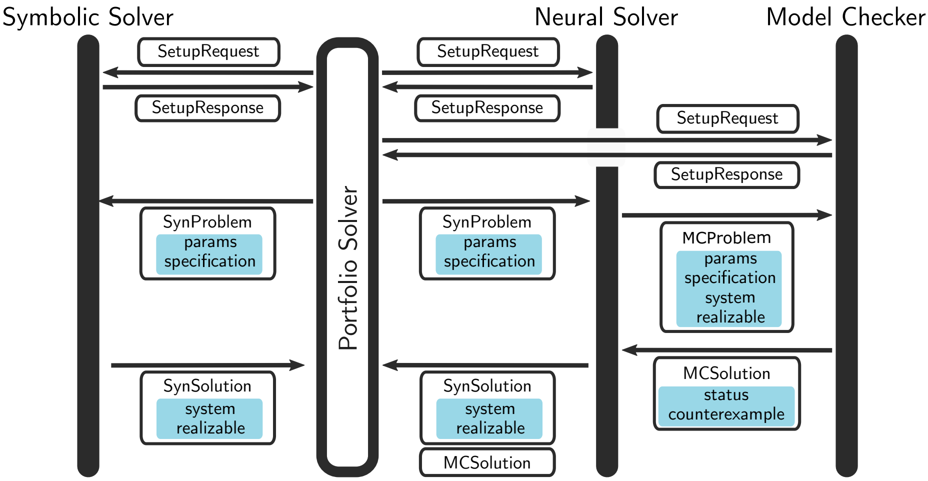


Fig. 2. Communication diagram of gRPC calls for a run of `NeuroSynt`, calling the Symbolic Solver and the neural solver, including model-checking.

solver makes one or multiple calls to the model checker with candidate solutions, the specification, and the information on whether the specification is suspected to be realizable. The model checker answers a status and optionally a counterexample. The neural solver then selects one solution if multiple candidates have been generated and responds to `NeuroSynt`. The following details the specific protobuf messages that can be exchanged between components.

SetupRequest and SetupResponse. As initialization, the components exchange simple messages through a JSON-like object. This message establishes the successful connection and allows the user to provide some tool- but not run-specific arguments. In the case of the neural solver, the model name and other parameters are transmitted to load the model into the memory. The component then responds with a simple success flag or error message.

SynProblem, SynSolution, and UnsoundSynSolution. The *SynProblem* (request) contains an LTL Specification and a set of JSON-like parameters to configure the run- and tool-specific arguments, such as timeout or the number of threads. The LTL specification is decomposed into guarantees and assumptions, both strings in infix or prefix notation. A *SynSolution* contains the system as the string representation of an AIGER circuit or mealy machine, a status (realizable, unrealizable, error, timeout, nonsuccess), the calculation duration, and the tool's name. No system must be returned if error, timeout, or nonsucces were reported. The *UnsoundSynSolution* consists of a *SynSolution* and *MCSolution* and is returned by the neural solver. We show the protobuf definition for the *SynSolution*, *SynProblem*, and Specification in Figure 3. More definitions can be found in the full version [20].

```
// An LTL Synthesis solution. Used as response message for Synthesis.
message LTLSynSolution {
  // AIGER circuit. It is allowed to pass no system, e.g. if a timeout
  // happened.
  optional AigerCircuit circuit = 1;
  // Shows, whether the specification was found to be realizable or
  // unrealizable. May not be set, e.g. if a timeout happened.
  optional bool realizable = 2;
  // A status that includes useful information about the run.
  LTLSynStatus status = 3;
  // Here additional information should be supplied if the status value
  // requires more details.
  string detailed_status = 4;
  // which tool has created the response.
  Tool tool = 5;
  // How long the tool took to create the result.
  optional google.protobuf.Duration time = 6;
}

message LTLSynProblem {
  // Defines run- and tool-specific parameters. As Map (Dict in Python).
  // Typical examples are threads, timeouts etc. Can be empty.
  map<string, string> parameters = 1;
  // A decomposed specification (assumptions + guarantees).
  DecompLTLSpecification decomp_specification = 2;
}

message DecompLTLSpecification {
  // All input atomic propositions that occur in guarantees or assumptions.
  repeated string inputs = 1;
  // All output atomic propositions that occur in guarantees or assumptions
  repeated string outputs = 2;
  // A set of guarantees that make up the specifications. All inputs and
  // outputs occurring in any guarantee must be part of input/output.
  repeated LTLFormula guarantees = 3;
  // A set of assumptions that make up the specifications. All inputs and
  // outputs occurring in any guarantee must be part of input/output.
  repeated LTLFormula assumptions = 4;
}
```

Fig. 3. The protobuf definition for a *SynSolution*, *SynProblem*, and decomposed LTL specification. Slightly simplified for easier comprehension. We refer the reader to the artifact and our repository for the full definitions.

MCProblem and MCSolution. A tool can request its candidate solutions to be model-checked by sending an *MCProblem* request. This message contains a set of JSON-like parameters to configure the run- and tool-specific arguments, an LTL specification (see *SynProblem*), and a system and status (see *SynSolution*). The *MCSolution* contains the status of the model-checking and, if violating, a counterexample in the form of an error trace and the duration of the computation. We show the relevant protobuf definitions in the full version [20].

NeuroSynt can be extended in three major ways. New neural solvers, symbolic solvers, and model-checking tools can be integrated. Although not required, we recommend wrapping all components into Docker containers as it helps reproducibility, portability, and isolation, especially when run on high-performance clusters.

Neural Solver. The neural solver sits at the core of the portfolio solver, with connections to both the model-checking component and the main portfolio solver. This component has to support receiving and responding to a *SetupRequest* and a *SetupResponse* for initialization. Furthermore, it should respond to *SynProblem* requests with *UnsoundSynSolution*. To verify candidate solutions, the neural solver should initiate communication with the model-checking component to verify candidate solutions. Therefore, it should also support sending *MCProblem* requests and receiving *MCSolution* responses. The neural solver can be independent of the ML2 library if it implements the two communication interfaces mentioned above. It can also be based on the ML2 library, where one could benefit from the existing infrastructure ML2 provides.

Model checking tools. A model checker should respond to a *SetupRequest* with a *SetupResponse* and receive the *MCProblem* request, perform model checking and answer with an *MCSolution*.

Symbolic Solver. New symbolic solvers can be integrated into `NeuroSynt` by implementing the server side of our generic protocol buffer interface for symbolic solvers. As for all components, a symbolic solver should implement a setup message (*SetupRequest*, *SetupResponse*). For a synthesis call, the symbolic solver receives a *SynProblem*, performs the synthesis task, and eventually responds with a *SynSolution*. At the time of writing, we do not require the output of synthesis tools to be model-checked. However, one can implement the interface to the model checking component to increase the trust in the output of new symbolic approaches.

4 The Neural Solver

The neural solver is at the heart of the portfolio solver and is developed jointly with `NeuroSynt`. We report on the methodology of the neural solver, including architecture, datasets, data generation, training, and evaluation. We clearly distinguish between previous work [57], introducing a neural approach for reactive synthesis, and improvements that are integrated into `NeuroSynt`, leading to the significantly increased performance on the SYNTCOMP benchmarks.

4.1 Data and Data Generation Improvement

We significantly improved the training data generation compared to previous work. While the basic algorithm is taken from [57], we scale the size of the training samples, tweak the data generation parameters to fit the larger samples, and combine multiple data generation strategies to lift previous limitations.

We aim for a dataset containing specifications (assumptions and guarantees) and circuits. Depending on the specification, the circuit is either a winning strategy for the system (realizable) or a winning strategy for the environment (unrealizable). For each sample, we use an additional token to show whether the system is realizable or unrealizable. The dataset is for supervised training, with the specification being the input and the circuit, along with the realizability token being the model's target.

In total, we combine three datasets and generation techniques. For the first two datasets, we utilize the generation method from [57] with 1) the minor tweak of having a variable number of inputs and outputs (up to five) in the circuit instead of exactly five (denoted `previous`), and 2) extensions to handle a larger number of patterns, larger patterns, and patterns with more atomic propositions (denoted `new`). The third dataset is a data augmentation method based on the result of `new` (denoted `augmented`).

Data Generation. We first report on the data generation algorithm for `previous` and `new`. The data generation has two major steps. In the first step, we mined LTL formula patterns that are common in research and practice. Considering formula patterns is a widespread idea, e.g., [24]. We collect patterns from 1075 (previously 346) benchmarks from the LTL synthesis track of SYNTCOMP 2022. We extract a list of 627 assumption patterns and 7948 guarantee patterns. An assumption restricts the environment, and a guarantee defines the implementation's behavior. To fit the model requirements, we filtered out LTL formulas with more than 15 inputs and 15 outputs (previously 5). Additionally, we filter out specifications with an abstract syntax tree (AST) size greater than 30 (formerly 25), resulting in 519 (formerly 157) assumption patterns and 6841 (previously 1942) guarantee patterns. In the second step, we constructed synthesis specifications by combining the mined patterns. For each specification, we alternate between sampling guarantees until the specification becomes unrealizable, and sampling assumptions until the specification becomes realizable. Whether we aim for a realizable or unrealizable specification, we either collect the last successfully mined specification (realizable) or the second-to-last mined specification (unrealizable). We aim for an even split between realizable and unrealizable specifications. To handle more atomic propositions while reducing patterns that do not share atomic propositions, we now favor atomic propositions present in the already constructed part of the specification with a bias of 4 when instantiating the patterns. We continue this process until we reach one of the following stopping criteria:

a) the specification has the maximal number of guarantees (10),
b) the specification has the maximal number of assumptions (3),
c) the synthesis tool timed out (120s timeout), or
d) no suitable assumption was found after 7 (formerly 5) attempts.

To ensure an even distribution of challenging instances, we filter AIGER circuits exceeding a maximum variable index of 60 and only allow a certain amount (20%) of circuits with the same number of AND gates.

Data Augmentation. We augment the dataset `new` as a third approach to artificially force larger properties for a share of the final dataset. For each specification

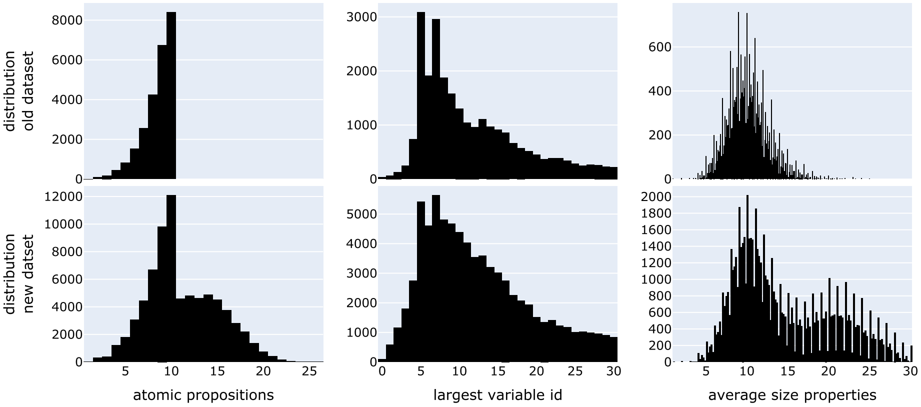


Fig. 4. Previous dataset[57], compared with the new final dataset. Comparing the number of atomic propositions in a sample, the largest variable id in the AIGER circuit, and the average size of properties.

in `new`, we combine multiple patterns into one property until we reach an AST size of 30. Having longer properties in the training dataset leads to better generalization to even larger properties. Compared to `new`, the augmented dataset has an average of 3 guarantees instead of 5.6, with an average size of 22.9 per guarantee instead of 12.3.

Final Dataset. All three resulting datasets are combined into a single dataset, consisting of 600 000 training samples and 75 000 validation and test samples. Figure 4, shows the key differences in features of the new final dataset compared to the previous dataset [57]. While the previous dataset used only up to 5 inputs and outputs in the specification, we now have up to 15 inputs and outputs, leading to up to 25 atomic propositions in a specification. We also have slightly more latches in the new dataset (1.23 instead of previously 1.16). Note that the same version and configuration of Strix [46] was used in both approaches. The most apparent distinction to previous datasets[57] is in the size of the properties, where we clearly see the effects of the data augmentation process.

4.2 Architecture & Training

Transformer Architecture. The core of the neural solver implemented in `NeuroSynt` is a Transformer neural network [63]. The vanilla Transformer architecture follows a basic encoder-decoder structure. The encoder constructs a hidden embedding z_i for each input embedding x_i of the input sequence $x = (x_0, \ldots, x_n)$ in parallel. An embedding is a mapping from plain input, for example, words or characters, to a high dimensional vector, for which learning algorithms and toolkits exist, e.g., word2vec [47]. Given the encoders output $z = (z_0, \ldots, z_k)$, the decoder generates a sequence of output embeddings

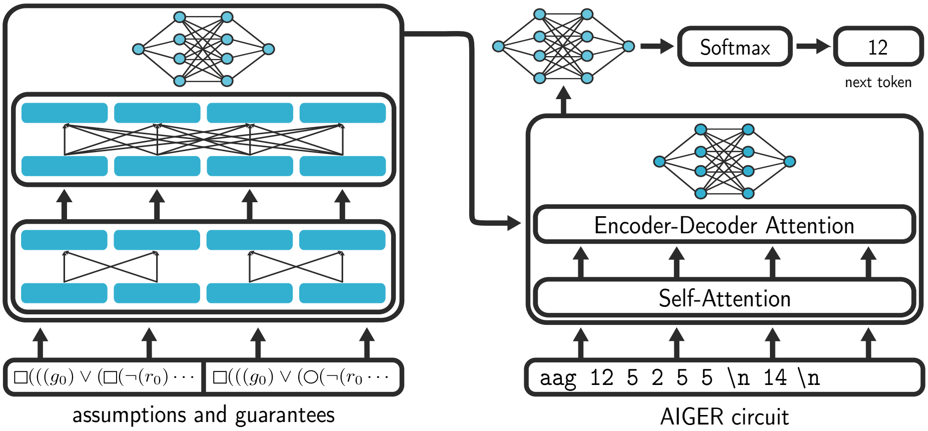


Fig. 5. Schematic view of the Hierarchical Transformer, with illustrated inputs/outputs of the reactive synthesis application. The encoder shows the hierarchical self-attention with separation into local and global layers. For simplicity, we show one local and global layer and only two assumptions and guarantees with two tokens each.

$y = (y_0, \ldots, y_m)$ autoregressively. Since the transformer architecture contains no recurrence nor convolution, we apply a tree positional encoding [60].

The main idea of the Transformer is a self-attention mechanism to compute a score for each pair of input elements, representing which positions in the sequence should be considered the most when computing the hidden embeddings. For each input embedding x_i, we compute 1) a query vector q_i, 2) a key vector k_i, and 3) a value vector v_i by multiplying x_i with weight matrices W_k, W_v, and W_q, which are learned during the training process. The embeddings can be calculated simultaneously using matrix operations [63]. Specifically, let Q, K, V be the matrices obtained by multiplying the input vector X consisting of all x_i with the weight matrices W_k, W_v, and W_q: $\mathit{Attention}(Q, K, V) = \mathit{softmax}(\frac{QK^T}{\sqrt{d_k}})V$, with d_k being the model's dimension. For details, we refer the interested reader to [63]. The Transformer variation used in this paper is a so-called hierarchical Transformer [44], separating the encoder self-attention into local and global layers. Local layers embed assumptions and guarantees individually and invariant against their order. Global layers calculate the self-attention across all assumptions and all guarantees. We show an illustration in Figure 5.

Model Hyperparameter & Training. We train our model on the 600 000 samples from our training dataset for 80 000 steps with early stopping and a batch size of 512. We show a plot of the accuracy per sequence in Figure 6. We train data parallel on two Nvidia A100 40GB from a Nvidia DGX A100 system, which takes approximately 10 hours. We use the Adam optimizer [40] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We use learning rate scheduling as proposed in [63] with 4000 warmup-steps. Our model consists of 4 local, 4 global, and 8 decoder layers, each having 4 heads. All feed-forward networks have 1024 nodes to which

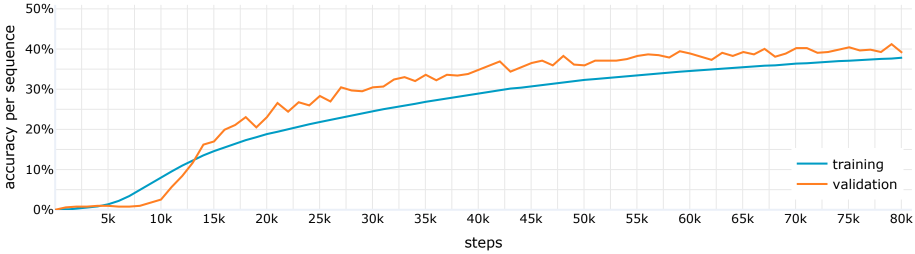


Fig. 6. Accuracy per sequence during training. Measured on training and validation data.

we apply a dropout of 0.2. Our model has a total size of 14 791 748 parameters. Input and output tokens have an embedding of size 256. The maximum input and target lengths are set according to the training data with at most 12 properties, a maximum AST size of 32 per property for the specification, and a maximum circuit length of 128 tokens after encoding.

We show that our model significantly improved compared to previous work [57] by reimplementing and adapting the previous model to evaluate the 2022 SYNTCOMP benchmarks. With 21.8%, the new model improved by 13 percentage points to 34.83%. We explore more details of the evaluation of the model in Section 5.1.

5 Experiments & Benchmarks

We split our experiments into two segments. In Section 5.1, we first perform generalization experiments on the integrated neural solver. The neural solver can generalize on its training distribution but also to more complex instances, longer specifications, and out-of-distribution instances, which we show using the datasets `test`, `large`, `timeouts`, and `syntcomp`.

Secondly, in Section 5.2, we evaluate the performance of the `NeuroSynt` framework on the SYNTCOMP 2022 benchmarks. To this end, we use `NeuroSynt` to compare the performance of the neural solver against multiple symbolic solvers and highlight efficiency gains and enhancements that arise from the combination of both methodologies. We show that the combined effort of neural and symbolic solvers leads to a performance gain that symbolic solvers alone could not achieve.

The evaluation is performed on a GPU cluster (1 Nvidia DGX A100 40GB, AMD EPYC 7F32 @ 1.8GHz base, 3.7GHz max, 8 cores + 8 SMT cores, 256GB RAM), on a CPU cluster (Intel Xeon E7-8867 v3 @ 2.50GHz, 64 cores + 64 HT cores, and 1536 GB RAM) and additionally did some early experiments on an Apple M1 Max (64GB memory, 10 cores, 32 neural cores).

Similar to different configurations of symbolic solvers, we have multiple models with slightly different performances. This paper reports the results of the model that performed best on the SYNTCOMP benchmark. Whenever we consider additional models, we mention that explicitly.

5.1 Generalization

We analyze the generalization capabilities of the model in the neural solver in four ways. Firstly on our `test` set, secondly on samples that are significantly larger than seen during training (`large`), thirdly samples that are arguably more difficult than training samples, and fourthly on out-of-distribution samples (`syntcomp`). Here, we consider instances that are not from the same data generation algorithm out-of-distribution samples. Results on these datasets are in Table 1.

Generalization on `test` and `large`. On the datasets `test` and `large`, additionally to measuring correct solutions (semantic accuracy, 84.2%), we collect how many solutions are syntactically identical to the solution from our data generation algorithm (38.6%). The large difference of 45.6 percentage points on our `test` dataset indicates that the neural solver generalizes to the semantics of the synthesis problem instead of learning the particularities of the data generator.

The dataset `large` consists of larger samples than seen during training. Samples in `large` have at least 10, on average 14.5 properties, and the largest property in each sample has an AST of 37.9 on average. In contrast, training samples have 5.3 properties on average, with the largest property having an AST of 22.2 on average.

For a more detailed analysis, we join datasets `test` and `large` and plot the share of correct solutions partitioned by the number of properties as well as the size of the largest property in each sample in Figure 7. The largest property seen during training is 30, and the largest number of properties per specification is 12. While we see a decrease in performance for larger samples, there is no clear drop after 12 or 30, respectively, which indicates generalization with the number of properties and the length of the properties. Note that results from larger sizes naturally have less significance as fewer samples per bucket exist. We refer to the full version [20] for more details on the count of samples in each displayed bucket.

Table 1. Performance of the neural model on different datasets.

	test	large	timeouts	syntcomp-small	syntcomp-large	syntcomp-full
syntactic accuracy	38.6%	10.2%	-	-	-	-
semantic accuracy	84.2%	57.7%	33%	65.8%	54.5%	34.83%

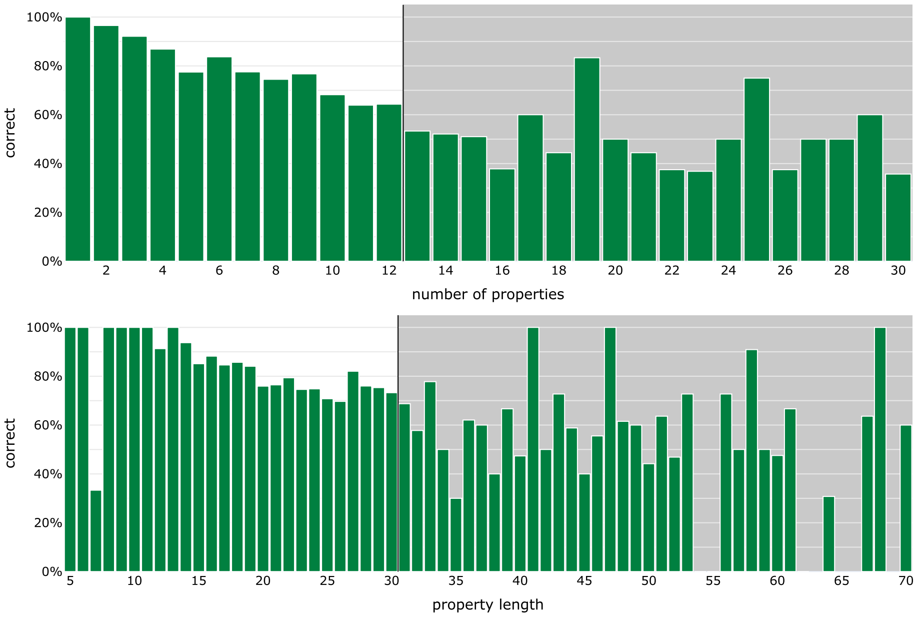


Fig. 7. Share of correct solutions on the joint dataset of `large` and `test` over the number of properties in a sample and the size of the largest property in each sample. A darker background indicates sizes larger than seen during training.

Generalization on `timeouts`*.* The dataset `timeouts` consists of samples on which Strix timed out after 120s during our data generation. Therefore, such samples can be seen as significantly harder, while not larger than samples in the training data. We achieve 33% correct solutions on this dataset, showing that our model generalizes from the training data to more challenging specifications and solutions that could not have not been solved by Strix during the data generation. This experiment prognosticates the potential of combining neural methods with symbolic methods.

Generalization on out-of-distribution dataset. While `large` and `timeouts` were generated with the same data generation approach as the training data, `syntcomp-full` consists of all 1075 real-world specifications collected in the SYNTCOMP benchmark, on which the neural solver achieves 34.83% accuracy (see Table 1). `syntcomp-large` contains all such samples that are in the size of our evaluation constraints (i.e. max 30 properties, max AST size of 70 per property, 54.5% accuracy). `syntcomp-small` contains only such samples that are in the training data size (i.e., max 12 properties, max AST size of 30 per property, 65.8% accuracy). We see a remarkable generalization to out-of-distribution samples with an accuracy of 64.8% on `syntcomp-small`. We additionally observe generalization on specification size that we also see on the `large` dataset.

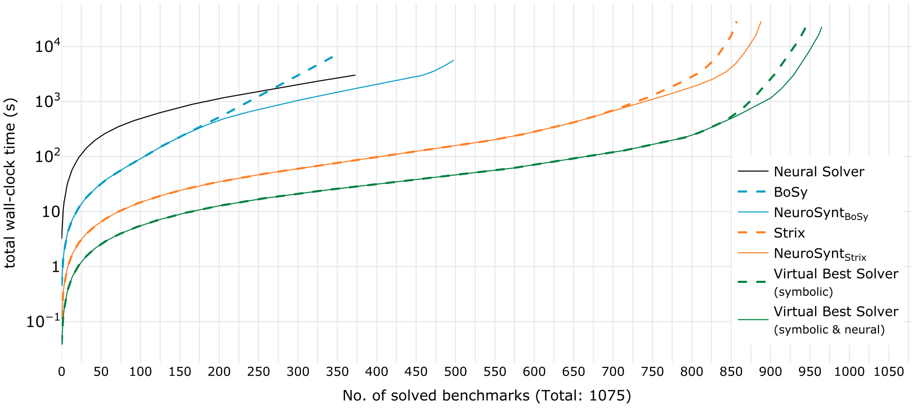


Fig. 8. A cactus plot showing the number of solved samples vs. accumulated wall-clock time. Each sample per solver is a dot on the respective line. The lower and further right a line, the better the solver. We compare the neural solver, Strix and BoSy alone, NeuroSynt$_{\mathrm{BoSy}}$ which couples BoSy and the neural solver and NeuroSynt$_{\mathrm{Strix}}$ which couples the neural solver and Strix. Further, we virtually combine the results of all tools and all configurations from the SYNTCOMP to a virtual best solver and compare that with the evaluations of multiple neural models.

5.2 Comparisons and Advantages of Combination

We demonstrate the advantage of NeuroSynt by comparing the neural solver to the performance of multiple symbolic solvers: Strix [46], the current state-of-the-art, BoSy [27], a bounded synthesis method, and additionally rely on the results of SYNTCOMP 2022 (ltlsynt [55], Otus [1], sdf [38]). Whenever we write SYNTCOMP in this paper, we refer to the 2022 iteration.

We initiate the evaluation by comparing the neural solver and NeuroSynt to the specified symbolic tools, illustrating the number of problems that can be solved within a specific time frame. Then we dive into details on instances that could only be solved by NeuroSynt and no other symbolic solver (*novel solves*), show details on the *time-to-solution* differences between the solvers, and lastly, look at *circuit sizes* of their respective solutions.

In Figure 8, we display the performance of the neural solver, the performance of several symbolic tools and the performance of NeuroSynt that unites the neural solver with a symbolic solver. We additionally show a Virtual Best Solver (VBS) of all SYNTCOMP 2022 results without and including the neural solver. We further report what the previously published neural reactive synthesis approach [57] would have achieved if it had been integrated into the portfolio solver. With 374 solved instances, the neural solver alone can already solve more samples than BoSy (347) with 120s timeout on the CPU cluster. Its true advantage becomes evident when combining the neural solver with symbolic solvers. NeuroSynt$_{\mathrm{BoSy}}$ solves 152 (previous: 59) samples more than BoSy alone, which is 20.8% of the samples that BoSy could not solve. Similarly, $\texttt{NeuroSynt}_{\text{Strix}}$ solves 31 (previous: 2) samples more than Strix alone ($1h$ timeout on the CPU cluster), which is 14.2% of the samples that Strix could not solve in $1h$. To show the full potential of NeuroSynt, we combined all results from the SYNTCOMP and our experiments with BoSy and Strix. All symbolic solvers combined were able to solve 945 instances of the total of 1075. Adding the neural solver of NeuroSynt to the virtual best solver, we solve an additional 20 (previous: 0) samples exclusively that no other tool tested did solve (*novel solves*). This is 15.4% of the samples that none of the symbolic tools could solve in $1h$. No other tool in SYNTCOMP 2022 except the state-of-the-art Strix, solved more samples that no other tool could solve. We refer to the full version [20] for exact numbers. This signifies that even for specifications that pose computational challenges to symbolic synthesis tools, there exist patterns that a neural network can recognize and exploit post-training.

Novel Solves. Of the 20 novel solves, 6 instances are parameterized versions of full arbiters with 3 processes. This version of the full arbiter is unrealizable as the specification additionally enforces two grants to hold at the same time step (step 11 to step 16 respectively). These are the largest parameterizations of this problem class in the SYNTCOMP dataset. Similarly, 11 instances are full arbiter with 3 processes, where two grants are enforced simultaneously (step 6 to 16, respectively). These parameterizations are also the largest parameterizations of this problem class in the SYNTCOMP dataset. One instance is a full arbiter with 6 processes and the requirement of two grants to hold at *any* time step. Finally, we have one instance of a load balancer with 6 grants and the additional unrealizable requirement of two grants at time step 5. This is also the largest parameterization of this problem class in the SYNTCOMP dataset. For examples of the novel solves, we refer the reader to the artifact or the full version [20]

Time To Solution. For experiments with NeuroSynt, we record the wall-clock time of the neural solver, the symbolic solver, and the model checker. The neural solver (including model checking) is fastest on the GPU cluster, with $8.6s$ and a standard deviation of only $3.3s$. The time for model-checking using NuXmv is almost negligible, with $0.35s$ on average per sample. The low standard deviation highlights the advantage of the neural solver, as the time does not depend on the complexity of the specification. Strix with a timeout of $1h$ on the CPU cluster takes $33.4s$ on average, with a standard deviation of $185.3s$. We find that the neural solver can also be run on CPU-only hardware (CPU cluster) with an average of $79.4s$ and on hybrid desktop hardware such as the Apple M1 Max with an average of $17.8s$. For an extensive overview over the experiments with different timeouts, we refer the reader to the full version [20].

Circuit Sizes. We find that on instances where the neural solver and the symbolic solver both found a solution, the solution by the neural solver is often smaller than the symbolic solver's. This holds for BoSy and Strix, but also for all other tools in SYNTCOMP (on the realizable fraction). On samples solved by Strix

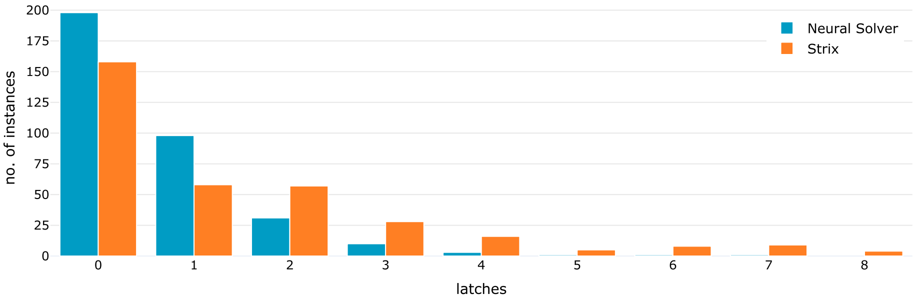


Fig. 9. No. of latches per instance. On instances that the neural solver and Strix commonly solved

and the neural solver, the solutions by the neural solver have 54.9% fewer latches than those by Strix. In Figure 9, we show the distribution of latches for this comparison. For more details, we refer the reader to the full version [20].

6 Conclusion

We introduced `NeuroSynt`, a neuro-symbolic portfolio solver for reactive synthesis. At the core of the portfolio solver lies an integrated neural solver that computes candidate implementations, which are automatically checked by model-checking tools. We reported on the neural solver's methodology and training and the API framework's implementation to isolate components. The open-source implementation of `NeuroSynt` provides an interface in which new neural and symbolic approaches alike can be seamlessly integrated.

Our experiments on the generalization capabilities of the Transformer show the ability to generalize to larger specifications, more difficult specifications, and out-of-distribution specifications. The relatively small size of the underlying Transformer neural network suggests that the overall performance of neural solvers can be further increased.

We evaluated the overall performance of `NeuroSynt`, enhancing the state-of-the-art in reactive synthesis with the integrated neural solver contributing novel solves in the SYNTCOMP 2022 benchmark. With the almost constant evaluation time of the neural solver, the portfolio solver is often faster than previous approaches. Furthermore, the integrated neural solver yields smaller implementations than state-of-the-art symbolic tools, including Strix and BoSy.

7 Data Availability Statement

`NeuroSynt` is published open-source on GitHub (https://github.com/reactive-systems/neurosynt). All data, models, and experiments supporting this paper's results are publicly available. A digital artifact is available at (https://doi.org/10.5281/zenodo.10046523).

References

1. Abraham, R.: Symbolic LTL reactive synthesis. Master's thesis, University of Twente, Enschede (Jul 2021)
2. Alet, F., Lopez-Contreras, J., Koppel, J., Nye, M., Solar-Lezama, A., Lozano-Perez, T., Kaelbling, L., Tenenbaum, J.: A large-scale benchmark for few-shot program induction and synthesis. In: International Conference on Machine Learning. pp. 175–186. PMLR (2021)
3. Alon, Y., David, C.: Using graph neural networks for program termination. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 910–921. ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA (Nov 2022). https://doi.org/10.1145/3540250.3549095
4. Balunovic, M., Bielik, P., Vechev, M.T.: Learning to solve SMT formulas. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 10338–10349 (2018)
5. Bansal, K., Loos, S.M., Rabe, M.N., Szegedy, C., Wilcox, S.: HOList: An Environment for Machine Learning of Higher-Order Theorem Proving. In: Proceedings of the 36th International Conference on Machine Learning. pp. 454–463. PMLR (May 2019). https://doi.org/10.48550/arXiv.1904.03241
6. Bansal, K., Szegedy, C., Rabe, M.N., Loos, S.M., Toman, V.: Learning to reason in large theories without imitation (Jun 2020). https://doi.org/10.48550/arXiv.1905.10501
7. Ben-Kiki, O., Evans, C., döt Net, I.: YAML Ain't Markup Language (YAML™) revision 1.2.2. Tech. rep. (Oct 2021)
8. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (October 2007, 2007)
9. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (July 2011, 2011)
10. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society **138**, 295–311 (1969). https://doi.org/10.2307/1994916
11. Cadilhac, M., Pérez, G.A.: Acacia-bonsai: a modern implementation of downset-based LTL realizability. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 192–207. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_14
12. 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.) Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017. pp. 252–263. ACM (2017). https://doi.org/10.1145/3055399.3055409
13. Cameron, C., Chen, R., Hartford, J., Leyton-Brown, K.: Predicting propositional satisfiability via end-to-end learning. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 3324–3331 (2020)

14. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
15. Church, A.: Logic, arithmetic, and automata (1962)
16. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. In: Summaries of the Summer Institute of Symbolic Logic. vol. 1, pp. 3–50. Cornell University, Ithaca, NY (1957)
17. 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: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification. pp. 359–364. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
18. Clymo, J., Manukian, H., Fijalkow, N., Gascón, A., Paige, B.: Data generation for neural programming by example. In: Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics. pp. 3450–3459. PMLR (2020)
19. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: interactively translating unstructured natural language to temporal logics with large language models. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 383–396. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_18
20. Cosler, M., Hahn, C., Omar, A., Schmitt, F.: NeuroSynt: A Neuro-symbolic Portfolio Solver for Reactive Synthesis (Full version) (Jan 2024). https://doi.org/10.48550/arXiv.2401.12131
21. Cosler, M., Schmitt, F., Hahn, C., Finkbeiner, B.: Iterative circuit repair against formal specifications. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023 (2023)
22. Drori, I., Zhang, S., Shuttleworth, R., Tang, L., Lu, A., Ke, E., Liu, K., Chen, L., Tran, S., Cheng, N., Wang, R., Singh, N., Patti, T.L., Lynch, J., Shporer, A., Verma, N., Wu, E., Strang, G.: A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. Proceedings of the National Academy of Sciences **119**(32), e2123433119 (Aug 2022). https://doi.org/10.1073/pnas.2123433119
23. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: what's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9
24. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering. pp. 411–420. ICSE '99, Association for Computing Machinery, New York, NY, USA (May 1999). https://doi.org/10.1145/302405.302672
25. Ehlers, R.: Unbeast: symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6605, pp. 272–275. Springer (2011). https://doi.org/10.1007/978-3-642-19835-9_25

26. Ellis, K., Wong, L., Nye, M., Sablé-Meyer, M., Cary, L., Anaya Pozo, L., Hewitt, L., Solar-Lezama, A., Tenenbaum, J.B.: DreamCoder: growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **381**(2251), 20220050 (Jun 2023). https://doi.org/10.1098/rsta.2022.0050
27. Faymonville, P., Finkbeiner, B., Tentrup, L.: BoSy: an experimentation framework for bounded synthesis. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 325–332. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_17
28. Fijalkow, N., Lagarde, G., Matricon, T., Ellis, K., Ohlmann, P., Potta, A.N.: Scaling neural program synthesis with distribution-based search. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 6623–6630 (Jun 2022). https://doi.org/10.1609/aaai.v36i6.20616
29. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesis from hyperproperties. Acta Informatica **57**(1-2), 137–163 (2020). https://doi.org/10.1007/s00236-019-00358-2
30. Finkbeiner, B., Klein, F.: Bounded cycle synthesis. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 118–135. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_7
31. First, E., Rabe, M.N., Ringer, T., Brun, Y.: Baldur: Whole-Proof Generation and Repair with Large Language Models (Mar 2023). https://doi.org/10.48550/arXiv.2303.04910
32. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: Roychoudhury, A., Cadar, C., Kim, M. (eds.) Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022. pp. 633–645. ACM, Singapore Singapore (Nov 2022). https://doi.org/10.1145/3540250.3549120
33. Hahn, C., Schmitt, F., Kreber, J.U., Rabe, M.N., Finkbeiner, B.: Teaching temporal logics to neural networks. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021 (2021)
34. Huang, D., Dhariwal, P., Song, D., Sutskever, I.: GamePad: a learning environment for theorem proving. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019 (2019)
35. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Piskac, R., Dimitrova, R. (eds.) Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. EPTCS, vol. 229, pp. 112–132 (2016). https://doi.org/10.4204/EPTCS.229.10
36. Jacobs, S., Perez, G.A., Abraham, R., Bruyere, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, K., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (SYNTCOMP): 2018-2021 (Jun 2022). https://doi.org/10.48550/arXiv.2206.00251
37. Jiang, A.Q., Welleck, S., Zhou, J.P., Lacroix, T., Liu, J., Li, W., Jamnik, M., Lample, G., Wu, Y.: Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023 (2023)

38. Khalimov, A.: Game-based bounded synthesis via BDDs
39. Khalimov, A., Jacobs, S., Bloem, R.: PARTY parameterized synthesis of token rings. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 928–933. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_66
40. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015). https://doi.org/10.48550/arXiv.1412.6980
41. Kreber, J.U., Hahn, C.: Generating symbolic reasoning problems with transformer GANs (May 2023). https://doi.org/10.48550/arXiv.2110.10054
42. Křetínský, J., Meggendorfer, T., Prokop, M., Rieder, S.: Guessing winning policies in LTL synthesis by semantic learning. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 390–414. Lecture Notes in Computer Science, Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37706-8_20
43. Lample, G., Charton, F.: Deep learning for symbolic mathematics. In: 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020 (2020)
44. Li, W., Yu, L., Wu, Y., Paulson, L.C.: IsarStep: a benchmark for high-level mathematical reasoning. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021 (2021)
45. Loos, S.M., Irving, G., Szegedy, C., Kaliszyk, C.: Deep network guided proof search. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 85–105. EasyChair (2017). https://doi.org/10.29007/8mwc
46. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 578–586. Lecture Notes in Computer Science, Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_31
47. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. In: Bengio, Y., LeCun, Y. (eds.) 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings (2013)
48. Paliwal, A., Loos, S.M., Rabe, M.N., Bansal, K., Szegedy, C.: Graph representations for higher-order logic and theorem proving. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020. pp. 2967–2974. AAAI Press (2020). https://doi.org/10.1609/aaai.v34i03.5689
49. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can large language models reason about program invariants? In: International Conference on Machine Learning. pp. 27496–27520. PMLR (2023)
50. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3855, pp. 364–380. Springer (2006). https://doi.org/10.1007/11609773_24
51. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57 (Oct 1977). https://doi.org/10.1109/SFCS.1977.32

52. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989. pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293
53. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II. pp. 746–757. IEEE Computer Society (1990). https://doi.org/10.1109/FSCS.1990.89597
54. Rabin, M.: Automata on infinite objects and church's problem. CBMS Regional Conference Series in Mathematics, vol. 13. American Mathematical Society, Providence, Rhode Island (1972). https://doi.org/10.1090/cbms/013
55. Renkin, F., Schlehuber, P., Duret-Lutz, A., Pommellet, A.: Improvements to ltlsynt (2022). https://doi.org/10.48550/arXiv.2201.05376
56. Ryan, G., Wong, J., Yao, J., Gu, R., Jana, S.: CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In: International Conference on Learning Representations (Sep 2019)
57. Schmitt, F., Hahn, C., Rabe, M.N., Finkbeiner, B.: Neural circuit synthesis from specification patterns. In: Advances in Neural Information Processing Systems. vol. 34, pp. 15408–15420. Curran Associates, Inc. (2021)
58. Selsam, D., Bjørner, N.S.: Guiding high-performance SAT solvers with unsat-core predictions. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 336–353. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_24
59. Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019 (2019)
60. Shiv, V.L., Quirk, C.: Novel positional encodings to enable tree-based transformers. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32. pp. 12058–12068. Vancouver, BC, Canada (Dec 2019)
61. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Advances in Neural Information Processing Systems. vol. 31 (2018)
62. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. Journal of the ACM (JACM) **32**(3), 733–749 (1985). https://doi.org/10.1145/3828.3837
63. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30. pp. 5998–6008. Long Beach, CA, USA (Dec 2017)
64. Wu, Y., Jiang, A.Q., Li, W., Rabe, M., Staats, C., Jamnik, M., Szegedy, C.: Autoformalization with large language models. Advances in Neural Information Processing Systems **35**, 32353–32368 (2022)

Testing and Verification

HaliVer: Deductive Verification and Scheduling Languages Join Forces

Lars B. van den Haak[1](✉), Anton Wijs[1], Marieke Huisman[2], and Mark van den Brand[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
{l.b.v.d.haak,a.j.wijs,m.g.j.v.d.brand}@tue.nl
[2] University of Twente, Enschede, The Netherlands
m.huisman@utwente.nl

Abstract. The HaliVer tool integrates deductive verification into the popular scheduling language Halide, used for image processing pipelines and array computations. HaliVer uses VerCors, a separation logic-based verifier, to verify the correctness of (1) the Halide algorithms and (2) the optimised parallel code produced by Halide when an optimisation schedule is applied to an algorithm. This allows proving complex, optimised code correct while reducing the effort to provide the required verification annotations. For both approaches, the same specification is used. We evaluated the tool on several optimised programs generated from characteristic Halide algorithms, using all but one of the essential scheduling directives available in Halide. Without annotation effort, HaliVer proves memory safety in almost all programs. With annotations HaliVer, additionally, proves functional correctness properties. We show that the approach is viable and reduces the manual annotation effort by an order of magnitude.

Keywords: Program correctness · Deductive verification · Scheduling language.

1 Introduction

To meet the continuously growing demands on software performance, parallelism is increasingly often needed [13]. However, introducing parallelism tends to increase the risk of introducing errors, as the interactions between parallel computations can be hard to predict. Moreover, a plethora of optimisation techniques exists [10], so identifying when an optimisation can be applied safely, without breaking correctness, can be very challenging. Also, applying optimisations tends to make a program more complex, making it harder to reason about.

To address this, on the one hand, various domain-specific languages (DSLs) have been proposed that separate the *algorithm* (*what* it does) from the parallelisation *schedule* (*how* it does it). These are called *scheduling languages* [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.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 71–89, 2024.
https://doi.org/10.1007/978-3-031-57256-2_4

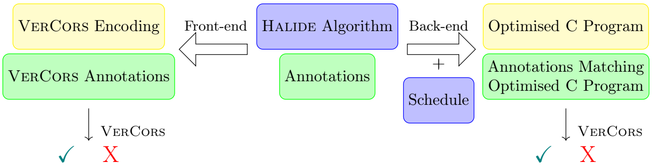


Fig. 1. High level overview of our approach.

6–8, 22, 23, 28]. Given an algorithm and a schedule, a compiler generates an optimised parallel program. This approach crucially depends on the schedule not introducing any errors in the functionality, which is not always obvious.

On the other hand, *deductive program verification* [9] has been successfully applied to verify the functionality of parallel programs [4]. This requires that the intended functionality is formalised as a contract, for instance using *permission-based separation logic* [1,5]. A major hurdle, preventing this technique from being adopted at a large scale, is that if a program becomes more complicated, the required annotations rapidly grow in size and complexity [25, 26].

In this paper, we combine the best of both worlds. We propose the HaliVer tool, which focusses on Halide [22,23], a scheduling language for portable image computations and array processing. It has been widely adopted in industry, for instance to produce parts of Adobe Photoshop and to implement the YouTube video-ingestion pipeline. For verification, we use the VerCors program verifier [4]. In this paper we define two verification approaches (1) *front-end* and (2) *back-end*, as seen in Figure 1. Our approaches verify that the program adheres to the *same* functional specification. This specification is detailed by annotating the algorithmic part of a Halide program, thereby keeping the annotations focussed on the functionality, and therefore relatively straightforward. With the *front-end* verification approach we verify the correctness of the algorithmic part of a Halide program. HaliVer transforms the algorithm and the annotations to an annotated VerCors program. With *back-end* verification approach we verify the C code that the Halide compiler generates, given a Halide algorithm and a schedule. HaliVer transforms the given annotations to match the generated code. Furthermore, where possible, HaliVer generates annotations, such as permission specifications, to relieve the user from having to manually write these. This contributes to making the annotation process straightforward.

In this way, HaliVer allows the user to succinctly specify the intended functionality of optimised, parallel code, and it checks that the resulting program indeed has the desired functionality. A major advantage of our approach is that it is flexible to use in a setting where multiple compiler passes are made. Also, it can be easily extended if a new compiler pass or schedule optimisation is added. An alternative would be to prove correctness of the compiler, but this would require a large amount of initial work and additionally for each change to the compiler.

Concretely, this paper provides the following contributions:

Listing 1. Halide blur example with annotations added to verify the code.

```
  requires inp.x.min == blur_y.x.min ∧ inp.x.max == blur_y.x.max+2 ∧ inp.y.min ==
       blur_y.y.min ∧ inp.y.max == blur_y.y.max+2;
  ensures ∀ x, y . blur_y.x.min≤x<blur_y.x.max ∧ blur_y.y.min≤y<blur_y.y.max ⇒
       blur_y(x,y) == ((inp(x,y)+inp(x+1,y)+inp(x+2,y))/3 + (inp(x,y+1) +inp(x+1,y+1)+
       inp(x+2,y+1))/3 + (inp(x,y+2)+inp(x+1,y+2) + inp(x+2,y+2))/3)/3);
void blur(Buffer<2,int> inp, Func &blur_y){
  Func blur_x; Var x, y;
  blur_x(x,y) = (inp(x,y) + inp(x+1,y) + inp(x+2,y))/3
  blur_x.ensures(blur_x(x,y) == (inp(x,y) + inp(x+1,y) + inp(x+2,y))/3);
  blur_y(x,y) = (blur_x(x,y) + blur_x(x,y+1) + blur_x(x,y+2))/3;
  blur_y.ensures(blur_y(x,y) == ((inp(x,y)+inp(x+1,y)+inp(x+2,y))/3 + (inp(x,y+1)+inp
       (x+1,y+1) + inp(x+2,y+1))/3 + (inp(x,y+2)+inp(x+1,y+2)+inp(x+2,y+2))/3)/3;}
```

- An annotation language to describe the functionality of Halide algorithms, which is integrated into the Halide algorithm language;
- Tool support for the front-end verification approach of Halide algorithms;
- Tool support for the back-end verification approach, which can verify programs generated by the Halide compiler from an algorithm and a schedule;
- Evaluation of the HaliVer tool on Halide examples using all but one of the essential scheduling directives available in various combinations. We evaluated the tool on 23 different optimised programs, generated from eight characteristic Halide algorithms, to prove memory safety with no annotation effort. For 21 cases, HaliVer proves safety, for the remaining two cases we discuss the limitations. For 20 programs, based on five algorithms, we also add annotations for functional correctness properties. For 19 of these programs HaliVer proves correctness, for the remaining one we run into a similar limitation.

The remainder of this paper is organised as follows. Section 2 gives brief background information on Halide and VerCors. Section 3 introduces Halide annotations, and describes how HaliVer supports the verification of an algorithm and an optimised program. The approach is illustrated on characteristic examples. Section 4 evaluates the HaliVer tool, and Sections 5 and 6 address related work, conclusions and future work.

2 Background

Halide. Halide is a DSL embedded in C++, targeting image processing pipelines and array computations [22, 23].[3] Halide separates the *algorithm*, defining what you want to calculate, from the *schedule*, defining how the calculation should be performed. Typically, when optimising code for a specific architecture, the code becomes much more complex and loses portability. By separating the schedule, the code expressing the functionality is not altered.

Listing 1 presents the Halide algorithm for a box filter, or blur function. The reader can ignore the **requires** and **ensures** annotations for now. Images

[3] A Halide tutorial can be found here: https://halide-lang.org/tutorials/.

Listing 2. A reduction to count the positive numbers of each row in matrix `inp`.

```
1  void cnt(Buffer<2,int> inp, Func count) {
2    Var x; RDom r(0,10);
3    count(x) = 0;
4    count.ensures(count(x) == 0);
5    count(x) = select(inp(x, r) > 0, count(x)+1, count(x))
6    count.invariant(0≤count(x)≤r);
7    count.ensures(0≤count(x)≤10);}
```

are represented as pure (side-effect free) functions that point-wise map coordinates to values. A blur function defines how every pixel, referred to by its two-dimensional coordinates, should be updated. In the example, the coordinates are represented by the variables `x` and `y`. Halide uses a functional style, allowing algorithms to be compact and loop-free. Halide functions are denoted by the keyword `Func`. In the example, the input image is stored in a two-dimensional integer buffer `inp`, and the output is given by defining the function `blur_y`, a reference to which is a parameter of `blur`. A *pipeline* of function calls is defined: the function `blur_x` is applied on the input image (line 5). The output of that function is used to compute the final image with the function `blur_y` (line 7). With `inp.x.min` and `inp.x.max` we refer to the minimum and maximum value of the dimension `inp.x`, respectively.

A function may involve *update definitions*, which (partially) update the value of a function. A *reduction domain* is a way to apply an update a finite number of times and is typically used to express sums or histograms in Halide. A function is called a *reduction* when such a domain is used, and an initialisation and an update definition are given. Listing 2 presents a reduction example. For now, ignore the **ensures** and **invariant** lines. The reduction domain (`RDom`) `r` ranges from `0` to `9`, i.e. it consists of `10` values. The initial value of the `count` function is defined at line 3, and line 5 is executed once for every value in `r`. The statement `select(a,b,c)` returns `b` if `a` evaluates to **true**, `c` otherwise. For a given matrix of integers `inp`, `cnt` counts the number of non-zeros at the first ten positions of each row in `inp`.

A Halide *schedule* is given in Listing 5 and further explained in Section 3.3.

VerCors. VerCors[4] [4] is a deductive verifier to verify the functional correctness of, possibly concurrent, software. Its specification language uses permission-based separation logic [5], a combination of first-order logic and read/write permissions. The latter are used for concurrency-related verification, to express which data can be accessed by a thread at which moment. Programs written in a number of languages, such as Java and C, can be verified. VerCors also has its own language, Pvl. VerCors's verification engine relies on Viper [16], which applies symbolic execution to analyse programs with persistent mutable state.

[4] An online tutorial can be found at https://vercors.ewi.utwente.nl/wiki/.

Intended functional behaviour can be specified by means of pre- and postconditions, indicated by the keywords `requires` and `ensures`, respectively. The statement `context P` is an abbreviation for `requires P; ensures P`. Loop invariants and assertions can be added to the code to help VerCors in proving the pre- and postconditions. We refer to the pre- and postconditions, loop invariants and assertions together as the *annotations* of a code fragment. A permission `Perm(x, f)` gives permission to memory location `x`, where `f` is a fractional, with `1\1` indicating a write and anything between `0\1` and `1\1` a read. For a statement s, we have the Hoare triple $\{P\}s\{Q\}$. This indicates that if P holds in the *pre-state* then after s, Q holds in the *post-state*. A *pure* function is without side-effects, thus can be used in annotations. It has the keyword `pure` in the header, and its body is a single expression. Annotations and pure function definitions in C files are given in special comments, like `//@` or `/*@...@*/` for multi-line comments. (See Listing 6 for examples.)

VerCors can prove termination of recursive functions. Whenever the clause `decreases r` is added to a function contract, VerCors will try to prove that the function terminates, by showing that all recursive calls will strictly decrease the value of `r` while `r` has a lower bound.

3 Verification of Scheduling Languages with HaliVer

HaliVer works directly on a Halide program and its intermediate representations, adding and transforming annotations where necessary. The tool is embedded in the Halide compiler. From a user's point of view, the general approach is as follows, using the front-end and back-end approach as in Figure 1.

1. **Write a Halide algorithm and add annotations**. Annotations are the functional specification of the Halide algorithm. Since a user can write an incorrect Halide algorithm, its correctness is ideally checked against a user-supplied specification.
2. **The front-end approach produces a Pvl encoding.** This encoding contains the algorithm and the specified annotations.
3. **VerCors verifies the encoding.** If verification succeeds, we know that the front-end algorithm conforms to the functional specification. Otherwise, the verification fails; VerCors produces a counterexample and we return to step 1.
4. **Write a Halide schedule**.
5. **The back-end approach produces an annotated C file.** The tool automatically generates permission annotations. These allow us to prove data-race freedom and the absence of out-of-bound errors. The tool transforms the annotations and generates additional annotations to match the scheduled back-end code. This is highly non-trivial, as each for-loop requires precise annotations to guide VerCors in the verification. However, it is ensured that the same property is verified.
6. **VerCors verifies the back-end C file.** If the verification fails, the lines of C code that caused the failure are given, which can be traced back to the Halide algorithm. The cause of a verification failure may be that

- The Halide compiler produced incorrect code w.r.t. the specifications.
- More *auxiliary* annotations from step 1 are needed to guide VerCors.
- A limitation has been encounter of the tools HaliVer relies on, e.g., VerCors or the underlying SMT solver.

In the remainder of this section we explain how to write annotations, and address front-end and back-end verification approaches. We also discuss the soundness and current limitations of the technique.

3.1 Halide Annotations

HaliVer makes it possible to add annotations when writing a Halide algorithm. Intuitively, these annotations are added as a Hoare triple. We consider three types of annotation: *pipeline*, *intermediate* and *reduction invariant* annotations.

In Listing 1 annotations have been added. The lines 1–2 are *pipeline annotations*: they specify the pre- and postconditions of the whole function and can only contain references to input buffers or output functions. Note that the results are stored directly in the `blur_y` function. Line 1 specifies how the input and output bounds should be related. Line 2 indicates what the output values are. One can add *intermediate annotations* after any (update) function call to specify state predicates for particular locations in the pipeline. Examples are the `blur_x.`**ensures** and `blur_y.`**ensures** state predicates of Listing 1 (lines 6 and 8).

Halide functions map coordinates to values pointwise. To achieve a one-to-one relationship between function and annotations, the intermediate annotations for a function should also specify how coordinates relate to values pointwise. However, input buffers can be used freely with any point. For example, `blur_x.`**ensures**`(blur_x(x,y)`$\geq$`inp(x+1,y))` is valid, but `blur_x.`**ensures**`(blur_x(x+1,y)`$\geq$`0)` is not, because the latter refers to `blur_x(x+1,y)` as opposed to `blur_x(x,y)`. HaliVer requires this because each point of the function may be computed in parallel in the back-end, so it must be possible to reason about the points individually.

For ease of annotation, HaliVer automatically generates a pipeline postcondition. This postcondition is derived from the intermediate annotation of the last pipeline function in the algorithm. For Listing 1, HaliVer can generate line 2, which is included here for completeness, based on line 8.

To prove that a *reduction* is correct, *reduction invariant* annotations must be provided for reduction domains. In Listing 2, an example is given of a reduction (line 5) together with its reduction invariant (line 6) and post-state predicate (line 7). Intuitively, a reduction invariant is similar to a loop invariant. First, it must hold before the reduction starts. In our example this means that `count(x)` has the value `0`, which is ensured by the previous definition of `count` (line 4). Second, it must be preserved by each step of the reduction. In our example, `count` is bounded by the reduction variable. Finally, after each reduction variable has reached its maximum value, the reduction invariant should imply the post-state

Listing 3. The front-end Pvl code for the blur example (Listing 1). We omitted the decreases clauses for brevity.

```
 1  pure int inp(int x, int y);
 2  pure int inp_x_min(); pure int inp_x_max(); pure int inp_y_min(); pure int inp_y_max();
 3  pure int blur_y_x_min(); pure int blur_y_x_max();
 4  pure int blur_y_y_min(); pure int blur_y_y_max();
 5
 6   ensures \result ≡ (inp(x, y) + inp(x+1, y) + inp(x+2, y))/3;
 7  pure int blur_x(int x, int y) = (inp(x, y) + inp(x+1, y) + inp(x+2, y))/3;
 8
 9   ensures \result ≡ ((inp(x, y) + inp(x+1, y) + inp(x+2, y))/3
10     + (inp(x, y+1) + inp(x+1, y+1) + inp(x+2, y+1))/3
11     + (inp(x,y+2) + inp(x+1,y+2) + inp(x+2,y+2))/3)/3;
12  pure int blur_y(int x, int y) = (blur_x(x, y) + blur_x(x, y+1) + blur_x(x, y+2))/3;
13
14   requires inp_x_min() ≡ blur_y_x_min() ∧ inp_x_max() ≡ blur_y_x_max()+2
15     ∧ inp_y_min() ≡ blur_y_y_min() ∧ inp_y_max() ≡ blur_y_y_max()+2;
16   ensures (∀ x, y; blur_y_x_min()≤x ∧ x<blur_y_x_max() ∧ blur_y_y_min()≤y ∧ y<
         blur_y_y_max();
17    blur_y(x,y) ≡ ((inp(x, y) + inp(x+1, y) + inp(x+2, y))/3
18     + (inp(x, y+1) + inp(x+1, y+1) + inp(x+2, y+1))/3
19     + (inp(x, y+2) + inp(x+1, y+2) + inp(x+2, y+2))/3)/3);
20  void pipeline() { }
```

predicate of the function. For the example, note that the invariant implies the post-state predicate when r has reached the value 10. The actual used value goes to 9, and r==10 indicates that the reduction is done.

3.2 Front-end Verification Approach

For verifying the algorithm part of a Halide program, an annotated Halide algorithm is encoded into annotated Pvl code. Listings 3 and 4 show how HaliVer translates the examples of Listings 1 and 2, respectively. Input buffers are translated into abstract functions to verify the pipeline w.r.t. arbitrary input. The bounds of input buffers and functions are modelled via functions that are abstract if the bound is unknown or otherwise return a concrete value. For example, the inp buffer of the blur example is translated to a function inp in Listing 3 (line 1), with its bounds represented by the pure functions on line 2.

Update-free Halide functions are translated directly into pure Pvl functions, and post-state predicates are translated into postconditions of these functions. In the example, blur_x and blur_y are translated to the functions on lines 6–7 and 9–12 of Listing 3, respectively, and the ensures lines express the postconditions of those functions, using \result to refer to the expected result.

The pre- and postconditions of a Halide algorithm are translated into a Pvl lemma to be checked by VerCors. In the example, lines 14–19 of Listing 3 address the pre- and postconditions on lines 1–2 of Listing 1. On line 20, a method called pipeline is given, which represents the Halide pipeline.

For an update definition, references to itself are replaced by references to the previous definition, thus the output of one definition is the input of the next.

Listing 4. The front-end PVL code for the reduction example of Listing 2.

```
1    decreases;
2  pure int inp(int x, int y);
3    decreases;
4  pure int inp_x_min(); pure int inp_x_max(); pure int inp_y_min(); pure int inp_y_max();
5
6   ensures \result ≡ 0;
7   decreases;
8  pure int count0(int x) = 0;
9
10  requires 0≤r ∧ r≤10;
11  ensures (0≤\result ∧ \result≤r);
12  decreases r;
13 pure int count1r(int x, int r) = r ≡ 0 ? count0(x)
14   : inp(x, r-1) > 0 ? count1r(x, r-1) + 1 : count1r(x, r-1);
15
16  ensures (0≤\result ∧ \result≤10);
17  decreases;
18 pure int count(int x) = count1r(x, 10);
```

For a reduction, the initialisation and update parts are translated into separate functions, and reduction domain variables are explicitly added as function parameters. Listing 4 illustrates this for the `cnt` example. The function `count0` on line 8 corresponds to the initialisation (line 3 of Listing 2), with the translated post-state predicate on line 6. The function `count1r` (lines 13–14) corresponds to the update function (line 5 of Listing 2). Note that the annotation on line 10 refers to the reduction domain. The reason for using references to `r-1` on line 14 is that the result of the whole computation corresponds to `r` with its maximum value `10` (see line 18). This is computed by recursively decrementing `r`. The invariant on line 6 of Listing 2 is translated into the postcondition of `count1r` (line 11), reflecting that the invariant should hold after each reduction iteration. For the `decreases r` annotation added on line 12, VERCORS will try to prove that this recursive function terminates. The reduction postcondition is represented by the `ensures` annotation on line 16.

Guarantees. For the front-end verification approach, HALIVER straightforwardly encodes a HALIDE function without reductions, as it defines the function pointwise in PVL. For reductions, HALIVER mimics the iterative updates with recursion, as shown in the `cnt` example of Listings 2 and 4. HALIVER adds `decreases` clauses to check that the recursive functions terminate.

With HALIVER's approach, functional correctness of the algorithm part can be proven. Since memory safety depends on how a HALIDE algorithm is compiled into actual code according to a schedule, this is checked using the back-end verification approach.

3.3 Back-end Verification Approach

For verifying a HALIDE algorithm with a schedule, HALIVER adds annotations to the generated C code that can be checked by VERCORS. First, HALIVER gen-

Listing 5. A schedule for the blur example (Listing 1), together with the loop nest the Halide compiler produces, given in the intermediate representation of Halide. The `blur_y` bounds are assumed to be from 0 up to 1,024 for dimensions `x` and `y`.

```
blur_y.split(y, yo, yi, 8).parallel(yo).split(x, xo, xi, 2).unroll(xi);
blur_x.store_at(blur_y, yo).compute_at(blur_y, yi).split(x, xo, xi, 2).unroll(xi);
// Below is the loop nest produced (not part of the schedule)
produce blur_y:
  parallel y.yo in [0, 127]:
    store blur_x:
      for y.yi in [0, 7]:
        produce blur_x:
          for y:
            for x.xo in [0, 511]:
              unrolled x.xi in [0, 2]:
                blur_x(...) = ...
        consume blur_x:
          for x.xo in [0, 511]:
            unrolled x.xi in [0, 2]:
              blur_y(...) = ...
```

erates read and write permissions and preconditions for functions used in definitions. This generation of permissions makes it possible to keep the annotations of Halide algorithms concise, since the user does not have to specify permissions. Second, HaliVer transforms the annotations and adds them to the intermediate representation used by the Halide compiler. Finally, HaliVer adds the annotations to the code, during the code generation of the Halide compiler.

Annotation Generation. Since Halide algorithms consist of pure point-wise functions, permissions are relatively straightforward: for a function `f(x,...)`, HaliVer generates the write permission `Perm(f(x,...),1\1)`. For the blur example from Listing 1, HaliVer generates `blur_x.context(Perm(blur_x(x,y), 1\1)` and `blur_y.context(Perm(blur_y(x,y), 1\1)` for function `blur_x` and `blur_y`, respectively.

For update functions and reductions, HaliVer generates (1) read permissions for function values that are not being updated, and (2) a pre-state predicate, using the post-state predicate of the previous update step.

Once a function is fully defined, read permission is given to all values wherever the function is used, along with a context predicate containing any intermediate annotations of the function.

Transformation of Annotations. Next, HaliVer transforms the annotations according to the schedule given by the user and associates them with the corresponding parts of the optimised Halide program expressed in Halide's intermediate language.

HaliVer supports the `split`, `fuse`, `parallel`, `unroll`, `store_at`, `reorder` and `compute_at` scheduling directives. Of the most commonly used directives in the Halide example apps[5], only `vectorize` is not supported because VerCors

[5] https://github.com/halide/Halide/tree/main/apps

does not yet support verification of vectorised code as produced by Halide.[6] With these directives, HaliVer provides the means to verify optimised programs w.r.t. memory locality, parallelism and recomputation. This is the optimisation space in which Halide resides [22]. We illustrate the meaning of these directives with an example. Listing 5 shows a schedule for blur on lines 1–2, and below that the *loop nest* structure of the resulting program. Loop nests are program statements of nested for loops. The loops can be sequentially executed or be parallelized, unrolled or vectorized. The allocation of space for a function result is indicated by `store`, and `produce` and `consume` refer to writing and reading function results, respectively. This loop nesting corresponds to the actual code produced by the Halide compiler.

Assuming that the output dimensions in the example are both of size 1,024, the directive `split(y, yo, yi, 8)` (line 1 of Listing 5) splits the dimension `y` into two nested dimensions `y.yo` (line 5) and `y.yi` (line 7) of sizes `128` and `8`, respectively. HaliVer similarly renames references to `y` in annotations. The `parallel(yo)` directive (line 1) expresses that `y.yo` should be executed in parallel (line 5). The `store_at(blur_y, yo)` directive (line 2) expresses that `blur_x` must be stored at the start of the `y.yo` loop (line 6). The directive `compute_at (blur_y, yi)` (line 2) defines that the values for `blur_x` should be produced at `y.yi` (line 8). The directive `unroll(xi)` (line 1 and 2) expresses that the dimension `xi` should be completely unrolled.

The `for` loops are sequential. In this example, `fuse` and `reorder` are not used; they express that two dimensions should be fused into one and the nesting order of the loops should be changed, respectively.

HaliVer moves bottom-up through the program, constructing loop invariants for each loop by taking the constructed state predicates from the loop body and extending them with quantifications over the loop variables. Below, we give an example of this exact process for the blur example of Listing 1.[7]

Encoding of Halide Program. Finally, HaliVer adds annotations to the C code during the code generation of the Halide compiler. As an example, we show how HaliVer adds annotations of the `blur_y` function of Listing 1 with the schedule of Listing 5. The result of this can be found in Listing 6. It shows the structure of the whole program, but is focussed on the code below the `consume blur_x` node (line 13 of Listing 5).

First, HaliVer updates its pipeline annotations (lines 1–2 of Listing 1), to match the flattened array structure the Halide back-end uses, and adds them to the function contract (lines 8–15 of Listing 6). HaliVer also uses the Halide definition of division (`hdiv`), i.e., Euclidean[8] [12] with $x/0 \equiv 0$.

[6] The vectorize scheduling directive is the same as the unroll directive from the perspective of transforming annotations. So they can be treated exactly the same and already are in HaliVer.

[7] For the interested reader, we explain the approach in a more general way in Appendix C of the version available at https://arxiv.org/abs/2401.10778.

[8] The Halide compiler uses bit operators to define euclidean division. However, bit operators are not supported in VerCors, so HaliVer uses an equivalent definition.

Listing 6. The C code and annotations the Halide compiler produces together with HaliVer for the function `blur_y`, focussing on the `consume blur_x` node (see line 13 of Listing 5). The complete encoding for the `blur_y` pipeline is available in Appendix B of the version available at https://arxiv.org/abs/2401.10778

```
struct halide_dimension_t {int32_t min, max;};
struct buffer {int32_t dimensions;struct halide_dimension_t *dim;int32_t *host;};
int div_eucl(int x, int y);
//@ pure int hdiv(int x, int y) = y ≡ 0 ? 0 : div_eucl(x, y);
//@ pure int p_i(int x);
/*@ ... // Buffers annotations
 context (∀ int x,int y;0≤x∧x<1026∧0≤y∧y<1026; inpb→host[y*1026+x]≡p_i(y*1026+x));
 // Pipeline preconditions
 requires inpb→dim[0].min≡blur_yb→dim[0].min∧ inpb→dim[0].max≡blur_yb→dim[0].max+2;
 requires inpb→dim[1].min≡blur_yb→dim[1].min∧ inpb→dim[1].max≡blur_yb→dim[1].max+2;
 // Pipeline postconditions
 ensures (∀ int x, int y; 0≤x∧x<1024∧0≤y& y<1024; blur_yb→host[y*1024+x] ≡ hdiv(
  hdiv(inpb→host[y*1026+x+1027]+inpb→host[y*1026+x+1028]+inpb→host[y*1026+x+1026],3)+
  hdiv(inpb→host[y*1026+x+2053]+inpb→host[y*1026+x+2054]+inpb→host[y*1026+x+2052],3)+
  hdiv(inpb→host[y*1026+x+1]+inpb→host[y*1026+x+2]+inpb→host[y*1026+x],3),3));@*/
int blur_3(struct buffer *inpb, struct buffer *blur_yb) {
 int32_t* blur_y = blur_yb→host;
 int32_t* inp = inpb→host;
 // produce blur_y
 #pragma omp parallel for
 for (int yo = 0; yo<0 + 128; yo++)
 ... // Annotations blur_y.y.yo
 {
   int64_t _2 = 10240;
   int32_t *blur_x = (int32_t *)malloc(sizeof(int32_t )*_2);
   int32_t _t11 = (yo * 8);
   ... // Annotations blur_y.y.yi
   for (int yi = 0; yi<0 + 8; yi++)
   {... // produce blur_x
    // consume blur_x
    int32_t _t16 = (yi + _t11) * 512;
    int32_t _t15 = yi * 512;
 /*@ loop_invariant 0≤xo ∧ xo≤0 + 512;
     loop_invariant (∀* int x, int y; 0≤x ∧ x<1024 ∧ yo*8≤y ∧ y<yo*8 + 10;
      Perm(&blur_x[(y-yo*8)*1024+x], 1\2));
     loop_invariant (∀ int xo, int y; 0≤xo ∧ xo<1024 ∧ yo*8+yi≤y ∧ y≤yo*8+yi+2;
      blur_x[(y-yo*8)*1024+xo] ≡ hdiv(p_i(y*1026+xo) + p_i(y*1026+xo+1) + p_i(y*1026+xo
            +2),3));
     loop_invariant (∀* int xif, int xof; 0≤xof ∧ xof<512 ∧ 0≤xif ∧ xif<2;
      Perm(&blur_y[(yo*8+yi)*1024+xof*2+xif], 1\1));
     loop_invariant (∀ int xof, int xif; 0≤xof ∧ xof<xo ∧ 0≤xif ∧ xif<2; blur_y[(
           yo*8+yi)*1024+xof*2+xif] ≡
      hdiv(hdiv(p_i((yo*8+yi)*1026+xof*2+xif) + p_i((yo*8+yi)*1026+xof*2+xif+1) + p_i((
            yo*8+yi)*1026+xof*2+xif+2), 3) +
      hdiv(p_i((yo*8+yi)*1026+xof*2+xif+1026) + p_i((yo*8+yi)*1026+xof*2+xif+1027) + p_i
            ((yo*8+yi)*1026+xof*2+xif+1028), 3) +
      hdiv(p_i((yo*8+yi)*1026+xof*2+xif+2052) + p_i((yo*8+yi)*1026+xof*2+xif+2053) + p_i
            ((yo*8+yi)*1026+xof*2+xif+2054), 3), 3)); @*/
    for (int xo = 0; xo<0 + 512; xo++)
    {
     int32_t _t9 = (xo + _t15);
     blur_y[(xo + _t16) * 2] = div_eucl(blur_x[_t9 * 2] + blur_x[_t9 * 2 + 1024] +
           blur_x[_t9 * 2 + 2048], 3);
     blur_y[(xo + _t16) * 2 + 1] = div_eucl(blur_x[_t9 * 2 + 1] + blur_x[_t9 * 2 + 1025]
            + blur_x[_t9 * 2 + 2049], 3);
    } // for xo
   } // for yi
   free(blur_x);
 } // for yo
 return 0;}
```

Next, HaliVer transforms the annotations added to the `blur_y` function, before it adds them to any loop nest. The Halide compiler flattens the two-dimensional function `blur_y(x,y)` into a one-dimensional array `blur_y[y*1024+ x]`, so HaliVer does the same for all function references in the annotations. Next, from the schedule, the directive `split(x, xo, xi, 2)` splits `x` into `xo` and `xi` of sizes `512` and `2`, respectively. A similar split is performed for `y`. The generated annotation **context** `(Perm(blur_y(x,y), 1\1))` becomes `context Perm(&blur_y[(yo*8+yi)+ xo*2+ xi], 1\1))`.

For the annotation **ensures**`(blur_y(x,y)==(((inp(x,y)+...`, HaliVer replaces the calls to `inp(x,y)` with calls to an abstract pure function `p_i`. This is done because quantification instantiation in VerCors can become unstable if `inp` is used frequently. Where `inp` is used in the code, HaliVer adds annotations stating that `inp` and `p_i` have the same value (line 7 of Listing 6).

HaliVer adds these annotations to the first loop nest, starting bottom up. In Listing 5, this is `xi`, but since this loop is unrolled, additional annotations are not needed. After passing this loop nest, anything for `xi=0` and `xi=1` now holds. HaliVer changes the annotations by quantifying over `xi`'s domain. It uses `xif` as variable and changes any references to `xi` towards `xif`. The resulting permissions are `(∀xif; 0≤xif ∧xif<2; Perm(blur_y[(yo*8+yi)+ xo*2+xif], 1\1))`. The other annotations are processed in a similar way.

Next, HaliVer arrives at the loop nest for `xo`, which needs loop invariants. First, the tool adds the bounds of the `xo` dimension (line 33 of Listing 6). The annotation is transformed depending on whether it was a `requires`, `ensures` or `context` annotation. The write permission (`context`), should hold before the loop starts and after the loop ends. Therefore, HaliVer adds the permission, but quantifies over dimension `xo`, which results in a loop invariant (lines 38–39 of Listing 6). The `ensure` annotation does not hold at the start of the loop, but after each iteration of the loop, one more value for `xo` holds. Therefore, HaliVer quantifies over `xof` bounded by zero and the iteration variable `xo`, and replaces occurrences to `xo` with `xof`, which leads to a loop invariant (lines 40–43 of Listing 6). For loops above this loop nest, the `ensure` annotations hold for the whole domain of `xo`, resulting in `ensures (∀xof, xif; 0≤xof ∧xof<512∧0 ≤xif ∧xif<2; blur_y[(yo*8+yi)*1024+xof*2+xif] ≡...`. This annotation is added to the parallel for loop.

After constructing the `produce` node for `blur_y`, the `produce` node for `blur_x` is constructed in a similar way. The bound inferencer of Halide detects it only needs to calculate for `y` values of `8*y0+yi` up to `8*y0+yi+2`. The annotations are transformed respecting that fact. After the `produce` node, the `blur_x` is consumed (line 30 of Listing 6). So for each loop below the `consume` statement, HaliVer adds read permission (lines 34–35 of Listing 6s) and the post-state predicate of `blur_x` (lines 36-37 of Listing 6) as context annotations. For the loop of `xo`, this means they are valid for any value of `xo`.

Guarantees. With the back-end verification approach, HaliVer can prove that the optimised code produced by the Halide compiler is correct w.r.t. specifications. Memory safety is proven without any additional effort, as the permission

Table 1. Number of lines of code and annotations for different HALIDE algorithms, schedules and resulting programs, and the verification times required by VERCORS to prove memory safety, given that no annotations are provided by the user. The letters after each schedule denote the used directives: `compute_at`, `fuse`, `parallel`, `reorder`, `split`, `store_at` and `unroll`. *F* stands for verification failed. Times with † are inconsistent, i.e. they are succesfully verified, but can also sometimes fail or timeout.

Name		HALIDE	Sched.	C			
		LoC	Dir.	LoC	LoA.	Loops	T. (s).
blur	V0	38	0	178	60	2	18
	V1-{f,p}	"	2	172	56	1	19
	V2-{c,p,r,s}	"	6	212	74	6	29
	V3-{c,p,s,st,u}	"	8	211	72	5	24
hist	V0	71	2	299	98	11	30
	V1-{c,p,r,u}	"	4	308	99	11	38
	V2-{c,p,r,u}	"	6	311	105	13	48
	V3-{c,p,r,u}	"	13	312	101	13	48
conv_	V0	44	0	273	148	7	90
layer	V1-{c,f,p,u}	"	4	281	145	8	97
	V2-{p,r,s,u}	"	6	302	166	10	209
	V3-{c,p,r,s,u}	"	15	279	148	7	168
gemm	V0	70	0	218	105	3	41
	V1-{c,p,r,s}	"	8	274	136	10	89
	V2-{c,p,r,s}	"	16	342	173	19	196†
	V3-{c,f,p,r,s,u}	"	24	451	221	31	F
auto_	V0	112	0	443	118	19	35
viz	V1-{c}	"	9	402	139	23	180
	V2-{c,p}	"	12	440	156	27	170
	V3-{c,p,r,s}	"	27	443	152	25	105
camera_pipe-{c,p,r,s,st}		345	27	701	236	25	F
bilateral_grid-{c,p,r,u}		88	18	562	180	39	140
depthwise_separable_conv-{c,p,r,s}		94	13	562	315	44	480

annotations for this are generated automatically. For functional correctness, a specification needs to be provided. For any non-inlined function, an intermediate annotation is required to guide VERCORS in correct functional verification.

The approach is sound, but not necessarily complete. One concern is that, since we have not formally proved the correctness of the transformation, our implementation could in principle be wrong. HALIVER addresses this by keeping the *pipeline* annotations very close to what the user has written as annotations. These pipeline annotations act as the formal contract that will be verified, and the user can inspect these at any time. If an intermediate annotation is not correctly transformed, the verification will fail, thus remaining sound but not complete. Of course we have not constructed any transformations to be wrong, but even if there is an oversight, we will remain sound. Moreover, in Section 4, we show that our approach works for real world examples.

4 Evaluation

The goal of the evaluation of HALIVER is four-fold. (1) We evaluate that the front-end verification approach of HALIVER can verify functional correctness properties for a representative set of HALIDE algorithms. (2) For the back-end verification approach, the annotations that HALIVER generates and transforms should lead to successful verification for a representative set of HALIDE programs, with schedules that use the most important scheduling directives in different

Table 2. Number of lines of code and annotations for different Halide algorithms, schedules and resulting programs, and the verification times required by VerCors.

Name		Halide		Front-end	Sched.	C				LoA
		LoC	LoA	T. (s)	LoC	LoC	LoA	Loops	T. (s)	incr.
blur	V0	38	2	8	0	178	63	2	21	31.5x
	V1-{f,p}	"	"	"	2	172	58	1	23	29.0x
	V2-{c,p,r,s}	"	"	"	6	212	83	6	52	41.5x
	V3-{c,p,s,st,u}	"	"	"	8	211	79	5	$97^{\dagger}$	39.5x
hist	V0	71	10	8	0	299	118	11	34	11.8x
	V1-{c,p,r,u}	"	"	"	4	308	118	11	47	11.8x
	V2-{c,p,r,u}	"	"	"	6	311	123	13	56	12.3x
	V3-{c,p,r,u}	"	"	"	13	312	125	13	64	12.5x
conv_	V0	44	7	8	0	273	177	7	111	25.3x
layer	V1-{c,f,p,u}	"	"	"	4	281	174	8	108	24.9x
	V2-{p,r,s,u}	"	"	"	6	302	204	10	283	29.1x
	V3-{c,p,r,s,u}	"	"	"	15	279	177	7	207	25.3x
gemm	V0	70	12	7	0	218	120	3	43	10.0x
	V1-{c,p,r,s}	"	"	"	8	274	169	10	133	14.1x
	V2-{c,p,r,s}	"	"	"	16	342	230	19	368	19.2x
	V3-{c,f,p,r,s,u}	"	"	"	24	451	310	31	F	25.8x
auto_	V0	112	15	8	0	443	158	19	$152^{\dagger}$	10.5x
viz	V1-{c}	"	"	"	9	402	210	23	216	14.0x
	V2-{c,p}	"	"	"	12	440	235	27	$230^{\dagger}$	15.7x
	V3-{c,p,r,s}	"	"	"	27	443	229	25	$192^{\dagger}$	15.3x

combinations. (3) We evaluate the verification speed for front-end and back-end verification. (4) Lastly, we evaluate how many annotations are needed in HaliVer compared to manually annotating the generated C programs.[9]

Set-up. We used a machine with an 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz with 32GB running Ubuntu 23.04.

We used eight characteristic programs from the Halide repository.[10] These are representative Halide algorithm examples. They cover all scheduling directives supported by HaliVer, in commonly-used combinations. We removed any scheduling directives that we do not support. VerCors is unable to deal with large dimensions that are unrolled, thus we removed some `unroll` directives as well.[11]

The original schedule, as found in the Halide repository, is indicated with V3 if there are multiple schedules present. For five of these programs we defined annotations that express functional properties. These five programs are also evaluated with the standard schedule (V0), which tries to inline functions as much as possible, and two additional schedules (V1 and V2) we constructed.

Memory Safety Results. We evaluate 8 Halide programs, with in total 23 schedules, and prove data race freedom and memory safety for 21 of them. No user provided annotations are needed. The results can be found in Table 1.

For each case, we provide: the number of lines of code (LoC)[12] for the Halide algorithm, without the schedule and number of scheduling directives (Sched.

[9] The experiments can be found at https://github.com/sakehl/HaliVerExperiments.

[10] https://github.com/halide/Halide/tree/main/apps `gemm` is part of `linear_algebra`.

[11] For the interested reader, we explain this further in Appendix A of the version available at https://arxiv.org/abs/2401.10778.

[12] These lines are counted automatically and indicate the size of the programs.

Dir.). For the generated programs (C) we list: lines of code (LoC), lines of annotations (LoA.), number of (parallel) loops (Loops). These numbers indicate how large programs tend to become w.r.t. Halide algorithms, and how much annotation effort would be required to manually annotate the programs. Verification running times (T. (s)) are given in seconds, averaged over five runs.

For `camera_pipe`, VerCors gives a verification failure. It could not prove a `loop_invariant`, but after simplifying parts of the generated C program not related to this specific invariant, it leads to a successful verification. This indicates that the program is too complex for the underlying solvers. We also coded this example in similar Pvl code instead of C, which verifies in 193s. We suspect the failure is caused by quantifier instantiation, which instantiates too many quantifiers, resulting in the SMT solver on which VerCors relies stopping the exploration of quantifiers that are needed for successful verification.

For `gemm V3`, verification fails due to VerCors not sufficiently rewriting annotations of the `fuse` directive.[13]

Functional correctness results. Next, we evaluate five[14] algorithms with annotations and 20 schedules, both for the front-end and back-end. HaliVer proves functional correctness for the front-end, and both functional correctness and data race freedom and memory safety for the back-end for 19 of the 20 schedules. These results are given in Table 2. The table additionally has the amount of user provided annotations (LoA.) and the last column (Ann. incr.) indicates the growth of the annotations. The annotations of the C file (LoC) contain both the generated annotations, which are already present in Table 1 and the transformed user annotations.

For optimised programs, the annotation size is strongly related to the number of loops, as each loop needs its own loop invariants. Front-end verification is successful for all examples and is relatively fast compared to back-end verification. In verification of the C files produced by the back-end verification approach, time increases as the number of scheduling directives increases. Here, `gemm V3` also fails for the same reason as outlined above.

Inconsistent Results. For `gemm V2` for the memory benchmarks and for `blur V3` and `auto_viz V0, V2` and `V3`, VerCors does not always succeed with the verification. In the case of `gemm V2`, the verification sometimes hangs, which is timed out after 10 minutes. In the other cases, VerCors sometimes gave a verification failure. This inconsistency is due to the non-deterministic nature of the underlying SMT solvers.

Conclusions. With the front-end verification approach of HaliVer we are able to prove functional correctness properties for representative Halide algorithms.

[13] For the interested reader, we explain this further in Appendix A of the version available at https://arxiv.org/abs/2401.10778.

[14] The other three algorithms from the memory safety results are typical image processing pipelines. They are therefore less suitable for checking functional correctness and are not used here.

Using HaliVer's back-end verification approach, the tool provides correct annotations for the generated C programs. VerCors successfully verifies all but two programs. However, in the unsuccessful cases, HaliVer runs into limitations of the underlying tools. The verified programs are all verified within ten minutes. Finally, the manual annotation effort required is an order of magnitude larger than the effort required for HaliVer's approach.

5 Related Work

There is much work on optimising program transformations, either applied automatically or manually [2,11], sometimes using scheduling languages [3,6–8,22, 23,28]. The vast majority of this does not address functional correctness.

Work on functional correctness consists of techniques that apply verification every time a program is transformed, and techniques that verify the compiler.

Liu *et al.* [15] propose an approach inspired by scheduling languages, with proof obligations generated when a program is optimised, for automatic verification using Coq. The Cogent language [20] uses refinement proofs, to be verified in Isabelle/HOL. However, it does not separate algorithms from schedules. In [17, 18] an integer constraint solver and a proof checker are used, respectively, to verify the transformation of a program. In all these approaches, semantics-preservation is the focus, as opposed to specifying the intended behaviour. Model-to-model transformations can be verified w.r.t. the preservation of functional properties [21]. However, that work targets models, not code.

Regarding the verification of compilers, CompCert [14] is a framework involving a formally verified C compiler. In [19], Halide's Term Rewriting System, used to reason about the applicability of schedules, is verified using Z3 and Coq. These approaches do not require verification every time an optimisation is applied, but verifying the compiler is time-consuming and complex, and has to be redone whenever the compiler is updated. Furthermore, they focus on semantics-preservation, not the intended behaviour of individual programs.

Alpinist [27] is most closely related. This tool automatically optimises Pvl code, along with its annotations, for verification with VerCors. It allows the specification of intended behaviour, but it does not separate algorithms from schedules, forcing the user to reason about the technical details of parallelisation.

6 Conclusions & Future Work

We presented HaliVer, a tool for verifying optimised code by exploiting the strengths of scheduling languages and deductive verification. It allows focussing on functionality when annotating programs, keeping annotations succinct.

For future work, we want to extend the HaliVer tool with aspects not directly supported by VerCors, such as vectorisation. The master thesis of [24] defines a natural semantics for Halide. We want to formalise our front-end Pvl encoding with an axiomatic semantics to match this semantics. We also want to investigate the inconsistent results and see whether annotations with quantifiers can be rephrased to allow VerCors to be more consistent. In this work we have focussed on parallel CPU code, but we have designed our approach to be extendable to GPU code produced by Halide.

With the current expressiveness of the annotations, when reduction domains are present, HaliVer proves functional correctness for specific inputs. For example, in Listing 2 we can prove that `count(x)==9` if we require that `input (x,y)==x`. This can also be done for any input if the reduction domain is of known size, but then many annotations are needed. To make the annotations concise, a user needs to be able to use axiomatic data types[15] and pure functions in their annotations. We expect that these annotations can be similarly transformed by our approach, and that is thus orthogonal to this contribution, but this is planned as future work.

Most Halide programs use floating point numbers. These are currently modelled as reals in VerCors. How to efficiently verify programs with floats using deductive verifiers is still an open research question. Once this is addressed, HaliVer will be able to give better guarantees.

We require that the bounds of a Halide program are set to concrete values for our back-end verification approach. HaliVer transforms the annotations the same way for not know bounds, but the underlying tools have difficulty verifying these programs. With unknown bounds, we end up with nonlinear arithmetic due to the flattening of multi-dimensional functions on one-dimensional arrays. This is generally undecidable, so the SMT solvers that VerCors rely on cannot handle it. We will investigate if there are ways to tackle this in our domain-specific case.

Acknowledgements We are grateful to the anonymous reviewers of TACAS 2024 for their thorough reading and constructive feedback. We want to thank Jan Martens for their discussions and feedback on this work.

References

1. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:2)2015
2. Bacon, D., Graham, S., Sharp, O.: Compiler Transformations for High-Performance Computing. ACM Computing Surveys **26**(4), 345–420 (1994). https://doi.org/10.1145/197405.197406
3. Baghdadi, R., Ray, J., Romdhane, M.B., Sozzo, E.D., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., Amarasinghe, S.P.: Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In: CGO. pp. 193–205. IEEE (2019). https://doi.org/10.1109/CGO.2019.8661197
4. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors Tool Set: Verification of Parallel and Concurrent Software. In: Polikarpova, N., Schneider,

[15] https://vercors.ewi.utwente.nl/wiki/#axiomatic-data-types

S. (eds.) Integr. Form. Methods. pp. 102–110. Lecture Notes in Computer Science, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7

5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL. pp. 259–270 (2005). https://doi.org/10.1145/1040305.1040327
6. Chame, C.C.J., Hall, M.: CHiLL: A framework for composing high-level loop transformations. 08-897, University of Southern California (2008)
7. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In: 13th USENIX Symp. Oper. Syst. Des. Implement. OSDI 18. pp. 578–594. USENIX Association, USA (2018)
8. Hagedorn, B., Elliott, A.S., Barthels, H., Bodik, R., Grover, V.: Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In: Proc. ACM Int. Conf. Parallel Archit. Compil. Tech. pp. 71–82. ACM, Virtual Event GA USA (Sep 2020). https://doi.org/10.1145/3410463.3414632
9. Hähnle, R., Huisman, M.: Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In: Computing and Software Science - State of the Art and Perspectives. LNCS, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
10. Hijma, P., Heldens, S., Sclocco, A., van Werkhoven, B., Bal, H.: Optimization Techniques for GPU Programming. ACM Computing Surveys **55**(11), 239:1–239:81 (2023). https://doi.org/10.1145/3570638
11. Kowarschik, M., Weiß, C.: An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In: Algorithms for Memory Hierarchies. LNCS, vol. 2625, pp. 213–232. Springer (2003). https://doi.org/10.1007/3-540-36574-5_10
12. Leijen, D.: Division and modulus for computer scientists (July 2003), https://www.microsoft.com/en-us/research/publication/division-and-modulus-for-computer-scientists/, short note about division definitions in programming languages
13. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the top: What will drive computer performance after Moore's law? Science **368**(6495) (2020). https://doi.org/10.1126/science.aam9744
14. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
15. Liu, A., Bernstein, G.L., Chlipala, A., Ragan-Kelley, J.: Verified tensor-program optimization via high-level scheduling rewrites. Proc. ACM Program. Lang. **6**(POPL), 55:1–55:28 (Jan 2022). https://doi.org/10.1145/3498717
16. Müller, P., Schwerhoff, M., Summers, A.: Viper - a verification infrastructure for permission-based reasoning. In: VMCAI (2016). https://doi.org/10.1007/978-3-662-49122-5_2
17. Namjoshi, K.S., Singhania, N.: Loopy: Programmable and formally verified loop transformations. In: International Static Analysis Symposium. pp. 383–402. Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_19
18. 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). https://doi.org/10.1007/978-3-030-67067-2_7

19. Newcomb, J.L., Adams, A., Johnson, S., Bodik, R., Kamil, S.: Verifying and Improving Halide's Term Rewriting System with Program Synthesis. Proc. ACM Program. Lang. **4**(OOPSLA), 166:1–166:28 (Nov 2020). https://doi.org/10.1145/3428234
20. O'Connor, L., Chen, Z., Rizkallah, C., Jackson, V., Amani, S., Klein, G., Murray, T., Sewell, T., Keller, G.: Cogent: Uniqueness Types and Certifying Compilation. Journal of Functional Programming **31**(e25), 1–66 (2021). https://doi.org/10.1017/S095679682100023X
21. de Putter, S., Wijs, A.: Verifying a verifier: on the formal correctness of an LTS transformation verification technique. In: FASE. pp. 383–400. Springer (2016). https://doi.org/10.1007/978-3-662-49665-7_23
22. Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Halide: Decoupling algorithms from schedules for high-performance image processing. Commun. ACM **61**(1), 106–115 (Dec 2017). https://doi.org/10.1145/3150211
23. 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. SIGPLAN Not. **48**(6), 519–530 (Jun 2013). https://doi.org/10.1145/2499370.2462176
24. Reinking, A., Bernstein, G., Ragan-Kelley, J.: Formal Semantics for the Halide Language. Master's thesis, EECS Department, University of California, Berkeley (2020, May)
25. 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). https://doi.org/10.1007/978-3-030-64276-1_10
26. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal Verification of Parallel Prefix Sum. In: Lee, R., Jha, S., Mavridou, A. (eds.) NASA Form. Methods. pp. 170–186. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_10
27. Sakar, Ö., Safari, M., Huisman, M., Wijs, A.: Alpinist: An Annotation-Aware GPU Program Optimizer. In: TACAS, LNCS, vol. 13244, pp. 332–352. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_18
28. Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., Amarasinghe, S.P.: GraphIt: a high-performance graph DSL. Proc. ACM Program. Lang. **2**(OOPSLA), 1–30 (2018). https://doi.org/10.1145/3276491

Gray-Box Fuzzing via Gradient Descent and Boolean Expression Coverage[⋆]

Martin Jonáš, Jan Strejček, Marek Trtík(✉), and Lukáš Urban

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{martin.jonas,strejcek,trtikm,492717}@mail.muni.cz

Abstract. We present a gray-box fuzzing approach based on several new ideas. While standard gray-box fuzzing aims to cover all branches of the input program, our approach primarily aims to cover both results of each Boolean expression. To achieve this goal, we track the distances to flipping these results and we dynamically detect the input bytes that influence the distance. Then we use this information to efficiently flip the results. More precisely, we apply gradient descent on the detected bytes or we create new inputs by using detected bytes from different inputs. We implemented our approach in a tool called Fizzer. An evaluation on the benchmarks of Test-Comp 2023 shows that Fizzer is fully competitive with the winning tools of the competition, which use advanced formal methods like symbolic execution or bounded model checking, usually in combination with fuzzing.

1 Introduction

Fuzzing is a technique for automated generation of test inputs for a given program. The goal of fuzzing is to generate tests with high code coverage and to quickly detect bugs in the code. We distinguish three basic kinds of fuzzing based on their use of the given program. *Black-box fuzzing* [18] only runs the given program on various inputs and observes the outputs. *Gray-box fuzzing* [18] first instruments the program to get some information about performed executions. The instrumented code typically tracks the information about the basic blocks visited during the execution. While black-box and gray-box fuzzing rely on dynamic analysis of the original or instrumented code, *white-box fuzzing* [18] combines dynamic analysis with some static analysis of the code, typically concolic execution, symbolic execution, or bounded model checking.

Black-box fuzzers have only limited efficiency due to the lack of information. Gray-box fuzzers and white-box fuzzers proved to be very efficient and they are routinely applied in software industry. For example, the gray-box fuzzer AFL [27] discovered dozens of bugs in many recognized open-source projects and the white-box fuzzer Sage [11] is intensively used in Microsoft.

The standard approach of successful gray-box fuzzers is to collect only a very limited information about each program execution and to quickly perform

[⋆] This work has been supported by the Czech Science Foundation grant GA23-06506S.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 90–109, 2024.
https://doi.org/10.1007/978-3-031-57256-2_5

as many executions as possible. In this paper we suggest an approach that gathers slightly more information about program executions and uses it to select uncovered parts of the code and make more targeted attempts to cover it. We can illustrate some ideas of our approach on a simple example. Consider a program that contains a branching statement `if (x > 42)` and assume that some program execution passed its `true` branch. During this program execution, we saved the value of `x - 42` to know the *distance* to entering the `false` branch. When we decide to cover the `false` branch, we first repeatedly execute the program on modified inputs to detect the bytes of the input that have some influence on the distance value. This is called a *sensitivity analysis* and the detected bytes are called *sensitive*. We then propose two analyses that use the sensitive bytes to cover the uncovered branch. One analysis performs a dynamic *gradient descent* on the sensitive bytes with the aim to minimize the absolute value of the distance and to enter the `false` branch. Alternatively, if we already know another input that entered the `false` branch of this statement in a different calling context, we can try to use the value of its sensitive bytes instead of the sensitive bytes of the current input. This analysis is called *byteshare analysis*. Now consider a slightly different program where the branching statement has the form `if (res)` where `res` is a Boolean variable assigned before by `res = x > 42`. Clearly, we want to track the distance to changing the value of `res`. Hence, we in fact do not track distances for branching conditions, but the distances for values of atomic Boolean expressions. Roughly speaking, our approach aims to generate tests such that each atomic Boolean expression in each calling context is evaluated to `true` and to `false` in some program executions. Our fuzzing approach tracks its progress with the use of *atomic Boolean execution tree* and we talk about *Boolean expression coverage*.

The following section introduces the basic terminology used in the paper and states our assumptions on the analysed programs. Section 3 then describes the basic concepts of our approach, in particular the Boolean expression coverage, the information we collect from each program execution and how we obtain this information, the atomic Boolean execution tree, and the fuzzing algorithm. This algorithm iteratively tries to close vertices of the tree by generating inputs in which each of the vertices evaluates both to `true` and to `false` in order to either increase the Boolean expression coverage or to discover new parts of the tree. These inputs are generated by sensitivity analysis, byteshare analysis, and gradient descent analysis presented in Section 4. The selection strategy of the vertex to be closed is briefly explained in Section 5. Note that the page limit does not allow describing all the technical details of the approach. They can be found in the corresponding technical report [15].

We have implemented the presented fuzzing approach in a tool called FIZZER. The architecture and some implementation aspects of the tool are described in Section 6. Further, we have run FIZZER on all benchmarks of the *Cover-Branches* category of the *Competition on Software Testing (Test-Comp) 2023* [5]. We evaluated the tests generated by FIZZER using the competition infrastructure which measures the achieved branch coverage. The results presented in Section 7 show that our tool is competitive with the top-ranking tools of Test-Comp 2023, namely FuSeBMC [2], VeriFuzz [21], and CoVeriTest [6]. Note that our tool is a pure gray-box fuzzer while FuSeBMC and VeriFuzz combine dynamic analysis with static analyses like symbolic execution and bounded model checking. CoVeriTest fully relies on static methods like predicate analysis with the CEGAR loop and value analysis. Finally, Section 8 discusses some related work and Section 9 sums up the presented results and outlines future work.

2 Preliminaries

The ideas presented in this paper can be adopted for various kinds of programs. For ease of presentation, here we consider sequential C programs that get input only via functions `nondet_char()`, `nondet_int()`, and `nondet_float()` which return values of the corresponding type.

For simplicity, we assume that these are the only types that can be read from the input and we define the set $\mathit{InputTypes} = \{\texttt{char}, \texttt{int}, \texttt{float}\}$. We define the set of typed values $\mathit{TypedValues} = \{(v, t) \mid t \in \mathit{InputTypes}, v \text{ is a value of type } t\}$ and denote the pairs $(v, t) \in \mathit{TypedValues}$ as $v : t$, e.g., $3 : \texttt{int}$ is the value 3 of type `int`. We also work with untyped inputs, which are arbitrary finite sequences of bits 0 and 1. Untyped inputs are denoted by a standard language-theoretic notation, e.g., 1^{12} is a sequence of 12 elements 1.

An expression occurring in a program is called an *atomic Boolean expression* (ABE) if it has type `bool` and it is not a variable, not a call of a function whose definition is a part of the program, and not a result of applying logical operators, i.e., conjunction, disjunction, and negation. For example, the expression `(x > 3) && foo(x,y) && cond`, where `foo` is a function defined in the program and `cond` is a variable, contains only one ABE `x > 3`. By ABE we always mean a particular occurrence of the expression in the program.

We assume that the control flow is fully determined by the values of ABEs. This property may not hold for programs with `switch` statements, function calls via input-dependent function pointers, etc. However, such programs can be transformed into equivalent ones satisfying our assumption.

By a *calling context* we mean the sequence of function calls that are currently being evaluated. The outermost function call is the first element of the sequence and the last one is the function whose body is executed at the moment. In other words, the calling context roughly corresponds to the call stack.

We sometimes denote a sequence $x_1 x_2 \dots x_n$ as $\langle x_1, x_2, \dots, x_n \rangle$ or $\langle x_i \rangle_{1 \le i \le n}$.

3 Overview of Our Fuzzing Approach

This section provides an overview of the key concepts that are used in our fuzzing algorithm and presents the high-level view of the algorithm. The key heuristics for input generation are explained later in Sections 4 and 5.

```
void main() {
  int x = nondet_int();
  if (x < 42) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(a) Trivial case.

```
void main() {
  int x = nondet_int();
  bool res1 = x < 42;
  x++;
  bool res2 = x < 42;
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(b) Depends on a non-local comparison.

```
bool compare(int v) {
  return v < 42;
}

void main() {
  int x = nondet_int();
  bool res1 = compare(x);
  x++;
  bool res2 = compare(x);
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(c) Depends on a comparison coming from a different scope.

Listing 1.1: Example C codes showing that the values that influence which branch is taken can be both lexically far away from the branching statement and can be behind several layers of indirection.

3.1 Branch Coverage via Boolean Expression Coverage

The main idea of the proposed approach is to assign to each executed branching statement a metric called *distance* reflecting how far the current program state is from evaluating the branching expression to the opposite Boolean value. Thanks to this metric, we can use gradient descent to generate inputs that either flip the Boolean value or are close enough to the flipping point so that the actual flip can be achieved by small mutations of the input.

It is easy to define the distance for branchings like `if (x > 42)`: we set the distance to $x-42$ and minimize the absolute value $|x-42|$ to get close to the point where the result of the branching expression changes. However, as Listing 1.1 shows, the situation can be far more complex. The comparison does not have to occur in the branching expression itself, but it can be precomputed earlier in the program, it can come from a function call or be read from an array, etc.

We sidestep this issue by assigning the distances to atomic Boolean expressions and trying to flip their values rather than doing the same for branching expressions. In other words, we approach the goal of generating tests with maximal branch coverage indirectly by maximizing *Boolean expression coverage.* Intuitively, we try to generate a set of inputs such that every *atomic Boolean expression* evaluates to `true` on some input and to `false` on some input. In fact, we want to generate inputs leading to both Boolean values of each ABE in *each possible calling context.* The importance of the calling context is illustrated on the ABE `v < 42` in the code of Listing 1.1c: we clearly want to distinguish the case when the value of `v < 42` is used to set the value of `res1` from the case when it is used to set the value of `res2`. The precise goal of our approach will be formulated later using the terms *atomic Boolean execution tree* and *covered vertex* of the tree introduced in Definitions 1 and 2, respectively.

Every time an ABE e is evaluated, its *distance* is computed by the expression

$$dist(e) = \begin{cases} value(l) - value(r), & \text{if } e = l \bowtie r \text{ where } \bowtie\, \in \{=, \neq, <, \leq, >, \geq\}, \\ value(e), & \text{otherwise.} \end{cases}$$

In the first case, the $value(l)$ and $value(r)$ refer to the numerical values of l and r, respectively, before the evaluation of e. In the second case, $value(e)$ is defined as 1 if e evaluates to `true` and it is defined as 0 if e evaluates to `false`.

Note that the branch coverage and the atomic Boolean expression coverage do not precisely match. For example, we can achieve the full branch coverage of the code in Listing 1.1b by two tests; with input values 40 : `int` and 41 : `int`. However, the `true` branch of the first ABE `x < 42` is not covered by either of these tests. Nevertheless, our experimental evaluation shows that maximizing the atomic Boolean expression coverage also leads to test inputs with high branch coverage.

3.2 Instrumentation and Execution

From each program execution, our approach needs to get the sequence of evaluated ABEs including their calling contexts, their Boolean values, and their distances. To obtain this information, the program is instrumented with the following functions:

- To track the calling context, we assign a unique identifier *id* to each function call (except `nondet_*` function calls) and insert `__instr_call(`*id*`)` before the call and `__instr_return()` after the call. The inserted function calls maintain the current stack of open function calls.
- To track all evaluated ABEs and their values, distances, and calling contexts, we assign a unique identifier *id* to each ABE e and insert the call `__instr_abe(`*id*`, `*e*`, `*dist*(*e*)`)` before the ABE. The calling context is internally retrieved from the tracked stack of open function calls.

Listing 1.2 provides the instrumented programs from Listings 1.1b and 1.1c.

Besides the inserted function calls, we also alter the functions `nondet_`*type*`()` to collect the information about the values and types read from the input stream and when they were read.

In the following, we assume that there exists a function `execute`$(P', input)$ that gets an instrumented program P' and an untyped input $input \in \{0,1\}^*$ and returns the *trace* of the execution of P' on $input.0^\omega$, i.e., *input* extended with infinitely many zero bits. The trace is a pair $(usedInput, \pi)$ where

- *usedInput* is the sequence of *TypedValues* that were read by the program P' during the execution.
- π is the sequence $\langle(e_i, c_i, r_i, d_i, n_i)\rangle_{1 \leq i \leq k}$ of tuples, where each tuple represents one evaluation of an ABE: e_i is the evaluated ABE, c_i is the calling context in which it was evaluated, r_i is the result of the evaluation, d_i is the corresponding value of $dist(e_i)$, and n_i is the number of bytes of the input that have been read before the evaluation.

```
void main() {
  int x = nondet_int();
  __instr_abe(1, x < 42, x - 42);
  bool res1 = x < 42;
  x++;
  __instr_abe(2, x < 42, x - 42);
  bool res2 = x < 42;
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(a) Instrumentation of Listing 1.1b

```
bool compare(int v) {
  __instr_abe(1, v < 42, v - 42);
  return v < 42;
}

void main() {
  int x = nondet_int();
  __instr_call(1);
  bool res1 = compare(x);
  __instr_return();
  x++;
  __instr_call(2);
  bool res2 = compare(x);
  __instr_return();
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(b) Instrumentation of Listing 1.1c

Listing 1.2: Instrumented programs from Listings 1.1b and 1.1c

Note that the trace is always finite as P' is executed with some limits on the number of evaluated ABEs.

Example 1. Let P' be the instrumented program from Listing 1.2b. The function $\texttt{execute}(P', 0^{32})$ returns the trace $(\langle 0 : \texttt{int}\rangle, \pi)$, where

$$\pi = \big\langle (\texttt{v < 42}, \langle 1\rangle, \texttt{true}, -42, 4), (\texttt{v < 42}, \langle 2\rangle, \texttt{true}, -41, 4)\big\rangle.$$

In other words, the execution read only a single `int` of value 0 from the input and these 4 bytes were read before the first ABE evaluation. Further, the ABE `v < 42` with identifier 1 (for readability denoted directly by the expression) has been evaluated twice: once in the calling context $\langle 1\rangle$, value `true`, and distance -42 and later with the calling context $\langle 2\rangle$, value `true`, and distance -41.

3.3 Atomic Boolean Execution Tree

Each execution trace $(\mathit{usedInput}, \langle (e_i, c_i, r_i, d_i, n_i)\rangle_{1 \le i \le k})$ determines the sequence $r_1 r_2 \ldots r_k$ of ABE values. Our fuzzing approach tracks the information about all such sequences seen so far by maintaining an *atomic Boolean execution tree.*

Definition 1 (atomic Boolean execution tree, ABET). *An* atomic Boolean execution tree *(*ABET*) is a nonempty prefix-closed finite set $T \subseteq \{\texttt{true}, \texttt{false}\}^*$. Elements of T are vertices, ε is the root, and elements $v.\texttt{true}, v.\texttt{false}$ are children of v. We assume that each vertex is either a leaf or it has two children, i.e., for each $v \in T$ it holds $v.\texttt{true} \in T \iff v.\texttt{false} \in T$.*

Our method starts with the tree $T = \{\varepsilon\}$. Whenever we obtain a trace $(\mathit{usedInput}, \langle (e_i, c_i, r_i, d_i, n_i)\rangle_{1 \le i \le k})$, we update T to contain the sequence $r_1 \ldots r_k$

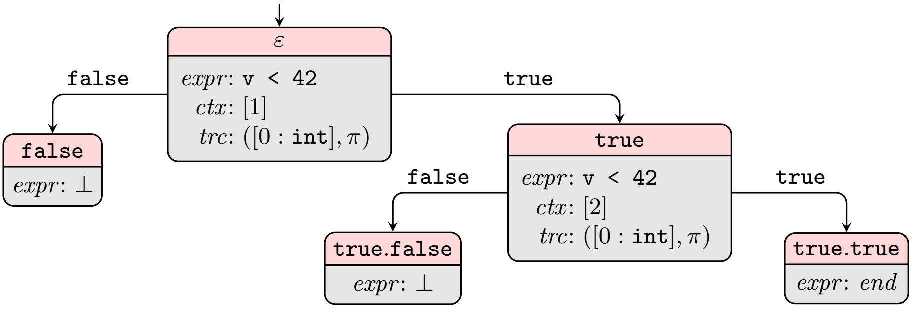


Fig. 1: An example of an ABET.

and all its prefixes. Further, with each newly added vertex we also add its sibling. We say that the trace *visits* a vertex v if v is a prefix of $r_1 r_2 \dots r_k$.

As we mentioned in the preliminaries, we assume that the next evaluated ABE of each program is fully determined by the values of ABEs evaluated before it. This means that each inner vertex $v \in T$ determines the corresponding ABE and its calling context. The ABE and the calling context corresponding to v are denoted by $expr(v)$ and $ctx(v)$. We extend the notation $expr(v)$ also to leaves. We set $expr(v) = end$ if we have seen a trace with the sequence v of ABE values. If this is not the case and v is in T only because of its sibling (or as the only node in the initial tree $\{\varepsilon\}$), we set $expr(v) = \bot$. Note that a leaf v with $expr(v) = \bot$ can become a leaf with $expr(v) = end$ or even an inner node if we later obtain a trace that visits v. Similarly, a leaf v with $expr(v) = end$ can become an inner node. This happens for example when v originally represents a trace that ends with an error (e.g., division by zero) and later we found a longer trace visiting v that avoids the error.

Finally, to each inner vertex $v \in T$ we associate some trace that visits it. The trace is denoted as $trc(v)$.

An example of an ABET can be found in Figure 1. It represents the tree for the instrumented program from Listing 1.2b after obtaining the first trace $(\langle 0 : \texttt{int}\rangle, \pi)$ given in Example 1.

Definition 2 (Covered vertex). *An inner vertex $v \in T$ is said to be* covered *if there are inner vertices $v_t, v_f \in T$ satisfying $expr(v) = expr(v_t) = expr(v_f)$, $ctx(v) = ctx(v_t) = ctx(v_f)$, $expr(v_t.\texttt{true}) \neq \bot$, and $expr(v_f.\texttt{false}) \neq \bot$. An inner vertex that is not covered is called* uncovered.

Definition 3 (Open/closed vertex). *An inner vertex $v \in T$ is said to be* open *if $expr(v.\texttt{true}) = \bot$ or $expr(v.\texttt{false}) = \bot$. An inner vertex that is not open is called* closed.

3.4 Fuzzing Algorithm

A high-level description of the fuzzing algorithm is given in Algorithm 1. The algorithm starts with instrumentation of the given program P (line 1) and ini-

Algorithm 1 Fuzzing algorithm

```
1: create instrumented program $P'$ from $P$ (see Section 3.2)
2: $T \leftarrow \{\varepsilon\}$
3: $(\mathit{usedInput}, \pi) \leftarrow$ execute$(P', \varepsilon)$
4: processTrace$(\mathit{usedInput}, \pi)$
5: while some inner vertex of $T$ is not covered do
6:     select an unprocessed open vertex $v$ from $T$ (see Section 5)
7:     if no $v$ is selected then end test generation
8:     try to close $v$ in $T$ using an input generation analysis (see Section 4)
```

tialization of the ABET T (line 2). Then it executes the instrumented program on the stream of zero bits (line 3) to obtain an initial trace $(\mathit{usedInput}, \pi)$ where π is of the form $\langle (e_i, c_i, r_i, d_i, n_i) \rangle_{1 \leq i \leq k}$.

On line 4, the trace is processed by `processTrace`$(\mathit{usedInput}, \pi)$. This function updates T with the sequence $r_1 r_2 \ldots r_k$ as described in Section 3.3. For each inner vertex $v \in T$ visited by the current trace and not visited by any trace before, we set $\mathit{trc}(v)$ to the current trace. Further, for each vertex $v \in T$ visited by the current trace and another trace before, we compute the value $\sum_{i=1}^{|v|+1} d_i^2$ and if it is smaller than the corresponding value for $\mathit{trc}(v)$, we set $\mathit{trc}(v)$ to the current trace. Our practical experiments showed that keeping the trace with the smaller sum of squares of d_i leads to better results than minimizing only the current distance $|d_{|v|+1}|$. Finally, the function `processTrace`$(\mathit{usedInput}, \pi)$ saves *usedInput* to the output test suite if it is in $\mathit{trc}(v)$ of some vertex v at this moment. Otherwise, the trace is completely discarded.

The main fuzzing loop (line 5) iterates until all vertices in T are covered. In each iteration, we select an unprocessed open vertex $v \in T$ (line 6). A vertex is *processed* if it has been analyzed by all input generation analyses. If we fail to select v, the fuzzing algorithm terminates (line 7). Otherwise, we try to close v by some input generation analysis (line 8). The selection process and the input generation analyses are described in Sections 5 and 4, respectively.

4 Input Generation

We propose three methods to generate new inputs with the aim to close the selected vertex: sensitivity analysis, byteshare analysis, and gradient descent. When a vertex is selected, we execute the first of these analyses that has not been executed yet for the vertex. The order is important, as byteshare and gradient descent analyses need the information about sensitive bytes, and byteshare analysis is significantly cheaper than the gradient descent analysis.

In all the analyses, v is the vertex we want to close, we assume without loss of generality that $\mathit{expr}(v.\mathtt{true}) = \bot$, and we define $l = |v| + 1$, i.e., the depth of v. The goal of all analyses is to generate an input for which the resulting trace visits v and continues to $v.\mathtt{true}$. In all the analyses, $\mathit{trc}(v) = (\mathit{usedInput}, \pi)$ denotes the current trace assigned to the vertex v, with the typed values read by the trace *usedInput* $= \langle in_i : t_i \rangle_{1 \leq i \leq n}$ and the sequence of ABE evaluations $\pi = \langle (e_i, c_i, r_i, d_i, n_i) \rangle_{1 \leq i \leq k}$. Moreover, whenever any of the analyses executes P', the resulting execution trace is processed by `processTrace` function.

4.1 Sensitivity Analysis

The goal of the analysis is twofold. First, it detects so-called *sensitive bytes* of vertex v, denoted as $\mathit{sbytes}(v)$. Let us denote as $b_{i,j}$ the j-th byte in the i-th typed value in_i. We check whether $b_{i,j}$ is sensitive by mutating each bit of the byte $b_{i,j}$ separately and executing the program P' on each one-bit mutation. If the resulting trace with $\pi' = \langle (e'_i, c'_i, r'_i, d'_i, n'_i) \rangle_{1 \leq i \leq k'}$ still visits v and the value of the distance function in the node v changes, i.e., $d'_l \neq d_l$, the whole byte is considered *sensitive* and is added to $\mathit{sbytes}(v)$. We also try changing the whole value in_i to several selected special values, e.g., the smallest and the greatest value of the type t_i and special floating-point values, if t_i is `float`.

Second, during the computation of sensitive bytes, we also extend the tree with each executed trace. The sensitivity analysis therefore also effectively works as a local neighborhood search around the previous input of the vertex v.

Observe that when computing sensitive bytes of the vertex v, we can simultaneously use the resulting traces to determine the sensitive bytes of all predecessors of v. We use this observation as an optimization in the implementation to reduce the number of sensitivity analysis executions.

4.2 Byteshare Analysis

Let u be an inner vertex of the current tree T with the same ABE as v (the contexts may differ), with a non-empty set of the sensitive bytes, and whose successor $u.\mathtt{true}$ is not a leaf. For each such vertex u, the analysis combines inputs from $\mathit{trc}(u.\mathtt{true})$ and $\mathit{trc}(v)$ into a new input. More precisely, the new input is the same as $\mathit{trc}(v)$, but for each $j \in \{1, 2, \ldots, \min(|\mathit{sbytes}(v)|, |\mathit{sbytes}(u)|)\}$, we replace the value of the j-th sensitive byte of v by the value of the j-th sensitive byte of u in $\mathit{trc}(u.\mathtt{true})$. The idea behind this construction is that we keep the new input similar to the original input of $\mathit{trc}(v)$ so that the execution trace will likely visit v, but we replace the sensitive bytes of v by those of $u.\mathtt{true}$, which might steer the execution to the desired child.

Note that $\mathit{sbytes}(v)$ and $\mathit{sbytes}(u)$ may be completely different bytes. The size of the sets may also differ. Since we lack information for building a mapping between $\mathit{sbytes}(v)$ and $\mathit{sbytes}(u)$, we simply map the bytes based on their order.

4.3 Gradient Descent with Multi-sampling and Locking

We extend the notion of sensitivity to the typed inputs. An element of the sequence *usedInputs* is called *sensitive* in v if it contains at least one byte sensitive in v. The gradient descent analysis tries to minimize the absolute value of the distance for v by changing only the sensitive typed inputs of the vertex v. We

Algorithm 2 Gradient descent for vertex v from seed $\boldsymbol{x}$ and distance $f(\boldsymbol{x})$

```
 1: while $v$ is open and the number of steps is below the predefined bound do
 2:     for all $i \in \{1, \dots, m\}$ do
 3:         compute $\nabla_i f(\boldsymbol{x})$ as $\frac{|\texttt{ComputeDistance}(x_1,\dots,x_{i-1},x_i+\Delta x_i,x_{i+1},\dots,x_m)|-|f(\boldsymbol{x})|}{\Delta x_i}$
 4:     lock each $\nabla_i f(\boldsymbol{x})$ which is not finite
 5:     while $||\nabla f(\boldsymbol{x})||^2$ is finite and non-zero do
 6:         $\lambda \leftarrow |f(\boldsymbol{x})| \;/\; ||\nabla f(\boldsymbol{x})||^2$
 7:         if $\lambda$ is zero or not finite then return
 8:         $V' \leftarrow \emptyset$
 9:         for all $e \in \{0, -1, 1, -2, 2, -3, 3\}$ do
10:             $\boldsymbol{x}' \leftarrow \boldsymbol{x} - 10^e \lambda \nabla f(\boldsymbol{x})$
11:             $V' \leftarrow V' \cup \{(\boldsymbol{x}', \texttt{ComputeDistance}(\boldsymbol{x}'))\}$
12:         let $(\boldsymbol{x}', f(\boldsymbol{x}')) \in V'$ be the pair with the smallest finite $|f(\boldsymbol{x}')|$
13:         if $|f(\boldsymbol{x}')| < |f(\boldsymbol{x})|$ then
14:             $\boldsymbol{x} \leftarrow \boldsymbol{x}'$, $f(\boldsymbol{x}) \leftarrow f(\boldsymbol{x}')$
15:             break
16:         else
17:             lock all extreme coordinates $\nabla_i f(\boldsymbol{x})$
18:             if no coordiate was locked then return
```

fix the values of the inputs that were not identified as sensitive as they likely do not influence the value of the distance. In particular, we minimize the function $f(\boldsymbol{x})$ that receives an input vector of m values that correspond to sensitive inputs of the vertex v. The value of the function $f(\boldsymbol{x})$ is computed by a function `ComputeDistance`$(\boldsymbol{x})$ that:

1. Creates the input sequence *input*$'$ by replacing the sensitive inputs of the original input from *trc*(v) by the values specified in $\boldsymbol{x}$.
2. Executes the program on *input*$'$ and obtains the trace $(\mathit{usedInput}', \pi')$, where $\pi' = \langle(e'_i, c'_i, r'_i, d'_i, n'_i)\rangle_{1 \le i \le k'}$.
3. If the trace π' does not visit v, returns ∞.
4. Otherwise returns the obtained distance value at the vertex v, i.e., d'_l.

The search for the desired values of $\boldsymbol{x}$ is motivated by the following idea. If $\boldsymbol{x}$ is chosen from a small neighborhood around the global minimum of $|f(\boldsymbol{x})|$, the value $f(\boldsymbol{x})$ has roughly the same chance of being positive as negative. I.e., there is roughly the same chance of *expr*(v) being evaluated to `true` as `false`. Therefore, we repeatedly run the gradient descent from randomly chosen seeds $\boldsymbol{x}$ to approach towards the minimum. Along the way, we perform sampling in the descent direction. This sampling also helps escaping from local minima by trying more values of the function $f(\boldsymbol{x})$.

Our gradient descent starting from one random seed $\boldsymbol{x}$ is formally described in Algorithm 2. We repeatedly perform gradient descent steps from the initial seed $\boldsymbol{x}$ until we generate an input that closes the open vertex v or reach the predefined bound on gradient descent steps. In the loop at line 2, we numerically compute coordinates $\nabla_i f(\boldsymbol{x})$, one for each variable x_i, of the gradient vector

$\nabla f(\boldsymbol{x})$. The coordinates are computed using forward differences, where $\Delta x_i > 0$ is the smallest change of that variable. Since the algorithm works only with finite values, all non-finite coordinates $\nabla_i f(\boldsymbol{x})$ are locked, i.e., they are set to zero and we do not move in these coordinates in the gradient step.

The loop at line 5 performs a single gradient step. It first computes the value of learning rate λ at line 6, which has the property that the *linear approximation* of the function f at $\boldsymbol{x}$ is zero at the input $\boldsymbol{x} - \lambda \nabla f(\boldsymbol{x})$. Next we compute a set V' of samples $\boldsymbol{x}'$ (see the loop at line 9), each representing a candidate for the gradient step. Observe that the samples are separated by multipliers 10^e ranging over several orders of magnitude. These are the samples we mentioned earlier, which can both explore the small neighborhood of the global minimum and escape from local minima. Only the sample $\boldsymbol{x}'$ with the smallest $|f(\boldsymbol{x}')|$ is considered in the gradient step (see line 12). If none of the samples decreases the value of the function, we are stuck in a local minimum and try to escape it by locking more coordinates of the gradient. Namely, we identify and lock coordinates with high absolute values compared to others as they dominate the descent direction. By their locking, we can dramatically change the descent direction and potentially move towards the global minimum. If all coordinates are locked, i.e., set to zero, $||\nabla f(\boldsymbol{x})||^2 = \sum_i (\nabla_i f(\boldsymbol{x}))^2$ will be zero and the gradient descent terminates.

The gradient descent algorithm is repeatedly called with randomly chosen seed inputs $\boldsymbol{x}$ and the starting distance $f(\boldsymbol{x}) = \texttt{ComputeDistance}(\boldsymbol{x})$, until the target vertex is closed[1] or we exceed the predefined bound on the number of seeds to try. We skip all the seeds $\boldsymbol{x}$ for which $\texttt{ComputeDistance}(\boldsymbol{x})$ is infinite. More details of the algorithm can be found in the technical report [15].

5 Target Vertex Selection

We now briefly describe how we select vertices that are targeted by the analyses from the previous section. First, the heuristic tries to select a suitable *uncovered* vertex that has not been processed yet. Second, if all *uncovered* vertices have been processed, it means that none of the analyses was able to cover them. In this case, we try to select an *open* unprocessed vertex and try to close it. The detailed description of the selection process is available in the technical report [15].

5.1 Selecting an Uncovered Vertex

Primarily, we want to target uncovered vertices. Before that, we want to explore program executions with diverse numbers of loop iterations. To this end, we would like to identify all *loop head* vertices in the ABET, which can be expensive. Therefore, we perform loop head detection lazily on the fly. We maintain a worklist of loop heads H and if it is not empty, we remove its random vertex

[1] In fact, Algorithm 2 is immediately terminated when the target vertex is closed by any execution of `ComputeDistance`.

and select it as the target. Only if the worklist H is empty, we select a suitable vertex v in the tree based on *vertex selection heuristics* and detect loop heads on the path to the vertex v. If there are loop heads on the path to v, we put some of them to H based on the *loop head selection heuristics* and randomly take one of them as the target vertex. If there are no loop heads on the path to v or the loop heads on the path to v have been processed, we select v itself as the target vertex. We now describe the heuristics that we use for selection the suitable vertex v and for selection of loop heads on the path to v.

Vertex Selection Heuristics The selection relies on the classification of the uncovered vertices into three categories: *input-sensitive* vertices with $\mathit{sbytes}(v) \neq \emptyset$, *input-insensitive* vertices with $\mathit{sbytes}(v) = \emptyset$, and vertices with *unknown sensitivity*, on which the sensitivity analysis has not been performed yet. Additionally, we call a vertex *likely input-insensitive* (LII), if it has unknown sensitivity and there is an input-insensitive vertex with the same ABE and calling context in the current ABET.

The input-insensitive vertices often arise in practice. For example, when processing the loop `for (int i = 0; i < 1000; ++i)`, all the ABET vertices with the ABE `i < 1000` will be input-insensitive as the number of iterations does not depend on the input. Moreover, both byteshare and gradient descent analyses are useless on input-insensitive vertices, so we prefer not processing the LII vertices to avoid useless sensitivity computations. However LII vertices cannot be ignored completely as they can be in fact input-sensitive. For this reason, we first try selecting uncovered vertices that are either input-sensitive, or that have unknown sensitivity but are not LII. We sort such vertices lexicographically according to the following criteria and select the best vertex v.

1. Input-sensitive vertices are preferred to vertices with unknown sensitivity as we want to exploit the computed information about sensitive bytes.
2. Vertices with fewer sensitive bytes are preferred, as the analyses are more expensive with more sensitive bytes.
3. Vertices with the number of input bytes closer to the half of the maximal number of input bytes of all ABET vertices are preferred, as it helps to explore loop iterations that are deep enough to be interesting and at the same time to keep the number of input bytes reasonably small.
4. Vertices closer to the root of the execution tree are preferred, as they are easier to process.

If no such vertex exists, we fall back to choosing an LII vertex. We use the distance function to select a promising LII vertex in the following way. We select the uncovered vertex v if it is LII and all identified input-insensitive vertices with the same ABE and context have greater absolute value of the distance function. If there are more such vertices v, we first prefer the ones with the smallest absolute value of the distance function and then according to the criteria similar to the previous ones.

Loop Head Selection Heuristics To fill the worklist H, we detect all loop heads on the path to v. The identified loop heads are grouped to buckets of exponentially increasing size according to the number of bytes read from the input. This ensures that we do not process too many loop heads to make the search impractical, but we still explore loop heads with diverse depths of loop iterations. We then pick from each bucket the vertex that lexicographically minimizes the number of input bytes and the depth and add it to the worklist H.

5.2 Selecting an Open Vertex

If the previous algorithm failed to select a vertex, it means that all uncovered vertices are processed. We try to make progress by selecting a vertex that is covered but still open. The rationale is that by exploring the open vertex, albeit otherwise covered, we hope to extend the ABET with new vertices where the analyses can continue further and some open vertices might become covered.

In particular, we choose an open *input-independent* vertex with a small value of the distance function and identify an earlier loop head on the path to the root as in the previous subsection. We then perform a random ABET traversal from the loop head and select the first open unprocessed vertex for which the search tries to visit to its unvisited child. If this search fails as well, then the analysis cannot make any further progress, returns `null` and the fuzzing loop terminates.

6 Implementation

We implemented the approach in an experimental tool called FIZZER. The tool is implemented in C++, consists of around 11,000 lines of code (in 125 files), and the only external tool it depends on is the CLANG compiler and its libraries. The tool is open-source and available under ZLIB license either as an artifact at Zenodo [13] or at the repository [14].

Given a C program to be analyzed, FIZZER first compiles it into LLVM bitcode using the CLANG compiler. The bitcode is then instrumented using our instrumenter, which first applies a standard LLVM pass to replace all `switch` instructions by sequences of `if-else` statements[2] and then finds and instruments all ABEs and function calls. Observe that we ignore `br` instructions, i.e., we do not care about the actual control flow. After the instrumentation, we link the instrumented LLVM bitcode with our implementations of `nondet_`*type*`()` and `__instr_*()` functions into the final executable program, called `target`, which will be repeatedly executed by the main FIZZER process.

Whenever FIZZER wants to execute the `target` with some input, it spawns a new process with the `target` executable. During the execution of `target`, the instrumented code tracks the current call context, collects data about the executed ABEs, and stores them to the shared memory, which is accessible by the parent FIZZER process. The separation of FIZZER and `target` to independent processes allows handling crashes of the `target`.

[2] We should also replace calls via function pointers by sequences of `if-else` statements. This pass is not implemented yet.

7 Evaluation

Experimental setup. For evaluation of the implemented tool FIZZER, we use all branch-coverage benchmarks from Test-Comp 2023, the 5th Competition on Software Testing [5]. The benchmark set consists of 2933 benchmarks divided into 16 families. For the presentation purposes, *"ReachSafety"* and *"SoftwareSystems"* substrings in the family names are shortened to *"rs"* and *"ss"*, respectively, in the rest of this section. For comparison, we used three best-scoring tools[3] from Test-Comp 2023, namely FUSEBMC [2], VERIFUZZ [21], and COVERITEST [6], in the versions in which they entered Test-Comp 2023. To obtain reproducible results, we asked the organizer of Test-Comp to evaluate FIZZER on the official infrastructure of Test-Comp and compare the obtained results with the official results of Test-Comp 2023. We stress out that this means that the results *were produced by an independent third party* and thus *are independently reproducible*. The resource limits of the competition are 15 minutes of CPU time and 15 GB of RAM. A detailed description of the infrastructure and the setting used for the experimental evaluation we refer to the competition report [5].

Results. The average branch coverage for each tool and each benchmark family is shown in Table 1. The table shows that the approach proposed in this paper and implemented in the tool FIZZER is competitive with FUSEBMC – the winner of Test-Comp 2023 – in most of the benchmark families except *rs-Combinations*, *rs-ECA*, and *rs-Sequentialized*. It is also competitive with the other state-of-the-art tools on all of the benchmark families. Although the table shows that FIZZER is the best on average in benchmark families *rs-ControlFlow* and *ss-SQLite-MemSafety*, we do not consider these particular results significant due to the small size of these families.

Figure 2 provides a comparison of the branch coverage achieved by FIZZER and the other considered tools on individual benchmarks. It can be seen that while on most of the benchmarks, FIZZER provides the same or worse coverage than FUSEBMC, there are some benchmarks where it provides better coverage. It is also comparable with VERIFUZZ and provides better branch coverage than COVERITEST on a large number of benchmarks.

Out of all 2933 evaluated programs, there are 145 programs where FIZZER provides better coverage than any other of the compared tools. For comparison, COVERITEST provides the best coverage for 129 programs, FUSEBMC for 318, and VERIFUZZ for 180. The distribution of these benchmarks to the individual benchmark families can be found in Table 2.

Finally, note that FIZZER participated in Test-Comp 2024 and placed third in the category *Cover-Branches* after FUSEBMC and FUSEBMC-AI.[4]

[3] We do not compare against FUSEBMC IA [1], the runner-up in Test-Comp 2023, as we want to compare only with the best variant of each individual tool, not all their variants.

[4] `https://test-comp.sosy-lab.org/2024/results/results-verified/`

Table 1: *Average branch-coverage* of the tests generated by the individual tools for individual benchmark families and for all benchmarks. The results are in percents. The best result of each benchmark family is printed typeset in bold.

Family	Family size	CoVeriTest	Fizzer	FuSeBMC	VeriFuzz
rs-Arrays	292	71.2	84.6	**86.5**	81.6
rs-BitVectors	61	78.8	77.3	**79.5**	73.8
rs-Combinations	671	34.8	42.0	**50.7**	37.6
rs-ControlFlow	11	4.0	**14.1**	13.7	13.5
rs-ECA	29	18.3	25.1	32.3	**34.9**
rs-Floats	197	46.9	48.2	**50.8**	49.8
rs-Heap	110	68.9	72.5	**72.7**	70.6
rs-Loops	661	79.6	80.3	**82.1**	81.4
rs-ProductLines	263	29.0	28.8	**29.2**	**29.2**
rs-Recursive	51	78.4	84.2	**85.8**	76.0
rs-Sequentialized	91	80.4	66.5	87.8	**88.4**
rs-XCSP	114	**99.8**	88.5	91.7	92.6
ss-BusyBox-MemSafety	62	16.9	32.9	**33.2**	0.0
ss-DeviceDriversLinux64-rs	287	**20.6**	20.5	**20.6**	19.7
ss-SQLite-MemSafety	1	0.0	**3.7**	3.4	3.5
Termination-MainHeap	32	**95.6**	95.3	95.1	90.9
All	2933	54.3	57.3	**61.0**	56.2

8 Related Work

The sensitivity analysis is a form of taint analysis, which is a technique popular in fuzzing [17,8,22,9,4,10,12,23,25,7,19,26]. The most frequent approach to taint analysis is propagating the taint information explicitly from taint sources (e.g., sources of input) through the program instructions [17,8,22,9,4,10,12,23,25]. Most of the approaches propagate taint information dynamically. However, some of them compute it statically [23], with use of control flow information [25], or using concrete and symbolic execution [7]. There are two papers [19,26] that compute the tainted bytes by identifying input bytes that lead to different program executions. This is most similar to our approach. But our approach also tries extreme values of typed inputs and performs more precise one-bit mutations, which are then extended to byte boundaries, while the mentioned papers [19,26] only mutate whole bytes.

Gradient descent is used in fuzzing [8,26,9,16,24] in different forms. For instance, there is a paper [8] that uses forward and backward method of finite differences for computation of the partial derivatives. Additional constraints appearing in the control flow have been also considered [9]. Another approach [16]

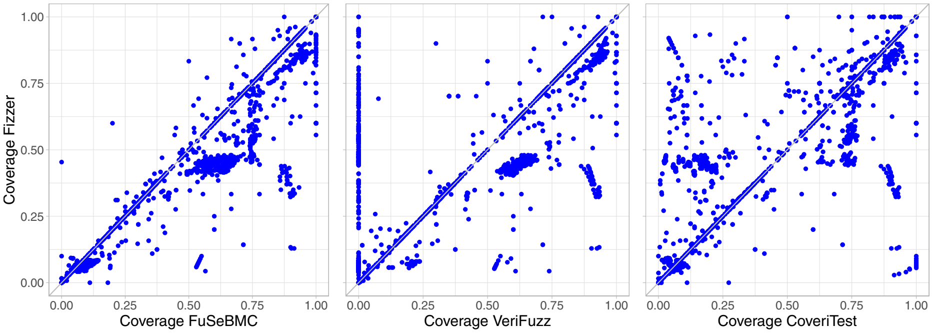


Fig. 2: Scatter plots comparing branch coverage achieved by Fizzer and the other considered tools.

exponentially decreases the learning rate λ as the gradient descent progresses. Our approach differs especially in taking multiple samples along the gradient direction in each descent step. The samples span several orders of magnitude along the line, which can both provide samples in the small region close to the global minimum and help escaping from local minima. We further compute the learning rate λ from the linear approximation of the function. Thanks to multi-sampling, this simplification is sufficient in practice. Lastly, our approach is extended with locking coordinates, which can contribute to escaping from local minima by avoiding extreme directions.

Our approach further uses a unique coverage goal. Other fuzzers monitor actual control flow of the program execution (to measure, e.g., branch coverage), while we ignore it completely. We instead monitor values of all abes and aim for their coverage. The byteshare analysis is also a novel approach inspired by genetic algorithms. The random search we apply to select the target vertex is novel among fuzzers, but it was used in the context of concolic execution [20].

The experimental evaluation of the paper compares the proposed approach with the best test-generation tools participating in Test-Comp 2023. All of these combine several analyses. Namely, FuSeBMC [2,3] combines bounded model-checking (bmc), symbolic execution, and two fuzzers (afl [27] and a selective fuzzer) and FuSeBMC IA [1] extends it further with interval analysis. VeriFuzz [21] is built on top of afl and an engine based on Coverage Guided Fuzzing, combined with the bounded model checker cbmc and the prism framework. CoVeriTest [6] combines several model checkers.

9 Conclusion

We presented a novel approach to gray-box fuzzing, which aims to generate tests that cover both possible values of each atomic Boolean expression. To reach this goal, our approach uses a dynamic computation to identify the bytes that influence the value of a given Boolean expression. Further, it employs two

Table 2: The numbers of benchmarks in individual benchmarks families where a given tool achieved better branch coverage than the other considered tools.

Family	CoVeriTest	Fizzer	FuSeBMC	VeriFuzz
ReachSafety-Arrays	0	12	18	4
ReachSafety-BitVectors	0	3	1	2
ReachSafety-Combinations	96	23	243	139
ReachSafety-ControlFlow	2	1	1	1
ReachSafety-ECA	1	1	6	15
ReachSafety-Floats	0	16	1	0
ReachSafety-Heap	1	8	0	2
ReachSafety-Loops	0	6	6	0
ReachSafety-ProductLines	0	33	0	3
ReachSafety-Recursive	0	1	4	0
ReachSafety-Sequentialized	0	2	16	14
ReachSafety-XCSP	14	0	0	0
SoftwareSystems-BusyBox-MemSafety	4	27	19	0
SoftwareSystems-DeviceDriversLinux64-ReachSafety	9	11	3	0
SoftwareSystems-SQLite-MemSafety	0	1	0	0
Termination-MainHeap	2	0	0	0
All	129	145	318	180

analyses to find the value of these bytes to get the desired value of the Boolean expression. One of these analyses is based on gradient descent.

We implemented the proposed approach in an experimental tool called Fizzer. An independent evaluation shows that, despite being a pure gray-box fuzzer, it is competitive with the state-of-the-art tools competing in Test-Comp 2023.

In future, we plan to add the support for calls via function pointers and gradient descent tailored for floating-point values. We will also investigate an extensible architecture that allows running different external analyses on the vertices of the execution tree. In particular, this would allow running techniques such as symbolic execution on vertices that cannot be covered by gradient descent alone, which could improve the performance of our tool even further.

Acknowledgement

The authors would like to thank Dirk Beyer for running the experiments on the original Test-Comp infrastructure and for his technical assistance.

References

1. Aldughaim, M., Alshmrany, K.M., Gadelha, M.R., de Freitas, R., Cordeiro, L.C.: FuSeBMC_IA: Interval analysis and methods for test case generation (competition contribution). In: Lambers, L., Uchitel, S. (eds.) Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 324–329. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_18, https://doi.org/10.1007/978-3-031-30826-0_18
2. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Loulergue, F., Wotawa, F. (eds.) Tests and Proofs - 15th International Conference, TAP 2021, Held as Part of STAF 2021, Virtual Event, June 21-22, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12740, pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6, https://doi.org/10.1007/978-3-030-79379-1_6
3. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC v4: Smart seed generation for hybrid fuzzing. In: Johnsen, E.B., Wimmer, M. (eds.) Fundamental Approaches to Software Engineering. pp. 336–340. Springer International Publishing, Cham (2022)
4. Bekrar, S., Bekrar, C., Groz, R., Mounier, L.: A taint based approach for smart fuzzing. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. p. 818–825. ICST '12, IEEE Computer Society, USA (2012). https://doi.org/10.1109/ICST.2012.182, https://doi.org/10.1109/ICST.2012.182
5. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Lambers, L., Uchitel, S. (eds.) Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 309–323. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_17, https://doi.org/10.1007/978-3-031-30826-0_17
6. Beyer, D., Jakobs, M.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transf. **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8, https://doi.org/10.1007/s10009-020-00587-8
7. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy. pp. 725–741 (2015). https://doi.org/10.1109/SP.2015.50
8. Chen, P., Chen, H.: Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 711–725 (2018). https://doi.org/10.1109/SP.2018.00046
9. Chen, P., Liu, J., Chen, H.: Matryoshka: Fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 499–513. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363225, https://doi.org/10.1145/3319535.3363225
10. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering. p. 474–484. ICSE '09, IEEE Computer Society, USA (2009). https://doi.org/10.1109/ICSE.2009.5070546, https://doi.org/10.1109/ICSE.2009.5070546

11. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Communications of the ACM **55**(3), 40–44 (2012)
12. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proceedings of the 22nd USENIX Conference on Security. p. 49–64. SEC'13, USENIX Association, USA (2013)
13. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Fizzer: Artifact for TACAS 2024 evaluation (Dec 2023). https://doi.org/10.5281/zenodo.10440311
14. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Fizzer: Git repository (2023), https://github.com/staticafi/sbt-fizzer
15. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Gray-box fuzzing via gradient descent and Boolean expression coverage. Tech. rep., Masaryk University, Brno (2024), https://arxiv.org/abs/2401.12643
16. Kim, Y., Yoon, J.: Maxafl: Maximizing code coverage with a gradient-based optimization technique. Electronics **10**(1) (2021). https://doi.org/10.3390/electronics10010011, https://www.mdpi.com/2079-9292/10/1/11
17. Liang, G., Liao, L., Xu, X., Du, J., Li, G., Zhao, H.: Effective fuzzing based on dynamic taint analysis. In: 2013 Ninth International Conference on Computational Intelligence and Security. pp. 615–619 (2013). https://doi.org/10.1109/CIS.2013.135
18. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. IEEE Transactions on Reliability **67**(3), 1199–1218 (2018). https://doi.org/10.1109/TR.2018.2834476
19. Liang, J., Wang, M., Zhou, C., Wu, Z., Jiang, Y., Liu, J., Liu, Z., Sun, J.: PATA: Fuzzing with path aware taint analysis. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1–17 (2022). https://doi.org/10.1109/SP46214.2022.9833594
20. Liu, D., Ernst, G., Murray, T., Rubinstein, B.I.P.: Legion: Best-first concolic testing. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. p. 54–65. ASE '20, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3324884.3416629, https://doi.org/10.1145/3324884.3416629
21. Metta, R., Yeduru, P., Karmarkar, H., Medicherla, R.K.: VeriFuzz 1.4: Checking for (non-)termination (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 594–599. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_42, https://doi.org/10.1007/978-3-031-30820-8_42
22. Paduraru, C., Melemciuc, M.C., Ghimis, B.: Fuzz testing with dynamic taint analysis based tools for faster code coverage. In: Proceedings of the 14th International Conference on Software Technologies. p. 82–93. ICSOFT 2019, SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT (2019). https://doi.org/10.5220/0007921300820093, https://doi.org/10.5220/0007921300820093
23. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: Application-aware evolutionary fuzzing. In: NDSS. vol. 17, pp. 1–14 (2017)
24. She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S.: Neuzz: Efficient fuzzing with neural program smoothing. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 803–817. IEEE (2019)

25. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy. pp. 497–512 (2010). https://doi.org/10.1109/SP.2010.37
26. You, W., Liu, X., Ma, S., Perry, D., Zhang, X., Liang, B.: SLF: Fuzzing without valid seed inputs. In: Proceedings of the 41st International Conference on Software Engineering. p. 712–723. ICSE '19, IEEE Press (2019). https://doi.org/10.1109/ICSE.2019.00080, https://doi.org/10.1109/ICSE.2019.00080
27. Zalewski, M.: American fuzzy lop (2013), http://lcamtuf.coredump.cx/afl/.

Fast Symbolic Computation of Bottom SCCs

Anna Blume Jakobsen, Rasmus Skibdahl Melanchton Jørgensen, Jaco van de Pol, and Andreas Pavlogiannis(✉)

Aarhus University, Aarhus, Denmark
{jaco,pavlogiannis}@cs.au.dk

Abstract. The computation of bottom strongly connected components (BSCCs) is a fundamental task in model checking, as well as in characterizing the attractors of dynamical systems. As such, symbolic algorithms for BSCCs have received special attention, and are based on the idea that the computation of an SCC can be stopped early, as soon as it is deemed to be non-bottom.

In this paper we introduce PENDANT, a new symbolic algorithm for computing BSCCs which runs in linear symbolic time. In contrast to the standard approach of *escaping* non-bottom SCCs, PENDANT aims to *start* the computation from nodes that are likely to belong to BSCCs, and thus is more effective in sidestepping SCCs that are non-bottom. Moreover, we employ a simple yet powerful *deadlock-detection* technique, that quickly identifies singleton BSCCs before the main algorithm is run. Our experimental evaluation on three diverse datasets of 553 models demonstrates the efficacy of our two methods: PENDANT is decisively faster than the standard existing algorithm for BSCC computation, while deadlock detection improves the performance of each algorithm significantly.

Keywords: BDDs · strongly connected components · symbolic algorithms

1 Introduction

The decomposition of a graph to its *strongly connected components* (*SCCs*) is one of the most standard tasks in automated system verification. For example, model checking against LTL and ω-regular properties reduces to computing cycles [30], while fairness conditions are typically checked given an SCC decomposition of the graph [21,34]. Of special interest are *bottom/terminal SCCs* (or *BSCCs*), i.e., SCCs that, once entered, cannot be escaped. BSCCs are used to speed up LTL model checking [28], and they capture the long-run properties of Markov Chains [4,11] and Markov Decision Processes [23,13], while they also correspond to the attractors of dynamical systems, as in signal transduction networks [29,33].

Large-scale model-checking settings comprise huge systems that suffer from the state-space explosion problem. These systems are usually represented compactly by a model, e.g., by means of a programming language, a logic or a reaction network, and have size that is exponentially large in its description. Nevertheless, the system typically exhibits numerous symmetries that can be preserved when

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 110–128, 2024.
https://doi.org/10.1007/978-3-031-57256-2_6

the state space is represented *symbolically* rather than *explicitly*. One predominant symbolic representation is via (reduced/ordered) Binary Decision Diagrams (BDDs) [10], which are found at the core of many classic and modern model checkers [14,24,20,26,5]. To benefit from the symbolic representation, analysis algorithms typically only have *coarse-grained* access to the graph, querying for the *successors* ($\mathrm{Post}(X)$) and *predecessors* ($\mathrm{Pre}(X)$) of a set of nodes X represented by a single BDD. Each such operation counts as a *symbolic step*. As symbolic steps are significantly slower than primitive operations, they serve as the complexity measure of symbolic algorithms [9,18,12,25].

Due to the prevalence of SCC decomposition, the problem has been studied extensively in the symbolic setting, starting with the Xie-Beerel algorithm [32] of symbolic complexity $O(n^2)$; LockStep [8] improves this bound to $O(n \log n)$, while Skeleton [17] achieves $O(n)$ time at the expense of $\Theta(n)$ symbolic space (i.e., number of BDDs). The most recent step in this progression is Chain [25] which achieves both $O(n)$ symbolic time and $O(\log n)$ symbolic space. In practice, heuristics aim to further improve the running time [31,16,34].

Naturally, the computation of BSCCs can be achieved by using one of the aforementioned algorithms to obtain an SCC decomposition, and check whether each SCC is indeed a BSCC. In practice, however, computing an SCC can be expensive, as it typically requires traversing it multiple times. For this reason, algorithms dedicated to BSCCs have received special attention. Although these do not offer theoretical improvements, they attempt to minimize the number of non-bottom SCCs computed and thus perform better in practice.

The predominant, general-purpose BSCC-decomposition algorithm is BwdFwd, which is a modification of Xie-Beerel [32], and has $O(n)$ complexity. Effectively, this algorithm aborts the computation of an SCC S as soon as it determines that S cannot be a BSCC, and removes it from the graph, as well as any node that can reach S. A recently-introduced preprocessing technique, called interleaved transition-guided reduction (ITGR) [6], aims to further detect and discard non-bottom SCCs before the main algorithm is run. ITGR is general-purpose, and was shown to be effective in handling asynchronous Boolean Network models [3,1,2]. However, as these algorithms are typically executed on huge inputs, issues of scalability often remain. We address this challenge here.

1.1 Our contributions

The Pendant algorithm. We develop a new, linear-time algorithm for symbolic BSCC computation, called Pendant, drawing inspiration from the recent Chain algorithm [25]. In contrast to the existing BSCC paradigm based on *stopping* the computation of SCCs that are deemed non-bottom, Pendant aims to *start* such computations from SCCs that are likely to be bottom. To achieve this, while Pendant computes an SCC, it also implicitly (at no extra cost) traverses the quotient graph Q downwards, making future SCC computations start from nodes that are close to the bottom of Q, and thus discover a BSCC quickly.

Deadlock detection. We employ a simple yet powerful preprocessing technique, called *deadlock-detection*. This is based on the insights that (i) each deadlock (singleton SCC) is a BSCC, and (ii) *all* deadlocks can be computed effectively in a *single* symbolic step.

Experimental evaluation. We implement PENDANT and the deadlock-detection preprocessing, and evaluate their performance on computing the BSCCs of a large pool of models from three diverse datasets, namely, (i) Petri Nets from the Model Checking Contest [22], (ii) DiVinE models from the Benchmark of Explicit Models [27], and (iii) Asynchronous Boolean Network models [3,1,2]. Our experiments conclude that (i) PENDANT is decisively more efficient than BWDFWD, (ii) deadlock-detection improves the performance of both algorithms, and (iii) after deadlock-detection, ITGR is scarcely effective.

2 Preliminaries

In this section we present standard definitions and the BWDFWD algorithm.

2.1 Graphs, Bottom SCCs and Symbolic Representations

Graphs. We consider directed graphs $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. We often write $u \to v$ to denote an edge $(u, v) \in E$. For a node v, the image of v is $\text{Post}(v) = \{u \mid v \to u\}$, while the pre-image of v is $\text{Pre}(v) = \{u \mid u \to v\}$. These notions are extended to sets of nodes X in the natural way, i.e., $\text{Post}(X) = \bigcup_{v \in X} \text{Post}(v)$ and $\text{Pre}(X) = \bigcup_{v \in X} \text{Pre}(v)$.

A path is a sequence $P = v_1 \to v_2 \to \cdots \to v_k$, in which case we also write $v_1 \rightsquigarrow v_k$, and say that v_k is reachable from v_1. The length of P is $|P| = k - 1$. For a set of nodes X we write $\text{Fwd}(X) = \{u \mid \exists v \in X, v \rightsquigarrow u\}$ for the forward set of X and $\text{Bwd}(X) = \{u \mid \exists v \in X, u \rightsquigarrow v\}$ for the backward set of X. We call a set $X \subseteq V$ *forward-closed* if $\text{Fwd}(X) \subseteq X$. The restriction of G on a set $X \subseteq V$ is the graph $G[X] = (X, (X \times X) \cap E)$. A node $v \in V$ is called a *deadlock* if it has no outgoing edges, i.e., $\text{Post}(v) = \emptyset$.

Bottom Strongly Connected Components (BSCCs). A *strongly connected component* (*SCC*) of G is a maximal set of nodes S such that for all $u, v \in S$ we have $u \rightsquigarrow v$. Each node v belongs to one SCC, written $\text{SCC}(v)$. A set $X \subseteq V$ is called *SCC-closed* if for each $v \in X$, we have $\text{SCC}(v) \subseteq X$. The *diameter* of an SCC S is the maximum distance between two nodes in S, i.e.,

$$\delta(S) = \max_{u,v \in S} \min_{P\colon u \rightsquigarrow v} |P|$$

The quotient graph of G represents each SCC of G by a single node, and has a directed edge $S \to S'$ iff $\text{Post}(S) \cap S' \neq \emptyset$, i.e., there exists nodes $u \in S$ and $v \in S'$ with $u \to v$. The quotient graph is a directed acyclic graph. The leaf nodes of a quotient graph represent the SCCs that have no outgoing edges to any other SCCs, called *bottom SCCS* (or *BSCCs*). We denote by SCCs(G) and BSCCs(G) the set of SCCs and BSCCs of G, respectively.

The problem targeted in this paper is the computation of BSSCs. The following two simple properties of BSCCs are used throughout the paper.

Proposition 1. *An SCC S is a BSCC if and only if* $\mathrm{Fwd}(S) = S$.

Proposition 2. *If S is a BSCC then there is no BSCC in* $\mathrm{Bwd}(S) \setminus S$.

Symbolic operations and complexity. In large-scale model-checking settings, graphs are typically represented symbolically. One popular symbolic representation is Binary Decision Diagrams (BDDs) [19]. In particular, the node set V and edge relation E are represented compactly as BDDs, while algorithms use BDDs as data structures for representing subsets of V and E. The basic BDD operations give only coarse-grained access to the graph: given a BDD representing a set of nodes X, an algorithm can access $\mathrm{Pre}(X)$ and $\mathrm{Post}(X)$, each of which counts as one *symbolic step*. The complexity of symbolic algorithms is measured in the number of symbolic steps they execute [12,25], since these are much slower than elementary operations (e.g., incrementing a counter). Basic set operations on BDDs (union, intersection, etc.) also do not count towards the time complexity*. Finally, given a set X represented as a BDD, we use a Pick(X) operation which returns an arbitrary node $v \in X$. This operation is natural and efficient for BDDs, and has been common in symbolic SCC algorithms [17,8,25].

2.2 The BwdFwd Algorithm for BSCCs

The symbolic computation of BSCCs(G) can be performed by computing each $S \in \mathrm{SCCs}(G)$ using some existing symbolic algorithm [32,17,8,25], and then reporting that S is a BSCC iff $\mathrm{Post}(S) \subseteq S$ (following Proposition 1). Although this approach runs in $O(n)$ symbolic steps when using Chain [25] or Skeleton [17], it can be unnecessarily slow in practice, as it typically spends considerable time computing SCCs that are not BSCCs. For this reason, the computation of BSCCs is targeted by algorithms dedicated to this task. The standard symbolic BSCC algorithm is BwdFwd, which we briefly present here.

The Backward-Forward BSCC algorithm. BwdFwd is an adaptation of the standard Xie-Beerel algorithm [32]. Algorithm 2 follows its recent presentation in [6], adapted to our setting. The algorithm uses the standard mechanism for computing SCCs symbolically: given a pivot node v, we have $\mathrm{SCC}(v) = \mathrm{Fwd}(v) \cap \mathrm{Bwd}(v)$. Given such a node v, BwdFwd first calls Algorithm 1 (Line 3) to retrieve the backward set $\mathrm{Bwd}(v)$ (called the *basin* of v) using a standard fixpoint computation. Then, it uses a similar fixpoint computation to retrieve $\mathrm{Fwd}(v)$ (Line 5) in F. This computation is terminated early

*For many algorithms, including ours, counting set operations does not affect the asymptotic complexity.

Algorithm 1: Bwd

```
   Input: A graph $G = (V, E)$ and a node $v \in V$
 1 $B = \{v\}$
 2 while $\mathrm{Pre}(B) \not\subseteq B$ do                  // Fixpoint not reached
 3     $B = B \cup \mathrm{Pre}(B)$                // Update with new predecessors
 4 return $B$
```

Algorithm 2: BwdFwd

```
   Input: A graph $G = (V, E)$
 1 if $V = \emptyset$ then return
 2 $v = \textsc{Pick}(V)$                                    // Pick a pivot
 3 $B = \textsc{Bwd}(G, v)$                     // Compute safe-to-remove nodes
 4 $F = \emptyset; \mathrm{Layer} = \{v\}$
 5 while $\mathrm{Layer} \neq \emptyset$ and $F \subseteq B$ do          // Compute and detect BSCC
 6     $F = F \cup \mathrm{Layer}$
 7     $\mathrm{Layer} = \mathrm{Post}(\mathrm{Layer}) \setminus F$
 8 if $F \subseteq B$ then                                   // Output if BSCC
 9     output Fwd
10 $\textsc{BwdFwd}(G[V \setminus B])$                // Recursive call w/o safe nodes
```

if the algorithm discovers that Fwd$(v) \not\subseteq$ Bwd(v), as then Fwd$(v) \not\subseteq$ SCC(v), and due to Proposition 1, we have that SCC(v) is not a BSCC. On the other hand, if the computation is carried to a fixpoint, we have that Fwd$(v) \subseteq$ Bwd(v) and thus Fwd$(v) =$ SCC(v); then, Proposition 1 guarantees that SCC(v) is a BSCC. Since the check in Line 9 succeeds, BwdFwd correctly outputs SCC(v) as a BSCC. Finally, Proposition 2 guarantees that the basin Bwd(v) contains no BSCC, except possibly SCC(v) which was just outputted. The algorithm hence safely removes Bwd(v) from G, and proceeds recursively (Line 10).

It is not hard to see that BwdFwd runs in $O(n)$ symbolic steps, but offers two practical improvements over general SCC-decomposition algorithms. In each recursive call, the algorithm avoids computing SCCs in Bwd$(v) \setminus$ SCC(v) as they are guaranteed to be non-bottom; nodes in this set are only accessed during the basin computation in Algorithm 1, which is cheaper. Moreover, it stops computing SCC(v) as soon as it discovers that Fwd$(v) \not\subseteq$ Bwd(v) (as SCC(v) is not a BSCC). However, the algorithm can spend significant time in computing Fwd(v) before it discovers that Fwd$(v) \not\subseteq$ Bwd(v), which results in wasteful symbolic operations. The following example illustrates this issue on a small graph.

Example. Fig. 1 shows a graph $G = (V, E)$ (a) and two recursion trees. The left-most tree (b) illustrates the execution of BwdFwd on G. Each node in the tree has its variables subscripted by the pivot node v chosen in the corresponding recursive call, with the variables showing their values in that recursive call. E.g.,

F_v is the value of F after the loop of Line 5 has completed, given that v was chosen as pivot in that recursive call. The number of a node is underlined in F_v if it is a node is outside the backward set B_v and cuts the computation of F_v short (Line 5). Observe that the algorithm makes four recursive calls, where the second ($v = 2$) and third ($v = 3$) call spend considerable time in the forward computation (of the sets F_2 and F_3, respectively), and essentially compute SCC(2) and SCC(3) before determining that these are not BSCCs.

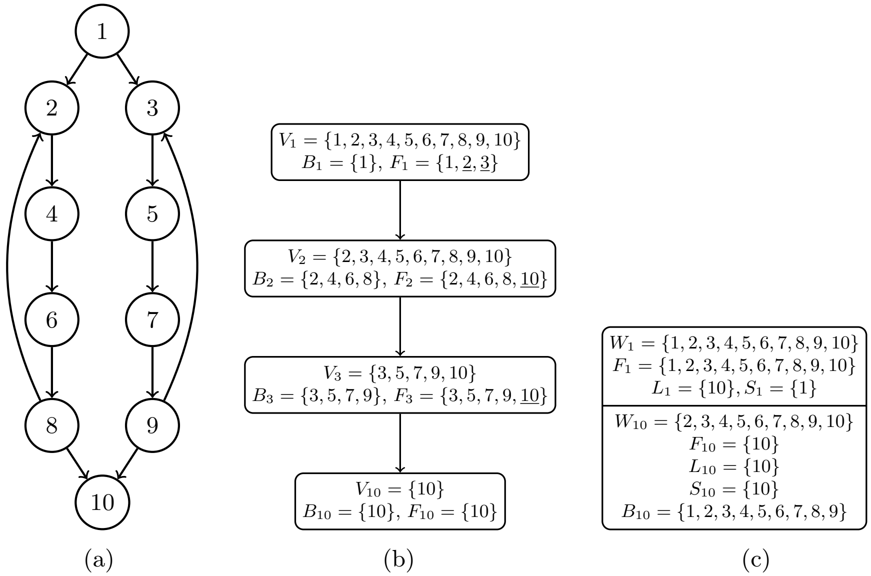

1
2
3
4
5
6
7
8
9
10
(a)
$V_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$B_1 = \{1\}$, $F_1 = \{1, \underline{2}, \underline{3}\}$
$V_2 = \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$B_2 = \{2, 4, 6, 8\}$, $F_2 = \{2, 4, 6, 8, \underline{10}\}$
$V_3 = \{3, 5, 7, 9, 10\}$
$B_3 = \{3, 5, 7, 9\}$, $F_3 = \{3, 5, 7, 9, \underline{10}\}$
$V_{10} = \{10\}$
$B_{10} = \{10\}$, $F_{10} = \{10\}$
(b)
$W_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$F_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$L_1 = \{10\}, S_1 = \{1\}$
$W_{10} = \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$F_{10} = \{10\}$
$L_{10} = \{10\}$
$S_{10} = \{10\}$
$B_{10} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
(c)

Fig. 1: An example input graph (a) and the recursion trees of the BwdFwd (b) and Pendant (c) algorithms on it.

3 The Pendant Algorithm for BSCCs

In this section we present our new algorithm, Pendant, for computing BSCCs symbolically. Like BwdFwd, Pendant spends linear time in the number of nodes of the input graph. In particular, we have the following theorem.

Theorem 1. *Given a graph* $G = (V, E)$ *of* n *nodes,* Pendant *computes BSCCs(G) in* $O(\sum_{S \in \mathrm{SCCs}(G)} \delta(S)) = O(n)$ *symbolic time.*

However, as we will see in Section 5, in practice Pendant typically requires fewer symbolic steps than BwdFwd. Intuitively, this is achieved by making, over time, smarter choices of pivot nodes v to start the SCC computation, meaning nodes v that are more likely to have SCC(v) close to the leaves of the quotient graph.

In turn, this reduces the number of non-bottom SCCs computed throughout the execution of the algorithm, which reduces the number of symbolic steps.

3.1 Pendant

Pendant is shown in Algorithm 4, and uses FwdLastLayer, shown in Algorithm 3, as a sub-procedure.

Algorithm 3: FwdLastLayer

```
Input: A graph $G = (V, E)$ and a node $v \in V$
$F = \emptyset$; Layer $= \{v\}$; $L = \emptyset$
while Layer $\neq \emptyset$ do                          // Fixpoint not reached
    $F = F \cup$ Layer                                  // Update with new successors
    $L =$ Layer                      // $L$ stores the last layer of nodes reached
    Layer $=$ Post (Layer) $\setminus F$                      // Compute the new layer
return $F, L$
```

FwdLastLayer. FwdLastLayer computes the forward set Fwd(v) of a node v using a standard fixpoint computation. The algorithm also keeps track of the last layer L of nodes discovered during the fixpoint computation, and returns both Fwd(v) (represented in F) and L. Intuitively, Fwd(v) is used by Pendant for computing SCC(v) and testing whether it is a BSCC, while L is used to guide the selection of future pivots downwards in the quotient graph.

Pendant. On input $G = (V, E)$, Pendant begins by Pick'ing an arbitrary pivot node v (Line 2), with the aim to compute SCC(v) and test whether it is a BSCC. For this purpose, it calls FwdLastLayer to retrieve $F = \text{Fwd}(v)$, and L being the last layer of Fwd(v) (Line 5). It then computes $S = \text{SCC}(v)$, by calling Bwd (Algorithm 1, Line 6) to compute the backward set of v restricted to Fwd(v). At this point, there are two cases.

– If $F \setminus S \neq \emptyset$, then S is not a BSCC. At this point, the set $W = F \setminus S$ is guaranteed to contain a BSCC, and the algorithm resumes its search for a BSCC in this set, running a new iteration of the main loop. Moreover, the algorithm attempts to pick a new pivot in the last layer of Fwd(v) (Line 10), as opposed to an arbitrary node in W. Intuitively, this effectively allows Pendant to traverse the quotient graph downwards towards its leaves, and thus quickly pick a pivot v such that SCC(v) is a BSCC.
– If $F \setminus S = \emptyset$, then $D = \text{SCC}(v)$ is guaranteed to be a BSCC; this is reported (Line 15), and the loop breaks (Line 4). Then the backwards set of B is computed and removed from the graph, as it is guaranteed to not contain any other BSCC, and the algorithm proceeds recursively in the remaining graph (Line 17). Note that the number of recursive calls of Pendant thus equals the number of BSCCs in the input graph.

Algorithm 4: PENDANT

```
   Input: A graph $G = (V, E)$
1  if $V = \emptyset$ then return
2  $v = $ PICK $(V)$                                   // Pick a pivot
3  $W = V; D = \emptyset$                              // $D$ stores a BSCC, once found
4  while $D = \emptyset$ do                            // Find a BSCC
5     $F, L = $ FWDLASTLAYER $(G[W], v)$               // Get $\mathrm{Fwd}(v)$ and its last layer
6     $S = $ BWD $(G[F], v)$                           // Compute $\mathrm{SCC}(v)$
7     if $F \setminus S \neq \emptyset$ then           // Not a BSCC
8        $W = F \setminus S$                           // $W$ contains a BSCC, continue here
9        if $L \cap W \neq \emptyset$ then             // If there are candidates in last layer,
10          $v = $ PICK $(L \cap W)$                   // pick new pivot from the last layer
11       else
12          $v = $ PICK $(W)$                          // otherwise, pick any $v$ from $W$
13    else
14       $D = S$
15       output $D$
16 $B = $ BWD $(D, G)$                                 // Compute safe-to-remove nodes
17 PENDANT $(G[V \setminus B])$                        // Recursive call w/o safe nodes
```

Observe the qualitative differences between PENDANT and BWDFWD. First, BWDFWD begins with a backward search from the pivot v, while PENDANT begins with a forward search from v. Second, BWDFWD removes the basin $\mathrm{Bwd}(v)$ from G as soon as $\mathrm{SCC}(v)$ is deemed to be non-bottom, while PENDANT delays this step, and only computes (and removes) the basin of BSCCs. Third, BWDFWD picks pivots completely arbitrarily, whereas PENDANT, any time it computes an SCC S that is not bottom, it picks the next pivot from a distant successor of S in the quotient graph, which allows it to discover BSCCs quickly.

Example. Let us revisit our example in Fig. 1. The right-most recursion tree (c) illustrates the computation of PENDANT. Since there is only one BSCC, there is only one recursive call, but the node is subdivided to show each iteration of the loop in Line 4. As before, variables are subscripted with the pivot node v of that iteration. Initially, PENDANT chooses arbitrarily $v = 1$, like BWDFWD, and computes $\mathrm{Fwd}(1)$. Then, it deems $\mathrm{SCC}(1)$ as a non-bottom SCC, and the next pivot is chosen from the last layer of $\mathrm{Fwd}(1)$, i.e., $v = 10$. Effectively, PENDANT has reached a leaf of the quotient graph (the only leaf, in this case), and thus identifies a BSCC quickly. Importantly, it skips the expensive computation of two SCCs with large diameters ($\mathrm{SCC}(2)$ and $\mathrm{SCC}(3)$), in contrast to BWDFWD.

3.2 Correctness

We now turn our attention to the correctness of PENDANT. We start with two simple lemmas regarding forward-closed sets.

Lemma 1. *Assume that $X \subseteq V$ is forward-closed, and $D \subseteq X$ is a BSCC. Then $X \setminus \mathrm{Bwd}(D)$ is forward-closed.*

Proof. For any node $v \in X$, if $\mathrm{Fwd}(v) \cap \mathrm{Bwd}(D) \neq \emptyset$ then clearly $v \in \mathrm{Bwd}(D)$ and hence $v \notin X \setminus \mathrm{Bwd}(D)$. Thus, for every node $v \in X \setminus \mathrm{Bwd}(D)$, we have $\mathrm{Fwd}(v) \cap \mathrm{Bwd}(D) = \emptyset$, and since X is forward-closed, we have $\mathrm{Fwd}(v) \subseteq X$. □

Lemma 2. *For any node v, the set $\mathrm{Fwd}(v) \setminus \mathrm{SCC}(v)$ is forward-closed.*

Proof. For any node $u \in \mathrm{Fwd}(v)$, if $\mathrm{Fwd}(u) \cap \mathrm{SCC}(v) \neq \emptyset$, then $u \in \mathrm{SCC}(v)$. Hence for every node $u \in \mathrm{Fwd}(v) \setminus \mathrm{SCC}(v)$, we have $\mathrm{Fwd}(u) \cap \mathrm{SCC}(v) = \emptyset$, and thus $\mathrm{Fwd}(u) \subseteq \mathrm{Fwd}(v) \setminus \mathrm{SCC}(v)$. The desired result follows. □

We now prove the soundness of Pendant, i.e., every SCC outputted in Line 15 is a BSCC. For this, we prove the following stronger lemma, which states three invariants maintained by the algorithm.

Lemma 3. *At each iteration of the main loop of* Pendant*, the following invariants hold: (a) V and W are forward-closed, (b) S is an SCC, and (c) D is a BSCC.*

Proof. Before entering the first iteration of the loop, we have that each of W and V is the whole node set of the input graph, hence both are trivially forward-closed. Now, assuming that W is forward-closed, we have that $F = \mathrm{Fwd}(v)$ in Line 5. In turn, this implies that $S = \mathrm{SCC}(v)$ in Line 6. Moreover, due to Proposition 1, if $F \subseteq S$ in Line 7, then S is a BSCC, thus D outputted in Line 15 is indeed a BSCC.

To complete the invariant proof, it remains to argue that V' and W' remain forward-closed after they have been updated. There are two cases.

1. If the algorithm proceeds with another iteration of the main loop, we have $V' = V$ and $W' = F \setminus S$. Since $F = \mathrm{Fwd}(v)$ and $S = \mathrm{SCC}(v)$, Lemma 2 implies that W' is forward-closed.
2. Otherwise, the algorithm proceeds with a new recursive call in Line 17. We have that $W' = V' = V \setminus B$, where $B = \mathrm{Bwd}(D)$, and D is a BSCC. By Lemma 1, we have that $V \setminus B$ is forward-closed, as desired. □

Observe that case (c) of Lemma 3 establishes the soundness of Pendant. Next we establish its completeness, thereby concluding the correctness of Pendant.

Lemma 4. Pendant *outputs every BSCC of the input graph.*

Proof. First, observe every time Pendant calls itself recursively in Line 17, it has outputted a BSCC D, and the recursion proceeds on the subgraph $V \setminus \mathrm{Bwd}(D)$. Due to Proposition 2, the algorithm has outputted all BSCCs in $V \setminus$

Bwd(D). Hence, in each recursive call on a graph $G = (V, E)$, the node set V contains all the BSCCs not already outputted by the algorithm. It thus suffices to argue that, in each recursive call, the main loop eventually terminates, as in doing so it outputs a BSCC.

In each iteration of the main loop, the set W is updated to $W' = F \setminus S$ (Line 8), where $F = \text{Fwd}(v)$ and $S = \text{SCC}(v)$, where v is the current pivot. Since $F \subseteq W$ and $S \neq \emptyset$, it follows that $W' \subsetneq W$, and thus the loop must eventually terminate. □

3.3 Complexity

Although the linear upper-bound of BwdFwd is trivial, the case of Pendant is more involved. This is because a call to FwdLastLayer may compute forward sets that consist of many layers (and thus cost many symbolic steps), while these sets are not immediately removed from the graph (as opposed to the backward set computed by BwdFwd), and are again accessed in future iterations of the algorithm. Nevertheless, a careful analysis shows that the complexity is indeed linear. We start with a simple lemma.

Lemma 5. *Assume that $X \subseteq V$ is forward-closed and $D \subseteq X$ is a BSCC. Then* $\text{Bwd}(D) \cap X$ *is SCC-closed.*

Proof. Consider any node $v \in \text{Bwd}(D) \cap X$. Since X is forward-closed, we have $\text{Fwd}(v) \subseteq X$ and thus $\text{SCC}(v) \subseteq X$. Moreover, $\text{Bwd}(v) \subseteq \text{Bwd}(D)$ and thus $\text{SCC}(v) \subseteq X$. Hence $\text{SCC}(v) \subseteq \text{Bwd}(D) \cap X$.

We now prove the complexity of Pendant.

Lemma 6. Pendant *runs in $O(\sum_{S \in \text{SCCs}(G)} \delta(S)) = O(n)$ symbolic steps.*

Proof. In each recursive call, Pendant makes symbolic steps to (i) compute the SCCs of the picked pivots (Lines 5 and 6), and (ii) compute the backwards set of the outputted BSCC (Line 16). We will argue that, in total, case (i) takes $\sum_{S \in \text{SCCs}(G)} 3\delta(S)$ time, while case (ii) takes $\sum_{S \in \text{SCCs}(G)} \delta(S)$ time.

We start with case (i). For a given pivot v, computing SCC(v) is done in two steps: (a) Line 5 computes the forward set F of v restricted to the node set W, while (b) Line 6 computes SCC(v) as the backward set of v restricted to F. Clearly, (b) takes $\delta(\text{SCC}(v))$ symbolic steps, thus summing over all pivots v, we have that Line 6 takes at most $\sum_{S \in \text{SCCs}(G)} \delta(S)$ time. To bound the time spent in (a), denote by Levels(v) the number of iterations executed in FwdLastLayer, i.e., Pendant spends Levels(v) symbolic steps in Line 5. If $F \setminus \text{SCC}(v) = \emptyset$ or $L \setminus \text{SCC}(v) = \emptyset$, we have $\text{Levels}(v) = \delta(\text{SCC}(v))$. Otherwise, the next pivot v' is Pick'ed from L (Line 10). Consider a shortest-path $P \colon v \rightsquigarrow v'$, and let $\{S_1, \dots S_k\}$ be the SCCs of nodes along P (except v), and note that $\text{Levels}(v) \leq$

$\sum_{i=1}^{k} \delta(S_i)$. Moreover, we have $S_i \subseteq \mathrm{Bwd}(v')$ for each $i \in \{1, \ldots, k\}$, and thus each S_i is not touched again by FwdLastLayer, except if $S_i = \mathrm{SCC}(v')$, but this case is accounted for already. Summing over all such S_i across all pivots v, we have that $\sum_v \mathrm{Levels}(v) \leq \sum_{S \in \mathrm{SCCs}(G)} \delta(S)$. Hence the total symbolic time spent for case (i) is bounded by $\sum_{S \in \mathrm{SCCs}(G)} 3\delta(S)$.

We now turn our attention to case (ii). Due to Lemma 3, W is forward closed and D is a BSCC. By Lemma 5, the set B computed in Line 16 is SCC-closed. The number of symbolic steps is hence bounded by $\sum_{S \in \mathrm{SCCs}(B)} \delta(S)$. Finally, B is removed from the graph in the recursive call, hence it will not be processed again. Thus the total time for case (ii) is $\sum_{S \in \mathrm{SCCs}(G)} \delta(S)$. □

4 Deadlock Detection

We now outline a simple but effective preprocessing technique for BSCCs.

Recall that a deadlock is a node v without outgoing edges, i.e., $\mathrm{Post}(v) = \emptyset$. Observe that all deadlocks are BSCCs: formally we have $\mathrm{Fwd}(\{v\}) = \{v\} = \mathrm{SCC}(v)$, and thus the statement follows from Proposition 1 (the opposite is, of course, not true in general). Thus, deadlock-detection can be seen as a natural preprocessing step to any BSCC algorithm.

The key observation in this approach is that *the set of all* deadlocks can be computed efficiently, in only one symbolic step; this is achieved by Algorithm 5. In particular, the deadlock set is computed as $D = V \setminus H$ where H is the set of nodes u that have a successor. In turn, H can be computed by a single Pre operation on the entire node set. Finally, due to Proposition 2, the set $\mathrm{Bwd}(D)$ is guaranteed to contain no BSCCs other than those in D, and thus it can be removed. The resulting graph is then passed to the main BSCC algorithm.

Algorithm 5: Deadlock detection (preprocessing)

```
Input: A graph $G = (V, E)$
$H = \mathrm{Pre}(V, G)$          // Compute all nodes that have a successor
$D = V \setminus H$                        // Compute all deadlocks
$B = \mathrm{Bwd}(D, G)$               // Compute safe-to-remove nodes
output each node in $D$                          // Output BSCCs
return $G[V \setminus B]$   // Return remaining graph for further computation
```

5 Experiments

Here we report on an implementation of Pendant, including the deadlock-detection technique, and an experimental evaluation of its performance on a large dataset of standard model-checking benchmarks across various domains.

Baselines. To assess the performance of Pendant and deadlock detection, we compare it with BwdFwd (Algorithm 2), as well as the recently introduced *interleaved transition guided reduction* (*ITGR*) [6], which we have implemented in our setting. ITGR is applicable when the transition relation is partitioned into a number of smaller relations $E = (R_1, \ldots, R_k)$ (as is the case in our setup), and works as a preprocessing step, much like our deadlock detection. At a high level, ITGR employs some local reasoning for each relation R_i to identify sets of nodes that do not contain BSCCs. Such sets can be removed, reducing the size of the graph that is further processed by a BSCC-computation algorithm.

Research Questions. Our setup is centered around the following questions.

RQ1 How does the performance of Pendant compare to that of BwdFwd?
RQ2 How does deadlock detection impact the performance of Pendant and BwdFwd?
RQ3 How does ITGR impact the performance of Pendant and BwdFwd?
RQ4 How does the performance of Pendant compare to the performance of BwdFwd when both use deadlock detection?
RQ5 How does ITGR impact the performance of Pendant after deadlock detection?

Datasets. We use benchmarks from the following categories.

- Petri Net models from MCC, the Model Checking Contest [22].
- DiVinE models from BEEM, the Benchmark of Explicit Models [27].
- Asynchronous Boolean Network models [3,1,2].

We do not apply any selection criteria, except discarding models that are too slow to handle by all algorithms in our timed experiments. This results in 553 models in total.[†] In each model, the edge relation is naturally partitioned into subrelations $R_1, \ldots, R_k$, following the structure of the high-level specifications (transitions in Petri Nets and DiVinE state machines, and reactions in the Boolean Networks). We use the language-independent model checker LTSmin [20] to generate symbolic graphs for the DVE and PNML models. Since LTSmin does not handle Boolean Networks, these graphs are generated by a custom parser. The time taken for the graph generation is not measured in the running time of each algorithm. We use the BDD package Sylvan as our symbolic representation [15].

Experimental setup. Our experiments are run on a Linux machine with 2.4GHz CPU speed and 60GB of memory. We measure both symbolic steps and run time, but only present the results on symbolic steps here, as they reflect the true symbolic time-complexity of the algorithms, and are independent of the choice of the underlying BDD package. The results on time are qualitatively the same. Each run is timed out after 400 seconds, indicated as the graph taking 10^9 symbolic steps on the figures. Since our input relation is partitioned into

[†] Tool and data set available at `https://doi.org/10.5281/zenodo.10427894`.

several sub-relations $E = (R_1, \ldots, R_k)$, each Pre/Post operation incurs k symbolic steps (for all algorithms). Our setup is completely deterministic, however certain operations, like Pick'ing a node, are executed arbitrarily.

Experimental results. We now present our experimental results for addressing the above research questions. Note that all figures are plotted in log-scale.

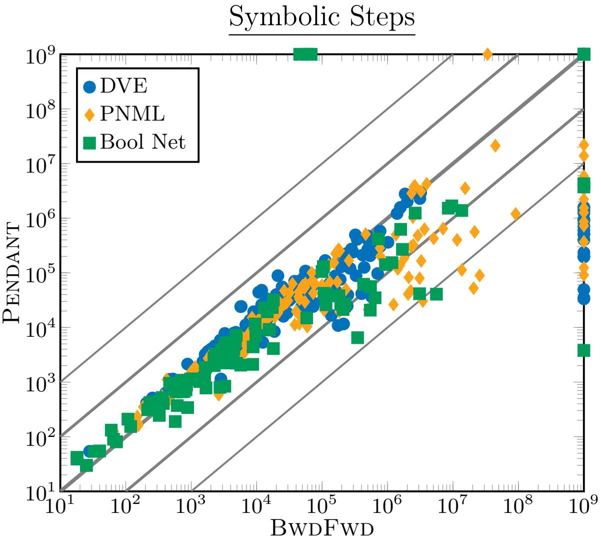


Fig. 2: The number of symbolic steps executed by Pendant and BwdFwd.

***RQ1**: Pendant vs BwdFwd.* The performance of Pendant and BwdFwd is shown in Fig. 2, across all three datasets. Both algorithms manage to handle many models within the time limit, though there are a few time outs. We see that Pendant is generally no slower than BwdFwd, with the clear exception of three timeout outliers. For the rest, the models that are slower for Pendant sit only slightly above the $x = y$ line, meaning that the slowdown is comparatively small. On the other hand, there are several models on which Pendant is generally faster than BwdFwd, and the speedup increases as we go towards more demanding benchmarks (more than two orders of magnitude). Finally, Pendant times out on much fewer models than BwdFwd. Overall, Pendant is measurably faster than BwdFwd, and this trend persists across all three datasets (DVE, PNML and Boolean Networks).

***RQ2**: The impact of deadlock detection.* The impact of deadlock detection to both Pendant and BwdFwd is shown in Fig. 3. We see that deadlock detection improves the performance of both algorithms significantly. Indeed, detecting

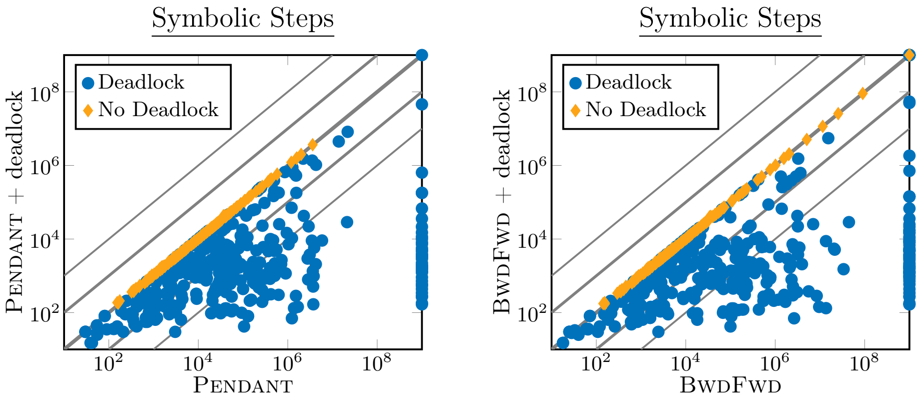


Fig. 3: The impact of deadlock detection in the number of symbolic steps executed by PENDANT (left) and BWDFWD (right). Data points are classified as those having at least one deadlock, and those having no deadlock.

deadlocks requires only one symbolic step (per relation R_i), hence it is natural to expect that it does not slow down any algorithm, and has no effect on models that have no deadlocks. On the other hand, it leads to a measurable speedup on the models that have deadlocks, and the impact varies depending on the fraction of the graph that is removed during deadlock removal. Interestingly, deadlock detection also reduces significantly the number of timeouts for both PENDANT and BWDFWD. In conclusion, deadlock detection helps both algorithms.

***RQ3**: The impact of ITGR.* The impact of ITGR to both PENDANT and BWDFWD is shown in Fig. 4. Perhaps surprisingly, we find that ITGR does not have a consistent effect: it can both speed up and slow down each of the algorithms. At closer inspection, we observe that ITGR has a positive effect on most Boolean Network models, which is indeed the context in which it was introduced [6]. On the other hand, it has both positive and negative effects on DVE and PNML models, and even makes both algorithms time out on instances that they could easily handle without ITGR.

***RQ4**: PENDANT vs BWDFWD, with deadlock detection.* Since deadlock detection has a clear positive effect on both algorithms, it is natural to revisit **RQ1** and ask about the performance of the two algorithms when also using deadlock detection. The result is shown in Fig. 5. Deadlock detection makes the performance of the two algorithms more similar in many benchmarks (i.e., more data points lie closer on the $x = y$ line). However, PENDANT remains decisively faster on many models, and thus its benefit is not overshadowed by the positive impact of deadlock detection. At closer inspection, we see that PENDANT is faster on DVE and PNML models, but not on Boolean Networks. This is due to the fact that

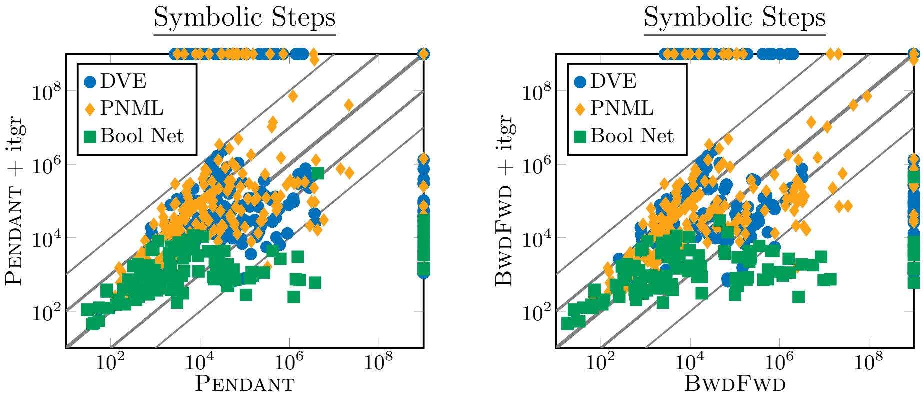


Fig. 4: The impact of ITGR in the number of symbolic steps executed by Pendant (left) and BwdFwd (right).

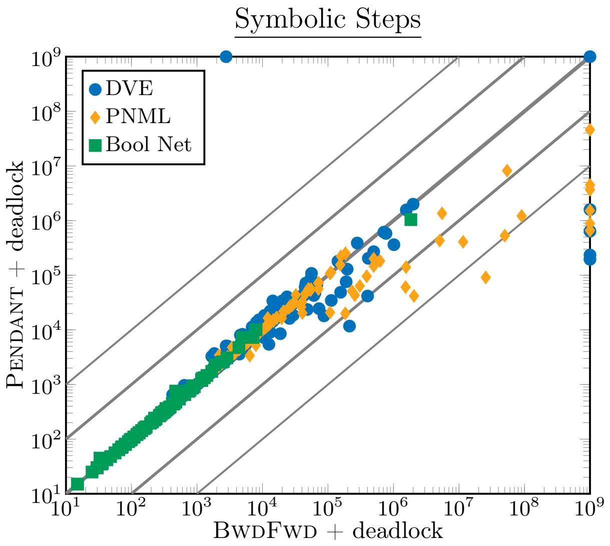


Fig. 5: The number of symbolic steps executed by Pendant and BwdFwd, when also using deadlock detection.

most Boolean Networks have many deadlocks, and thus the common deadlock-detection component simplifies such models considerably, making the remaining performance of the two algorithms similar.

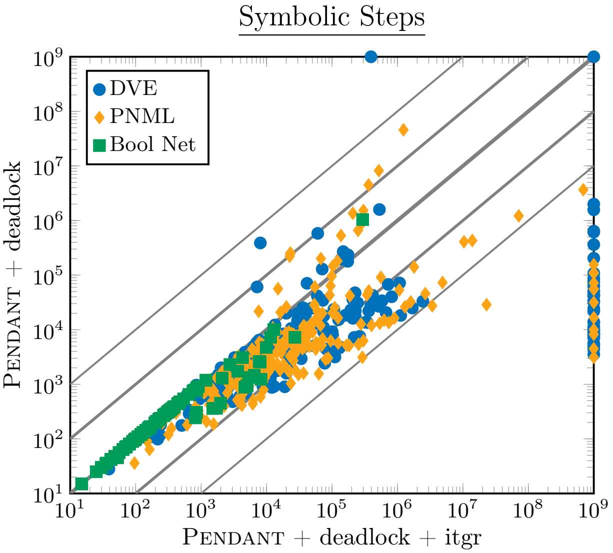


Fig. 6: The impact of ITGR after using deadlock detection.

RQ5: *The impact of ITGR after deadlock detection.* Finally, in Fig. 6 we examine whether ITGR improves the performance of Pendant after deadlock detection has run. Although ITGR improves the performance on a few models, it generally leads to a slowdown, as well as to more timeouts. Interestingly, ITGR has the fewest positive effects (on top of deadlock detection) for Boolean Network models, for which it was originally introduced. Since these models have several deadlocks, the fast deadlock-detection preprocessing simplifies them considerably, at which point the cost of ITGR is not worth its little (or no) impact.

6 Conclusion

We have introduced Pendant, a new symbolic algorithm for computing BSCCs, as well as a deadlock-detection technique for this task. Though both Pendant and the standard BwdFwd have $O(n)$ symbolic-time complexity, our experimental results show that Pendant is typically faster, and thus to be preferred for this task. Moreover, deadlock-detection is an efficient and effective preprocessing technique for reporting singleton BSCCs (and removing their basin), before handing the computation to the general algorithm. Finally, the recently introduced ITGR, although effective on Boolean Network models, has mixed effects on DVE and PNML models, while its effect is often negative after deadlock detection (but not always). Some opportunities for future research include introducing saturation techniques [34] to Pendant, extending the algorithm to symbolically handle colored graphs [7,25], and understanding better the settings in which ITGR is effective.

Acknowledgements. This work was supported in part by Villum Fonden (Project VIL42117).

References

1. EMBL-EBI's BioModels model repository (2023), https://www.ebi.ac.uk/biomodels/, Last accessed on 2023-10-10
2. PyBoolNet model repository (2023), https://github.com/hklarner/pyboolnet/tree/master/pyboolnet/repository, Last accessed on 2023-10-10
3. SBML models repository (2023), https://github.com/sybila/biodivine-lib-param-bn/tree/master/sbml_models, Last accessed 2023-10-10
4. Abraham, E., Jansen, N., Wimmer, R., Katoen, J.P., Becker, B.: DTMC model checking by SCC reduction. In: Proceedings of the 2010 Seventh International Conference on the Quantitative Evaluation of Systems. p. 37–46. QEST '10, IEEE Computer Society, USA (2010). https://doi.org/10.1109/QEST.2010.13
5. Amparore, E.G., Donatelli, S., Gallà, F.: starMC: an automata based CTL* model checker. PeerJ Comput. Sci. **8**, e823 (2022)
6. Benes, N., Brim, L., Pastva, S., Šafránek, D.: Computing bottom SCCs symbolically using transition guided reduction. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, CAV 2021, Part I. LNCS, vol. 12759, pp. 505–528. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_24
7. Benes, N., Brim, L., Pastva, S., Šafránek, D.: BDD-based algorithm for SCC decomposition of edge-coloured graphs. Logical Methods in Computer Science **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:38)2022
8. Bloem, R., Gabow, H.N., Somenzi, F.: An algorithm for strongly connected component analysis in n log n symbolic steps. Formal Methods in System Design **28**(1), 37–56 (2006)
9. Bloem, R., Ravi, K., Somenzi, F.: Efficient decision procedures for model checking of linear time logic properties. In: Proceedings of the 11th International Conference on Computer Aided Verification. p. 222–235. CAV '99, Springer (1999)
10. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
11. Buchholz, P., Katoen, J.P., Kemper, P., Tepper, C.: Model-checking large structured Markov chains. The Journal of Logic and Algebraic Programming **56**(1), 69–97 (2003). https://doi.org/https://doi.org/10.1016/S1567-8326(02)00067-X, probabilistic Techniques for the Design and Analysis of Systems
12. Chatterjee, K., Dvořák, W., Henzinger, M., Loitzenbauer, V.: Lower bounds for symbolic computation on graphs: Strongly connected components, liveness, safety, and diameter. In: Proc. 29th ACM-SIAM Symp. on Discrete Algorithms. p. 2341–2356. SODA '18, Soc. for Industrial and Applied Mathematics, USA (2018)

13. Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: Proc. 22nd ACM-SIAM Symp. on Discrete Algorithms. p. 1318–1336. SODA '11, Society for Industrial and Applied Mathematics, USA (2011)
14. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: CAV. LNCS, vol. 2404, pp. 359–364. Springer (2002)
15. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int. Journal on Software Tools for Technology Transfer **19**(6), 675–696 (2017)
16. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is there a best symbolic cycle-detection algorithm? In: Proc. 7th IC on Tools and Algorithms for the Construction and Analysis of Systems. p. 420–434. TACAS 2001, Springer (2001)
17. Gentilini, R., Piazza, C., Policriti, A.: Computing strongly connected components in a linear number of symbolic steps. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. p. 573–582. SODA '03, Society for Industrial and Applied Mathematics, USA (2003)
18. Hardin, R.H., Kurshan, R.P., Shukla, S.K., Vardi, M.Y.: A new heuristic for bad cycle detection using BDDs. Form. Methods Syst. Des. **18**(2), 131–140 (mar 2001). https://doi.org/10.1023/A:1008727508722
19. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press (2004)
20. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-performance language-independent model checking. In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer (2015)
21. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model checking with strong fairness. Formal Methods Syst. Des. **28**(1), 57–84 (2006)
22. Kordon, F., Garavel, H., Hillah, L., Paviot-Adet, E., Jezequel, L., Hulin-Hubard, F., Amparore, E.G., Beccuti, M., Berthomieu, B., Evrard, H., Jensen, P.G., Botlan, D.L., Liebke, T., Meijer, J., Srba, J., Thierry-Mieg, Y., van de Pol, J., Wolf, K.: MCC'2017 - the seventh model checking contest. Trans. Petri Nets Other Model. Concurr. **13**, 181–209 (2018)
23. Kučera, A., Stražovský, O.: On the controller synthesis for finite-state Markov decision processes. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science. pp. 541–552. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
24. Kwiatkowska, M.Z., 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)
25. Larsen, C.A., Schmidt, S.M., Steensgaard, J., Jakobsen, A.B., van de Pol, J., Pavlogiannis, A.: A truly symbolic linear-time algorithm for SCC decomposition (2023)
26. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transf. **19**(1), 9–30 (2017)
27. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: SPIN. Lecture Notes in Computer Science, vol. 4595, pp. 263–267. Springer (2007)
28. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Strength-Based Decomposition of the Property Büchi Automaton for Faster Model Checking. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 580–593. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_42

29. Saadatpour, A., Albert, I., Albert, R.: Attractor analysis of asynchronous boolean models of signal transduction networks. Journal of Theoretical Biology **266**(4), 641–656 (2010). https://doi.org/https://doi.org/10.1016/j.jtbi.2010.07.022
30. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: TACAS. Lecture Notes in Computer Science, vol. 3440, pp. 174–190. Springer (2005)
31. Wang, C., Bloem, R., Hachtel, G.D., Ravi, K., Somenzi, F.: Divide and compose: SCC refinement for language emptiness. In: Proceedings of the 12th International Conference on Concurrency Theory. p. 456–471. CONCUR '01, Springer-Verlag, Berlin, Heidelberg (2001)
32. Xie, A., Beerel, P.A.: Implicit enumeration of strongly connected components and an application to formal verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19**(10), 1225–1230 (2000)
33. Yuan, Q., Mizera, A., Pang, J., Qu, H.: A new decomposition-based method for detecting attractors in synchronous boolean networks. Science of Computer Programming **180**, 18–35 (2019). https://doi.org/https://doi.org/10.1016/j.scico.2019.05.001
34. Zhao, Y., Ciardo, G.: Symbolic computation of strongly connected components and fair cycles using saturation. Innov. Syst. Softw. Eng. **7**(2), 141–150 (2011)

Btor2-Cert: A Certifying Hardware-Verification Framework Using Software Analyzers

Zsófia Ádám[1,2], Dirk Beyer[2], Po-Chun Chien[2], Nian-Ze Lee[2], and Nils Sirrenberg[2]

[1] Department of Measurement and Information Systems,
Budapest University of Technology and Economics, Budapest, Hungary
[2] Department of Computer Science, LMU Munich, Munich, Germany

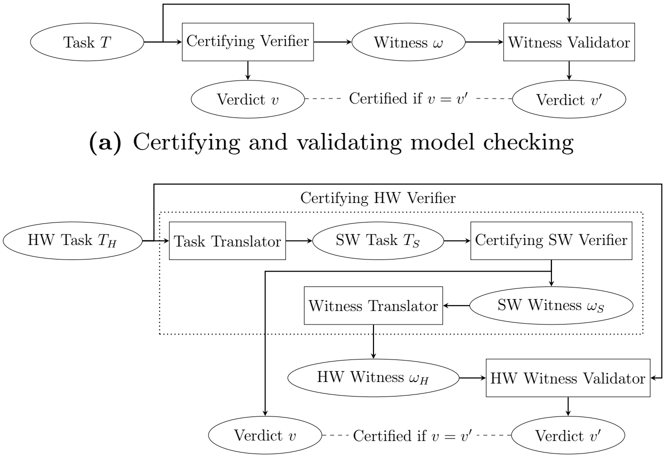
Abstract. Formal verification is essential but challenging: Even the best verifiers may produce wrong verification verdicts. *Certifying* verifiers enhance the confidence in verification results by generating a *witness* for other tools to validate the verdict independently. Recently, translating the hardware-modeling language Btor2 to software, such as the programming language C or LLVM intermediate representation, has been actively studied and facilitated verifying hardware designs by software analyzers. However, it remained unknown whether witnesses produced by software verifiers contain helpful information about the original circuits and how such information can aid hardware analysis. We propose a certifying and validating framework Btor2-Cert to verify safety properties of Btor2 circuits, combining Btor2-to-C translation, software verifiers, and a new witness validator Btor2-Val, to answer the above open questions. Btor2-Cert translates a software *violation witness* to a Btor2 violation witness; As the Btor2 language lacks a format for *correctness witnesses*, we encode invariants in software correctness witnesses as Btor2 circuits. The validator Btor2-Val checks violation witnesses by circuit simulation and correctness witnesses by *validation via verification*. In our evaluation, Btor2-Cert successfully utilized software witnesses to improve quality assurance of hardware. By invoking the software verifier Cbmc on translated programs, it uniquely solved, with confirmed witnesses, 8 % of the unsafe tasks for which the hardware verifier ABC failed to detect bugs.

Keywords: Hardware verification · Software verification · Verification witnesses · Witness validation · Word-level circuit · Btor2 · SMT · SAT

1 Introduction

Certifying algorithms [1] generate a *certificate* alongside the computed solution such that *proof checkers* can independently validate the solution to increase users' trust and the explainability of the results. In the model-checking community, a certificate to explain a verdict for a verification task is called a *witness* [2], and verifiers able to generate witnesses are called *certifying verifiers*. Witnesses can be independently checked by *witness validators* to confirm the verification

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 129–149, 2024.
https://doi.org/10.1007/978-3-031-57256-2_7



(a) Certifying and validating model checking

(b) Translating SW witnesses to HW for validation

Fig. 1: A certifying hardware-verification framework using software analyzers

results. Figure 1a shows a generic workflow for *certifying and validating model checking*. After a certifying verifier produces a verdict v and a witness ω on a task T, a witness validator takes T and ω as input and checks if the information in ω is enough to reestablish the results of the verifier on T. The outcome of the verifier is *certified* if its verdict v and the validator's verdict v' are consistent. In the rest of the paper, we use *certifying model checking* interchangeably with *certifying and validating model checking* when it is clear from the context that a framework contains both a certifying verifier and a witness validator. For reachability properties, if a model violates a safety specification, a *violation witness* [3] may contain external inputs to the model to replay the erroneous execution trace. If the safety specification is satisfied, a *correctness witness* [4] could record invariants of the model to reconstruct a safety proof. Section 2 presents a brief survey on witness validation in the formal-methods community.

Recently, hardware-to-software translators [5, 6] from the hardware-modeling language Btor2 [7], a prevailing format for word-level hardware model checking used in the Hardware Model Checking Competitions (HWMCC) [8, 9], have been proposed to facilitate the application of software analyzers to hardware circuits. Tools Btor2C [5] and Btor2MLIR [6] translate Btor2 circuits to behaviorally equivalent imperative software in the programming language C [10] and the intermediate representation used by the compilation toolchain LLVM [11], respectively, and enable any software analyzer for C or LLVM-bytecode programs to inspect Btor2 circuits. In an experiment on more than 1 000 Btor2 circuits [5], software verifiers for C programs are shown to detect more bugs than the best hardware model checkers by preprocessing the original circuit with Btor2C and analyzing the translated C program. However, in this previous work [5], only the verdicts of software verifiers but not the witnesses, which contain the information and reasoning behind a verdict, are transferred back to the hardware domain. In other words, the results of software verifiers on Btor2 circuits are not certified, and hence hardware designers may not trust software verifiers for analyzing their circuits.

1.1 Our Motivation and Contributions

Motivated to mitigate the aforementioned threat to reliability and leverage the capability of software verifiers to generate witnesses, we investigate the following open questions in this work: (1) *whether the software witnesses for translated programs contain useful information about original circuits* and (2) *how to employ the information to aid hardware quality assurance*. Our contributions are summarized below.

A Certifying Framework for HW Verification with SW Analyzers. Figure 1b shows the proposed certifying and validating hardware-verification framework based on software analyzers to approach the open questions. The framework translates a hardware-verification task T_H to a software task T_S and applies software verifiers to T_S. After obtaining a software witness ω_S, it encodes relevant information from ω_S in the form of a hardware witness ω_H and validates the verdict returned by software verifiers with ω_H. We instantiate the framework in a tool Btor2-Cert for verifying Btor2 circuits with certified verdicts. In addition to preprocessing Btor2 circuits with Btor2C [5] and invoking model checkers for the translated C programs, such as CPAchecker [12], Cbmc [13], Esbmc [14], and UAutomizer [15], Btor2-Cert features a **translator** from software witnesses to Btor2 witnesses and a **witness validator** Btor2-Val to check Btor2 witnesses. Section 4 shows our tool architecture.

Note that the framework in Fig. 1b is not limited to Btor2C and verifiers for C programs. For example, one could also materialize the concept with the translator Btor2MLIR [6], analyzers for LLVM-bytecode programs [11], such as Klee [16], Smack [17], and SeaHorn [18], and a corresponding LLVM-to-Btor2 witness translator. There also exist translators [19, 20, 21] from Verilog [22] circuits to C programs or SMV [23] models. We choose Btor2C for task translation because many verifiers for C programs participating in the International Competitions on Software Verification (SV-COMP) [24] can generate witnesses in a standardized and exchangeable format [2].

A Translator from Software Witnesses to Btor2 Witnesses. Btor2-Cert translates software violation witnesses in the format used in SV-COMP [24] to the format defined by the Btor2 language [7]. For tasks satisfying their specifications, as there is no native format for correctness witnesses in Btor2, Btor2-Cert extracts the invariants in software witnesses and represents them as Btor2 circuits, whose inputs refer to the state variables of the original circuit. The advantages of not inventing a new format but reusing the existing Btor2 language are twofold: First, Btor2 extends SMT-LIB 2 [25] and provides the required operations on the word level to accommodate most invariants derived by software verifiers. Second, Btor2 is supported by many hardware model checkers participating in HWMCC [8, 9] and offers a suite Btor2Tools [26] of utility tools for parsing and simulation, which simplifies further development around the Btor2 format.

A Validator for Btor2 Witnesses. To validate the witnesses for Btor2 circuits, we develop Btor2-Val, a portfolio-based witness validator involving hardware simulators and verifiers. Btor2-Val validates violation witnesses by invoking the simulator BtorSim from Btor2Tools [26]. For correctness witnesses, Btor2-Val follows the *validation-via-verification* approach [27] by instrumenting the original Btor2 circuit with the circuit representing the invariant and verifying the instrumented circuit. The instrumented circuit satisfies the modified safety property if the invariant can be used to reconstruct the proof of correctness. Hardware verifiers are employed to check the instrumented circuits. Btor2-Val leverages CoVeriTeam [28], a framework for cooperative verification, to coordinate the underlying hardware simulators and verifiers.

Enhancing Confidence in SW Verifiers on HW Designs. We evaluate Btor2-Cert on more than 1 000 Btor2 circuits to study its capability of providing certified verification results using software analyzers. In the experiment,

- the witness translator was able to translate every violation witness and 97 % of the correctness witnesses produced by software verifiers,
- the combination of witness translation and Btor2-Val outperformed mature software witness validators in both effectiveness and efficiency, and
- Btor2-Cert provided **certified** results computed by software verifiers on some Btor2 circuits that the best hardware model checkers failed to verify.

The conceptual message conveyed by Btor2-Cert is *software analyzers can derive useful information about circuits and complement conventional hardware model checkers with trustworthy results.* Our contributions have a positive impact on analyzing hardware designs with software verifiers. The proposed framework Btor2-Cert is open-source and available online (more information in Sect. 4).

2 Related Work

Generating and validating witnesses for analysis results have been studied throughout the entire verification toolchain from satisfiability solvers to model checkers. In the following, we briefly review witness validation and compare our work to a recent certifying verification framework [29, 30, 31] targeting k-induction [32].

2.1 Witness Validation

For satisfiability solving, the competitions on propositional SAT solvers [33, 34] use the DRAT format [35] to encode the certificates of unsatisfiability and independent validators [36, 37] to check the proofs. The competitions on SMT solving verify models to satisfiable formulas with the tool Dolmen [38]. Certifications for quantified Boolean formulas have also been investigated [39, 40].

For model checking, an early work [41] suggests generating a deductive proof from the run of model checkers with extra bookkeeping steps. In HWMCC [8, 9], the Btor2 [7] language defines a format for violation witnesses as a sequence of input values and initial values for registers that lead to an erroneous execution. However, Btor2 has no format for correctness witnesses. The competitions on automated termination analysis [42] use the format CPF [43], and in SV-COMP [24], a GraphML-based format [2] is used to describe software witnesses as automata. In addition to the properties commonly used in tool competitions, a recent work extends proof generation of model checking to full LTL properties [44].

Numerous approaches have been invented for validating software witnesses. Methods to validate correctness witnesses include a parallel extension [45] of k-induction, program instrumentation with invariants and re-verification [27] (referred to as *validation via verification* in the publication), and program decomposition into several straight-line sub-programs [46]. *Execution-based validation* [47] is an elegant approach to validate violation witnesses. It extracts a sequence of external input values from a violation witness and employs debuggers or simulators to testify the reachability of an error location. Our witness validator Btor2-Val leverages validation via verification and execution-based validation. More details are given in Sect. 5 and Sect. 6, respectively. In our evaluation, the proposed validator Btor2-Val (together with the witness translator) competed well against the winners in the witness-validation track of SV-COMP 2023 [24].

2.2 Validating k-Inductiveness of Properties in Hardware Models

Given a sequential circuit and a number k as input, the tool Certifaiger [29, 30] aims to validate that the safety property of the input circuit is k-inductive. Composing a k-induction-based hardware model checker and Certifaiger yields a certifying and validating model checker (as depicted in Fig. 1a), whose witnesses are the inductive length k. The key differences between the proposed framework in Fig. 1b and this framework [29, 30] for k-inductiveness are as follows.

First, our validator Btor2-Val expects a candidate invariant in the correctness witness but does not restrict the algorithms used by software verifiers. In contrast, Certifaiger expects a candidate inductive length k and thus can only validate results of k-induction-based model checkers. Second, to validate witnesses, Btor2-Val relies on validation via verification [27] and invokes model checkers because the candidate invariant may not be inductive. In comparison, Certifaiger avoids model checking and reduces the validation problem to several SAT checks since it assumes the safety property to be k-inductive. To sum up, our framework complements the existing work [29, 30] by considering candidate invariants as witnesses. Its applicability to algorithms other than k-induction comes at the expense of potentially more complex validation procedure. Certifaiger is further extended to accommodate temporal decomposition [48] as preprocessing to simplify the verification tasks [31], which has not yet been considered in our framework and is an important direction of future work.

3 Background

To facilitate the discussion in the rest of this manuscript, we provide prerequisite knowledge on model checking and witness validation from the literature.

A state-transition system $\mathcal{M}$ is described by two predicates $I(s)$ and $TR(s, s')$ over states s and s' of $\mathcal{M}$, which encode the *initial states* and *transition relation* ($TR(s, s')$ is true if s can transit to s' via one step) of $\mathcal{M}$, respectively. An

```
1 sort bitvec 8
2 sort bitvec 1
3 constd 1 42
4 constd 1 2
5 zero 1
6 state 1 ; a
7 state 1 ; b
8 input 1 ; in
9 init 1 6 4 ; a init to 2
10 init 1 7 5 ; b init to 0
11 eq 2 6 5 ; a == 0
12 eq 2 7 4 ; b == 2
13 eq 2 8 3 ; in == 42
14 and 2 11 12
15 and 2 13 14
16 bad 15
17 one 1
18 srl 1 6 17
19 xor 1 7 17
20 next 1 6 18
21 next 1 7 19
```

(a) Btor2 circuit

```
extern void abort(void);
extern unsigned char nondet_uchar();
void main() {
  typedef unsigned char SORT_1;
  SORT_1 a = nondet_uchar();
  SORT_1 b = nondet_uchar();
  a = 2;
  b = 0;
  for (;;) {
    SORT_1 in = nondet_uchar();
    if (a == 0 && b == 2 && in == 42) {
      ERROR: abort();
    }
    a = a >> 1;
    b = b ^ 1;
  }
}
```

(b) C program (simplified for demo)

Fig. 2: An example Btor2 circuit and its translated C program

invariant $Inv(s)$ of a system $\mathcal{M}$ is a predicate over states of $\mathcal{M}$ such that $Inv(s)$ *is true for every reachable state* s *of* $\mathcal{M}$. We denote "Inv is an invariant of $\mathcal{M}$" by $\mathcal{M} \models Inv$. A *safety-verification task* consists of a state-transition system $\mathcal{M}$ and a safety property $P(s)$. We say a safety-verification task (or a verification task for short) is *safe* if $\mathcal{M} \models P$ and *unsafe* otherwise. Given a verification task of $\mathcal{M}$ and P, the problem of *model checking* asks whether $\mathcal{M} \models P$ or not. In practice, state-transition systems manifest themselves as sequential digital circuits or programs. In the following, we briefly introduce the modeling languages used in HWMCC [8, 9] and SV-COMP [24] with a running example.

3.1 The Btor2 Language for Word-Level Circuits

The Btor2 hardware-modeling language [7] was invented to describe model-checking problems of word-level sequential circuits. It extends the bit-level AIGER format [49] with data sorts of bit-vectors and arrays and inherits word-level operations from SMT-LIB 2 [25]. Figure 2a shows an example Btor2 circuit. The circuit has two state variables `a` and `b` and an external input `in`, defined in lines `6`-`8`, respectively. The states and input are bit-vectors of width 8 (the sort `bitvec 8` defined in line `1`). Variables `a` and `b` are initialized to 2 and 0, respectively. In each iteration, variable `a` is right-shifted by 1 bit (line `18`), and variable `b` is bitwise XOR-ed with 1 (line `19`). Indicated by the keyword `bad` in line `16`, a property violation happens if variable `a` equals 0, variable `b` equals 2, and input `in` equals 42. The example Btor2 circuit satisfies its safety property because variable `b` never equals 2. However, if variable `b` is initialized to a different value at line `10` (marked in red), say 2, a property violation will be triggered after two steps of state transition if 42 is given as the external input in the last iteration.

Translating Btor2 Circuits to C Programs. Btor2C [5] is a lightweight translator from the Btor2 language to the programming language C [10]. It encodes Btor2 data sorts with unsigned integers and static arrays, expresses Btor2 operations with corresponding operators of C, and uses an infinite loop to model the execution of a sequential circuit. Given the example Btor2 circuit in Fig. 2a as input, Btor2C generates a translated C program[1] shown in Fig. 2b. Btor2C follows the rules of SV-COMP [24] to encode safety-verification tasks for C programs, so *compositional* hardware model checkers for Btor2 circuits can be readily formed by combining software verifiers participating in SV-COMP as verification engines and Btor2C as preprocessing. In an extensive experiment [5], software verifiers are shown to detect more bugs in Btor2 circuits than the best *conventional* hardware model checkers, such as ABC [50] and AVR [51].

3.2 Representing Software Witnesses as Automata

Software witnesses can be represented as *protocol automata* [2], describing program invariants needed to construct a safety proof or program paths leading to a property violation. A letter in the alphabet of such a protocol automaton is a pair of a set of program edges and a condition over program variables. The set of program edges indicates the control flow, and the condition can be used to restrict the state space of the program. Program invariants that should hold at a certain program location can be annotated to a protocol automaton. In the following, we give an example correctness witness for the C program in Fig. 2b and an example violation witness for the same C program but with line 8 commented out.

Correctness Witnesses. Figure 3 shows an example correctness witness for the C program in Fig. 2b. The correctness witness shows that a program invariant `b>=0 && b<=1` is established once line 8 is executed. Indeed, variable `b` switches between 0 and 1 after being initialized, and `b>=0 && b<=1` is an invariant at the loop head of the program. A program invariant is stored as a C expression in a software witness and hence potentially more compact than invariants represented in other formalisms, e.g., a bit-level AIGER [49] circuit.

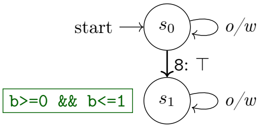


Fig. 3: A correctness witness

Violation Witnesses. Figure 4 shows an example violation witness for the modified C program with variable `b` uninitialized (by commenting out line 8 in Fig. 2b). The violation witness shows how to reach the error in line 12 of the C program. First, it assumes the value of variable `b` to be 2 via the condition when line 6 is executed. Second, it goes to the next state when line 10 is executed for the first two times. Third, it assumes the external input to be 42 when line 10 is executed for the third time. Indeed, the error in line 12 can be reached if variable `b` gets an initial value of 2 and the external input equals 42 in the third loop iteration.

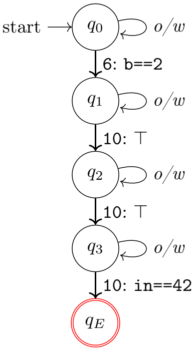
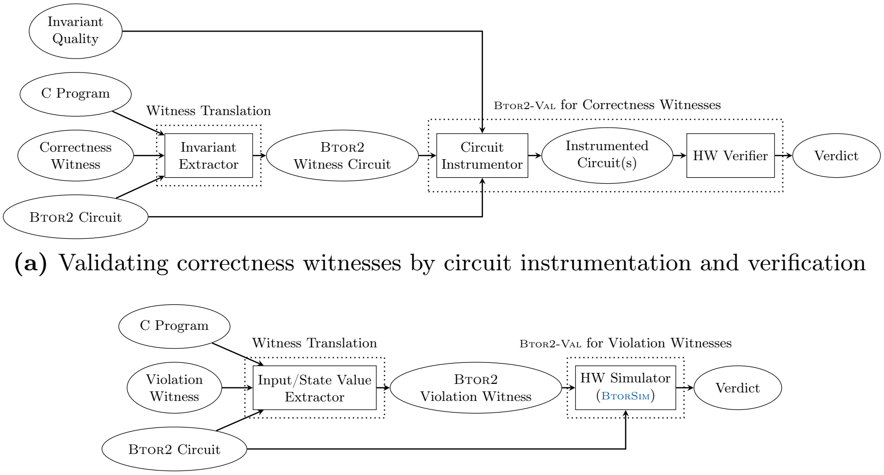


Fig. 4: A violation witness

[1] The intermediate variables in the actual output program of Btor2C are omitted.



(a) Validating correctness witnesses by circuit instrumentation and verification

(b) Validating violation witnesses by circuit simulation

Fig. 5: Witness translation and validation in Btor2-Cert and Btor2-Val

4 Architecture of Btor2-Cert and Btor2-Val

We instantiate the proposed certifying and validating hardware-verification framework in Fig. 1b as Btor2-Cert[2] with the Btor2-to-C translator Btor2C [5], model checkers for C programs [52] that can produce verification witnesses in the format discussed in Sect. 3, a C-to-Btor2 witness translator, and the witness validator Btor2-Val. Figure 5 shows the translation and validation flows for correctness (in Fig. 5a) and violation witnesses (in Fig. 5b). Both the translator and the validator Btor2-Val for Btor2 witnesses are implemented in Python 3. Btor2-Val is based on a portfolio of hardware verifiers and simulators, with different tools coordinated by the cooperative-verification framework CoVeriTeam [28].

4.1 Validating Correctness Witnesses

Given a safe Btor2 circuit, its translated C program, and a correctness witness produced by some software verifier, Btor2-Cert certifies the results of the software verifier in two steps, as depicted in Fig. 5a. In the first step of witness translation, Btor2-Cert extracts the invariant at the loop head of the C program and represents it as a Btor2 circuit. The Btor2 circuit is named a *witness circuit* and refers to the state variables of the original circuit from its primary inputs. Second, in the validation step, Btor2-Val takes as input the original circuit, the witness circuit, and a user-defined parameter called *invariant quality* that specifies the level of strictness imposed on the invariant. Btor2-Val offers three levels of invariant quality to users, based on which it instruments the original circuit. Hardware verifiers are invoked on the *instrumented circuit* and will deem it safe if the invariant meets the specified invariant quality for reconstructing a safety proof. The details of validating correctness witnesses are presented in Sect. 5.

[2] https://gitlab.com/sosy-lab/software/btor2-cert

4.2 Validating Violation Witnesses

Given an unsafe Btor2 circuit, its translated C program, and a violation witness produced by some software verifier, Btor2-Cert certifies the results of the software verifier in two steps, as depicted in Fig. 5b. In the first step of witness translation, Btor2-Cert extracts the values for external inputs and uninitialized states from the software violation witness and encodes the information as a Btor2 violation witness [7]. Second, in the validation step, Btor2-Val invokes BtorSim [26], a simulator for Btor2 circuits, to decide whether the Btor2 violation witness can trigger a bug in the original circuit. The details of validating violation witnesses are presented in Sect. 6.

5 Certifying Results of Software Verifiers: Correctness

In this section, we describe how Btor2-Cert certifies verification results for **safe** verification tasks. The Btor2 circuit and its translated C program in Fig. 2 as well as the software correctness witness in Fig. 3 will be used to explain the translation and validation of correctness witnesses, as outlined in Fig. 5a.

5.1 Witness Translation

Given a software correctness witness with a predicate annotated at the loop head of the translated C program, which some software verifier claims to be an invariant,[3] Btor2-Cert considers the predicate as a *candidate invariant* for the original Btor2 circuit and extracts it to reconstruct a safety proof. We encode the candidate invariant, written as an expression in the programming language C, into a combinational Btor2 circuit whose inputs refer to the state variables of the original Btor2 circuit and unique output asserts the predicate. Translating C expressions into Btor2 circuits is feasible thanks to the word-level data sorts and operations in the Btor2 language [7]. We name the combinational Btor2 circuit a *witness circuit* and refer to it as a Btor2 correctness witness. Note that our notion of a witness circuit is different from Certifaiger's definition of a *k-witness circuit* [29], which is a sequential circuit simulating k-step execution of the original circuit in one step. Figure 6 shows the witness circuit generated from the software correctness witness in Fig. 3. The input defined in line 5 refers to state variable b of the Btor2 circuit in Fig. 2a. The output defined in line 9 asserts the candidate invariant `b >= 0 && b <= 1`.

```
1 sort bitvec 8
2 sort bitvec 1
3 zero 1
4 one 1
5 input 1 ; state "b"
6 ugte 2 5 3 ; b >= 0
7 ulte 2 5 4 ; b <= 1
8 and 2 6 7
9 output 8
```

Fig. 6: A witness circuit

5.2 Witness Validation via Verification

Following the idea of validation via verification [27], the validator Btor2-Val in Btor2-Cert checks Btor2 correctness witnesses by instrumenting the original circuit with the witness circuit and invoking hardware model checkers. It distinguishes three levels of quality for a candidate invariant computed by software

[3] Many mature verifiers in SV-COMP derive invariants at loop-head locations.

Table 1: Candidate invariants at the loop head of the program in Fig. 2b

Predicate	Quality	Reason
$\bot$	not invariant	$\mathcal{M} \models Inv$ fails.
$\top$	invariant but unsafe	$Inv \Rightarrow P$ fails.
`b!=2`	safe invariant but not inductive	$Inv(s) \wedge TR(s, s') \Rightarrow Inv(s')$ fails.
`b>=0 && b<=1`	safe and inductive invariant	All checks succeed.

verifiers. According to the notation introduced in Sect. 3, we denote the state-transition system of the original Btor2 circuit by $\mathcal{M}$, with initial states $I(s)$, a transition relation $TR(s, s')$, and a safety property $P(s)$. A predicate $Inv(s)$ is

- an invariant if $\mathcal{M} \models Inv$,
- a *safe* invariant if $\mathcal{M} \models Inv$ and $Inv(s) \Rightarrow P(s)$, and
- a *safe* and *inductive* invariant if (1) $Inv(s) \Rightarrow P(s)$, (2) $I(s) \Rightarrow Inv(s)$, and (3) $Inv(s) \wedge TR(s, s') \Rightarrow Inv(s')$.

In the literature [29], the three conditions for safe and inductive invariants are also named *consistency*, *initiation*, and *consecution*, respectively. Table 1 shows four predicates and highlights their respective quality as an invariant at the loop head of the program in Fig. 2b (P is the negated error condition).

Btor2-Val takes the original Btor2 circuit, the witness circuit, and a user-specified invariant quality for the correctness witness as input and instruments the original circuit accordingly. To check if $Inv(s)$ is an invariant helpful to reestablish a proof of P, Btor2-Val combines the witness circuit and the original circuit by connecting the state variables of the original circuit to the corresponding inputs of the witness circuit. That is, Btor2-Val builds a circuit that encodes $\mathcal{M} \models Inv \wedge P$. The instrumented circuit is given to hardware model checkers, which will utilize the information provided by the witness circuit to find a proof of correctness or refute the predicate if it is not an invariant. Note that the verification time of the instrumented circuit is expected to be shorter than that of the original circuit because the predicate can guide the search of hardware model checkers.

To implement the consistency, initiation, and consecution checks for safe or inductive invariants, Btor2-Val also relies on circuit instrumentation and hardware model checkers. While the three checks are not model checking but satisfiability in essence, it is convenient to encode them as combinational Btor2 circuits. Moreover, some hardware model checkers, such as ABC [50], can simplify the circuits before performing satisfiability solving, which is usually faster than solving the queries directly with satisfiability solvers.

6 Certifying Results of Software Verifiers: Violation

In this section, we describe how Btor2-Cert certifies verification results for **unsafe** verification tasks. The unsafe versions of the Btor2 circuit and its translated C program in Fig. 2 with the state variable `b` being uninitialized (namely, with line `10` in Fig. 2a and line `8` in Fig. 2b commented out) as well as the software violation witness in Fig. 4 will be used to explain the translation and validation of violation witnesses, as outlined in Fig. 5b.

The Btor2 language defines a format for violation witnesses [7]. A Btor2 violation witness contains a sequence of input values fed to the Btor2 circuit in each cycle and the initial values for uninitialized state variables. Figure 7 shows an example violation witness for the unsafe version of the Btor2 circuit in Fig. 2a. It demonstrates how to trigger the error specified by the 0th `bad` statement (indicted by `b0`) via giving the initial value 2 to the 1st state variable `b` (under `#0`; `a` is the 0th state variable) and 42 to the 0th input `in` in the 2nd cycle (indicated by `@2`). The simulator BtorSim [26] takes a Btor2 circuit and a Btor2 violation witness and executes the circuit with the values for inputs and states in the witness. It confirms the violation witness if an error is triggered. The violation witness in Fig. 7 does not specify input values in the first two cycles because they are irrelevant to the error. In this case, BtorSim will assume the unspecified values to be zero.

6.1 Witness Translation

Given a software violation witness of the translated C program, Btor2-Cert extracts the conditions over program variables from the protocol automaton. These conditions are used by the software violation witness to prune out irrelevant program paths and highlight an error path. Btor2-Cert uses such information to give values to the corresponding Btor2 inputs and state variables in the form of a Btor2 violation witness. For example, the software violation witness in Fig. 4 will be translated to the Btor2 violation witness in Fig. 7.

```
sat
b0
#0
1 00000010 ; b==2
@0
@1
@2
0 00101010 ; in==42
.
```

Fig. 7: A Btor2 violation witness

6.2 Witness Validation via Execution

Following the idea of execution-based witness validation [47], Btor2-Val checks Btor2 violation witnesses by invoking the simulator BtorSim on the original Btor2 circuit and the translated Btor2 violation witness. An advantage of execution-based witness validation is its speed: In our evaluation, Btor2-Val was able to validate Btor2 violation witnesses translated from software violation witnesses much faster than software verifiers for finding the bugs. The speed of Btor2-Val minimizes the overhead to validate the alarms reported by software verifiers and makes the results of software verifiers more trustworthy and transparent for hardware designers.

7 Evaluation

To address the open questions highlighted in Sect. 1.1, we evaluated the proposed certifying hardware-verification framework Btor2-Cert on more than 1 000 Btor2 circuits and the witness validator Btor2-Val prepended with witness translation against the top contenders in the witness-validation track of SV-COMP 2023 [24]. Our experiment is designed to answer the following research questions:

- **RQ1**: Can Btor2-Cert translate software witnesses to Btor2 witnesses?
- **RQ2**: Is Btor2-Val prepended with witness translation **effective** compared to state-of-the-art software witness validators?
- **RQ3**: Is Btor2-Val prepended with witness translation **efficient** compared to state-of-the-art software witness validators?
- **RQ4**: Is the run-time consumed by witness validators shorter than the run-time consumed by software verifiers?
- **RQ5**: Can Btor2-Cert complement conventional hardware model checking by providing additional certified verification results?

7.1 Benchmark Set

We executed our experiments on a benchmark set consisting of 1214 safety-verification tasks of Btor2 circuits, among which 758 are safe and 456 are unsafe. The verification tasks are collected from HWMCC as well as other sources and were used to compare the performance of hardware and software model checkers [5].

7.2 Experimental Settings

All experiments were conducted on machines running Ubuntu 22.04 (64 bit), each with a 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM. The resource limits imposed on verifying translated C programs and validating generated witnesses are both set to 2 CPU cores, 15 min of CPU time, and 15 GB of RAM. We used BenchExec [53] to ensure reliable resource measurement and reproducible results. Btor2-Cert uses Btor2C at commit `36c1ad52` for translating a Btor2 circuit to a C program. In our experiment, we configure the witness validator Btor2-Val to use the PDR [54] implementation in ABC [50] at commit `65ccd3cc` and BtorSim [26] as the underlying hardware model checker and simulator, respectively.[4] We also tried AVR [51] for validating correctness witnesses, but it encountered errors on many instrumented circuits even though the circuits are syntactically valid according to Btor2Tools [26].

7.3 Evaluated Verifiers and Validators

To verify the translated C programs, we used CPAchecker [12] at revision `44619` and UAutomizer [15] at commit `6fd36663` on safe tasks because they are good at constructing invariants in the competitions. We configured CPAchecker to run four algorithms based on Craig interpolation [55], including IMC [56, 57], ISMC [58], Impact [59], and predicate abstraction [60]. On unsafe tasks, we evaluated the BMC [61] implementations in CPAchecker, Cbmc [13], and Esbmc [14] because BMC is the prevailing technique for bug hunting. Both Cbmc and Esbmc were downloaded from the archiving repository of SV-COMP 2023 [52]. For UAutomizer, we used its default settings in SV-COMP for both safe and unsafe tasks.

To evaluate Btor2-Val, we prepended it with the witness-translation step and compared the combination, which takes software witnesses as input, to validators for software witnesses. For correctness witnesses, we evaluated the first place

[4] As ABC works on the bit level, we bit-blasted Btor2 circuits into the AIGER format with Btor2AIGER [26] before invoking ABC.

winner UAutomizer of the witness-validation track in SV-COMP 2023 [24]. We also used an emerging validator LIV [46] at commit cf736e45, which decomposes a program into straight-line sub-programs to check inductive invariants. We cannot compare Btor2-Val to Certifaiger [29, 30] because Certifaiger consumes a candidate inductive length as input, while Btor2-Val expects an invariant from the witnesses. For violation witnesses, we compared Btor2-Val to execution-based validators [47] CPA-w2t and FShell-w2t. The former is of the same version as CPAchecker (i.e., at revision 44619) and the latter was downloaded from the tool archive of SV-COMP 2023 [52]. We also evaluated MetaVal [27], a tool using validation via verification, but it did not terminate when instrumenting the translated C programs and failed to validate any witness in our experiment.

7.4 Results

RQ1: SW-to-HW Witness Translation. The upper part of Table 2 (resp. Table 3) shows the numbers of correctness (resp. violation) witnesses produced by the software verifiers and those successfully translated by the witness translator in Btor2-Cert. Table 2 additionally shows in its 2nd row the numbers of software witnesses with candidate invariants annotated to the loop head of a translated C program. About 97 % of the candidate invariants in software correctness witnesses can be translated to Btor2 witness circuits. The CPAchecker's 14 candidate invariants that cannot be translated were due to the C-expression parser[5] exceeding the time limit when constructing abstract syntax trees. This is a technical limitation orthogonal to the proposed approach. Furthermore, all 4 candidate invariants of UAutomizer that could not be translated refer to undeclared program variables, rendering the witnesses to be syntactically incorrect.[6]

For software violation witnesses, all of them were successfully translated by Btor2-Cert. The median translation time was below 2 s for both correctness and violation witnesses. Moreover, measured by the number of lines of a Btor2 witness, the translated correctness witnesses have a median size of 321, and the violation witnesses have a median size of 308. The results show the feasibility to translate and represent the information found by software verifiers in a native hardware-modeling format.

RQ2: Effectiveness of Btor2-Val. The lower part of Table 2 (resp. Table 3) summarizes the numbers of correctness (resp. violation) witnesses that were validated by Btor2-Val and the compared validators.

Btor2-Val was able to validate the correctness witnesses produced by both CPAchecker and UAutomizer. When configured to accept safe and inductive invariants (recall the three levels of invariant quality in Sect. 5), it validates 329 out of 576 correctness witnesses translated to Btor2 witness circuits. In contrast, UAutomizer, the winner of the witness-validation track in SV-COMP 2023 [24], was not able to validate any correctness witness produced by CPAchecker (the corresponding cells are marked as "-"). LIV is designed to confirm safe and inductive invariants [46] and accepted 305 correctness witnesses in total, similar

[5] Btor2-Cert uses pycparser 2.21 (https://github.com/eliben/pycparser).

[6] https://github.com/ultimate-pa/ultimate/issues/660

Table 2: Summary of results on validating correctness witnesses

	Verif.	CPAchecker				UAutomizer	Sum of each analysis		
Val.		IMC	ISMC	Impact	PredAbs		accepted	rejected	others
(proofs)		119	85	155	182	79	620	-	-
w/ candidate inv.		114	79	148	178	75	594	-	-
translated		113	79	139	174	71	576	-	-
Bt-Val	invariant	77	66	117	119	67	446	105	69
	safe	27	47	90	118	45	327	228	65
	safe & inductive	28	47	90	118	46	329	243	48
LIV		15	32	95	122	41	305	252	63
UAutomizer		-	-	-	-	74	74	2	3

Table 3: Summary of results on validating violation witnesses

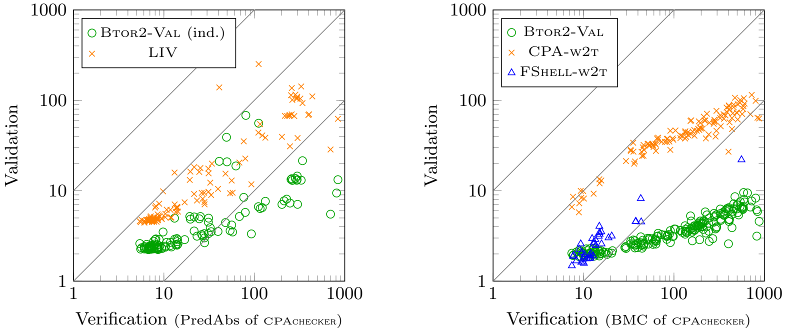
Verif.	Cbmc	CPAchecker	Esbmc	UAutomizer	Sum of each analysis		
Val.					accepted	rejected	others
(alarms)	369	197	302	31	899	-	-
Btor2-Val	59	197	295	27	578	321	0
CPA-w2t	0	122	0	0	122	-	777
FShell-w2t	44	38	44	24	150	-	749

to Btor2-Val. Btor2-Val and LIV agreed on the majority of the correctness witnesses, and the cases where they computed different verdicts were caused by a bug[7] in LIV, which has been fixed by its developers. The results show that Btor2-Val is more robust than UAutomizer and achieves similar effectiveness as LIV. We manually inspected several witnesses rejected by both Btor2-Val and LIV and found that they indeed contain incorrect candidate invariants that do not overapproximate the reachable state spaces. Such invalid invariants might be caused by bugs in the conversion step of software verifiers from its internal formula representation back to the programming language C.

Table 2 also reports the results when Btor2-Val is configured to accept correctness witnesses with different levels of invariant quality. Overall, 77 % of the candidate invariants derived by software verifiers passed the invariant check of Btor2-Val, but only 57 % are deemed safe and inductive. As expected, the number of rejections increases with the strictness for invariant quality. However, there are 2 instances in Table 2 that passed the level “safe & inductive” but were not confirmed at the level “safe” by Btor2-Val. Such cases occurred because ABC, the backend verifier of Btor2-Val, ran into timeout when performing model checking, whereas the consistency, initiation, and consecution checks based on satisfiability easily went through. Among the four interpolation-based algorithms in CPAchecker, predicate abstraction is the best in terms of invariant quality: It generated the most safe and inductive invariants. The results demonstrate the unique value of Btor2-Val to quantify the quality of invariants derived by software verifiers.

For violation witnesses, Btor2-Val was far more effective than CPA-w2t and FShell-w2t in our experiment. Among 899 violation witnesses generated by software verifiers, Btor2-Val was able to validate 578 cases; It rejected

[7] https://gitlab.com/sosy-lab/software/liv/-/issues/2



(a) Correctness witnesses

(b) Violation witnesses

Fig. 8: CPU-time comparison of verification and witness validation (unit: s)

other witnesses because they contain an incomplete or infeasible error path. In comparison, CPA-w2t and FShell-w2t only confirmed 122 and 150 witnesses, respectively. The numbers of rejected witnesses for CPA-w2t and FShell-w2t are not listed in Table 3 as the tools do not distinguish rejection of witnesses from other errors. We also observed that only 11 violation witnesses produced by CPAchecker, Esbmc, and UAutomizer were not validated by Btor2-Val, but witnesses generated by Cbmc suffered from a high rejection rate. This is because the violation witnesses of Cbmc often report an infeasible error path. Moreover, we notice that for many cases, different error paths are printed in Cbmc's violation witnesses and the console logs for its execution.[8] If we extract Btor2 violation witnesses from the console logs instead, Btor2-Val could validate 359 out of the 369 cases where Cbmc found an alarm. The effectiveness of Btor2-Val in confirming translated Btor2 violation witnesses showcases the value of Btor2-Cert because hardware designers can now trust software verifiers to detect bugs in their circuits and obtain a certified test case to trigger an error if software verifiers reported one.

RQ3: Efficiency of Btor2-Val. We compared the CPU time required for Btor2-Val and other state-of-the-art validators. From our experimental results, Btor2-Val (configured to accept safe and inductive invariants) achieved a median speedup of 2.2× over LIV for correctness witness validation, and a median speedup of 11× and 1.1× over CPA-w2t and FShell-w2t for violation witness validation, respectively. In addition, Fig. 8 shows the scatter plots for the CPU time consumption of the compared validators. A data point (x, y) in the plots corresponds to a case where CPAchecker took x seconds to produce a witness and a validator took y seconds to validate the witness. Observe that most data points of Btor2-Val are below those of other validators. The efficiency of the proposed certifying framework in translating and validating violation witnesses minimizes the overhead to apply software analyzers to find hardware bugs and makes the results of software verifiers trustworthy for hardware designers.

[8] https://github.com/diffblue/cprover-sv-comp/issues/70

RQ4: Verfication versus Validation Time. Figure 8a (resp. Figure 8b) compares the CPU time for CPAchecker to compute a verdict and generate a correctness (resp. violation) witness to the CPU time for a validator to check the witness. We can see that almost all data points are below the diagonal, indicating that validation time is typically shorter than verification time. Such speedup shows that the validators are able to utilize the information in witnesses to reconstruct proofs of correctness or violation more efficiently than verifying the task from scratch.

RQ5: Complementing HW Model Checking with Btor2-Cert. The empirical evaluation in the TACAS 2023 publication [5] on Btor2C demonstrates that software verifiers are able to complement the state-of-the-art hardware model checkers by finding more bugs and uniquely solving dozens of tasks. We take a step further and investigate whether the verification results of those additional alarms and uniquely solved tasks can be certified by Btor2-Cert.

Btor2-Cert **certified** 37, 1, and 4 alarms found by the BMC implementations of Cbmc, CPAchecker, and Esbmc, respectively, which cannot be detected by the BMC implementation of ABC.[9] The additional alarms found by Cbmc alone account up to 8 % of unsafe tasks in our benchmark set. With the help of Btor2-Cert, the violation witnesses generated by software verifiers can be translated to Btor2 witnesses and validated by BtorSim. That is, the property violation reported by software verifiers can be replayed fully in the hardware domain, demonstrating the unique ability of Btor2-Cert to provide trustworthy verification results obtained by software analyzers.

For property satisfaction, although the previous study shows that software verifiers are not as good at finding proofs for correctness as their hardware counterparts, we still observed a case where ABC (the backend verifier used by Btor2-Val) went into timeout but only required less than 3 s to reconstruct a proof using the invariant generated by CPAchecker, and another case with a 5× run-time speedup.

Summaries of Results. From the reported results, we conclude that (1) software witnesses can be translated to hardware witnesses (Table 2 and Table 3), (2) Btor2-Val is effective (Table 2 and Table 3) and efficient (Fig. 8), (3) witness validation by Btor2-Val consumes less time than software verification (Fig. 8), and (4) Btor2-Cert complements state-of-the-art hardware model checkers.

As a by-product of this work, our intensive investigation of software witnesses led to the discovery of several bugs in software verifiers. We reported the issues to the developers of the tools, and some of the bugs have been fixed. A complete list of issues that we found in software analyzers during this project is available on the supplementary webpage [62].

7.5 Threats to Validity

For **external validity**, our claims are established on a large set of Btor2 circuits to increase confidence, but it is unclear if they will hold on tasks with different

[9] We considered the 359 validated witnesses translated from console logs of Cbmc.

features that are not covered in the used benchmark set. For **construct validity**, we report that witness validation is faster than verification, but validation and verification were done on behaviorally equivalent but syntactically different models (namely, a Btor2 circuit vs. a C program). While the setting is not exactly the same as in a previous publication [4], it is necessary because our experiment is designed to investigate how information in software witnesses can be used by hardware analyzers. We compared Btor2-Val prepended with witness translation to software witness validators. The former also uses the original Btor2 circuit as input, but the validators for software do not leverage circuit information. We performed the comparison this way because the hardware witness validator Certifaiger [29] does not accept an invariant as input. For **internal validity**, we ran the experiments with the popular benchmarking framework BenchExec [53] to guarantee reproducibility.

8 Conclusion

Validating verification results is vital to make formal methods applicable in practice, as it reinforces the trust of users and offers more insights into the analyzed model. In this manuscript, we proposed Btor2-Cert, a certifying and validating hardware-verification framework built upon translators and software analyzers. Btor2-Cert is an open-source toolchain, involving the Btor2-to-C translator Btor2C, certifying verifiers for C programs, a C-to-Btor2 witness translator, the Btor2 simulator BtorSim, and the validator Btor2-Val. We evaluated Btor2-Cert's capability of transferring the information across software and hardware analyzers and providing certified verification results on a large benchmark set. By employing software model checkers for hardware verification, we identified and certified 8 % of the unsafe tasks in our benchmark set that the state-of-the-art conventional hardware model checker ABC overlooked. For future work, we will augment Btor2-Cert to accommodate temporal decomposition [48], a preprocessing technique used to simplify sequential circuits before model checking. Such extension [31] has been made to k-inductiveness validators [29, 30].

Data-Availability Statement. All verification tasks, tools, and experimental results from our evaluation are available in the reproduction artifact [63]. A previous version [64] of the reproduction package was reviewed by the Artifact Evaluation Committee. The updated version [63] fixes some bugs in the witness translator. More information is available on the supplementary webpage [62].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY). Zsófia Ádám is supported partially by the Doctoral Excellence Fellowship Programme, which is funded by the National Research, Development and Innovation Fund of the Ministry of Culture and Innovation, and the Budapest University of Technology and Economics, under a grant agreement with the National Research, Development and Innovation Office.

References

1. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009
2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
3. 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
4. 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
5. Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In: Proc. TACAS. pp. 1–21. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12
6. Tafese, J., Garcia-Contreras, I., Gurfinkel, A.: Btor2MLIR: A format and toolchain for hardware verification. In: Proc. FMCAD. pp. 55–63. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_13
7. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
8. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Proc. FMCAD. p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233
9. Biere, A., Froleyks, N., Preiner, M.: 11th Hardware Model Checking Competition (HWMCC 2020). http://fmv.jku.at/hwmcc20/, accessed: 2023-01-29
10. ISO/IEC JTC 1/SC 22: ISO/IEC 9899-2018: Information technology — Programming Languages — C. International Organization for Standardization (2018), https://www.iso.org/standard/74528.html
11. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO. pp. 75–88. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665
12. 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
13. 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
14. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Proc. ASE. pp. 888–891. ACM (2018). https://doi.org/10.1145/3238147.3240481
15. 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
16. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008). https://dl.acm.org/doi/10.5555/1855741.1855756

17. 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
18. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
19. Mukherjee, R., Tautschnig, M., Kroening, D.: v2c: A Verilog to C translator. In: Proc. TACAS. pp. 580–586. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_38
20. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2SMV: A tool for word-level verification. In: Proc. DATE. pp. 1156–1159 (2016), https://ieeexplore.ieee.org/document/7459485
21. Minhas, M., Hasan, O., Saghar, K.: Ver2Smv: A tool for automatic Verilog to SMV translation for verifying digital circuits. In: Proc. ICEET. pp. 1–5 (2018). https://doi.org/10.1109/ICEET1.2018.8338617
22. IEEE standard for Verilog hardware description language (2006). https://doi.org/10.1109/IEEESTD.2006.99495
23. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An open-source tool for symbolic model checking. In: Proc. CAV. pp. 359–364. LNCS 2404, Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
24. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
25. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., University of Iowa (2010), https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf
26. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Source-code repository of Btor2, BtorMC, and Boolector 3.0. https://github.com/Boolector/btor2tools, accessed: 2023-01-29
27. 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
28. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
29. Yu, E., Biere, A., Heljanko, K.: Progress in certifying hardware model checking results. In: Proc. CAV. pp. 363–386. LNCS 12760, Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_17
30. Yu, E., Froleyks, N., Biere, A., Heljanko, K.: Stratified certification for k-induction. In: Proc. FMCAD. pp. 59–64. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_11
31. Yu, E., Froleyks, N., Biere, A., Heljanko, K.: Towards compositional hardware model checking certification. In: Proc. FMCAD. pp. 1–11. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_12
32. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, pp. 127–144. LNCS 1954, Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
33. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT competition 2020. Artif. Intell. **301**, 103572:1–103572:25 (2021). https://doi.org/10.1016/j.artint.2021.103572

34. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine **33**(1) (2012). https://doi.org/10.1609/aimag.v33i1.2395
35. Heule, M.J.H.: The DRAT format and DRAT-TRIM checker. CoRR **1610**(06229) (October 2016). https://doi.org/10.48550/arXiv.1610.06229
36. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z
37. 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
38. Bury, G., Bobot, F.: Verifying models with DOLMEN. In: Proc. SMT Workshop. CEUR Workshop Proceedings, CEUR (2023). https://ceur-ws.org/Vol-3429/short9.pdf
39. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF - (tool presentation). In: Proc. SAT. pp. 430–435. LNCS 7317, Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_33
40. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. Formal Methods Syst. Des. **41**(1), 45–65 (2012). https://doi.org/10.1007/s10703-012-0152-6
41. Namjoshi, K.S.: Certifying model checkers. In: Proc. CAV. pp. 2–13. LNCS 2102, Springer (2001). https://doi.org/10.1007/3-540-44585-4_2
42. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_6
43. Sternagel, C., Thiemann, R.: The certification problem format. In: Proc. UITP. pp. 61–72. EPTCS 167, EPTCS (2014). https://doi.org/10.4204/EPTCS.167.8
44. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. In: Proc. FMCAD. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603022
45. Kahsai, T., Tinelli, C.: PKIND: A parallel k-induction based model checker. In: Proc. Int. Workshop on Parallel and Distributed Methods in Verification. pp. 55–62. EPTCS 72, EPTCS (2011). https://doi.org/10.4204/EPTCS.72.6
46. Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00214
47. 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
48. Case, M.L., Mony, H., Baumgartner, J., Kanzelman, R.: Enhanced verification by temporal decomposition. In: Proc. FMCAD. pp. 17–24. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351146
49. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Institute for Formal Models and Verification, Johannes Kepler University (2007). https://doi.org/10.35011/fmvtr.2007-1
50. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proc. CAV. pp. 24–40. LNCS 6174, Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
51. Goel, A., Sakallah, K.: AVR: Abstractly verifying reachability. In: Proc. TACAS. pp. 413–422. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23

52. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627829
53. 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
54. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proc. FMCAD. pp. 125–134. FMCAD Inc. (2011). https://dl.acm.org/doi/10.5555/2157654.2157675
55. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
56. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
57. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. arXiv/CoRR **2208**(05046) (July 2022). https://doi.org/10.48550/arXiv.2208.05046
58. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD. pp. 1–8. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351148
59. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14
60. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
61. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
62. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Supplementary webpage for TACAS 2024 article 'Btor2-Cert: A certifying hardware-verification framework using software analyzers'. https://www.sosy-lab.org/research/btor2-cert/
63. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Reproduction package for TACAS 2024 article 'Btor2-Cert: A certifying hardware-verification framework using software analyzers'. Zenodo (2023). https://doi.org/10.5281/zenodo.10548597
64. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Reproduction package for TACAS 2024 submission 'Btor2-Cert: A certifying hardware-verification framework using software analyzers'. Zenodo (2023). https://doi.org/10.5281/zenodo.10013059

Games

Auction-Based Scheduling

Guy Avni[1], Kaushik Mallik[2](✉), and Suman Sadhukhan[1]

[1] University of Haifa, Haifa, Israel
gavni@cs.haifa.ac.il,
ssadhukh@campus.haifa.ac.il
[2] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
kaushik.mallik@ist.ac.at

Abstract. Sequential decision-making tasks often require satisfaction of multiple, partially-contradictory objectives. Existing approaches are monolithic, where a single *policy* fulfills all objectives. We present *auction-based scheduling*, a *decentralized* framework for multi-objective sequential decision making. Each objective is fulfilled using a separate and independent policy. Composition of policies is performed at runtime, where at each step, the policies simultaneously bid from pre-allocated budgets for the privilege of choosing the next action. The framework allows policies to be independently created, modified, and replaced. We study path planning problems on finite graphs with two temporal objectives and present algorithms to synthesize policies together with bidding policies in a decentralized manner. We consider three categories of decentralized synthesis problems, parameterized by the assumptions that the policies make on each other. We identify a class of assumptions called *assume-admissible* for which synthesis is always possible for graphs whose every vertex has at most two outgoing edges.

1 Introduction

Sequential decision-making tasks often require satisfaction of multiple, partially-contradictory objectives. For example, the control *policy* of a traffic light may need to choose signals in a way that the traffic throughput is maximized *while* the maximum waiting time is minimized [34], the control policy operating an unmanned aerial vehicle may need to navigate in a way that the destination is reached *while* no-fly zones are avoided [33], the policy of an operating-system resource manager needs to allocate resources to tasks in a way that deadlocks are avoided *while* fairness is maintained [3].

We propose a decentralized synthesis framework for policies when tasks are given as a conjunction of two objectives Φ_1 and Φ_2, and the policies need to choose actions from a common action space. The key idea is that Φ_1 and Φ_2 will be accomplished, respectively, using two *action policies* α_1 and α_2—designed independently, and the composition of α_1 and α_2 at runtime will generate a policy for $\Phi_1 \wedge \Phi_2$. The challenge is that at each time point, one action needs to be chosen, whereas α_1 and α_2 might select conflicting actions. For example,

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 153–172, 2024.
https://doi.org/10.1007/978-3-031-57256-2_8

when developing a plan for a robot, Φ_1 and Φ_2 might specify two target locations, and α_1 and α_2 may select opposite directions in a location.

We propose a novel composition mechanism called *auction-based scheduling*: both policies are allocated bounded monetary *budgets*, and at each point in time, an auction (aka *bidding*) is held, where the policies *bid* from their budgets for the privilege to get scheduled for choosing the action. More formally, we equip each action policy α_i, for $i \in \{1, 2\}$, with a *bidding policy* β_i, which is a function that proposes a bid from the available budget based on the history of the interaction. A *tender* for objective Ψ is a triple $\tau = \langle \alpha, \beta, \mathbb{B} \rangle$, where α is an *action policy*, β is a *bidding policy*, and $\mathbb{B} \in (0, 1)$ is a minimal budget required for the tender to guarantee Ψ. Two tenders τ_1 and τ_2 are *compatible* if $\mathbb{B}_1 + \mathbb{B}_2 < 1$, which is when they can be composed at runtime as follows. Each Tender i, for $i \in \{1, 2\}$, is allocated an initial budget that exceeds $\mathbb{B}_i$, where the sum of budgets equals 1. At each point in time, the tenders simultaneously choose bids using their bidding policies, the higher bidder chooses an action using its action policy and pays the bid to the other tender. Thus, the sum of budgets stays constant at 1. Note that the composition gives rise to a path in the graph. The decentralized synthesis problem asks: Given a graph $\mathcal{G}$ and objectives Φ_1, Φ_2 such that $\Phi_1 \wedge \Phi_2 \neq \texttt{false}$, for each Φ_i compute τ_i such that no matter which tender it is composed with, the composition generates a path that fulfills Φ_i. The framework is sound-by-construction, namely the composition of compatible tenders satisfies $\Phi_1 \wedge \Phi_2$.

The advantage of auction-based scheduling is modularity at two levels. First, since the designs of policies do not depend on each other, they can be created independently and in parallel, e.g., by different vendors or in a parallel computation. Second, since the policies operate independently, they can be modified and replaced separately. For example, when only the objective Φ_1 changes, there is no need to alter the policy α_2, and vice versa.

Bidding for the next action encourages the policy with higher scheduling urgency to bid higher, and at the same time, the bounds on budgets ensure *fairness*, namely that no policy is starved. Auction-based scheduling adds new, complementary features to the arsenal of modular approaches in multi-objective decision-making. With the conventional decentralized synthesis approaches, the policies are composed either *concurrently* [39] or in a *turn-based* manner [23]. Concurrent actions are meaningful if each policy needs to act on its own local control variables, e.g., when the local control policies of two robots concurrently move the robot towards their destinations in a shared workspace. In our case, the set of actions is common between policies, and the concurrent interaction is unsuitable. Likewise, turn-based actions are also unsuitable in our setting because it is unclear how to assign turns to policies apriori. We will demonstrate (Ex. 2) that an inappropriate turn-assignment to policies may violate some of the objectives, while auction-based scheduling will succeed to fulfil all of them.

We study auction-based scheduling in the context of path planning on finite directed graphs with pairs of ω-regular objectives on its paths, and present algorithms for the decentralized synthesis problem with increasing levels of assumptions made by the tenders on each other: (a) Strong synthesis, with no assumptions and the most robust solution, (b) assume-admissible synthesis, with the assumption that the other tender is not purely cynical and behaves rationally with respect to its own objective, and (c) assume-guarantee synthesis, with explicit contract-based pre-coordination. We show for graphs whose every vertex has at most two outgoing edges, for every pair of ω-regular objectives Φ_1, Φ_2, and for all three classes of problems (a), (b), and (c), there exist PTIME decentralized synthesis algorithms that either compute compatible tenders or output that no compatible tenders with the respective assumptions exist; surprisingly, we show that compatible tenders always exist for (b). For general graphs, we show that the problems are in NP $\cap$ coNP. All our algorithms internally solve *bidding games* using known algorithms from the literature [38,37]. Due to the lack of space, some proofs are omitted, but can be found in the extended version [17].

2 Preliminaries

Let Σ be a finite alphabet. We use Σ^* and Σ^ω to respectively denote the set of finite and infinite words over Σ, and Σ^∞ to denote $\Sigma^* \cup \Sigma^\omega$. Let for two words $u \in \Sigma^*$ and $v \in \Sigma^\infty$, $u \leq v$ denote that u is a prefix of v, i.e., there exists a w such that $v = uw$. Given a language $L \subseteq \Sigma^\infty$, define $\mathsf{pref}(L)$ to be the set of every finite prefix in L, i.e., $\mathsf{pref}(L) := \{u \in \Sigma^* \mid \exists v \in L \,.\, u \leq v\}$.

Graphs. We formalize path planning problems on *graphs.* A graph $\mathcal{G}$ is a tuple $\langle V, v^0, E \rangle$ where V is a finite set of vertices, v^0 is a designated initial vertex, and $E \subseteq V \times V$ is a set of directed edges. If $(u, v) \in E$, then v is a *successor* of u. A *binary graph* is a graph whose every vertex has at most two successors. A *path* over $\mathcal{G}$ is a sequence of vertices $v^0 v^1 \ldots$ so that every $(v^i, v^{i+1}) \in E$. Unless explicitly mentioned, paths always start at v^0. We use $\mathit{Paths}^{\mathsf{fin}}(\mathcal{G})$ and $\mathit{Paths}^{\mathsf{inf}}(\mathcal{G})$ to denote the sets of finite and infinite paths, respectively.

A *strongly connected component* (SCC) of the graph $\mathcal{G}$ is a set S of vertices, such that there is a path between every pair of vertices of S. An SCC S is called a *bottom strongly connected component* (BSCC) if there does not exist any edge from a vertex in S to a vertex outside of S. The graph $\mathcal{G}$ is itself called strongly connected if V is an SCC.

Objectives. Fix a graph $\mathcal{G}$. An *objective Φ in $\mathcal{G}$* is a set of infinite paths, i.e., $\Phi \subseteq \mathit{Paths}^{\mathsf{inf}}(\mathcal{G})$. For an infinite path ρ, we use $\mathit{Inf}(\rho)$ to denote the set of vertices that ρ visits infinitely often. We focus on the following objectives:

Reachability: for $S \subseteq V$, $\mathit{Reach}_{\mathcal{G}}(S) := \left\{ v^0 v^1 \ldots \in \mathit{Paths}^{\mathsf{inf}}(\mathcal{G}) \mid \exists i \geq 0 \,.\, v^i \in S \right\}$,

Safety: for $S \subseteq V$, $\mathit{Safe}_{\mathcal{G}}(S) := \left\{ v^0 v^1 \ldots \in \mathit{Paths}^{\mathsf{inf}}(\mathcal{G}) \mid \forall i \geq 0 \,.\, v^i \in S \right\}$,

Büchi: for $S \subseteq V$, $\mathit{Büchi}_{\mathcal{G}}(S) := \left\{ \rho \in \mathit{Paths}^{\mathsf{inf}}(\mathcal{G}) \mid \mathit{Inf}(\rho) \cap S \neq \emptyset \right\}$,

Parity (max, even): for $\mathit{Col}\colon V \to [0; k]$ for some $k > 0$, $\mathit{Parity}_{\mathcal{G}}(\mathit{Col}) :=$ $\left\{ \rho \in \mathit{Paths}^{\mathsf{inf}}(\mathcal{G}) \mid \max \{ i \mid \exists v \in \mathit{Inf}(\rho) \,.\, \mathit{Col}(v) = i \} \text{ is even} \right\}$,

Given an objective Φ, we will use Φ^c to denote its complement, i.e., $\Phi^c = \mathit{Paths}^{\mathsf{inf}}(\mathcal{G}) \setminus \Phi$. Observe that $(\mathit{Reach}_{\mathcal{G}}(S))^c = \mathit{Safe}_{\mathcal{G}}(V \setminus S)$.

Action policies. Fix a graph $\mathcal{G}$. An *action policy* is a function $\alpha\colon \mathit{Paths}^{\mathsf{fin}}(\mathcal{G}) \to V$, choosing the next vertex to extend any given finite path ρv, where $\langle v, \alpha(\rho v)\rangle \in E$. The action policy α is *memoryless* if for every pair of distinct finite paths $\rho v, \rho' v$ that end in the same vertex v, it holds that $\alpha(\rho v) = \alpha(\rho' v)$; in this case we simply write $\alpha(v)$. An action policy α generates a unique infinite path in $\mathcal{G}$, denoted $\mathit{out}(\alpha)$, and defined inductively as follows. The initial vertex is v^0. For every prefix $v^0, \ldots, v^i$ of $\mathit{out}(\alpha)$, for $i \geq 0$, $v^{i+1} = \alpha(v^0, \ldots, v^i)$. We say that the policy α *satisfies* a given objective Φ, written $\alpha \models \Phi$ iff $\mathit{out}(\alpha) \in \Phi$.

3 The Auction-Based Scheduling Framework

Consider a graph $\mathcal{G} = \langle V, v^0, E\rangle$. A pair of objectives $\Phi_1, \Phi_2 \subseteq V^\omega$ in $\mathcal{G}$ are called *overlapping* if they have nonempty intersection (i.e., $\Phi_1 \cap \Phi_2 \neq \emptyset$). The *multi-objective planning problem* asks to synthesize an action policy that satisfies the *global* objective $\Phi_1 \cap \Phi_2$ for overlapping Φ_1, Φ_2.

We propose a decentralized approach to the problem. Our goal is to design two action policies α_1 and α_2 for Φ_1 and Φ_2, respectively. We will equip each action policy with a *bidding policy*, which it will use at runtime to bid for choosing the action at each time point. We formalize this below.

Definition 1 (Bidding policies). *A* bidding policy *is a function* $\beta\colon V \times [0,1] \to [0,1]$ *with the constraint that* $\beta(v, B) \leq B$ *for every vertex* v *and every amount of available budget* $B \in [0,1]$.

We equip a pair of action and bidding policies with a *threshold budget*, which represents the greatest lower bound on the initial budget needed for the policies to guarantee their objective, and we call the resulting triple a *tender.*

Definition 2 (Tenders). *A* tender *for a given graph* $\mathcal{G}$ *is a triple* $\langle \alpha, \beta, \mathbb{B}\rangle$ *of an action policy* α*, a bidding policy* β*, and a* threshold budget $\mathbb{B} \in [0,1]$*. The set of all tenders for* $\mathcal{G}$ *is denoted* $\mathcal{T}^{\mathcal{G}}$*. A tender* τ satisfies *an objective* Φ*, denoted* $\tau \models \Phi$*, iff* $\alpha \models \Phi$ *(i.e., when the tender is operating alone on the graph).*

Next, we formalize the composition of two tenders at runtime, which produces an action policy that uses a register of memory to keep track of the available budgets. We introduce some notation. A *configuration* is a pair $\langle v, B_1\rangle$, where v is a vertex and B_1 is the budget available to the first tender. We normalize the sum of budgets to 1, hence implicitly, the budget available to the second tender is $B_2 = 1 - B_1$. Let $\mathcal{C} = V \times [0,1]$ be the set of all configurations. For a given sequence of configurations $s = (v^0, B_1^0)(v^1, B_1^1) \ldots \in \mathcal{C}^\infty$, let $\mathsf{proj}_V(s)$ denote the path $v^0 v^1 \ldots$. A *history* is a finite sequence of configurations $\langle v^0, B_1^0\rangle \ldots \langle v^k, B_1^k\rangle \in \mathcal{C}^*$ with the constraint that $\mathsf{proj}_V(s) \in \mathit{Paths}^{\mathsf{fin}}(\mathcal{G})$. Let $\mathcal{H}$ be the set of all histories.

Definition 3 (Composition of tenders). *Let* $\mathcal{G}$ *be a graph, and* $\tau_1 = \langle \alpha_1, \beta_1, \mathbb{B}_1\rangle$ *and* $\tau_2 = \langle \alpha_2, \beta_2, \mathbb{B}_2\rangle$ *be two tenders. The tenders* τ_1 *and* τ_2 *are* compatible *iff* $\mathbb{B}_1 + \mathbb{B}_2 < 1$*. If compatible, then their* composition*, denoted* $\tau_1 \bowtie \tau_2$*, is a function*

$\tau_1 \bowtie \tau_2 \colon \mathcal{H} \to \mathcal{C}$ defined as follows. Given a history $h = \langle v^0, B_1^0 \rangle \dots \langle v^k, B_1^k \rangle \in \mathcal{H}$, let $b_1 \coloneqq \beta_1(v^k, B_1^k)$ and $b_2 \coloneqq \beta_2(v^k, 1 - B_1^k)$. Then,

- *if $b_1 \geq b_2$, then $\tau_1 \bowtie \tau_2(h) = (\alpha_1(\rho v), B_1 - b_1)$, and*
- *if $b_1 < b_2$, then $\tau_1 \bowtie \tau_2(h) = (\alpha_2(\rho v), B_1 + b_2)$.*

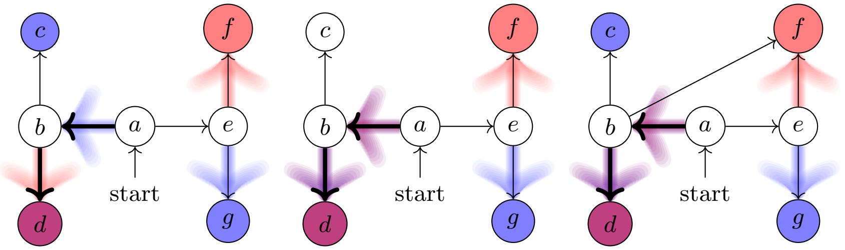
Given an initial configuration $\langle v^0, B_1^0 \rangle$ with $B_1^0 > \mathbb{B}_1$ and $B_2^0 = 1 - B_1^0 > \mathbb{B}_2$, the composition outputs an infinite sequence of configurations, denoted $\mathrm{out}(\tau_1 \bowtie \tau_2)$, where $\mathrm{out}(\tau_1 \bowtie \tau_2) \coloneqq \langle v^0, B_1^0 \rangle \langle v^1, B_1^1 \rangle \dots \in \mathcal{C}^\omega$ such that for every k, $\langle v^k, B_1^k \rangle = \tau_1 \bowtie \tau_2 \left(\langle v^0, B_1^0 \rangle \dots \langle v^{k-1}, B_1^{k-1} \rangle \right)$. We will say $\tau_1 \bowtie \tau_2$ satisfies a given objective Φ, written $\tau_1 \bowtie \tau_2 \models \Phi$, iff $\mathsf{proj}_V(\mathrm{out}(\tau_1 \bowtie \tau_2)) \in \Phi$.

We will often use the index $i \in \{1,2\}$ to refer to either of the two tenders or their attributes, and will use $-i = 3 - i$ for the "other" one, e.g., τ_i and τ_{-i}. Notice the difference between $\mathbb{B}_i$ and B_i^0: $\mathbb{B}_i$ is the threshold budget at v^0 which is a constant attribute of τ_i, whereas B_i^0 is the actual budget initially allocated to τ_i whose value can be anything above $\mathbb{B}_i$.

3.1 Classes of decentralized synthesis problem

In this section, we describe three classes of decentralized synthesis problems that we study. Throughout this section, fix a graph $\mathcal{G}$ and a given pair of overlapping objectives Φ_1 and Φ_2.

Strong decentralized synthesis. Here, tenders make no assumptions on each other, thus the solutions provide the strongest (the most robust) guarantees. Formally, for each $i \in \{1,2\}$, the goal is to construct τ_i such that for every compatible τ_{-i}, we have $\tau_i \bowtie \tau_{-i} \models \Phi_i$.



(a) Strong (b) Assume-admissible (c) Assume-guarantee

Fig. 1: Graphs with two reachability objectives given by targets: T_{blue}, depicted in blue, T_{red} depicted in red, and $T_{\mathsf{blue}} \cap T_{\mathsf{red}}$ depicted in purple. The action policies of the red and blue tenders choose edges with, respectively, red and blue shadows (shared edges are in purple). If no edges from a vertex have red or blue shadow, then the respective tender is indifferent about the choice made. Thick edges depict the paths taken by the compositions of tenders.

Example 1. Consider the graph depicted in Fig. 1a with a pair of *reachability* objectives having the targets $T_{\mathsf{blue}} = \{c, d, g\}$ and $T_{\mathsf{red}} = \{d, f\}$, respectively. Their intersection $\{d\}$ is depicted in purple. We present a pair of robust tenders τ_{blue} and τ_{red} with $\mathbb{B}_{\mathsf{blue}} = 1/4$ and $\mathbb{B}_{\mathsf{red}} = 1/2$, so that τ_{blue} and τ_{red} are compatible. We will show that τ_{blue} guarantees that no matter which compatible tender it is composed with, eventually T_{blue} is reached, and similarly τ_{red} ensures that T_{red} is reached. Therefore, $\tau_{\mathsf{blue}} \bowtie \tau_{\mathsf{red}}$ ensures that d is reached.

We first describe τ_{blue}. Consider an initial configuration $\langle a, 1/4 + \epsilon\rangle$, for any $\epsilon > 0$. Note that the other tender's budget is $3/4 - \epsilon$. The first action of τ_{blue} is $\langle b, 1/4\rangle$. There are two possibilities. First, τ_{blue} wins the bidding, then we reach the configuration $\langle b, \epsilon\rangle$, and since both successors of b are in T_{blue}, the objective is satisfied in the next step. Second, τ_{blue} loses the bidding, meaning that the other tender bids at least $1/4$, and in the worst case, we proceed to the configuration $\langle e, 1/2 + \epsilon\rangle$. Next, τ_{blue} chooses $\langle g, 1/2\rangle$ and necessarily wins as τ_{red}'s budget is only $1/2 - \epsilon$, and we reach $g \in T_{\mathsf{blue}}$. We stress that τ_{blue} can be entirely oblivious about τ_{red}, except for the implicit knowledge of τ_{red}'s budget.

We describe τ_{red}. Consider an initial configuration $\langle a, 1/2 + \epsilon\rangle$, for any $\epsilon > 0$. Initially, τ_{red} bids 0, because it does not have a preference between going left or right. In the worst case, the budget stays $1/2 + \epsilon$ in the next turn. Since both b and e have single successors in T_{red}, thus τ_{red} must win the bidding. It does so by bidding $1/2$, which exceeds the available budget $1/2 - \epsilon$ of τ_{red}. △

We now use the same problem as in Ex. 1, and show that the conventional turn-based interaction may fail to fulfill both objectives.

Example 2. Consider again the graph depicted in Fig. 1a with the targets $T_{\mathsf{blue}} = \{c, d, g\}$ and $T_{\mathsf{red}} = \{d, f\}$. Suppose α_{blue} and α_{red} are the two respective action policies, and we arbitrarily decide to make their interaction turn-based, where α_{red} chooses actions in a and α_{blue} chooses actions in b and e. It is clear that no matter which edge α_{red} chooses from a, it cannot guarantee satisfaction of T_{red}, because α_{blue} can take the game to c or g depending on α_{red}'s choice. △

Assume-admissible decentralized synthesis. While the guarantees of strong decentralized synthesis are appealing, it often fails as each tender makes the pessimistic assumption that the other tender can behave arbitrarily—even adversarially. We consider *admissibility* [23] as a stronger assumption based on rationality, ensuring compatible tenders to exist even when strong synthesis may fail. We illustrate the idea in the following example.

Example 3. Consider the graph in Fig. 1b, with reachability objectives given by targets $T_{\mathsf{blue}} = \{d, g\}$ and $T_{\mathsf{red}} = \{d, f\}$. We argue that strong decentralized synthesis is not possible. Indeed, using the same reasoning for τ_{red} in Ex. 1, we have $Th_{\mathsf{blue}}(a) = Th_{\mathsf{red}}(a) = 0.5$. On the other hand, observe that when synthesizing τ_{red}, since $c \notin T_{\mathsf{blue}}$, we know that a "rational" τ_{blue}—formally, *admissible* τ_{blue} (see Sec. 6)—will not proceed from b to c, and we can omit the edge. In turn, the threshold in a decreases to $1/4$ for both objectives. Since the sum of thresholds is now less than 1, two compatible tenders can be obtained. △

In general, we seek an *admissible-winning tender*, which ensures that its objective is satisfied when composed with any admissible tender. Admissible-winning tenders are modular because they can be reused provided that the set of admissible actions of the other tender remains unchanged. For example, even when vertex g is added to the red target set, the blue tender can be used with no change. Somewhat surprisingly, we show that in graphs in which all vertices have out-degree at most 2, assume-admissible decentralized synthesis is always possible, and a pair of admissible-winning tenders can be found in PTIME.

Assume-guarantee decentralized synthesis. Sometimes, even the admissibility assumption is too weak, and we need more direct synchronization of the tenders. We consider assume-guarantee decentralized synthesis, where each tender needs to respect a pre-specified *contract*, and as a result, their composition satisfies both objectives. We illustrate the idea below.

Example 4. Consider the graph depicted in Fig. 1c, with reachability objectives given by targets $T_{\mathsf{blue}} = \{c, d, g\}$ and $T_{\mathsf{red}} = \{d, f\}$. Here, the strong decentralized synthesis fails due to reasons similar to Ex. 3. The assume-admissible decentralized synthesis fails because from e, both objectives cannot be fulfilled, and from b, no matter which tender wins the bidding can use an admissible edge that violates the other objective (e.g., (b, c) is admissible for τ_{blue} but violates T_{red}). We consider the *contract* $\langle G_{\mathsf{blue}}, G_{\mathsf{red}}\rangle = \langle \mathbf{G}\,\neg c, \mathbf{G}\,\neg f\rangle$, which is satisfied when (a) if α_{blue} fulfills G_{blue}, then α_{red} fulfills G_{red}, and (b) if α_{red} fulfills G_{red}, then α_{blue} fulfills G_{blue}. Now whichever tender wins the bidding at b needs to fulfill its guarantee, because it cannot judge from the past interaction if the other tender violates its guarantee. Therefore, from b, the next vertex will be d under the contract, and using the same tenders from Ex. 3, both objectives will be fulfilled.

4 An Aside on Bidding Games on Graphs

All our synthesis algorithms internally solve *bidding games*, which we briefly review here; see the survey [8] for more details. A (two-player) bidding game is played between *Player X* and *Player Y*, and is a tuple $\langle \mathcal{G}, \Phi\rangle$, where $\mathcal{G} = \langle V, E\rangle$ is the (finite, directed) graph and $\Phi \subseteq V^\omega$ is the objective for *Player X*. The game is *zero-sum*, meaning that the objective of *Player Y* is $V^\omega \setminus \Phi$, i.e., the violation of Φ. This differs from auction-based scheduling where objectives overlap; otherwise, the interaction between *Player X* and *Player Y* is the same as the one between tenders. A *strategy* for a player is a pair $\langle \alpha, \beta\rangle$ where α is an action policy and β is a bidding policy. As in the composition of tenders, two strategies and an initial configuration $\langle v, B_1\rangle$ give rise to an infinite sequence of configurations called a *play*. A strategy is *winning* if no matter which strategy the opponent follows, the play satisfies the player's objective. A central quantity in bidding games is the *threshold budget* in a vertex v, which is intuitively, a necessary and sufficient initial budget for *Player X* to guarantee winning.

Definition 4 (Threshold budgets). *Consider a bidding game $\langle \mathcal{G}, \Phi\rangle$ with $\mathcal{G} = \langle V, E\rangle$. The* threshold *of Player X is given by $Th^{\mathcal{G}}_{\Phi} : V \to [0, 1]$, where for every $v \in V$, we have $Th^{\mathcal{G}}_{\Phi}(v) = \inf_B \{$Player X has a winning strategy from $\langle v, B\rangle\}$.*

The threshold of *Player Y* is denoted as $Th^{\mathcal{G}}_{\Phi^c}(v)$. The following theorem characterizes the structure of thresholds and states that the two players' thresholds are complementary. The intuition can be found in the full version [17], where we also show how winning strategies can be constructed from thresholds.

Theorem 1 ([38]). *Consider a reachability bidding game $\langle \mathcal{G}, \Phi \rangle$ where Φ is $Reach_{\mathcal{G}}(T)$ where, without loss of generality, T is a given set of sink vertices. For every vertex v, we have $Th^{\mathcal{G}}_{\Phi}(v) = 1 - Th^{\mathcal{G}}_{\Phi^c}(v)$. Moreover, for every sink vertex t, we have $Th^{\mathcal{G}}_{\Phi}(t) = 0$, if $t \in T$, and $Th^{\mathcal{G}}_{\Phi}(t) = 1$ otherwise. For every vertex v, we have $Th^{\mathcal{G}}_{\Phi}(v) = 0.5 \cdot (Th^{\mathcal{G}}_{\Phi}(v^+) + Th^{\mathcal{G}}_{\Phi}(v^-))$, where v^- and v^+ are successors of v, such that for every other successor u, we have $Th^{\mathcal{G}}_{\Phi}(v^-) \leq Th^{\mathcal{G}}_{\Phi}(u) \leq Th^{\mathcal{G}}_{\Phi}(v^+)$. Verifying if $Th^{\mathcal{G}}_{\Phi}(v) > 0.5$ for a given vertex v is in NP$\cap$ coNP in general and is in PTIME for binary graphs.*

For infinite-horizon objectives, like parity, it is known that eventually one of the BSCCs will be reached, and inside every BSCC every vertex can be reached by both players infinitely often with every arbitrary initial budget. This implies that for every parity objective, the threshold of every vertex inside every BSCC in a game graph is either 0 or 1, and fulfilling a given parity objective is equivalent to *reaching* a BSCC whose every vertex has threshold 0. We state this formally.

Theorem 2 ([10]). *Consider a bidding game $\langle \mathcal{G}, \Phi \rangle$ with a parity objective Φ. Let S be a BSCC of $\mathcal{G}$. Every vertex in S has threshold either* 0 *or* 1*, and it is* 1 *iff the highest parity index in S is odd. Moreover, for a vertex v not in a BSCC, we have $Th^{\mathcal{G}}_{\Phi}(v) = Th^{\mathcal{G}}_{Reach_{\mathcal{G}}(T)}(v)$, where T is the union of BSCCs whose vertices have threshold* 0*.*

5 Strong Decentralized Synthesis

We study the *strong decentralized synthesis* problem, where the goal is to synthesize two compatible *robust tenders*, i.e., tenders that guarantee the fulfillment of their objectives when composed with *any* compatible tender.

Definition 5 (Robust tenders). *Let $\mathcal{G}$ be a graph and Φ_i be an objective in $\mathcal{G}$. A tender τ_i is* robust *for Φ_i if for every other compatible tender $\tau_{-i} \in \mathcal{T}^{\mathcal{G}}$, we have $\tau_i \bowtie \tau_{-i} \models \Phi_i$.*

Problem 1 (STRONG-SYNT). *Define* STRONG-SYNT *as the problem whose input is a tuple $\langle \mathcal{G}, \Phi_1, \Phi_2 \rangle$, where $\mathcal{G}$ is a graph and Φ_1 and Φ_2 are overlapping ω-regular objectives in $\mathcal{G}$, and the goal is to decide whether there exists a pair of tenders $\tau_1, \tau_2 \in \mathcal{T}^{\mathcal{G}}$ such that:*

(I) τ_1 and τ_2 are compatible,
(II) τ_1 is robust for Φ_1, and
(III) τ_2 is robust for Φ_2.

Since each robust tender τ_i guarantees that Φ_i is satisfied when composed with *any* tender, the composition of two robust tenders satisfies both objectives:

Proposition 1 (Sound composition of robust tenders). *Let τ_1 and τ_2 be two compatible robust tenders for $\langle \mathcal{G}, \Phi_1, \Phi_2 \rangle$. Then $\tau_1 \bowtie \tau_2 \models \Phi_1 \cap \Phi_2$.*

We reduce the strong decentralized synthesis problem to the solution of two independent bidding games, both played on the graph $\mathcal{G}$, one with *Player X*'s objective Φ_1 and the other one with *Player X*'s objective Φ_2. When the sum of thresholds in v^0 is less than 1, we set the two tenders to be winning *Player X* strategies in the two games with the threshold budgets of the tenders being set as the respective thresholds in v^0. It follows from the construction that both tenders are robust, and hence their composition will fulfill both objectives (Prop. 1).

Theorem 3 (Strong decentralized synthesis). *Let $\mathcal{G} = \langle V, v^0, E \rangle$ be a graph and Φ_1 and Φ_2 be a pair of overlapping ω-regular objectives. A pair of robust tenders exists iff $Th^{\mathcal{G}}_{\Phi_1}(v^0) + Th^{\mathcal{G}}_{\Phi_2}(v^0) < 1$. Moreover,* $\mathsf{STRONG\text{-}SYNT}$ *is in NP $\cap$ coNP in general and is in PTIME for binary graphs.*

Proof. First, assume that $Th^{\mathcal{G}}_{\Phi_1}(v^0) + Th^{\mathcal{G}}_{\Phi_2}(v^0) < 1$. For $i \in \{1, 2\}$, let $\langle \alpha_i, \beta_i \rangle$ denote a winning *Player X* strategy in the bidding game $\langle \mathcal{G}, \Phi_i \rangle$ from every configuration $\langle v^0, B \rangle$ with $B > Th^{\mathcal{G}}_{\Phi_i}(v^0)$. We argue that the render $\tau_1 = \langle \alpha_1, \beta_1, Th^{\mathcal{G}}_{\Phi_1}(v^0) \rangle$ is robust for Φ_1, and the proof for τ_2 is dual. Indeed, for any compatible tender $\tau_2' = \langle \alpha_2', \beta_2', \mathbb{B}_2' \rangle$, the pair $\langle \alpha_2', \beta_2' \rangle$ corresponds to a *Player Y* strategy in the bidding game $\langle \mathcal{G}, \Phi_1 \rangle$. The resulting play coincides with $out(\tau_1 \bowtie \tau_2')(\langle v^0, B \rangle)$ and satisfies Φ_1 since the strategy $\langle \alpha_1, \beta_1 \rangle$ is winning.

Second, suppose that $Th^{\mathcal{G}}_{\Phi_1}(v^0) + Th^{\mathcal{G}}_{\Phi_2}(v^0) \geq 1$. For any allocation $\mathbb{B}_1 + \mathbb{B}_2 < 1$, there is an $i \in \{1, 2\}$ such that $\mathbb{B}_i \leq Th^{\mathcal{G}}_{\Phi_i}(v^0)$. Assume WLog that $\mathbb{B}_1 \leq Th^{\mathcal{G}}_{\Phi_1}(v^0)$. Consider a winning *Player Y* strategy $\langle \alpha_2, \beta_2 \rangle$ in the bidding game $\langle \mathcal{G}, \Phi_1 \rangle$ from $\langle v^0, \mathbb{B}_1 \rangle$. The tender $\tau_2' = \langle \alpha_2, \beta_2, 1 - \mathbb{B}_1 \rangle$ is compatible and $out(\tau_1 \bowtie \tau_2'(\langle v^0, \mathbb{B}_1 \rangle))$ violates Φ_1.

Finally, in order to obtain the complexity bounds, we guess memoryless action policies in both games, which are known to exist [37], and verify that they are optimal. Based on the guess, we devise a linear program to compute the thresholds. Finally, we verify that the sum of thresholds in v^0 is less than 1. For binary graphs, there is no need to guess the action policy in order to find thresholds (Thm. 1). □

We identify a setting where strong decentralized synthesis is always possible. The following theorem follows from the result that threshold budgets in strongly-connected Büchi games containing at least one accepting vertex are 0.

Theorem 4 (Strong decentralized synthesis on SCCs). *Consider a strongly-connected graph $\mathcal{G}$ and a pair of non-empty Büchi objectives in $\mathcal{G}$. Then, a pair of robust tenders exists in $\mathcal{G}$.*

We demonstrate the effectiveness of strong synthesis using path planning problems with two reachability objectives. Consider a fixed grid but four different instances of the problem, as shown in Fig. 2. For the first three cases, we successfully obtain pairs of robust tenders whose compositions fulfill both objectives. Moreover, since the blue target remained the same in all cases, we needed to redesign only the red tender, saving us a significant amount of computation.

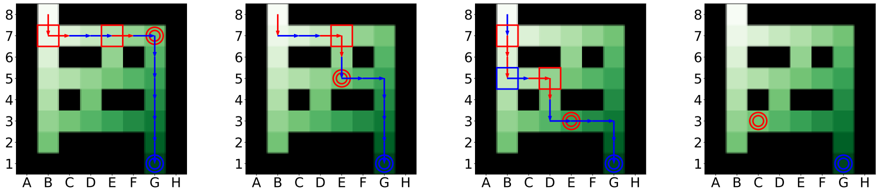


Fig. 2: Robust tenders for path planning with two reachability objectives on a one-way grid, where the black cells are obstacles and *the only permissible moves are from lighter to darker green cells*—and not the other way round. The cell B8 is the initial location. The cells with double circles of colors red (respectively, G7, E5, E3, C3) and blue (G1) are the targets to reach. The path shows the output of the composition of the two tenders, where the red and blue segments are actions which were chosen by the red and blue tenders, respectively. The cells with red and blue squares are locations where the respective tenders win the bidding; in the rest of the cells on the paths, the bidding ended in ties which were resolved randomly. Strong synthesis was successful in the first three instances and failed in the last one. The pairs of thresholds of red and blue targets are, respectively (left to right): $(0.75, 0.125)$, $(0.625, 0.125)$, $(0.75, 0.125)$, $(0.875, 0.125)$.

6 Assume-Admissible Decentralized Synthesis

In assume-admissible decentralized synthesis, each tender assumes that the other tender is *rational* and pursues its own objective. We formalize rationality by adapting the well-known concepts of *dominance* and *admissibility* from game theory [1,22]. Intuitively, τ_i dominates τ_i' if τ_i is always at least as good as τ_i and sometimes strictly better than τ_i; therefore, there is no reason to use τ_i'. An admissible tender is one that is not dominated by any other tender.

Definition 6 (Dominance, admissibility). *Let $\mathcal{G}$ be a graph and Φ be an objective. We provide definitions for the first tender and the definitions for the second tender are dual. Let $\mathbb{B}_1 < 1$. For two tenders τ_1 and τ_1' that have the same budget allocation, τ_1* dominates *τ_1' when*

(a) τ_1 performs as well as τ_1' when composed with any compatible τ_2; formally, for every compatible tender τ_2, $\tau_1' \bowtie \tau_2 \models \Phi$ implies $\tau_1 \bowtie \tau_2 \models \Phi$, and

(b) there is a compatible tender τ_2 for which τ_1 performs better than τ_1'; formally, there exists a compatible τ_2 with $\tau_1 \bowtie \tau_2 \models \Phi$, and $\tau_1' \bowtie \tau_2 \not\models \Phi$.

*A tender τ_1 is called Φ-*admissible *in $\mathcal{G}$ iff it is not dominated by any other tender. We denote the set of Φ-admissible tenders in $\mathcal{G}$ by $Adm^{\mathcal{G}}(\Phi)$.*

Next, we define admissible-winning tenders, which are tenders that fulfill their objectives when composed with *any* admissible tender.

Definition 7 (Admissible-winning tenders). *Let $\mathcal{G}$ be a graph and Φ_1, Φ_2 be a pair of overlapping objectives in $\mathcal{G}$. A tender τ_i is called Φ_{-i}-*admissible-winning *for Φ_i if and only if $\tau_i \in Adm^{\mathcal{G}}(\Phi_i)$, and for every other tender $\tau_{-i} \in Adm^{\mathcal{G}}(\Phi_{-i})$ compatible with τ_i, we have $\tau_i \bowtie \tau_{-i} \models \Phi_i$.*

When the objectives are clear from the context, we will omit them and will simply write a tender is "admissible tender," "admissible-winning tender," etc.

Problem 2 (AA-SYNT). *Define* AA-SYNT *as the problem whose input is a tuple $\langle \mathcal{G}, \Phi_1, \Phi_2 \rangle$, where $\mathcal{G}$ is a graph and Φ_1 and Φ_2 are overlapping ω-regular objectives in $\mathcal{G}$, and the goal is to decide whether there exists a pair of tenders $\tau_1 \in Adm^{\mathcal{G}}(\Phi_1)$ and $\tau_2 \in Adm^{\mathcal{G}}(\Phi_2)$ such that:*

(I) τ_1 and τ_2 are compatible,
(II) τ_1 is Φ_2-admissible-winning for Φ_1, and
(III) τ_2 is Φ_1-admissible-winning for Φ_2.

The following proposition follows from the requirement that τ_1 and τ_2 are admissible.

Proposition 2 (Sound composition of admissible-winning tenders). *Let τ_1 and τ_2 be tenders that fulfill the requirements stated in Prob. 2. Then, $\tau_1 \bowtie \tau_2 \models \Phi_1 \cap \Phi_2$.*

Remark 1. Note that the synthesis procedure for each Φ_{-i}-admissible-winning tender τ_i for Φ_i requires the knowledge of $Adm^{\mathcal{G}}(\Phi_{-i})$. Assume-admissible decentralized synthesis is modular in the following sense. First, the specific implementation of the tender τ_{-i} with which each τ_i is composed is not known during synthesis. All that is known is the objective Φ_{-i} for which τ_{-i} is synthesized. Second, each τ_i can remain unchanged even when Φ_{-i} changes to Φ'_{-i}, as long as $Adm^{\mathcal{G}}(\Phi'_{-i}) \subseteq Adm^{\mathcal{G}}(\Phi_{-i})$.

6.1 Reachability objectives

Throughout this section we focus on overlapping reachability objectives $\Phi_1 = Reach_{\mathcal{G}}(T_1)$ and $\Phi_2 = Reach_{\mathcal{G}}(T_2)$ with $T_1, T_2 \subseteq V$ being sets of sink target vertices. This is without loss of generality, as every graph with non-sink target vertices can be converted into a graph with sink target vertices by adding memory (see the full version [17]).

We reduce the decentralized assume-admissible synthesis problem to solving a pair of zero-sum bidding games on a sub-graph of $\mathcal{G}$. Intuitively, an edge $e = \langle u, v \rangle$ is *dominated* for the i-th tender, for $i \in \{1, 2\}$, if it is possible to achieve the objective Φ_i from u but not from v. Clearly, a tender that chooses e is dominated and is thus not admissible (see the proof of the lemma in the full version [17]). Recall that $Th^{\mathcal{G}}_{\Phi_i}(v)$ denotes the threshold in the zero-sum bidding game played on $\mathcal{G}$ with the *Player X* objective Φ_i, and that $Th^{\mathcal{G}}_{\Phi_i}(v) = 1$ means there is no path from v to T_i.

Lemma 1 (A necessary condition for admissibility). *For every vertex u having at least two successors v, w with $Th^{\mathcal{G}}_{\Phi_i}(v) < 1$ and $Th^{\mathcal{G}}_{\Phi_i}(w) = 1$, if a Player i tender $\langle \alpha_i, \beta_i, \mathbb{B}_i \rangle$ is in $Adm^{\mathcal{G}}(\Phi_i)$, then $\alpha_i(u) \neq w$, for both $i \in \{1, 2\}$.*

Proof. We argue that choosing w from u is dominated by the action of choosing v from u, no matter what the budget at u is. Firstly, Cond. (a) of Def. 6 trivially holds. Secondly, consider the other tender τ_{-i} which bids zero at u, and later cooperates with τ_i to satisfy Φ_i. Clearly, the τ_i's action policy that selects v at u will be able to satisfy Φ_i, but the one that selects w will not. □

We obtain the reduced graph by omitting edges that are dominated for both players. For example, in Fig. 1b, the edge $\langle b, c \rangle$ is dominated for both players (see Ex. 3) and in Fig. 1c, no edge is dominated for both players (see Ex. 4).

Definition 8 (Largest admissible sub-graphs for reachability). *The largest admissible sub-graph of $\mathcal{G}$ with respect to two reachability objectives Φ_1 and Φ_2 is $\widehat{\mathcal{G}}_{\Phi_1,\Phi_2} = \langle V', E' \rangle$ with $V' = V \setminus \left\{ v \in V \mid Th^{\mathcal{G}}_{\Phi_1}(v) = 1 \wedge Th^{\mathcal{G}}_{\Phi_2}(v) = 1 \right\}$ and $E' = (V' \times V') \cap E$. We omit Φ_1, Φ_2 from $\widehat{\mathcal{G}}_{\Phi_1,\Phi_2}$ when it is clear from the context.*

For a vertex v in $\mathcal{G}$ and $i \in \{1, 2\}$, recall that $Th^{\Phi_i}_{\mathcal{G}}(v)$ denotes the threshold in $\mathcal{G}$ for objective Φ_i, and $Th^{\Phi_i}_{\widehat{\mathcal{G}}}(v)$ denotes the threshold in the reduced graph. Observe that a winning strategy in $\mathcal{G}$ will never cross a dominated edge. Removing dominated edges restricts the opponent, thus $Th^{\Phi_i}_{\mathcal{G}}(v) \geq Th^{\Phi_i}_{\widehat{\mathcal{G}}}(v)$. The next lemma shows that, surprisingly, the decrease in sum of thresholds is guaranteed to be significant. The proof (see the full version [17]) which holds for non-binary graphs, intuitively follows from observing that in $\widehat{\mathcal{G}}$, necessarily a sink that is a target for one of the players is reached, and since there is an overlap in at least one sink, the sum of thresholds is at most 1.

Lemma 2 (On the sum of thresholds in $\widehat{\mathcal{G}}$). *For every vertex v, we have $Th^{\widehat{\mathcal{G}}}_{\Phi_1}(v) + Th^{\widehat{\mathcal{G}}}_{\Phi_2}(v) \leq 1$. Moreover, if $\mathcal{G}$ is binary then $Th^{\widehat{\mathcal{G}}}_{\Phi_1}(v) + Th^{\widehat{\mathcal{G}}}_{\Phi_2}(v) < 1$.*

Our synthesis procedure proceeds as in strong decentralized synthesis: Find and output a pair of robust tenders in $\widehat{\mathcal{G}}$, which are guaranteed to exist when $\mathcal{G}$ is binary. In order to maintain soundness (see Prop. 2), it is key to show that a robust tender τ_i in $\widehat{\mathcal{G}}$ is admissible in $\mathcal{G}$. The proof of the following lemma is intricate (see the full version [17]). We show that even when one can find τ'_i and τ_{-i} such that $\tau_i \bowtie \tau_{-i} \not\models \Phi_i$ but $\tau'_i \bowtie \tau_{-i} \models \Phi_i$, it is possible to construct τ'_{-i} for which $\tau_i \bowtie \tau'_{-i} \models \Phi_i$ but $\tau'_i \bowtie \tau'_{-i} \not\models \Phi_i$, thus τ'_i does not dominate τ_i. Furthermore, such a tender wins against a set of tenders which *over-approximates* admissible tender for Φ_{-i}.

Lemma 3 (Algorithm for computing admissible-winning tenders). *For $i \in \{1, 2\}$, a robust tender for Φ_i in $\widehat{\mathcal{G}}$ is Φ_{-i}-admissible-winning for Φ_i in $\mathcal{G}$.*

The following theorem is obtained by combining Lemmas 2 and 3.

Theorem 5 (Assume-admissible decentralized synthesis for reachability). *The problem* $\mathsf{AA\text{-}SYNT}$ *is a tautology for binary graphs: for every binary graph and two overlapping reachability objectives, there exists a pair of compatible admissible-winning tenders. Moreover, the tenders can be computed in PTIME.*

Remark 2. For general (i.e., non-binary) graphs, $\mathsf{AA\text{-}SYNT}$ is not a tautology anymore; a counter-example is given in Ex. 4. However, the same PTIME algorithm for computing tenders can still be used to obtain a sound solution; the completeness question is left open for future work.

6.2 Büchi objectives

In this section, we consider binary graphs with a pair of overlapping Büchi objectives. We first demonstrate that, unlike reachability, it is not guaranteed that an assume-admissible decentralized solution exists.

Example 5. Consider the graph depicted in Fig. 3 with the Büchi objectives given by the accepting vertices $S_{\mathsf{red}} = \{b, d\}$ and $S_{\mathsf{blue}} = \{a, c\}$. Note that the objectives are overlapping since the path $(bc)^\omega$ satisfies both. We argue that no pair of compatible admissible-winning tenders exist. Note that a robust (hence dominant) red tender forces reaching d, thus forcing Φ_{red} to be satisfied. Dually, a robust blue tender forces Φ_{blue} in a. It can be shown that $Th^{\mathcal{G}}_{\Phi_{\mathsf{red}}}(b) = 2/3$ and $Th^{\mathcal{G}}_{\Phi_{\mathsf{blue}}}(b) = 1/3$. Thus, for any $\mathbb{B}_{\mathsf{red}}$ and $\mathbb{B}_{\mathsf{blue}}$ with $\mathbb{B}_{\mathsf{red}} + \mathbb{B}_{\mathsf{blue}} < 1$, there is a robust tender that violates the other tender's objective. △

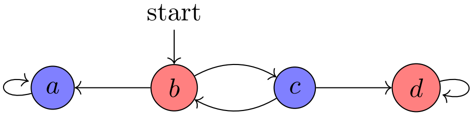


Fig. 3: A graph with no assume-admissible decentralized solution.

We generalize the concept of largest admissible subgraphs to Büchi objectives. It is not hard to show that proceeding into a BSCC with an accepting state is admissible. Indeed, Thm. 2 shows that there is a robust (hence admissible) tender in such BSCCs. On the other hand, proceeding to a BSCC with no accepting vertex is clearly not admissible. The largest admissible subgraph is obtained by repeatedly removing BSCC that are not admissible for both tenders. Formally, for a given action policy α and a given vertex v of $\mathcal{G}$, we will write $\alpha \not\models_v \Phi_1 \cup \Phi_2$ to indicate that the action policy cannot fulfill $\Phi_1 \cup \Phi_2$ from the initial vertex v.

Definition 9 (Largest admissible sub-graphs for Büchi). *The largest admissible sub-graph* $\widehat{\mathcal{G}}_{\mathcal{B}}$ *of* $\mathcal{G}$ *for the Büchi objectives* Φ_1, Φ_2 *is the graph* $\langle V', E' \rangle$ *with* $V' = V \backslash \{v \in V \mid \forall$ *action policy* α . $\alpha \not\models_v \Phi_1 \cup \Phi_2\}$, *and* $E' = (V' \times V') \cap E$.

We describe a reduction to reachability games. For $i \in \{1, 2\}$, let T_i denote the union of BSCCs of $\widehat{\mathcal{G}}_{\mathcal{B}}$ in which there is at least one Büchi accepting vertex for Φ_i. We call $\langle T_1, T_2 \rangle$ the *reachability core* of $\langle \Phi_1, \Phi_2 \rangle$ in $\widehat{\mathcal{G}}_{\mathcal{B}}$. Let $\Phi'_1 = \mathit{Reach}_{\widehat{\mathcal{G}}_{\mathcal{B}}}(T_1)$ and $\Phi'_2 = \mathit{Reach}_{\widehat{\mathcal{G}}_{\mathcal{B}}}(T_2)$. We proceed as in strong decentralized synthesis: we find

$Th^{\widehat{\mathcal{G}}_{\mathcal{B}}}_{\Phi_1'}(v^0)$ and $Th^{\widehat{\mathcal{G}}_{\mathcal{B}}}_{\Phi_2'}(v^0)$ and return a pair of robust tenders if their sum is strictly less than 1. Note that unlike reachability objectives, in Büchi objectives the sum might be 1 as in Ex. 5. Moreover, as Ex. 5 illustrates, when the sum is 1, no pair of admissible-winning tenders exist. By adapting results from the previous section, we obtain the following.

Theorem 6 (Assume-admissible decentralized synthesis for Büchi). *Let $\mathcal{G}$ be a binary graph and Φ_1, Φ_2 be a pair of overlapping Büchi objectives. Let $\langle T_1, T_2\rangle$ be the reachability core of $\langle \Phi_1, \Phi_2\rangle$ in the largest admissible sub-graph of $\mathcal{G}$ (for Φ_1, Φ_2). A pair of admissible-winning tenders exists iff $T_1 \cap T_2 \neq \emptyset$. Moreover,* AA-SYNT *for Büchi objectives is in PTIME.*

Like reachability (see Rem. 2), for Büchi objectives the same algorithm for AA-SYNT for binary graphs can be used to obtain a sound solution for general graphs, and the completeness question is left open for future work.

7 Assume-Guarantee Decentralized Synthesis

We present the assume-guarantee decentralized synthesis problem, the one with the highest degree of synchronization among the tenders, with the benefit of the most applicability. In this synthesis procedure, we assume that we are given a pair of languages $A_1, A_2 \subseteq V^\omega$, called the *assumptions*. Intuitively, each tender τ_i can *assume* A_i is fulfilled by the other tender, and, in return, needs to *guarantee* that A_{-i} is fulfilled, in addition to fulfilling own objective.

Definition 10 (Contract-abiding tenders). *Let $\mathcal{G}$ be a graph, Φ_i be an ω-regular objective, and A_1, A_2 be a pair of ω-regular assumptions in $\mathcal{G}$. We say a tender $\tau_i = \langle \alpha_i, \cdot, \cdot\rangle \in \mathcal{T}^{\mathcal{G}}$ fulfills Φ_i under the contract $\langle A_1, A_2\rangle$, written $\tau_i \models \langle A_i \triangleright \Phi_i \triangleright A_{-i}\rangle$, iff*

(a) for every finite path ρ, if ρ is in $\mathsf{pref}(A_i)$, then $\rho \cdot \alpha_i(\rho) \in \mathsf{pref}(A_1 \cap A_2)$, and
(b) for every other compatible tender $\tau_{-i} \in \mathcal{T}^{\mathcal{G}}$, we have $\tau_i \bowtie \tau_{-i} \models \mathit{Safe}_{\mathcal{G}}(\mathsf{pref}(A_1 \cap A_2)) \implies (A_{-i} \wedge (A_i \implies \Phi_i))$.

Here, each tender τ_i only make safety assumption on the other tender (Cond. (a)), namely that the path does not leave the safe set $\mathsf{pref}(A_i)$, and in return, provides full guarantee on A_{-i} (Cond. (b)). Normally, safety assumptions are not enough for fulfilling liveness guarantees and objectives [5]. But in bidding games, within the safe set, the players can use a known bidding tactic [9] to accumulate enough budgets from time to time to reach the liveness goals always eventually. We use A_1, A_2 as ω-regular sets, though we conjecture that safety assumptions suffice. The assume-guarantee distributed synthesis problem asks to compute a pair of tenders that fulfill their objectives under the given contract, as stated below.

Problem 3 (AG-SYNT). *Define* AG-SYNT *as the problem that takes as input a tuple $\langle \mathcal{G}, \Phi_1, \Phi_2, A_1, A_2\rangle$, where $\mathcal{G}$ is a graph, Φ_1 and Φ_2 are overlapping objectives in $\mathcal{G}$, and A_1 and A_2 are two ω-regular languages over V with $v^0 \in \mathsf{pref}(A_1 \cap A_2)$, and the goal is to decide whether there exists a pair of tenders $\tau_1, \tau_2 \in \mathcal{T}^{\mathcal{G}}$ such that:*

(I) τ_1 *and* τ_2 *are compatible,*
(II) $\tau_1 \models \langle A_1 \triangleright \Phi_1 \triangleright A_2 \rangle$*, and*
(III) $\tau_2 \models \langle A_2 \triangleright \Phi_2 \triangleright A_1 \rangle$*.*

When the assumptions allow all behaviors, i.e., $A_1 = A_2 = V^\omega$, then $\mathsf{AG\text{-}SYNT}$ is equivalent to $\mathsf{STRONG\text{-}SYNT}$. On the other hand, when the assumptions allow only each other's objectives, i.e., $A_1 = \Phi_1$ and $A_2 = \Phi_2$, then we obtain a purely cooperative synthesis algorithm. We prove that satisfaction of the contracts by a pair of tenders will imply satisfaction of $\Phi_1 \cap \Phi_2$.

Proposition 3 (Sound composition of contract-abiding tenders). *Let* τ_1 *and* τ_2 *be tenders that fulfill the requirements stated in Prob. 3. Then,* $\tau_1 \bowtie \tau_2 \models \Phi_1 \cap \Phi_2$*.*

Proof. In the following, for a given language $L \in V^\omega$, we write $\mathit{Safe}_\mathcal{G}(\mathsf{pref}(L))$ to denote the set of infinite paths which can always be extended to L, i.e., $\left\{ v^0 v^1 \ldots \in \mathit{Paths}^{\mathsf{inf}}(\mathcal{G}) \mid \forall i \geq 0 \,.\, v^0 \ldots v^i \in \mathsf{pref}(L) \right\}$.

We claim that both assumptions A_1, A_2 will be fulfilled, from which Cond. (b) of Def. 10 will imply satisfaction of both Φ_1 and Φ_2 by τ_1 and τ_2, respectively. Let $A = A_1 \cap A_2$, and A can be decomposed into safety and liveness components as $A = \mathit{Safe}_\mathcal{G}(\mathsf{pref}(A)) \cap \big(\mathit{Safe}_\mathcal{G}(\mathsf{pref}(A)) \implies A\big)$. We prove the claim on the two components separately. Firstly, the fact that $\tau_1 \bowtie \tau_2$ implements $\mathsf{pref}(A)$ on $\mathcal{G}$ can be proven by induction over the length of the generated path: The base case is given by the assumption $v^0 \in \mathsf{pref}(A_1 \cap A_2)$, and for every finite path ρ, if τ_i wins the bidding and if $\rho \in \mathsf{pref}(A_1 \cap A_2) \subset \mathsf{pref}(A_i)$ then τ_i needs to ensure that the next vertex v' satisfies $\rho v' \in \mathsf{pref}(A_i \cap A_{-i})$ (consequence of Cond. II-III of Prob. 3 and Cond. (a) of Def. 10), thereby implying that the path will always remain inside $\mathsf{pref}(A_1 \cap A_2)$, proving the safety part.

For the liveness part, we use known results from Richman bidding games, which guarantee that in an infinite horizon game, with any arbitrary positive initial budget, players can always eventually visit any vertex that can be reached [37]. This implies that if the invariance $\mathit{Safe}_\mathcal{G}(\mathsf{pref}(A_1 \cap A_2))$ holds, then each tender τ_i can actually fulfill A_{-i} (they are required to do so by Cond. (b) of Def. 10) when composed with any compatible tender in the long run. Therefore, $A_1 \cap A_2$ will be fulfilled. □

In bidding games literature, it is unknown how to compute strategies for objectives which can be violated if a given assumption is violated by the opponent, like in Cond. (a) in Def. 10. The challenge stems from the lack of separation of the set of available actions to the players, preventing us to impose assumptions only on the opponent's behavior. We present a practically motivated sound, but possibly incomplete, solution for the decentralized synthesis problem, by using a stronger way of satisfying the contract, namely by requiring each tender τ_i to use actions so that the generated path remains in $\mathsf{pref}(A_1 \cap A_2)$ all the time. Formally, we say that the tender τ_i *strongly* fulfills Φ_i under the contract $\langle A_1, A_2 \rangle$, written $\tau_i \models_s \langle A_i \triangleright \Phi_i \triangleright A_{-i} \rangle$, if, instead of Cond. (a) of Def. 10, for every finite path ρ, we have $\rho \cdot \alpha_i(\rho) \in \mathsf{pref}(A_1 \cap A_2)$, regardless of whether $\rho \in \mathsf{pref}(A_i)$

or not, and moreover Cond. (b) of Def. 10 is fulfilled. It is easy to show that $\tau_i \models_s \langle A_i \triangleright \Phi_i \triangleright A_{-i}\rangle$ implies $\tau_i \models \langle A_i \triangleright \Phi_i \triangleright A_{-i}\rangle$, so that $\Phi_1 \cap \Phi_2$ will be fulfilled.

Similar to AA-SYNT, we extract a sub-graph $\mathcal{G}'$ of $\mathcal{G}$, called the *largest contract-satisfying sub-graph*, whose every path belongs to $\mathsf{pref}(A_1 \cap A_2)$, and vice versa; we omit the construction, which follows usual automata-theoretic procedure from the literature [4]. For example, in Ex. 4, the largest contract-satisfying sub-graph of the graph in Fig. 1c is the one that only excludes the vertices c and f. It follows that when the tenders strongly fulfill their objectives under the contracts, it is guaranteed that every path always remains in $\mathcal{G}'$.

Theorem 7 (Assume-guarantee decentralized synthesis). *Let $\mathcal{G} = \langle V, v^0, E\rangle$ be a graph, Φ_1 and Φ_2 be a pair of overlapping ω-regular objectives, and A_1 and A_2 be ω-regular assumptions. Let $\mathcal{G}'$ be the largest contract-satisfying sub-graph of $\mathcal{G}$. A pair of robust tenders exist if $Th^{\mathcal{G}'}_{A_2\cap\Phi_1}(v^0) + Th^{\mathcal{G}'}_{A_1\cap\Phi_2}(v^0) < 1$. Moreover,* AG-SYNT *is in PTIME.*

8 Related Work

Shielding [35] is a framework in which a runtime monitor called a *shield* enforces an unverified policy π (e.g., generated using reinforcement learning [7]) to satisfy a given specification. A shield operates by observing, at each point in time, the action proposed by π and can alter it, e.g., if safety is violated. The choice of who acts at each point in time, π or the shield, can be seen as a scheduling choice similar to our setting. However, the goals of the two approaches are different: our goal is to design tenders for modular policy synthesis, whereas a shield is meant as a verified "wrapper" for a complex policy. Technically, in auction-based scheduling, the scheduling depends on the auction which is external to the policies, whereas in shielding, it is the shield who chooses whether to override π.

In *distributed reactive synthesis* [42,36,32], the goal is to design a collection of Mealy machines whose communication is dictated by a given *communication architecture*. Distributed synthesis is well studied and we point to a number of works that considered objectives that are a conjunction $\Phi_1 \wedge \Phi_2 \wedge \ldots$ of sub-objectives $\Phi_1, \Phi_2, \ldots$ [30,21,39,31,18,6]. While there is a conceptual similarity between our synthesis of tenders and the synthesis of Mealy machines, there is a fundamental difference between the approaches. Namely, our composition is based on scheduling, i.e., exactly one policy is scheduled at each point in time, whereas in distributed synthesis, the composition of the Mealy machines is performed in parallel, i.e., they all read and write at each point in time.

Our algorithms build upon the rich literature on bidding games on graphs. The bidding mechanism that we focus on is called *Richman* bidding [38,37,10]. Other bidding mechanisms have been studied: *poorman* [11], *taxman* [13], and *all-pay* [14,15]. Auction-based scheduling can be instantiated with any of these mechanisms and the properties from bidding games transfer immediately (which differ significantly for quantitative objectives). Of particular interest in practice is *discrete bidding*, in which the granularity of the bids is restricted [27,2,16]. To the best of our knowledge, beyond our work, non-zero-sum bidding games have only been considered in [40]. The solution concept that they consider is *subgame perfect equilibrium* (SPE). While it is suitable to model the interaction between selfish agents, we argue that it is less suitable in decentralized synthesis.

There are many works on designing optimal policies for multi-objective sequential decision making problems for various different system models; see the survey by Roijers et al. [43] and works on multi-objective stochastic games [20,21,24]. To the best of our knowledge, no prior work considers the decomposition of the problem into individual task-dependent policies like us. Auctions to distribute tasks to agents have been considered extensively [29,44,25,28,19,26,41]. Their goal is very different: their agents bid for tasks, that is, a bid represents an agent's cost (e.g., in terms of resources) for performing a task. The auction then allocates the tasks to agents so as to minimize the individual costs, giving rise to an efficient global policy.

9 Conclusion and Future Work

We present the auction-based scheduling framework. Rather than synthesizing a monolithic policy for a conjunction of objectives $\Phi_1 \wedge \Phi_2$, we synthesize two independent tenders for each of the objectives and compose the tenders at runtime using auction-based scheduling. A key advantage of the framework is modularity; each tender can be synthesized and modified independently. We study three instantiations of decentralized synthesis in planning problems with varying degree of flexibility and practical usability, and develop algorithms based on bidding games. Interestingly, we show that a pair of admissible-winning tenders always exists in binary graphs for reachability objectives and they can be found in PTIME. This positive result illustrates the strength and potential of the auction-based scheduling framework.

There are plenty of directions of future research and we list a handful. First, we consider only qualitative objectives and it is interesting to lift the results to *quantitative* objectives, where one can quantify the fairness achieved by the scheduling mechanism in a fine-grained manner. Moreover, it is appealing to employ the rich literature on *mean-payoff* bidding games. Second, we consider a conjunction of two objectives, and it is interesting to extend the approach to a conjunction of multiple objectives. This will require extending the theory of bidding games to the multi-player setting, which have not yet been studied. Finally, it is particularly interesting to extend the technique of auction-based scheduling beyond path-planning problems, for example, it is interesting to consider decentralized synthesis of controllers that operate in an adversarial or probabilistic environment. Again, the corresponding bidding games need to be studied (so far only sure winning has been considered for bidding games played on MDPs [12]).

Acknowledgements

This work was supported in part by the ERC project ERC-2020-AdG 101020093 and by ISF grant no. 1679/21.

References

1. Adam, B., Amanda, F., Keisler, H.J.: Admissibility in games. In: Econometrica (2008)
2. Aghajohari, M., Avni, G., Henzinger, T.A.: Determinacy in discrete-bidding infinite-duration games. Log. Methods Comput. Sci. **17**(1) (2021)
3. de Alfaro, L., Faella, M., Majumdar, R., Raman, V.: Code aware resource management. In: Proceedings of the 5th ACM international conference on Embedded software. pp. 191–202 (2005)
4. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed computing **2**, 117–126 (1987)
5. Amla, N., Emerson, E.A., Namjoshi, K., Trefler, R.: Assume-guarantee based compositional reasoning for synchronous timing diagrams. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 465–479. Springer (2001)
6. Anand, A., Nayak, S.P., Schmuck, A.K.: Contract-based distributed synthesis in two-objective parity games. arXiv preprint arXiv:2307.06212 (2023)
7. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T.A., Könighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: Proc. 31st CAV. pp. 630–649 (2019)
8. Avni, G., Henzinger, T.A.: A survey of bidding games on graphs. In: Proc. 31st CONCUR. LIPIcs, vol. 171, pp. 2:1–2:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
9. Avni, G., Henzinger, T.A., Chonev, V.: Infinite-duration bidding games. In: Proc. 28th CONCUR. LIPIcs, vol. 85, pp. 21:1–21:18 (2017)
10. Avni, G., Henzinger, T.A., Chonev, V.: Infinite-duration bidding games. J. ACM **66**(4), 31:1–31:29 (2019)
11. Avni, G., Henzinger, T.A., Ibsen-Jensen, R.: Infinite-duration poorman-bidding games. In: Proc. 14th WINE. LNCS, vol. 11316, pp. 21–36. Springer (2018)
12. Avni, G., Henzinger, T.A., Ibsen-Jensen, R., Novotný, P.: Bidding games on markov decision processes. In: Proc. 13th RP. pp. 1–12 (2019)
13. Avni, G., Henzinger, T.A., Zikelic, D.: Bidding mechanisms in graph games. J. Comput. Syst. Sci. **119**, 133–144 (2021)
14. Avni, G., Ibsen-Jensen, R., Tkadlec, J.: All-pay bidding games on graphs. In: Proc. 34th AAAI. pp. 1798–1805. AAAI Press (2020)
15. Avni, G., Jecker, I., Žikelić, Đ.: Infinite-duration all-pay bidding games. In: Proc. 32nd SODA. pp. 617–636 (2021)
16. Avni, G., Sadhukhan, S.: Computing threshold budgets in discrete-bidding games. In: Proc. 42nd FSTTCS. LIPIcs, vol. 250, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
17. Avni, G., Mallik, K., Sadhukhan, S.: Auction-based scheduling. CoRR **abs/2310.11798** (2023), `https://doi.org/10.48550/arXiv.2310.11798`
18. Bansal, S., De Giacomo, G., Di Stasio, A., Li, Y., Vardi, M.Y., Zhu, S.: Compositional safety ltl synthesis. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 1–19. Springer (2022)
19. Basile, F., Chiacchio, P., Di Marino, E.: An auction-based approach to control automated warehouses using smart vehicles. Control Engineering Practice **90**, 285–300 (2019)
20. Basset, N., Kwiatkowska, M., Topcu, U., Wiltsche, C.: Strategy synthesis for stochastic games with multiple long-run objectives. In: Tools and Algorithms for

the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21. pp. 256–271. Springer (2015)

21. Basset, N., Kwiatkowska, M., Wiltsche, C.: Compositional strategy synthesis for stochastic games with multiple objectives. Information and Computation **261**, 536–587 (2018)
22. Berwanger, D.: Admissibility in infinite games. In: Thomas, W., Weil, P. (eds.) STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4393, pp. 188–199. Springer (2007)
23. Brenguier, R., Raskin, J.F., Sankur, O.: Assume-admissible synthesis. arXiv preprint arXiv:1507.00623 (2015)
24. Chatterjee, K., Piterman, N.: Combinations of qualitative winning for stochastic parity games. arXiv preprint arXiv:1804.03453 (2018)
25. Chong, C.Y., Kumar, S.P.: Sensor networks: evolution, opportunities, and challenges. Proceedings of the IEEE **91**(8), 1247–1256 (2003)
26. De Ryck, M., Versteyhe, M., Debrouwere, F.: Automated guided vehicle systems, state-of-the-art control algorithms and techniques. Journal of Manufacturing Systems **54**, 152–173 (2020)
27. Develin, M., Payne, S.: Discrete bidding games. The Electronic Journal of Combinatorics **17**(1), R85 (2010)
28. Dias, M.B., Zlot, R., Kalra, N., Stentz, A.: Market-based multirobot coordination: A survey and analysis. Proceedings of the IEEE **94**(7), 1257–1270 (2006)
29. Farber, D.J., Larson, K.C.: The structure of a distributed computing system–software. In: Proceedings of the Symposium on Computer-communications Networks and Teletraffic. pp. 539–545 (1972)
30. Filiot, E., Jin, N., Raskin, J.F.: Compositional algorithms for ltl synthesis. In: International Symposium on Automated Technology for Verification and Analysis. pp. 112–127. Springer (2010)
31. Finkbeiner, B., Passing, N.: Compositional synthesis of modular systems. Innovations in Systems and Software Engineering **18**(3), 455–469 (2022)
32. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05). pp. 321–330. IEEE (2005)
33. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Multi-objective omega-regular reinforcement learning. Formal Aspects of Computing (2023)
34. Houli, D., Zhiheng, L., Yi, Z.: Multiobjective reinforcement learning for traffic signal control using vehicular ad hoc network. EURASIP journal on advances in signal processing **2010**, 1–7 (2010)
35. Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. Formal Methods in System Design **51**(2), 332–361 (2017)
36. Kupermann, O., Varfi, M.: Synthesizing distributed systems. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. pp. 389–398. IEEE (2001)
37. Lazarus, A.J., Loeb, D.E., Propp, J.G., Stromquist, W.R., Ullman, D.H.: Combinatorial games under auction play. Games and Economic Behavior **27**(2), 229–264 (1999)
38. Lazarus, A.J., Loeb, D.E., Propp, J.G., Ullman, D.: Richman games. Games of No Chance **29**, 439–449 (1996)

39. Majumdar, R., Mallik, K., Schmuck, A.K., Zufferey, D.: Assume–guarantee distributed synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(11), 3215–3226 (2020)
40. Meir, R., Kalai, G., Tennenholtz, M.: Bidding games and efficient allocations. Games and Economic Behavior **112**, 166–193 (2018)
41. Ouelhadj, D., Petrovic, S.: A survey of dynamic scheduling in manufacturing systems. Journal of scheduling **12**, 417–431 (2009)
42. Pneuli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science. pp. 746–757. IEEE (1990)
43. Roijers, D.M., Vamplew, P., Whiteson, S., Dazeley, R.: A survey of multi-objective sequential decision-making. Journal of Artificial Intelligence Research **48**, 67–113 (2013)
44. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on computers **29**(12), 1104–1113 (1980)

Most General Winning Secure Equilibria Synthesis in Graph Games[*]

Satya Prakash Nayak(✉) and Anne-Kathrin Schmuck

Max Planck Institute for Software Systems, Kaiserslautern, Germany
{sanayak,akschmuck}@mpi-sws.org

Abstract This paper considers the problem of co-synthesis in k-player games over a finite graph where each player has an individual ω-regular specification ϕ_i. In this context, a secure equilibrium (SE) is a Nash equilibrium w.r.t. the lexicographically ordered objectives of each player to first satisfy their own specification, and second, to falsify other players' specifications. A *winning secure equilibrium* (WSE) is an SE strategy profile $(\pi_i)_{i\in[1;k]}$ that ensures the specification $\phi := \bigwedge_{i\in[1;k]} \phi_i$ if no player deviates from their strategy π_i. Distributed implementations generated from a WSE make components *act rationally* by ensuring that a deviation from the WSE strategy profile is *immediately punished* by a *retaliating strategy* that makes the involved players lose.

In this paper, we move from *deviation punishment* in WSE-based implementations to a *distributed, assume-guarantee based realization* of WSE. This shift is obtained by generalizing WSE from *strategy profiles* to *specification profiles* $(\varphi_i)_{i\in[1;k]}$ with $\bigwedge_{i\in[1;k]} \varphi_i = \phi$, which we call *most general winning secure equilibria (GWSE)*. Such GWSE have the property that each player can individually pick a strategy π_i winning for φ_i (against all other players) and all resulting strategy profiles $(\pi_i)_{i\in[1;k]}$ are guaranteed to be a WSE. The obtained flexibility in players' strategy choices can be utilized for robustness and adaptability of local implementations. Concretely, our contribution is three-fold: (1) we formalize GWSE for k-player games over finite graphs, where each player has an ω-regular specification ϕ_i; (2) we devise an *iterative semi-algorithm* for GWSE synthesis in such games, and (3) obtain an *exponential-time algorithm* for GWSE synthesis with parity specifications ϕ_i.

Keywords: Distributed Synthesis, Parity Games, Secure Equilibria, Assume-Guarantee Contracts

1 Introduction

Games over graphs provide a well known abstraction for many challenging correct-by-construction synthesis problems for software and hardware in embedded cyber-physical applications. In particular, the correct-by-construction co-synthesis of

[*] Authors are supported by the DFG project 389792660 TRR 248-CPEC. Additionally A.-K. Schmuck is supported by the DFG project SCHM 3541/1-1.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 173–193, 2024.
https://doi.org/10.1007/978-3-031-57256-2_9

multiple interacting (reactive) components – each with its own correctness specification – poses, as of today, severe challenges in automated system design.

While many of these challenges arise from the fact that not every component has the same information about all relevant variables in the system, even in the seemingly simple setting of *full information* – where all components see the valuation to all variables – finding the right balance between centralized and local reasoning for co-synthesis is surprisingly challenging. While assuming all players to cooperate might demand too much commitment from individual components, a fully adversarial setting where all other components are assumed to harm a local implementation (independently of their own objective) might not capture a realistic scenario either.

To address this issue, starting with the seminal work of Chatterjee et al. [13], the concept of *rationality* – stemming from classical game theory – was brought to graph games in order to formalize a more realistic model for interaction of multiple components in co-synthesis. The main conceptual contribution of [13] was the introduction of *secure equilibria* (SE) – a special sub-class of Nash equilibria – given as particular strategy profiles. Intuitively, an SE is a Nash equilibrium w.r.t. the lexicographically ordered objectives of each player to first satisfy their own specification, and only second, to falsify other players' specifications. More specifically, it is a strategy profile, i.e., a tuple $(\pi_i)_i$, with π_i being the strategy of Player i, such that no player can improve w.r.t. their lexicographically ordered objective by deviating from this strategy.

As stated by [13, p.68], an SE can thus be interpreted as a contract between the players which enforces cooperation: any unilateral selfish deviation by one player cannot put the other players at a disadvantage if they follow the SE. While this property makes SE very desirable, their main draw-back, as most prominently pointed out by [5], is their restriction to a *single* strategy profile. This, in combination with classical reactive synthesis engines typically preferring small and goal-oriented strategies, incentivizes "immediate punishment" of deviations from an SE strategy profile in the final implementation.

Motivating Example. To illustrate this effect, let us consider the game depicted in Fig. 1, taken from [13]. Here, an SE can be described as follows: if Player 1 always chooses $v_3 \to v_1$ (forming π_1) and Player 2 always chooses $v_0 \to v_2$ and $v_2 \to v_3$ (forming π_2), then they both satisfy their specifications; if Player 1 deviates by choosing $v_3 \to v_2$ (risking falsification of ϕ_2), then Player 2 can retaliate by choosing $v_2 \to v_4$ (ensuring falsification of both ϕ_i); similarly, if Player 2 deviates by choosing $v_0 \to v_3$ (risking falsification of ϕ_1), then Player 1 retaliate by choosing $v_3 \to v_4$ (ensuring falsification of both ϕ_i). Clearly, the strategy profile (π_1, π_2) is an SE. It is, in particular, a *winning* SE as both players sat-

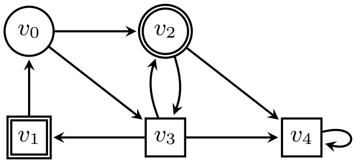


Figure 1: A two-player game with Player 1's vertices (squares), Player 2's vertices (circles) where Player i's specification $\phi_i = \Box\Diamond v_i$ is to visit v_i infinitely often.

isfy their specifications when following it. However, as the outlined retaliating strategies (π'_1, π'_2) are also part of the final implementation generated from this SE, any play that deviates from (π_1, π_2) *only once*, makes the game end up in a loop at v_4 resulting in neither player satisfying their objectives. Intuitively, this way of implementing SE-based strategies makes components *act rationally* by ensuring that a deviation from the contract is *immediately punished.*

Having the interpretation of an SE as a contract in mind, it is however very appealing to think about the *realization* of this contract in the final implementation in a more *permissive* way. Intuitively, in the game depicted in Fig. 1, both players can satisfy their specifications ϕ_i without the help by the other player, as long as the play does not go to v_4. In particular, whenever both players independently chose a strategy π_i which ensures that they (i) never take their edge to v_4 and (ii) satisfy ϕ_i for every strategy π_{-i} of the other player that also never takes their edge to v_4, forms an SE strategy profile (π_1, π_2). These *minimal cooperation obligations* for an SE can be interpreted as a *specification profile* (φ_1, φ_2), s.t. $\varphi_1 := \psi_1 \wedge (\psi_2 \Rightarrow \phi_1)$ and $\varphi_2 := \psi_2 \wedge (\psi_1 \Rightarrow \phi_2)$, where $\psi_1 = \Box\neg(v_3 \wedge \bigcirc v_4)$ and $\psi_2 = \Box\neg(v_2 \wedge \bigcirc v_4)$ express the above discussed assumption that Player i does not move to v_4 from their vertex. It turns out, that this new *specification profile* (φ_1, φ_2) has three nice properties: (i) it is *most general* meaning it does not lose any cooperative solution, i.e., $\phi_1 \wedge \phi_2 = \varphi_1 \wedge \varphi_2$, (ii) it is *realizable*, i.e., Player i has a strategy π_i that satisfies φ_i in a zero-sum sense, (i.e., no matter what the other player does) and, most importantly (iii) it is *secure (winning)*, i.e., every strategy profile (π_1, π_2), where Player i's strategy π_i satisfies φ_i (in a zero-sum sense) is a *winning* SE. While properties (i) and (iii) motivated us to call the set of new specifications a *most general winning secure equilibrium* (GWSE), property (ii) ensures that any specification φ_i from this tuple is locally and fully independently realizable by every component. Conceptually, this allows us to move from *deviation-punishment* in SE-based implementations to a *distributed, assume-guarantee based realization* of SE.

Contribution. By moving from *strategy profiles* (WSE) to *specification profiles* (GWSE) for SE realizations, our approach takes the conceptualisation of rationality for distributed synthesis to an extreme: as we are in the position to *design* every component (as it is a computer system not a human that actually acts rationally) we can enforce that implementations respect the new specifications φ_i. We only use the *concept of rationality* encoded in WSE to automatically obtain *meaningful and implementable* distributed specifications φ_i for this co-design process. Thereby the implementation of an accompanying *punishment mechanism* to enforce rationality of players becomes obsolete. The obtained flexibility in players' strategy choices can be utilized for robustness and adaptability of local implementations, which makes GWSE particularly suited for embedded systems applications.

Concretely, our contribution is three-fold: (1) We formalize GWSE for k-player games over finite graphs, where each player has an ω-regular specification. (2) We devise an *iterative semi-algorithm*[1] for GWSE synthesis under ω-regular

[1] A semi-algorithm is an algorithm that is not guaranteed to halt on all inputs.

specifications. (3) We give a (sound but incomplete) *exponential-time algorithm* for GWSE synthesis under *parity* specifications.

Other Related Work. After the introduction of *secure equilibria* (SE) by Chatterjee et al. [13], there has been several efforts on extending the notion to other classes of games, e.g., games with sup, inf, lim sup, lim inf, and mean-payoff measures [9], multi-player games with probabilistic transitions [17] or quantitative reachability games [8]. Furthermore, a variant of secure equilibria, called *Doomsday equilibria* was studied in [12], where if any coalition of players deviates and violates one players' objective, then the objective of every player is violated. Moreover, the notion of secure equilibria has been applied effectively in the synthesis of mutual-exclusion protocols [15,4] and fair-exchange protocols [21,23].

Motivated by similar insights, other concepts of rationality have also been introduced in multi-player games, e.g. subgame perfect equilibria [29,7,28,10,6] or rational synthesis [20,22,18]. Similar to the implementations of SE by [13], these works restrict implementations to a *single* strategy profile. In contrast, our work introduces a more flexible concept of rationality that is closely related to contract-based distributed synthesis, as in [24,19,16,2]. Here, an assume-guarantee contract is synthesized, such that every strategy realizing the guarantee is ensured to win whenever the other players satisfy the assumption. While this is conceptually similar to our synthesis of GWSE, these works do not consider the players to be adversarial, and hence, there is no notion of *equilibria*.

To the best of our knowledge, the only other work that also combines flexibility with equilibria is *assume-admissible (AA) synthesis* [5]. Their work utilizes a different, incomparable definition of rationality based on a dominance order. Both approaches are incomparable – there exist co-synthesis problems where our approach successfully synthesizes a GWSE and no AA contract exists, and vice versa (see Ex. 1 for details). Conceptually, AA contracts still require *rational* behaviour of players within the contract, while our approach only uses rationality as a *concept* to synthesize meaningful local specifications which can then be implemented in an arbitrary (non-rational) manner. We believe that this is a superior strength of our approach compared to AA synthesis.

2 Preliminaries

Notation. We use $\mathbb{N}$ to denote the set of natural numbers including zero. Given $a, b \in \mathbb{N}$ with $a < b$, we use $[a; b]$ to denote the set $\{n \in \mathbb{N} \mid a \leq n \leq b\}$. For any given set $[a; b]$, we write $i \in_{\text{even}} [a; b]$ and $i \in_{\text{odd}} [a; b]$ as short hand for $i \in [a; b] \cap \{0, 2, 4, \ldots\}$ and $i \in [a; b] \cap \{1, 3, 5, \ldots\}$ respectively. For a finite alphabet Σ, Σ^* and Σ^ω denote the set of finite and infinite words over Σ, respectively.

Linear Temporal Logic (LTL). Given a finite set $\mathtt{AP}$ of atomic propositions, *linear temporal logic (LTL)* formulas over $\mathtt{AP}$ are defined by the grammar:

$$\phi \coloneqq p \in \mathtt{AP} \mid \phi \vee \phi \mid \neg\phi \mid \bigcirc\phi \mid \phi \,\mathcal{U}\, \phi,$$

where $\vee$, $\neg$, $\bigcirc$, and $\mathcal{U}$ denotes the operators *disjunction*, *negation*, *next*, and *until*, respectively. Furthermore, we use the usual derived operators, $\mathtt{True} = p \vee \neg p$,

$\mathsf{False} = \neg\mathsf{True}$, *conjunction* $\phi \wedge \phi' = \neg(\neg\phi \vee \neg\phi')$, *implication* $\phi \Rightarrow \phi' = \neg\phi \vee \phi'$, and other temporal operators such as *finally* $\Diamond\phi = \mathsf{True}\,\mathcal{U}\,\phi$ and *globally* $\Box\phi = \neg\Diamond\neg\phi$. The semantics of LTL formulas are defined as usual (see standard textbooks [3]).

Game Graphs. A *k-player (turn-based) game graph* is a tuple $G = (V, E, v_0)$ where (V, E, v_0) is a finite directed graph with *vertices* V and *edges* E, and $v_0 \in V$ is an initial vertex. For such a game graph, let $\mathsf{P} = [1; k]$ be the set of players such that $V = \bigcup_{i \in \mathsf{P}} V_i$ is partioned into vertices of k players in P. We write E_i, $i \in \mathsf{P}$, to denote the edges from Player i's vertices, i.e., $E_i = E \cap (V_i \times V)$. Further, we write V_{-i} and E_{-i} to denote the set $\bigcup_{j \neq i} V_j$ and $\bigcup_{j \neq i} E_j$, respectively. A *play* from a vertex u_0 is a finite or infinite sequence of vertices $\rho = u_0 u_1 \ldots$ with $(u_j, u_{j+1}) \in E$ for all $j \geq 0$.

Specifications. Given a game graph G, we consider *specifications* specified using a LTL formula Φ over the vertex set V, that is, we consider LTL formulas whose atomic propositions are sets of vertices V. In this case the set of desired infinite plays is given by the semantics of ϕ over G, which is an ω-regular language $\mathcal{L}(G, \phi) \subseteq V^\omega$. We just write $\mathcal{L}(\phi)$ to denote this language when the game graph G is clear in the context. Every game graph with an arbitrary ω-regular set of desired infinite plays can be reduced to a game graph (possibly with an extended set of vertices) with an LTL objective, as above. The standard definitions of ω-regular languages are omitted for brevity and can be found in standard textbooks [3]. To simplify notation we use $e = (u, v)$ in LTL formulas as syntactic sugar for $u \wedge \bigcirc v$.

Games and Strategies. A *k-player game* is a pair $\mathcal{G} = (G, (\phi_i)_{i \in \mathsf{P}})$ where G is a k-player game graph and each ϕ_i is an *objective* for Player i over G. A strategy of Player i, $i \in \mathsf{P}$, is a function $\pi_i\colon V^* V_i \to V$ such that for every $\rho v \in V^* V_i$, it holds that $(v, \pi_i(\rho v)) \in E$. A strategy profile for a set of players $\mathsf{P}' \subseteq \mathsf{P}$ is a tuple $\Pi = (\pi_i)_{i \in \mathsf{P}'}$ of strategies, one for each player in P'. To simplify notation, we write P_{-i} and π_{-i} to denote the set $\mathsf{P} \setminus \{i\}$ and their strategy profile $(\pi_j)_{j \in \mathsf{P} \setminus \{i\}}$, respectively. Given a strategy profile $(\pi_i)_{i \in \mathsf{P}'}$, we say that a play $\rho = u_0 u_1 \ldots$ is $(\pi_i)_{i \in \mathsf{P}'}$*-play* if for every $i \in \mathsf{P}'$ and for all $\ell \geq 1$, it holds that $u_{\ell-1} \in V_i$ implies $u_\ell = \pi_i(u_0 \ldots u_{\ell-1})$.

Satisfying Specifications. Given a game graph G and a specification ϕ, a play ρ *satisfies* ϕ if $\rho \in \mathcal{L}(\phi)$. A strategy profile $(\pi_i)_{i \in \mathsf{P}'}$ satisfies/winning w.r.t. a specification ϕ, from a vertex v, denoted by $(\pi_i)_{i \in \mathsf{P}'} \vDash_v \phi$, if every $(\pi_i)_{i \in \mathsf{P}'}$-play from v satifies ϕ. We just write $(\pi_i)_{i \in \mathsf{P}'} \vDash \phi$ if v is the initial vertex. We collect all vertices from which there exists a strategy profile for players in P' that satisfies ϕ in the winning region[2] $\langle\!\langle \mathsf{P}' \rangle\!\rangle(G, \phi)$. We just write $\langle\!\langle \mathsf{P}' \rangle\!\rangle\phi$ to denote this set if game graph G is clear in the context. Furthermore, we write ϕ_{-i} to denote $\bigwedge_{j \in \mathsf{P}_{-i}} \phi_j$.

Parity Specifications. Give a game graph $G = (V, E, v_0)$, a specification ϕ is called *parity* if $\phi = \mathit{Parity}(\Omega) \coloneqq \bigwedge_{i \in_{\text{odd}}[0;d]} \left(\Box\Diamond\Omega_i \Rightarrow \bigvee_{j \in_{\text{even}}[i+1;d]} \Box\Diamond\Omega_j\right)$, with $\Omega_i = \{v \in V \mid \Omega(v) = i\}$ for some priority function $\Omega : V \to [0; d]$ that assigns each vertex a priority. A play satisfies such a specification if the maximum of priorities seen infinitely often is even.

[2] Slightly abusing notation, we write $\langle\!\langle i \rangle\!\rangle\phi$ for singleton sets of players $\mathsf{P}' = \{i\}$.

3 Most General Winning Secure Equilibria

This section formalizes most general winning secure equilibria (GWSE). In order to do so, we first recall the notion of secure equilibria from [13].

Secure Equilibria. Given a *k-player game* $\mathcal{G} = (G, (\phi_i)_{i\in\mathsf{P}})$ and a strategy profile $\Pi := (\pi_i)_{i\in\mathsf{P}}$ one can define a payoff profile, denoted by $\mathsf{payoff}(\Pi)$, as the tuple $(p_i)_{i\in\mathsf{P}}$ s.t. $p_i = 1$ iff $\Pi \vDash \phi_i$. With this, we can define a Player j preference order $\prec_j$ on payoff profiles lexicographiacally, s.t.

$$(p_i)_{i\in\mathsf{P}} \prec_j (p'_i)_{i\in\mathsf{P}} \text{ iff } (p_j < p'_j) \vee \big((p_j = p'_j) \wedge (\forall i \neq j.p_i \geq p'_i) \wedge (\exists i \neq j.p_i > p'_i)\big).$$

Intuitively, this preference order captures the fact that every player's main objective is to satisfy their own specification ϕ_i, and, as a secondary objective, falsify the specifications of the other players.

Definition 1. *Given a* k-player game $\mathcal{G} = (G, (\phi_i)_{i\in\mathsf{P}})$*, a strategy profile* $\Pi := (\pi_i)_{i\in\mathsf{P}}$ *is a* secure equilibrium *(SE) if for all* $i \in \mathsf{P}$*, there does not exist a strategy* π'_i *of Player* i *such that* $\mathsf{payoff}(\Pi) \prec_i \mathsf{payoff}(\pi'_i, \pi_{-i})$*.*

It is well known that every secure equilibrium is also a nash equilibrium in the classical sense. Within this paper, we only consider *winning secure equilibria* (WSE) i.e., SE with the payoff profile $(p_i = 1)_{i\in\mathsf{P}}$. As WSE have a trivial payoff profile, they can be characterized without referring to payoffs as formalized next.

Definition 2. *Give a k-player game* $(G, (\phi_i)_{i\in\mathsf{P}})$*, a* winning secure equilibrium (WSE) *is a strategy profile* $(\pi_i)_{i\in\mathsf{P}}$ *such that (i)* $(\pi_i)_{i\in\mathsf{P}} \vDash \bigwedge_{i\in\mathsf{P}} \phi_i$*; and (ii) for every strategy* π'_i *of Player* i*, if* $(\pi'_i, \pi_{-i}) \nvDash \phi_{-i}$ *holds, then* $(\pi'_i, \pi_{-i}) \nvDash \phi_i$ *holds.*

Intuitively, item i ensures that the strategy profile satisfies all player's objective, whereas item ii ensures that no player can improve, i.e., falsify another player's objective without falsifying their own objective, by deviating from the prescribed strategy.

Most General Winning Secure Equilibria. As illustrated by the motivating example in Sec. 1, we aim at generalizing WSE from single *strategy profiles* to *specification profiles* that capture an infinite number of WSE. These specification profiles $(\varphi_i)_{i\in\mathsf{P}}$, which we call *most general winning secure equilibria* (GWSE), allow each player to locally (and fully independently) pick a strategy π_i that is winning for φ_i (in a zero-sum sense). It is then guaranteed that any resulting strategy profile $(\pi_i)_{i\in\mathsf{P}}$ is indeed a WSE. This is formalized next.

Definition 3. *Give a k-player game* $(G, (\phi_i)_{i\in\mathsf{P}})$*, a tuple* $(\varphi_i)_{i\in\mathsf{P}}$ *of specifications is said to be a* most general winning secure equilibrium (GWSE) *if it is*

(i) (most) general: $\mathcal{L}(\bigwedge_{i\in\mathsf{P}} \varphi_i) = \mathcal{L}(\bigwedge_{i\in\mathsf{P}} \phi_i)$*;*
(ii) realizable: $v_0 \in \langle\!\langle i \rangle\!\rangle \varphi_i$ *for all* $i \in \mathsf{P}$*; and*
(iii) secure (winning): every strategy profile $(\pi_i)_{i\in\mathsf{P}}$ *with* $\pi_i \vDash \varphi_i$ *is a WSE.*

Intuitively, generality ensures that the transformation of the specifications $(\phi_i)_{i\in\mathsf{P}}$ into new specifications $(\varphi_i)_{i\in\mathsf{P}}$ does not lose any winning play. Further, realizability ensures that every single player can enforce φ_i (without the help of other players) from the initial vertex. Finally, security ensures that any locally chosen strategy π_i winning for φ_i fors a strategy profile which is indeed a WSE.

4 Computing GWSE in ω-regular Games

This section proposes an *iterative semi-algorithm*[3] to compute GWSE in this paper which utilizes the concept of *adequately permissive assumptions* (APA) introduced by Anand et al. [1]. Given a k-player game $(G, (\phi_i)_{i \in \mathtt{P}})$, an APA is a specification ψ_i that collects all Player i strategies which allow for a cooperative solution if other players cooperate. It therefore overapproximates the set of all Player i strategies which could possibly form a WSE with the other players. As a consequence, the intersection $\bigwedge_{i \in \mathtt{P}} \psi_i$ is an overapproximation of a GWSE. In order to refine this approximation, the next computation round can now use the APA's of other players when computing new local APA's. In order to properly formalize this idea, we first recall the concept of APA's from [1].

4.1 Adequately Permissive Assumptions

Following [1], we define an *adequately permissive assumption* (APA) as follows.

Definition 4. *Given a k-player game graph $G = (V, E, v_0)$ and a specification ϕ, we say that a specification ψ_i is an* adequately permissive assumption (APA) *on Player i for ϕ if it is:*

(i) sufficient*: there exists a strategy profile π_{-i} such that for every Player i strategy π_i with $\pi_i \vDash \psi_i$, we have $(\pi_i, \pi_{-i}) \vDash \phi$;*
(ii) implementable*: $\langle\!\langle i \rangle\!\rangle \psi_i = V$; and*
(iii) permissive*: $\mathcal{L}(\psi_i) \supseteq \mathcal{L}(\phi)$.*

The intuition behind an APA is that even if a player can not realize a specification ϕ, they should at least satisfy an APA on them as it will allow them to realize ϕ if the other players are willing to help (sufficiency). Further, such a behavior by Player i does not prevent any WSE (permissiveness), and Player i can individually choose to follow an APA (implementability).

Remark 1. While Def. 4 is an almost direct adaptation from [1, Def. 2-5] to k-player games, it has a couple of noteable differences. First, Anand et al. define APA's for 2-player games and, conceptually, use APA's to constraint the opponents moves. While we can simply view the k-player game as a 2-player game between the protagonist Player i and (the collection of) its opponents $\mathtt{P}_{-i}$, we will use the computed assumption ψ_i to constrain the *protagonist's* moves (not the opponent) in Def. 4. Second, the sufficiency condition for an APA in [1, Def. 2] does not depend on an initial vertex. An APA always exists in their setting (possibly being True when $\langle\!\langle \mathtt{P} \rangle\!\rangle \phi = \varnothing$). In contrast, the k-player games in this paper have a designated initial vertex, hence, an APA only exists iff $v_0 \in \langle\!\langle \mathtt{P} \rangle\!\rangle \phi$.

With this insight, we can use the algorithm from [1] to compute APA's for parity specificatios $\phi = \mathit{Parity}(\Omega)$ in polynomial time.

[3] A semi-algorithm is an algorithm that is not guaranteed to halt on all inputs.

Lemma 1 ([1, Thm. 4]). *Given a k-player game graph $G = (V, E, v_0)$ and a parity specification $\phi = Parity(\Omega)$, an APA on Player i for ϕ can be computed, if one exists, in time $\mathcal{O}(|V|^4)$.*

Let us write ComputeAPA(G, ϕ, i) to denote the procedure that returns this APA if it exists; otherwise, it returns False.

Remark 2. We note that *Lem.* 1 also gives a method to compute APA's for games with LTL- or ω-regular specifications as such games can be converted into parity games (possibly with an extended game graph) by standard methods [3]. Therefore, with a slight abuse of notation, we will also call the algorithm ComputeAPA(G, ϕ, i) if ϕ is not a parity specification, which the understanding, that the game is always converted into a parity game first. This might incur an exponential blowup of the state space. As we call ComputeAPA repeatedly to compute GWSE's, this blowup might cause non-termination (see Sec. 4.6 for details). In order to obtain a (non-optimal but) terminating algorithm for GWSE computation, we will mitigate this blowup later in Sec. 5.

4.2 Iterative Computation of APA's

Given the results of the previous section, we can use the algorithm ComputeAPA on a given game $(G, (\phi_i)_{i \in \mathsf{P}})$ to compute APA's for each player, i.e., $\psi_i := $ ComputeAPA(G, ϕ_i, i). Intuitively, ψ_i overapproximates the set of all Player i strategies which could possibly form a WSE with the other players. As a consequence, the intersection $\bigwedge_{i \in \mathsf{P}} \psi_i$ is an overapproximation of the GWSE.

As outlined previously, we will iteratively refine these computed APA's to finally compute the GWSE. In order to do so, we want to condition the computation of the next-round APA ψ_i' on the previous-round APA's of all other players ψ_{-i}, as any secure strategy of players in P_{-i} is incentivized to comply with ψ_{-i}. The most intuitive method to do this is to simply consider $\psi_{-i} \Rightarrow \phi_i$ as the specification for APA computation in the next round. However, the way sufficiency is formulated for APA's prevents this approach, as the implication $\psi_{-i} \Rightarrow \phi_i$ is true if ψ_{-i} is false. As there obviously exists a strategy profile π_{-i} which violates ψ_{-i}, the sufficiency condition becomes meaningless for this specification.

However, as we know that ψ_{-i} are APA's, their implementability constraint (Def. 4.ii) ensures that Player i can neither enforce nor falsify them. Therefore, a new specification $\phi_i' := \psi_{-i} \wedge \phi_i$ still puts all the burden of satisfying ψ_{-i} to players in P_{-i} and hence, implicitly constrains the choices of P_{-i} to strategies complying with ψ_{-i} for sufficiency of the new APA. However, using $\phi_i' := \psi_{-i} \wedge \phi_i$ indeed weakens the permissiveness requirement $\mathcal{L}(\psi_i') \supseteq \mathcal{L}(\phi \wedge \psi_{-i})$, i.e., the new APA ψ_i' needs to be more general than the specification ϕ, only when the assumption ψ_{-i} holds. With these refined conditions for sufficiency and permissiveness, it becomes evident that an APA for specification ϕ under assumption ψ_{-i} is equivalent to an APA for the modified specification $\psi_{-i} \wedge \phi$, as formalized below.

Definition 5. *Given a k-player game, a specification ϕ_i and an assumption ψ_{-i}, we say that the specification ψ_i is an* APA on Player i for ϕ_i under ψ_{-i} *if it is an APA on Player i for specification $\psi_{-i} \wedge \phi$.*

Following Rem. 2, we denote by $\textsc{Compute}\text{APA}(G, \psi_{-i} \wedge \phi, i)$ the algorithm which computes *APA's* on Player i for ϕ *under assumptions* ψ_{-i}, even though $\psi_{-i} \wedge \phi$ is typically not a parity specification over G anymore.

4.3 Computing GWSE

Using all the intuition discussed before, we now give a semi-algorithm in Algo. 1 to compute GWSE for k-player games with ω-regular specifications for all players. The main idea is to iteratively compute assumptions $(\psi_i)_{i \in \mathtt{P}}$ on every player and check if they are stable enough so that every player can satisfy their actual specification ϕ_i under the assumption ψ_{-i}. If not, then, in the next iteration, we compute new assumptions $(\psi'_i)_{i \in \mathtt{P}}$ that are stricter than earlier ones, i.e., $\mathcal{L}(\psi'_i) \subseteq \mathcal{L}(\psi_i)$ but still more general than their specifications under the earlier assumption, i.e., $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\psi_{-i} \wedge \phi_i)$.

Algorithm 1 $\textsc{Compute}\text{GE}(\mathcal{G})$

```
Require: A k-player game $\mathcal{G}$ with game graph $G = (V, E, v_0)$ and parity specifications
    $(\phi_i)_{i \in \mathtt{P}}$.
Ensure: Either a GWSE $(\varphi_i)_{i \in \mathtt{P}}$ or False.
 1: $\psi_i \leftarrow$ True $\forall i \in \mathtt{P}$
 2: return $\textsc{Recursive}\text{GE}(\mathcal{G}, (\psi_i)_{i \in \mathtt{P}})$

 3: procedure $\textsc{Recursive}\text{GE}(\mathcal{G}, (\psi_i)_{i \in \mathtt{P}})$
 4:     $\varphi_i \leftarrow \psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)$ $\forall i \in \mathtt{P}$
 5:     if $v_0 \in \bigcap_{i \in \mathtt{P}} \langle\langle i \rangle\rangle \varphi_i$ then
 6:         return $(\varphi_i)_{i \in \mathtt{P}}$
 7:     $\psi'_i \leftarrow \psi_i \wedge \textsc{Compute}\text{APA}(G, \psi_{-i} \wedge \phi_i, i)$ $\forall i \in \mathtt{P}$
 8:     if $\psi'_i = \psi_i$ for all $i \in \mathtt{P}$ then
 9:         return False
10:     return $\textsc{Recursive}\text{GE}(\mathcal{G}, (\psi'_i)_{i \in \mathtt{P}})$
```

More specifically, we start with ψ_i = True for each $i \in \mathtt{P}$ in the first iteration (line 1), and then in every iteration, we want each player to satisfy $\varphi_i = \psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)$ (computed in line 4) *by themselves*, i.e., always satisfy their assumption ψ_i and satisfy specification ϕ_i whenever others satisfy their assumptions ψ_{-i}. Note that, in this part of the algorithm it is correct to use this implication-style specification, as it is used for solving a *zero-sum* 2*-player game* between Player i and its opponent (i.e., the collection of all other players in $\mathtt{P}_{-i}$) for the specification φ_i. The winning regions $\langle\langle i \rangle\rangle \varphi_i$ for each such zero-sum 2-player game are then intersected in line 5 to obtain the winning region that is achievable by any strategy profile $(\pi_i)_{i \in \mathtt{P}}$ where π_i is a winning strategy of Player i w.r.t. φ_i (in a zero-sum sense). If this resulting winning region contains the initial vertex, we return the specification $(\varphi_i)_{i \in \mathtt{P}}$ (line 6), which is proven to indeed be a GWSE in Thm. 1.

If this is not the case, we keep on strengthening APA's, as discussed in Sec. 4.2, to make the above mentioned zero-sum 2-player games easier to solve (as they can rely on tighter assumptions now). Hence, we call ComputeAPA with the modified specifications $\phi'_i := \psi_{-i} \wedge \phi_i$ for all players (line 7). If this assumption refinement step was unsuccessful, i.e., assumptions have not changed (line 8), we give up and return False. Otherwise, we recheck the termination condition for the newly computed APA's.

Example 1. Before proving the correctness of the (semi) Algo. 1, let us first illustrate the steps using an example depicted in Fig. 2. In line 1, we begin with $\psi_1 = \psi_2 = \mathsf{True}$ and run the recursive procedure RecursiveGE in line 2.

Within the first iteration of RecursiveGE, in line 4, we set $\varphi_i = \phi_i$ as $\psi_i = \mathsf{True}$ for all $i \in [1;2]$. Then, in line 5, we check whether each player can satisfy $\varphi_i = \phi_i$ without cooperation (i.e., in a zero-sum sense), from the initial vertex v_0. As no player can ensure that, we move to line 7. Here, as $\psi_i = \mathsf{True}$ for $i \in [1;2]$, the new assumptions ψ'_i is an APA computed by ComputeAPA(G, ϕ_i, i). This gives us $\psi'_1 = \square\neg(e_{12} \wedge e_{34}) \wedge \lozenge \square \neg e_{10}$ and $\psi'_2 = \lozenge \square \neg e_{00}$, where $e_{ij} = v_i \wedge \bigcirc v_j$. Intuitively, ψ'_1 ensures that edges, i.e., $v_1 \rightarrow v_2$ and $v_3 \rightarrow v_4$, leading to the region from which it is not possible to satisfy ϕ_1 are never taken; and the edge, i.e., $v_1 \rightarrow v_0$, restricting the play to progress towards target vertex v_5 (as in ϕ_1) is eventually not taken. Similarly, ψ_2 ensures that the edge $v_0 \rightarrow v_0$ is eventually not taken that ensures progress towards ϕ_2's target vertices $\{v_4, v_5\}$. As $\psi'_i \neq \psi_i$ for all $i \in [1;2]$ in line 8, we go to the next iteration of RecursiveGE.

In the second iteration, we again compute the new potential GWSE (φ_1, φ_2) with $\varphi_i = \psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)$ in line 4. In line 5, we find that $v_0 \notin \langle\langle 1 \rangle\rangle \varphi_1$. That is because Player 1 cannot ensure satisfying ϕ_1 even when Player 2 satisfies ψ_2 as Player 2 can always use edge $v_0 \rightarrow v_3$ leading to the play $(v_0 v_3)^\omega \not\models \phi_2$. Hence, in line 7, the APA under ψ_1 gives a more restricted assumptions on Player 2: $\psi'_2 = \lozenge \square \neg(e_{00} \wedge e_{03})$. As the assumption ψ_2 on Player 2 was very weak, the APA for Player 1 under ψ_2 results in the same assumption as ψ_1, and hence, $\psi'_1 = \psi_1$. Then, we move to the third iteration.

In this iteration, we find that both players can indeed satisfy their new specification φ_i from the initial vertex in line 5. Hence, we finally return a GWSE (φ_1, φ_2) with $\varphi_i = \psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)$ where $\psi_2 = \lozenge \square \neg(e_{00} \wedge e_{03})$ and $\psi_1 = \square\neg(e_{12} \wedge e_{34}) \wedge \lozenge \square \neg(e_{10})$.

Remark 3. Let us remark that for the game depicted in Fig. 2, assume-admissible (AA) synthesis [5] has no solution. AA-synthesis utilizes a different, incomparable definition of rationality based on a dominance order. In their framework, a

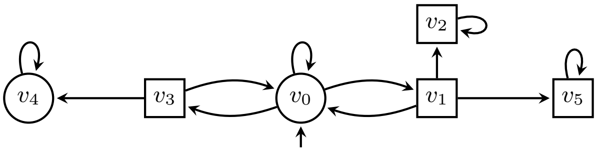


Figure 2: A two-player game with initial vertex v_0, Player 1's vertices (squares), Player 2's vertices (circles) and specifications $\phi_1 = \lozenge \square \{v_5\}$ and $\phi_2 = \lozenge \square \{v_4, v_5\}$.

Player i strategy π_i is said to be *dominated* by π'_i if the set of strategy profiles that π' is winning against (i.e., satisfies Player i's specification) is strictly larger than that of π. A strategy not dominated by any other strategy is called *admissible.* In AA-synthesis, one needs to find an admissible strategy π_i for Player i such that for every admissible strategy π'_{-i} for the other player, $(\pi_i, \pi'_{-i}) \models \phi_i$. In this example, Player 1 has only one admissible strategy π_1 that always uses $v_1 \rightarrow v_5$ and $v_3 \rightarrow v_0$. However, with the admissible strategy π'_2 of Player 2 that always uses $v_0 \rightarrow v_3$, we have $(\pi_1, \pi'_2) \not\models \phi_1$.

The next theorem shows that Algo. 1 is indeed sound.

Theorem 1. *Let $\mathcal{G}$ be a k-player game with game graph $G = (V, E, v_0)$ and parity specifications $(\phi_i)_{i \in \mathsf{P}}$ such that $(\varphi_i^*)_{i \in \mathsf{P}} =$ ComputeGE$(\mathcal{G})$, then $(\varphi_i^*)_{i \in \mathsf{P}}$ is a GWSE for $\mathcal{G}$.*

Proof. First, observe that ComputeGE did not return False by the premise of the theorem. So, if ComputeAPA returned False in line 7, i.e., $\psi'_i = \mathsf{False}$ for some $i \in \mathsf{P}$, in some n-th iteration, then in the $n+1$-th iteration, we have $\psi_i = \mathsf{False}$ and $\psi_{-j} = \mathsf{False}$ for all $j \in \mathsf{P}_{-i}$. So, it holds that $v_0 \notin \langle\!\langle i \rangle\!\rangle \varphi_i = \langle\!\langle i \rangle\!\rangle \mathsf{False} = \varnothing$ and hence, it does not return in line 6. Furthermore, as $\psi_{-j} \wedge \phi_j = \mathsf{False}$ for all $j \in \mathsf{P}_{-i}$, by sufficiency, ComputeAPA returns False for all $j \in \mathsf{P}_{-i}$. Hence, $\psi'_j = \mathsf{False}$ for all $j \in \mathsf{P}$. This would imply (by similar arguments), in $(n+2)$-th iteration, $\psi'_j = \psi_j = \mathsf{False}$ for all $j \in \mathsf{P}$ and hence, the algorithm would return False. Therefore, we can assume ComputeAPA never returned False in any iteration.

Now, let us claim that in every iteration of RecursiveGE, for all $i \in \mathsf{P}$:

(claim 1) $\mathcal{L}(\psi_i) \supseteq \mathcal{L}(\bigwedge_{j \in \mathsf{P}} \phi_j)$, and (claim 2) $\mathcal{L}(\psi_i) \supseteq \mathcal{L}(\psi_{-i} \wedge \phi_i)$.

We will prove the claim using induction on the number of itereative calls to RecursiveGE. For the base case, observe $\psi_i = \mathsf{True}$ for all $i \in \mathsf{P}$, hence, the claim holds trivially. For the induction step, assume that claim 1+2 hold in the n-th iteration. Then, for all $i \in \mathsf{P}$, as ψ'_i (computed in line 7) is ψ in the next iteration, it suffices to show that $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\bigwedge_{j \in \mathsf{P}} \phi_j)$ and $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\psi'_{-i} \wedge \phi_i)$.

By permissiveness of APA (as in Def. 4), for all $i \in \mathsf{P}$, we have $\mathcal{L}(\text{ComputeAPA}(G, \psi_{-i} \wedge \phi_i, i)) \supseteq \mathcal{L}(\psi_{-i}) \cap \mathcal{L}(\phi_i)$. Hence, by line 7, for all $i \in \mathsf{P}$, we have $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\psi_i) \cap \mathcal{L}(\psi_{-i}) \cap \mathcal{L}(\phi_i) = \left(\bigcap_{j \in \mathsf{P}} \mathcal{L}(\psi_j)\right) \cap \mathcal{L}(\phi_i)$, and hence, by claim 1, $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\bigwedge_{j \in \mathsf{P}} \phi_j)$.

Similarly, for all $i \in \mathsf{P}$, as $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\psi_i) \cap \mathcal{L}(\psi_{-i}) \cap \mathcal{L}(\phi_i)$, by claim 2, we also have $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\psi_{-i}) \cap \mathcal{L}(\phi_i)$. Furthermore, by line 7, for all $j \in \mathsf{P}$, we have $\mathcal{L}(\psi_j) \supseteq \mathcal{L}(\psi'_j)$, and hence, $\mathcal{L}(\psi_{-i}) = \bigcap_{j \neq i} \mathcal{L}(\psi_j) \supseteq \bigcap_{j \neq i} \mathcal{L}(\psi'_j) = \mathcal{L}(\psi'_{-i})$. Therefore, for all $i \in \mathsf{P}$, we have $\mathcal{L}(\psi'_i) \supseteq \mathcal{L}(\psi'_{-i}) \cap \mathcal{L}(\phi_i) = \mathcal{L}(\psi'_{-i} \wedge \phi_i)$.

Now, we show that Def. 3 (i)-(ii) indeed holds for the tuple $(\varphi_i^*)_{i \in \mathsf{P}}$.
(i) (general) By construction, $\varphi_i^* = \psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)$ for the specifications $(\psi_i)_{i \in \mathsf{P}}$ computed in last iteration. Hence, it holds that

$$\mathcal{L}\left(\bigwedge_{i \in \mathsf{P}} \varphi_i^*\right) = \bigcap_{i \in \mathsf{P}} \mathcal{L}(\psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)) = \bigcap_{i \in \mathsf{P}} \mathcal{L}(\psi_i) \cap \bigcap_{i \in \mathsf{P}} \mathcal{L}(\psi_{-i} \Rightarrow \phi_i)$$

$$= \bigcap_{i \in \mathsf{P}} \mathcal{L}(\psi_{-i}) \cap \bigcap_{i \in \mathsf{P}} \mathcal{L}(\psi_{-i} \Rightarrow \phi_i) = \bigcap_{i \in \mathsf{P}} \mathcal{L}(\psi_{-i} \wedge (\psi_{-i} \Rightarrow \phi_i)) \subseteq \bigcap_{i \in \mathsf{P}} \mathcal{L}(\phi_i) = \mathcal{L}\left(\bigwedge_{i \in \mathsf{P}} \phi_i\right).$$

For the other direction, it holds that

$$\mathcal{L}(\varphi_i^*) = \mathcal{L}(\psi_i) \wedge \mathcal{L}(\psi_{-i} \Rightarrow \phi_i) \supseteq \mathcal{L}(\psi_i) \cap \mathcal{L}(\phi_i) \tag{1}$$

Then, by claim 1, for all $i \in \mathtt{P}$, we have $\mathcal{L}(\varphi_i^*) \supseteq \mathcal{L}\left(\bigwedge_{i\in\mathtt{P}} \phi_i\right)$, and hence, $\mathcal{L}\left(\bigwedge_{i\in\mathtt{P}} \varphi_i^*\right) \supseteq \mathcal{L}\left(\bigwedge_{i\in\mathtt{P}} \phi_i\right)$. Therefore, $(\varphi_i^*)_{i\in\mathtt{P}}$ is general.
(ii) (realizable) Holds trivially by line 5.
(iii) (secure) Let $(\pi_i)_{i\in\mathtt{P}}$ be a strategy profile with $\pi_i \vDash \varphi_i^*$. Then, every $(\pi_i)_{i\in\mathtt{P}}$-play from v_0 satisfies φ_i^* for all $i \in \mathtt{P}$, and hence, $(\pi_i)_{i\in\mathtt{P}} \vDash \bigwedge_{i\in\mathtt{P}} \varphi_i^*$. So, by generality, we have $(\pi_i)_{i\in\mathtt{P}} \vDash \bigwedge_{i\in\mathtt{P}} \phi_i$.

Now, to prove item ii of Def. 2, let π_i' be a strategy of Player i, and let ρ be the (π_i', π_{-i})-play from v_0. As before, for all $j \in \mathtt{P}$, we have $\varphi_j^* = \psi_j \wedge (\psi_{-j} \Rightarrow \phi_j)$. So, for every $j \neq i$, $\rho \in \mathcal{L}(\varphi_j^*) \subseteq \mathcal{L}(\psi_j)$. Hence, we have $\rho \in \bigcap_{j\neq i} \mathcal{L}(\psi_j) = \mathcal{L}(\psi_{-i})$.

Now, if $\rho \in \mathcal{L}(\phi_i)$, then $\rho \in \mathcal{L}(\psi_{-i} \wedge \phi_i)$. Then, by (1) and claim 2, we have $\rho \in \mathcal{L}(\varphi_i^*)$. Furthermore, as $\pi_{-i} \vDash \varphi_{-i}^*$, we have $\rho \in \mathcal{L}(\varphi_{-i}^*)$. Therefore, $\rho \in \mathcal{L}(\varphi_i^* \wedge \varphi_{-i}^*)$, and by generality, $\rho \in \mathcal{L}(\phi_i \wedge \phi_{-i}) \subseteq \mathcal{L}(\phi_{-i})$. Then, by contraposition, item ii of Def. 2 holds for $(\pi_i)_{i\in\mathtt{P}}$. Hence, $(\pi_i)_{i\in\mathtt{P}}$ is an SE, and hence, $(\varphi_i^*)_{i\in\mathtt{P}}$ is secure. □

4.4 Games with an Environment Player

Up to this point, we have only considered games played between k players, each representing a distinct system. However, in the context of reactive synthesis problems, a different setup is often encountered. Here, the system players play against an environment player, who is considered as being adversarial toward all the system players. Consequently, the system players must fulfill their objectives against all possible strategies employed by the environment player.

Interestingly, this framework can be seen as equivalent to a $(k+1)$-player game with the original k system players and a $(k+1)$-th player, representing the environment. For this new player, the objective is simply $\phi_{k+1} = \mathsf{True}$. Then, it is easy to see that an APA for such specification ϕ_{k+1} under any assumption is True. Hence, in each iteration of RecursiveGE in Algo. 1, the associated assumption ψ_{k+1} is also True, and thus, $\varphi_{k+1} = \mathsf{True} \wedge ((\bigwedge_{i\in[1;k]} \psi_i) \Rightarrow \mathsf{True}) \equiv \mathsf{True}$. Consequently, if ComputeGE yields a GWSE $(\varphi_i^*)_{i\in[1;k+1]}$, the new objective of the environment player, $\varphi_{k+1}^* = \mathsf{True}$, doesn't impose any constraints on the environment's actions. Therefore, the tuple $(\varphi_i^*)_{i\in[1;k]}$ remains secure (as in Def. 3) for the k system players because the environment player can never violate its new specification φ_{k+1}. In sum, games featuring an environment player can be effectively handled as a special case, as formally summarized below:

Corollary 1. *Let $G = (V, E)$ be a game graph with k system players, i.e., $\mathtt{P} = [1;k]$, and an environment player* env *such that $V = (\bigcup_{i\in\mathtt{P}} V_i) \uplus V_{\mathtt{env}}$. Let $(\phi_i)_{i\in\mathtt{P}}$ be the tuple of specifications, one for each system player. Then, a tuple $(\varphi_i)_{i\in\mathtt{P}}$ is a GWSE for $(G, (\phi_i)_{i\in\mathtt{P}})$ if and only if $(\varphi_i)_{i\in[1;k+1]}$ with $\varphi_{k+1} = \mathsf{True}$ is a GWSE for the $k+1$-player game $(G, (\phi_i)_{i\in[1;k+1]})$ with $\phi_{k+1} = \mathsf{True}$.*

Furthermore, in synthesis problems, the choices of the environment are sometimes restricted based on a certain assumption $\phi_{\mathtt{env}}$. In such scenarios, a viable approach involves updating each system player's specification ϕ_i to $\phi_{\texttt{env}} \Rightarrow \phi_i$ and subsequently utilizing Cor. 1 to compute a GWSE. An alternative approach is to consider a $(k+1)$-player game with specification $\phi_{k+1} = \phi_{\texttt{env}}$ for the $(k+1)$-th player. With this approach, the solution becomes more meaningful, as any strategy profile for the system players satisfying the resulting GWSE allows the environment to satisfy its own assumptions $\phi_{\texttt{env}}$. This approach nicely complements existing works [14,25] that aim to synthesize strategies for systems while allowing the environment to fulfill its own requirement.

4.5 Partially Winning GWSE

In the preceding sections, we have presented a method for computing *winning* SE, i.e., equilibria where all players satisfy their objectives. However, it's worth noting that in certain scenarios, WSE might not exist (see e.g. [13] for a detailed discussion). In such cases, a subset $\mathtt{P}'$ of players can still form a coalition, which serves their interests by enabling them to compute a GWSE for their coalition only, while treating the remaining players in $\mathtt{P} \setminus \mathtt{P}'$ as part of the environment. This can be accomplished by computing a GWSE with updated specifications denoted as $(\phi'_i)_{i \in \mathtt{P}}$, wherein $\phi'_i = \phi_i$ for all $i \in \mathtt{P}'$ and $\phi'_i = \mathsf{True}$ for all $i \notin \mathtt{P}'$. This scenario aligns with the concept of considering an environment from Sec. 4.4.

It is important to emphasize that for instances where no WSE exists, there might not even exist a *unique maximal* outcome for which an SE is feasible, see [13, Sec. 5] for a simple example. As a result, there may be multiple coalitions that can offer different advantages to individual players from the initial vertex. This scenario presents an intriguing, unexplored challenge for future research.

4.6 Computational Tractability and Termination

While Algo. 1 has multiple desirable properties, additionally supported by the possible extensions discussed in Sec. 4.4 and 4.5, its computational tractability and termination is questionable for the full class of ω-regular games.

As pointed out in Rem. 2, the application of ComputeAPA might require changing the game graph for if the input is not a parity specification. While the *language* of the computed APA is guarantee to shrink in every iteration (see the proof of Thm. 1), this does not guarantee termination of Algo. 1 as such a language still contains an infinite number of words. Due to the possibly repeated changes in the game graph for APA computation, the finiteness of the underlying model can also not be used as a termation argument.

In addition, the need to change game graphs induces a severe computational burden. While this might be not so obvious for the polynomial time algorithm ComputeAPA, this is actually also the case for the (zero-sum) game solver that needs to be invoked line 5 of Algo. 1. As the specification for these games also keeps changing in each iteration, a new parity game needs to be constructed in each iteration, which might be increasingly harder to solve, depending on the nature of the added assumptions. We will see in Sec. 5 how these problems can be resolved by a suitable restriction of the considered assumption class.

5 Optimized Computation of GWSE in Parity Games

As discussed in Sec. 4.6, the potential need to repeatedly change game graphs in the computations of lines 5 and 7 in Algo. 1 might incur increasing computational costs and prevents a termination guarantee. To circumvent these problems, this section proposes a different algorithm for GWSE synthesis which overapproximates APA's by a simpler assumption class, called UCA's. The resulting algorithm is computationally more tractable and ensured to terminate. Nevertheless, unlike the semi-algorithm discussed in the previous section, this algorithm may not be able to compute a GWSE in all scenarios where the semi-algorithm can.

5.1 From APA's to UCA's

One of the main features of APA's on Player i computed by ComputeAPA from [1], is the fact that they can be expressed by well structured templates using Player i's edges, namely *unsafe-edge-*, *colive-edge-*, and *(conditional)-live-group-templates.* Unsafe- and colive-edge-templates are structurally very simple. Given a set of unsafe edges $S \subseteq E_i$ and colive edges $C \subseteq E_i$ the respective assumption templates $\psi_{\text{UNSAFE}}(S) := \bigwedge_{e \in S} \square \neg e$ and $\psi_{\text{COLIVE}}(C) := \bigwedge_{e \in C} \lozenge \square \neg e$ simply assert that unsafe (resp. colive) edges should never (resp. only finitely often) be taken. We call an assumption which can be expressed by these two types of templates an **U**nsafe- and **C**olive-edge-template **A**ssumption (UCA), as defined next.

Definition 6. *Given a k-player game graph $G = (V, E)$, a specification ψ is called an unsafe- and colive-edge-template assumption (UCA) for Player i, if there exist sets $S, C \subseteq E_i$ s.t. $\psi := \psi_{\text{UNSAFE}}(S) \wedge \psi_{\text{COLIVE}}(C)$. We write $\psi^{[S,C]}$ to denote such assumptions.*

It was recently shown by Schmuck et al. [27] that two-player (zero-sum) parity games under UCA assumptions, i.e., games $(G, \psi \Rightarrow \phi)$ where ψ is an UCA and ϕ is a parity specification over G, can be directly solved over G without computational overhead, compared to the non-augmented version (G, ϕ) of the same game. Interestingly, the synthesis problem under assumptions becomes proveably harder if live-group-templates ψ_{COND} are needed to express an assumption, requiring a change of the game graph in most cases. Conditional-live-group-templates ψ_{COND}, are structurally more challenging than UCA's, as they impose a Streett-type fairness conditions on edges in G (see [1, Sec.4] for details).

Motivated by this result, we will restrict the assumption class used for GWSE computation to UCA's in this section. Unfortunately, UCA's are typically not expressive enough to capture APA's for parity games. This follows from one of the main results of Anand et al., which shows that APA's computed by ComputeAPA for parity games are expressible by a conjunctions of all *three* template types, as re-stated in the following proposition.

Proposition 1 ([1, Thm. 3]). *Given the premisses of Lem. 1, the APA computed by ComputeAPA on Player i can be written as the conjunction $\psi := \psi_{\text{UNSAFE}}(S) \wedge \psi_{\text{COLIVE}}(C) \wedge \psi_{\text{COND}}$ where $S, C \subseteq E_i$.*

We therefore need to overapproximate APA's by UCA's, by simply dropping the ψ_{COND}-term from their defining conjunction, as formalized next.

Definition 7. *Given the premisses of Lem. 1, let $\psi := \textsc{ComputeAPA}(G, \phi, i) = \psi_{\textsc{unsafe}}(S) \wedge \psi_{\textsc{colive}}(C) \wedge \psi_{\textsc{cond}}$. Then we denote by $\textsc{ApproxAPA}(G, \phi, i)$ the algorithm that computes $\psi^{[S,C]}$ by first executing $\textsc{ComputeAPA}(G, \phi, i)$ and then dropping all $\psi_{\textsc{cond}}$-terms from the resulting APA.*

It is easy to see that $\mathcal{L}(\psi) \subseteq \mathcal{L}(\psi^{[S,C]})$. Therefore, it also follows that $\psi^{[S,C]}$ is implementable and permissive (i.e., Def. 4(ii) and (iii) holds). Unfortunatly, $\psi^{[S,C]}$ is in general no longer sufficient (i.e., Def. 4(i) does not necessarily hold). As the proof of Thm. 1 only uses permissiveness of APA, even though sufficiency is lost for UCA's, replacing ComputeAPA by ApproxAPA in Algo. 1 does not mitigate soundness, i.e., whenever ComputeGE terminates in line 6 with a specification profile $(\varphi_i)_{i \in \mathtt{P}}$, this profile is indeed a GWSE, even if APA's are over-approximated by UCA's. This is formalized next.

Theorem 2. *Let AComputeGE be the algorithm obtained by replacing procedure ComputeAPA by ApproxAPA in Algo. 1. Then, given a k-player game $\mathcal{G}$ with parity specifications such that $(\varphi_i^*)_{i \in \mathtt{P}} = \textsc{AComputeGE}(\mathcal{G})$, the tuple $(\varphi_i^*)_{i \in \mathtt{P}}$ is a GWSE for $\mathcal{G}$.*

The rest of this section will now show how the restriction to UCA's allows to execute lines 5 and 7 in Algo. 1 efficiently and allows to prove termination of the resulting algorithm for GWSE computation.

5.2 Iterative Computation of UCA's

We have seen in the previous section that UCA's can be computed by utilizing ComputeAPA and dropping all ψ_{COND} terms (called ApproxAPA). Of course, this can be done in every iteration of ComputeGE. However, ComputeAPA expects a party game as an input, and from the second iteration of ComputeGE onward the input to ComputeAPA is given by $(G, \psi_{-i} \wedge \phi_i, i)$, where ψ_{-i} is an assumption on players in $\mathtt{P}_{-i}$, which is not necessarily a parity game.

This section therefore provides a new algorithm, called ComputeUCA and given in Algo. 2 which computes UCA's for Player i directly on the game graph G for games $(G, \psi \wedge \phi)$ where $\psi = \psi^{[S,C]}$ is an UCA for $\mathtt{P}_{-i}$ with unsafe edges $S \subseteq E_{-i}$ and colive edges $C \subseteq E_{-i}$, and ϕ is a parity specification, both over G. Intuitively, ComputeUCA first slightly modifies G to a new two-player game graph $\hat{G}$ (lines 1 and 2) s.t. the specification $\psi \wedge \phi$ can be directly expressed as a parity specification $\hat{\phi}$ on $\hat{G}$ (line 4). This allows to apply ApproxAPA to construct and return an UCA for Player 1 on $\hat{G}$ (line 5). As the resulting UCA is for Player i, the unsafe edge and colive edge sets are subsets of E_i. Further, due to the mild modifications from G to $\hat{G}$, the edges of Player i are retained in $\hat{G}$ as E_1, hence, the resulting UCA is a well-defined UCA for Player i in G.

We have the following soundness result for showing equivalence between the UCA's computed by ComputeUCA and ApproxAPA for UCA assumptions, proven in extended version of this paper [26, App. A].

Algorithm 2 COMPUTEUCA$(G, \psi^{[S,C]} \wedge \phi, i)$

```
Require: A $k$-player game graph $G = (V, E, v_0)$ and specification $\psi \wedge \phi$ with UCA
    $\psi = \psi^{[S,C]}$ for $\mathtt{P}_{-i}$, i.e., $S, C \subseteq E_{-i}$, and $\phi = \mathit{Parity}(\Omega)$ s.t. $\Omega : V \to [0; 2d+1]$.
Ensure: An UCA $\psi^{[S',C']}$ for Player $i$.
1: $\hat{V}_1 \leftarrow V_i$ and $\hat{V}_2 \leftarrow V_{-i} \uplus C$
2: $\hat{E}_1 \leftarrow E_i$ and $\hat{E}_2 \leftarrow E_{-i} \setminus (S \cup C) \cup \{(u,c),(c,v) \mid c = (u,v) \in C\}$
3: $\hat{\Omega} = \begin{cases} \Omega(v) & \text{if } v \in V \\ 2d+1 & \text{otherwise.} \end{cases}$
4: $\hat{G} = (\hat{V}_1 \uplus \hat{V}_2, \hat{E}_1 \uplus \hat{E}_2, v_0)$; $\hat{\phi} \leftarrow \mathit{Parity}(\hat{\Omega})$
5: return APPROXAPA$(\hat{G}, \hat{\phi}, 1)$
```

Proposition 2. *Given game graph $G = (V, E, v_0)$ with parity specification ϕ and an UCA $\psi = \psi^{[S,C]}$ for $\mathtt{P}_{-i}$, let $\psi' :=$ APPROXAPA$(G, \psi \wedge \phi, i)$ and $\psi'' :=$ COMPUTEUCA$(G, \psi \wedge \phi, i)$ then $\mathcal{L}(\psi') = \mathcal{L}(\psi'')$. Furthermore, COMPUTEUCA terminates in time $\mathcal{O}((|V| + |E|)^4)$.*

The proof of this result is given in extended version [26, App. A], and essentially relies on the observation that the parity specification $\hat{\phi}$ in $\hat{G}$ expresses the language $\mathcal{L}(\psi \wedge \phi)$ when restricted to V, i.e, $\mathcal{L}(\hat{G}, \hat{\phi})|_V = \mathcal{L}(G, \phi \wedge \psi)$ and the fact that every UCA for Player 1 in $\hat{G}$ is also an UCA for Player i in G.

The usefulness of expressing the computed assumptions as unsafe and colive edge sets S, C over the input game graph G is that there are only a finite number of edges in that graph. Therefore, there obviously also exists only a finite number of unsafe or colive edge sets, which could all be enumerated in the worst case. Therefore, computing UCA's on the same game graph in every iteration, will ensure termination of the overall computation of GWSE.

5.3 Solving Parity Games under UCA's

As the final step towards an optimized version of Algo. 1, we now address the computations required in line 5 of Algo. 1. Observe that this line requires to check $v_0 \in \bigcap_{i \in \mathtt{P}} \langle\langle i \rangle\rangle \varphi_i$ for $\varphi_i = \psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)$. If this check returns True the algorithm terminates, if it returns False new assumptions are computed. In both cases, the game graph used to check this conditional will not have any effect on the future behavior of the algorithm.

Nevertheless, we utilize the recent result by Schmuck et al. [27] to compute $\langle\langle i \rangle\rangle \varphi_i$ more efficiently if ψ_i and ψ_{-i} are UCA's on Player i and $\mathtt{P}_{-i}$, respectively. The construction uses the same idea as presented in Algo. 2 to encode UCA's into a new, slightly modified two-player parity game $(\hat{G}, \hat{\phi})$ which can then be solved by a standard parity solver, such as Zielonka's algorithm [30], which return the winning region $\mathcal{W}$ of Player 1 in this new game that corresponds to the winning region of Player i in G. The resulting algorithm is called COMPUTEWIN given in the extended version [26, Algo. 3] and has the property that $v_0 \in \langle\langle i \rangle\rangle(G, \varphi)$ if and only if $v_0 \in \mathcal{W}$. This is formalized and proven in the extended version [26, Prop. 3].

5.4 Computation of GWSE via UCA's

With the previously discussed algorithms in place, we are now in the position to propose an optimized, surely terminating algorithm to compute GWSE, called OComputeGE. Within ComputeGE the recursive procedure RecursiveGE is replaced by one which uses the algorithms ComputeUCA and ComputeWin for UCA's from Sec. 5.2 and 5.3, as follows

```
procedure RecursiveGE($\mathcal{G}, (\psi_i)_{i \in \mathtt{P}}$)
    $\varphi_i \leftarrow \psi_i \wedge (\psi_{-i} \Rightarrow \phi_i)\ \forall i \in \mathtt{P}$
    $\mathcal{W}_i \leftarrow$ ComputeWin($\mathcal{G}, \varphi_i, i$)
    if $v_0 \in \bigcap_{i \in \mathtt{P}} \mathcal{W}_i$ then
        return $(\varphi_i)_{i \in \mathtt{P}}$
    $\psi'_i \leftarrow \psi_i \wedge$ ComputeUCA($G, \psi_{-i} \wedge \phi_i, i$) $\forall i \in \mathtt{P}$
    if $\psi'_i = \psi_i$ for all $i \in \mathtt{P}$ then
        return False
    return RecursiveGE($\mathcal{G}, (\psi'_i)_{i \in \mathtt{P}}$)
```

We have the following main result of this section.

Theorem 3. *Let $\mathcal{G}$ be a k-player game with game graph $G = (V, E, v_0)$ and parity specifications $(\phi_i)_{i \in \mathtt{P}}$ such that $(\varphi^*_i)_{i \in \mathtt{P}} =$ OComputeGE($\mathcal{G}$), then $(\varphi^*_i)_{i \in \mathtt{P}}$ is a GWSE for $\mathcal{G}$. Moreover, OComputeGE terminates in time $\mathcal{O}(k^2 |E| \cdot (2|V| + 2|E|)^{d+2})$, where d is the number of priorities used in the parity specifications.*

Proof. Combining results from Thm. 1 with Thm. 2 and Prop. 2 gives us that $(\varphi^*_i)_{i \in \mathtt{P}}$ is indeed a GWSE for $\mathcal{G}$. Furthermore, as ψ_i (for all $i \in \mathtt{P}$) in each iteration of the algorithm either remains the same or add more unsafe/colive edges, it can only change $2|E|$ times. Hence, as there are k players, the algorithm OComputeGE will terminate within $2k|E|$ iterations. Moreover, each iteration involves k calls to both ComputeWin and ComputeUCA. Using Zielonka's algorithm[4] [30] for solving parity games, each iteration will take $\mathcal{O}((2|V| + 2|E|)^{d+2})$ time for d priorities (by Prop. 2). In total, this gives us that OComputeGE terminates in time $\mathcal{O}(k^2 |E| \cdot (2|V| + 2|E|)^{d+2})$. □

Remark 4. As Anand et al. show that APA's for games with co-Büchi specifications (i.e., $\phi = \Diamond \Box T$ for some $T \subseteq V$) are always expressible by UCA's [1, Thm. 3], we note that ComputeAPA and ApproxAPA coincide for such games. This implies that no over approximation of assumptions is needed in this case an the optimizations discussed for ComputeUCA and ComputeWin can be directly applied for APA's.

We further note that OComputeGE also efficiently computes GWSE for games with more expressive specifications than co-Büchi . For instance, all games discussed in this paper as well as the mutual exclusion protocol discussed in [15] can be solved by OComputeGE.

[4] We note that the time complexity is exponential as we use Zielonka's algorithm [30] to solve parity games. One can also use a quasi-polynomial algorithm [11] for solving parity games to get a quasi-polynomial time complexity for OComputeGE.

References

1. Anand, A., Mallik, K., Nayak, S.P., Schmuck, A.: Computing adequately permissive assumptions for synthesis. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 211–228. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_15, https://doi.org/10.1007/978-3-031-30820-8_15
2. Anand, A., Nayak, S.P., Schmuck, A.: Contract-based distributed synthesis in two-objective parity games. CoRR **abs/2307.06212** (2023). https://doi.org/10.48550/ARXIV.2307.06212, https://doi.org/10.48550/arXiv.2307.06212
3. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
4. Bloem, R., Chatterjee, K., Jacobs, S., Könighofer, R.: Assume-guarantee synthesis for concurrent reactive programs with partial information. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 517–532. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_50, https://doi.org/10.1007/978-3-662-46681-0_50
5. Brenguier, R., Raskin, J., Sankur, O.: Assume-admissible synthesis. Acta Informatica **54**(1), 41–83 (2017). https://doi.org/10.1007/s00236-016-0273-2, https://doi.org/10.1007/s00236-016-0273-2
6. Brice, L., Raskin, J., van den Bogaard, M.: Subgame-perfect equilibria in mean-payoff games. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 8:1–8:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.8, https://doi.org/10.4230/LIPIcs.CONCUR.2021.8
7. Brice, L., Raskin, J., van den Bogaard, M.: Rational verification for nash and subgame-perfect equilibria in graph games. In: Leroux, J., Lombardy, S., Peleg, D. (eds.) 48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France. LIPIcs, vol. 272, pp. 26:1–26:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.MFCS.2023.26, https://doi.org/10.4230/LIPIcs.MFCS.2023.26
8. Brihaye, T., Bruyère, V., De Pril, J.: On equilibria in quantitative games with reachability/safety objectives. Theory Comput. Syst. **54**(2), 150–189 (2014). https://doi.org/10.1007/s00224-013-9495-7, https://doi.org/10.1007/s00224-013-9495-7
9. Bruyère, V., Meunier, N., Raskin, J.: Secure equilibria in weighted games. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. pp. 26:1–26:26. ACM (2014). https://doi.org/10.1145/2603088.2603109, https://doi.org/10.1145/2603088.2603109
10. Bruyère, V., Roux, S.L., Pauly, A., Raskin, J.: On the existence of weak subgame perfect equilibria. Inf. Comput. **276**, 104553 (2021). https://doi.org/10.1016/j.ic.2020.104553, https://doi.org/10.1016/j.ic.2020.104553

11. 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.) Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017. pp. 252–263. ACM (2017). https://doi.org/10.1145/3055399.3055409, https://doi.org/10.1145/3055399.3055409
12. Chatterjee, K., Doyen, L., Filiot, E., Raskin, J.: Doomsday equilibria for omega-regular games. Inf. Comput. **254**, 296–315 (2017). https://doi.org/10.1016/j.ic.2016.10.012, https://doi.org/10.1016/j.ic.2016.10.012
13. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Games with secure equilibria. Theor. Comput. Sci. **365**(1-2), 67–82 (2006). https://doi.org/10.1016/j.tcs.2006.07.032, https://doi.org/10.1016/j.tcs.2006.07.032
14. Chatterjee, K., Horn, F., Löding, C.: Obliging games. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6269, pp. 284–296. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_20, https://doi.org/10.1007/978-3-642-15375-4_20
15. Chatterjee, K., Raman, V.: Assume-guarantee synthesis for digital contract signing. Formal Aspects Comput. **26**(4), 825–859 (2014). https://doi.org/10.1007/s00165-013-0283-6, https://doi.org/10.1007/s00165-013-0283-6
16. Damm, W., Finkbeiner, B.: Automatic compositional synthesis of distributed systems. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8442, pp. 179–193. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_13, https://doi.org/10.1007/978-3-319-06410-9_13
17. De Pril, J., Flesch, J., Kuipers, J., Schoenmakers, G., Vrieze, K.: Existence of secure equilibrium in multi-player games with perfect information. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II. Lecture Notes in Computer Science, vol. 8635, pp. 213–225. Springer (2014). https://doi.org/10.1007/978-3-662-44465-8_19, https://doi.org/10.1007/978-3-662-44465-8_19
18. Filiot, E., Gentilini, R., Raskin, J.: Rational synthesis under imperfect information. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 422–431. ACM (2018). https://doi.org/10.1145/3209108.3209164, https://doi.org/10.1145/3209108.3209164
19. Finkbeiner, B., Passing, N.: Compositional synthesis of modular systems. Innov. Syst. Softw. Eng. **18**(3), 455–469 (2022). https://doi.org/10.1007/s11334-022-00450-w, https://doi.org/10.1007/s11334-022-00450-w
20. Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6015, pp. 190–204. Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_16, https://doi.org/10.1007/978-3-642-12002-2_16

21. Kremer, S., Raskin, J.: A game-based verification of non-repudiation and fair exchange protocols. J. Comput. Secur. **11**(3), 399–430 (2003). https://doi.org/10.3233/jcs-2003-11307, https://doi.org/10.3233/jcs-2003-11307
22. Kupferman, O., Shenwald, N.: The complexity of LTL rational synthesis. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 25–45. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_2, https://doi.org/10.1007/978-3-030-99524-9_2
23. Li, X., Li, X., Xu, G., Hu, J., Feng, Z.: Formal analysis of fairness for optimistic multiparty contract signing protocol. J. Appl. Math. **2014**, 983204:1–983204:10 (2014). https://doi.org/10.1155/2014/983204, https://doi.org/10.1155/2014/983204
24. Majumdar, R., Mallik, K., Schmuck, A., Zufferey, D.: Assume-guarantee distributed synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(11), 3215–3226 (2020). https://doi.org/10.1109/TCAD.2020.3012641, https://doi.org/10.1109/TCAD.2020.3012641
25. Majumdar, R., Piterman, N., Schmuck, A.: Environmentally-friendly GR(1) synthesis. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11428, pp. 229–246. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_13, https://doi.org/10.1007/978-3-030-17465-1_13
26. Nayak, S.P., Schmuck, A.K.: Most general winning secure equilibria synthesis in graph games. CoRR **abs/2401.09957** (2024). https://doi.org/10.48550/arXiv.2401.09957, https://doi.org/10.48550/arXiv.2401.09957
27. Schmuck, A.K., Thejaswini, K.S., Sağlam, I., Nayak, S.P.: Solving two-player games under progress assumptions. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 208–231. Springer Nature Switzerland, Cham (2024)
28. Steg, J.: On identifying subgame-perfect equilibrium outcomes for timing games. Games Econ. Behav. **135**, 74–78 (2022). https://doi.org/10.1016/j.geb.2022.05.012, https://doi.org/10.1016/j.geb.2022.05.012
29. Ummels, M.: Rational behaviour and strategy construction in infinite multiplayer games. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4337, pp. 212–223. Springer (2006). https://doi.org/10.1007/11944836_21, https://doi.org/10.1007/11944836_21
30. 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, https://doi.org/10.1016/S0304-3975(98)00009-7

On-The-Fly Algorithm for Reachability in Parametric Timed Games [⋆]

Mikael Bisgaard Dahlsen-Jensen[1](✉), Baptiste Fievet[2], Laure Petrucci[2], and Jaco van de Pol[1]

[1] Aarhus Univ rsity, Aarhus, D nmark
{mikael,jaco}@cs.au.dk
[2] LIPN, CNRS UMR 7030, Univ rsit´ Sorbonn Paris Nord, Vill tan us , Franc
{Baptiste.Fievet,Laure.Petrucci}@lipn.univ-paris13.fr

Abstract. Param tric Tim d Gam s (PTG) ar an xt nsion o th mod l o Tim d Automata. Th y allow or th v rification and synth sis o r al-tim syst ms, r activ to th ir nvironm nt and d p nding on adjustabl param t rs. Giv n a PTG and a r achability obj ctiv , w synth siz th valu s o th param t rs such that th gam is winning or th controll r. W adapt and impl m nt th On-Th -Fly algorithm or param t r synth sis or PTG. S v ral pruning h uristics ar introduc d, to improv t rmination and sp d o th algorithm. W valuat th asibility o param t r synth sis or PTG on two larg cas studi s. Finally, w inv stigat th corr ctn ss guarant o th algorithm: though th probl m is und cidabl , our s mi-algorithm produc s all corr ct param t r valuations “in th limit”.

1 Introduction

The seminal model of Timed Automata (TA) [1] equips finite automata with real-valued clocks, to verify real-time reactive systems. Numerous extensions of TA have been proposed. Timed Games (TG) [18] distinguish controllable and uncontrollable actions, to study the interaction of a controller with its environment (e.g. the plant, an attacker, or a system-under-test). Here, we focus on reachability objectives, which require a strategy for the controller to schedule controllable actions such that — no matter which and when uncontrollable actions are executed by the environment — a desirable state is reached.

Since precise timing constraints are not always known, one might replace concrete values by symbolic parameters, to study a whole family of timed systems. This leads to the model of Parametric Timed Automata (PTA) [2]. The problem is to find (some or all) values for the parameters such that the system satisfies a desired property. Most problems on PTA are undecidable [3], in particular the reachability problem. Several decidable fragments are known, e.g. by restricting the number of clocks or the positions of the parameters, as in L/U PTA [14].

[⋆] This work was partially support d by CNRS int rnational PhD programm , th CNRS Int rnational R s arch N twork CLoV and Innovations ond n Danmark’s DIREC proj ct SIoT (S cur Int rn t o Things).

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 194–212, 2024.
https://doi.org/10.1007/978-3-031-57256-2_10

This paper tackles the parameter synthesis problem for Parametric Timed Games (PTG) [15] with reachability objectives. We provide the first implementation of a semi-algorithm for PTG parameter synthesis. It operates on-the-fly, i.e. it starts solving the game while the symbolic state space is being generated. To avoid the generation of the full, potentially infinite, state space, we also implement several state space reductions. These improve the termination and efficiency of parameter synthesis. In particular, we lift inclusion/subsumption from TA to PTG, generalize coverage pruning and losing state propagation from TG to PTG, and we port cumulative pruning from PTA to PTG.

Interestingly, unlike the situation in PTA [5] and TG [10], the algorithm for PTG is not guaranteed to terminate, even if the symbolic state space is finite. But we claim that if the algorithm terminates, it produces the precise constraints under which there exists a winning strategy. If the algorithm does not terminate, the stronger guarantee holds, that (in the limit) it produces all valid parameter valuations, provided the waiting list is handled fairly.

The implementation allows us to study the feasibility of parameter synthesis for larger case studies in PTG. In particular, we synthesize parameters for the correctness of a game version of the Bounded Retransmission Protocol [13] and a parametric version of the Production Cell [19,10]. We measure the effectiveness of the individual pruning heuristics on these case studies. It appears that the state space reduction techniques are essential for feasible parameter synthesis.

Related Work. For TG, Maler et al. [18] proposed a strategy synthesis algorithm based on classical reachability games, handling the uncountable set of clock values using symbolic regions. Cassez et al. [10] improved the efficiency of TG strategy synthesis by an on-the-fly algorithm, and working with symbolic zones, represented by DBMs as implemented in UPPAAL Tiga [8]. Previous work on PTG initially focused on decidable subcases, like the case for bounded integers [16] and the fragment of L/U PTG [15,17]. The latter two papers also provide semi-algorithms for general PTG, either based on backward fixed points [17], or an on-the-fly algorithm [15], directly extending the work on Timed Games [10]. That paper leaves an implementation of the algorithm (and hence an evaluation on larger case studies) as future work. Our implementation extends the infrastructure of Imitator [4], which so far could only handle PTA. The symbolic data structure is based on Parma's convex Polyhedra Library [7].

Contributions. (1) We provide the first implementation of a parameter synthesis algorithm for PTG (Sec. 4), and integrate this on-the-fly algorithm in the Imitator toolset [4] (Sec. 6).

(2) We devise and implement several pruning heuristics to speed up parameter synthesis (Sec. 5).

(3) We evaluate the feasibility of parameter synthesis for PTG on two large case studies, and measure the effect of the various pruning techniques (Sec. 6).

(4) We carefully introduce the model (Sec. 2) and solution principles (Sec. 3), pointing out several semantic subtleties, and find that the semi-algorithm yields all valid parameters in the limit (Sec. 4).

2 Model of Parametric Timed Games

A Parametric Timed Game (PTG) is a structure based on timed automata (TA). Similarly to classical automata, it is composed of locations connected by discrete transitions. Moreover, it is equipped with clocks. Locations are associated a condition on clock valuations (invariant) that must be satisfied while staying in the location. An action in a timed automaton is either to take a discrete transition or to let some time pass. Discrete transitions have a guard that must be satisfied in order to take the transition. In a parametric setting, these conditions use linear terms over clocks and parameters. Parameters hold an unspecified value, and remain constant during a run. A discrete transition also has a subset of clocks which are reset when the transition is taken.

In a two-player timed game, discrete transitions are partitioned between controllable transitions and uncontrollable environment transitions.

Definition 1 (PTG). *A* Parametric Timed Game *is a tuple of the form* $G = (L, X, P, Act, T_c, T_u, \ell_0, Inv)$ *such that*

- *L, X, P, Act are sets of* locations, clocks, parameters, transition labels.
- *$T = T_c \cup T_u$ is the set of* transitions *in* $L \times \mathcal{G}(X, P) \times Act \times \mathcal{P}(X) \times L$, *partitioned into sets T_c of* controllable *and T_u of* uncontrollable *transitions of the form $(\ell, g, \mathrm{a}, Y, \ell')$; ℓ, ℓ' are source and target locations; $g \in \mathcal{G}(X, P)$ is the guard (see Def. 4); a is the label; Y is the set of clocks to reset.*
- *ℓ_0 is the initial location.*
- *$Inv : L \to \mathcal{G}(X, P)$ associates an* invariant *with each location.*

Example 1. Fig. 1 shows the example of a coffee machine. The controller represents the coffee machine and the environment represents the user. Uncontrollable transitions are depicted as dashed arcs. From idle, the user can ask_coffee. It resets clock y that will measure the time since the demand. The machine is then preparing_coffee. Action serve_coffee can happen after p_3 (parameter featuring the time to pour the coffee) and no later than p_4 after the request. While the coffee is being prepared, the user may add_sugar. Adding sugar does not interrupt the pouring of the coffee and lasts p_2. The coffee cannot be served while

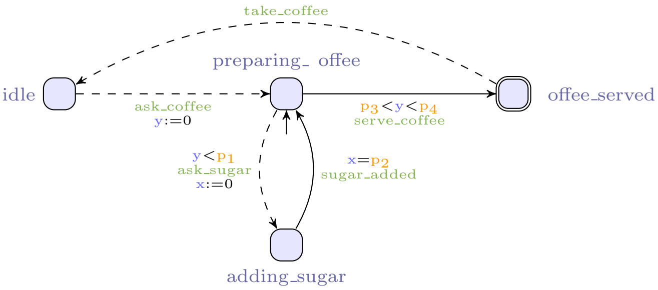


Fig. 1. Param tric Tim d Gam o th coff machin .

sugar is being added. A situation that may arise is that sugar is being added to the coffee when the time limit p_4 is met, making it impossible for the coffee to be served on time. To avoid this issue, ask_sugar is disabled after waiting p_1.

Our goal is to synthesize the constraints on parameters p_1 to p_4 for the coffee to be timely served. Hence, the initial location is set to preparing_coffee, with both clocks at 0. One possible solution to the problem is $\mathrm{p}_1 + \mathrm{p}_2 \leq \mathrm{p}_4 \wedge \mathrm{p}_3 < \mathrm{p}_4$.

2.1 Semantics of Parametric Timed Games

A *state* of a PTG consists of a location and a valuation of clocks and parameters.

Definition 2 (valuations). *A* clock valuation *is a function* $v_X \in \mathbb{R}^X_{\geq 0}$ *assigning a positive real value to each clock. A* parameter valuation $v_P \in \mathbb{Q}^P_{\geq 0}$ *assigns a positive rational value to each parameter. A* valuation of the game G *is a pair* $v = (v_X, v_P)$*. The set of all valuations of the game is denoted* $V = \mathbb{R}^X_{\geq 0} \times \mathbb{Q}^P_{\geq 0}$.

A *guard* is a constraint that can be satisfied by some valuations of the game.

Definition 3 (linear terms). *A* linear term *over* P *is a term defined by the following grammar:* $plt \ := \ k \mid k\mathrm{p} \mid plt + plt$ *where* $k \in \mathbb{Q}$ *and* $\mathrm{p} \in P$.

Definition 4 (guards). *The set of* guards $\mathcal{G}(X, P)$ *is the set of formulas defined inductively by the following grammar:*

$$\phi \ := \top \mid \phi \wedge \phi \mid \mathrm{x} \sim plt \mid plt' \sim plt \ \ ,$$

where $\mathrm{x} \in X$, $\sim \in \{<; \leq; =; \geq; >\}$ *and* plt, plt' *are linear terms over* P.

We now introduce the notion of *zone* which will be used to solve a PTG.

Definition 5 (zones). *The set of* parametric zones $\mathcal{Z}(X, P)$ *is the set of formulas defined inductively by the following grammar:*

$$\phi \ := \ \top \mid \phi \wedge \phi \mid \mathrm{x} \sim plt \mid \mathrm{x} - \mathrm{y} \sim plt \mid plt' \sim plt \ \ ,$$

where $\mathrm{x}, \mathrm{y} \in X$, $\sim \in \{<; \leq; =; \geq; >\}$ *and* plt *and* plt' *are linear terms over* P.

Function v_P is naturally extended to linear terms on parameters, by replacing each parameter in the term with its valuation. With $v \models \phi$, we denote that valuation $v = (v_X, v_P)$ *satisfies* a guard or a zone ϕ, which is defined in the expected manner. Zones, guards and invariants can also be seen as a convex set in the space of valuations of the game by considering those valuations that satisfy the condition.

Transitions modify clock valuations by letting time pass or resetting clocks.

Definition 6 (time delays). *Let* $v = (v_X, v_P)$ *be a valuation of the game and* $\delta \geq 0$ *a delay.*

- $\forall \mathrm{x} \in X : (v_X + \delta)(\mathrm{x}) = v_X(\mathrm{x}) + \delta$
- $v + \delta = (v_X + \delta, v_P)$

Definition 7 (clock resets). *Let $v = (v_X, v_P)$ be a valuation of the game and $Y \subseteq X$. $v_X[Y := 0]$ is the valuation obtained by resetting the clocks in Y, i.e.:*

- $\forall \mathrm{x} \in Y : v_X[Y := 0](\mathrm{x}) = 0$ *and* $\forall \mathrm{x} \in X \setminus Y : v_X[Y := 0](\mathrm{x}) = v_X(\mathrm{x})$
- $v[Y := 0] = (v_X[Y := 0], v_P)$

We can now define the semantics of a Parametric Timed Game.

Definition 8 (state). *A* state *of a PTG is a pair (ℓ, v) where ℓ is a location and v a valuation of the game satisfying its invariant: $v \models Inv(\ell)$. The* state space *is then* $\mathbb{S} = \{(\ell, v) \in L \times V \mid v \models Inv(\ell)\} = \bigcup_{\ell \in L} \{\ell\} \times Inv(\ell)$.

From a state in this state space, timed and discrete transitions can happen.

Definition 9 (timed and discrete transitions). *Let $\delta \in \mathbb{R}_{\geq 0}$ be a time delay. A* timed transition *is a relation $\rightarrow^{\delta} \in \mathbb{S} \times \mathbb{S}$ s.t. $\forall (\ell, v), (\ell', v') \in \mathbb{S} : (\ell, v) \rightarrow^{\delta} (\ell', v')$ iff $\ell = \ell'$ and $v' = v + \delta$.*

Let $t = (\ell, g, \mathrm{a}, Y, \ell') \in T$ be a transition. A discrete transition *is a relation $\rightarrow^{t} \in \mathbb{S} \times \mathbb{S}$ s.t. $\forall (\ell, v), (\ell', v') \in \mathbb{S} : (\ell, v) \rightarrow^{t} (\ell', v')$ iff $v \models g$ and $v' = v[Y := 0]$.*

Let $\vec{0}$ be the clock valuation where all clocks have value 0. The set of possible initial states of the PTG is $\xi_0 = \{(\ell_0, (\vec{0}, v_P)) \mid v_P \in \mathbb{Q}^{P}_{\geq 0} : (\vec{0}, v_P) \models Inv(\ell_0)\}$.

Definition 10 (run). *A* run *of the PTG G is a finite or infinite sequence of states $s_0 s_1 s_2 \ldots$ s.t. $s_0 \in \xi_0$ and $\forall i \in \mathbb{N},\ s_{2i} \rightarrow^{\delta} s_{2i+1} \rightarrow^{t} s_{2i+2}$. $\mathcal{R}(G)$denotes the set of runs, and $\mathcal{R}(G)(s)$ the set of those starting from state s.*

A run alternates between (potentially null) delays and discrete transitions, avoiding runs that let only time pass. However, there might still be Zeno runs where infinitely many discrete transitions are taken in a finite amount of time. When there is no ambiguity, we omit G in the notations.

Example 2. Let us consider again the coffee machine in Fig. 1. Assume the parameter valuations are: $v_P(\mathrm{p}_1) = 5$, $v_P(\mathrm{p}_2) = 2$, $v_P(\mathrm{p}_3) = 5$ and $v_P(\mathrm{p}_4) = 6$. Let $v_X = (v_X(\mathrm{x}), v_X(\mathrm{y}))$. We get the sequence: $(\text{preparing\_coffee}, ((0, 0), v_P)) \rightarrow^{4} (\text{preparing\_coffee}, ((4, 4), v_P)) \rightarrow^{\text{ask\_sugar}} (\text{adding\_sugar}, ((0, 4), v_P)) \rightarrow^{2} (\text{adding\_sugar}, ((2, 6), v_P)) \rightarrow^{\text{sugar\_added}} (\text{preparing\_coffee}, ((2, 6), v_P))$.

Definition 11 (history). *A* history *is a finite prefix of a run. The set of histories of game G is denoted $\mathcal{H}(G)$, and those starting in state s by $\mathcal{H}(G)(s)$.*

The notion of coverage allows for capturing all states that can occur up to some time, without a discrete transition.

Definition 12 (coverage). *Let $s, s' \in \mathbb{S}$ and $\delta \geq 0$ such that $s \rightarrow^{\delta} s'$. The* coverage *of the timed transition is the set of intermediate states traversed:* $Cover(s \rightarrow^{\delta} s') = \{s'' \in \mathbb{S} \mid \exists \delta' : 0 \leq \delta' \leq \delta\ \wedge\ s \rightarrow^{\delta'} s''\}$.

The coverage of state s is the set of states obtained from s with timed transitions only: $Cover(s) = \{s' \in \mathbb{S} \mid \exists \delta \geq 0\ s \rightarrow^{\delta} s'\}$.

The coverage of a run $r = s_0 s_1 s_2 \ldots$ is the union of the coverage of its timed transitions. When finite, it includes the coverage of its last state $ls(r)$: $Cover(r) = \left(\bigcup_{i \in \mathbb{N}} Cover(s_{2i} \rightarrow^{\delta} s_{2i+1})\right) \cup Cover\big(ls(r)\big)$.

Definition 13 (reachability objective and winning runs). *Let $R \subseteq L$ be a* reachability objective*. The set of* winning runs $\Omega_{Re\ ch}(R)$ *is the subset of runs that visit R:* $\Omega_{Re\ ch}(R) = \{r \in \mathcal{R} \mid \exists\, \ell \in R, \exists v \in V : (\ell, v) \in Cover(r)\}$.

Example 3. In the coffee machine, the objective is to reach from the initial location prepare_coffee the location coffee_served. The reachability objective is thus $R = \{$coffee_served$\}$, and the set of winning runs is $\Omega_{Re\ ch}(\{$coffee_served$\})$.

2.2 Strategies in Parametric Timed Games

We introduce a definition of a strategy that deviates from [10], where at each moment, a player decides to either wait, or take a discrete transition. So their strategy returns values in $T \cup \{wait\}$. The problem with their strategy is that it is not always clear what should happen: for instance, given a delay δ, a history $h = s_0 \rightarrow^{\delta} s_1$ and a strategy σ, where $\sigma(h) = wait$ for $0 < \delta \leq 1$ and $\sigma(h) = t$ for $\delta > 1$, it is not clear when transition t happens: there is no minimal $\delta > 1$. Although this works formally, it is less clear what the allowed behaviour of the winning player is precisely. For that reason, in our definition of strategy, players must decide in advance which delay they will take. This makes the definition more constructive, clarifying what move the winning player will actually take (i.e. perform an action or decide to wait for some particular time) and in the end simplifies the definition of what is winning.

Furthermore, following [10], the definition of strategy is asymmetric for controller and environment: If both wish to do a discrete transition, we provide priority to the environment; this corresponds to the safest situation from a software controller point of view. Another subtle asymmetry is that the controller cannot assume that the environment will take some uncontrollable transition, even when waiting any longer would violate the location invariant. While this is in line with the formal definition of strategy in TG [10], experiments with UPPAAL Tiga [8] reveal that in that tool, an uncontrollable discrete transition is actually forced when reaching the boundary of violating an invariant.

Definition 14 (strategy). *A* controller strategy σ_c *(resp.* environment strategy σ_e*) models decision-making. It is a function, depending on a history, deciding either to wait some amount of time (possibly infinite) or to take a discrete transition:* $\sigma_c : \mathcal{H} \rightarrow \mathbb{R}^{\infty}_{\geq 0} \cup T_c$*,* $\sigma_e : \mathcal{H} \rightarrow \mathbb{R}^{\infty}_{\geq 0} \cup T_u$ *s.t.* $\forall h \in \mathcal{H}$ *and* $\sigma \in \{\sigma_c, \sigma_e\}$,

1. *If* $\sigma(h) = (\ell, g, \mathrm{a}, Y, \ell') \in T$
   then $ls(h) = (\ell, v)$ *such that* $v \models g$ *and* $v[Y := 0] \models Inv(\ell')$
2. *If* $(\sigma(h) = \delta \in \mathbb{R}_{\geq 0})$ *and the transition* $\rightarrow^{\delta}$ *is available in ls(h)*
   then $\sigma(h \rightarrow^{\delta} s) \in T$

where $h \rightarrow^{\delta} s$ *denotes the history obtained by adding the delay* δ *at the end of* h.

A strategy can return a discrete transition if its guard is satisfied and the resulting state satisfies the destination invariant (1). To respect the alternation between timed and discrete transitions, we require that a strategy which returns a finite delay $\delta \geq 0$ on a history returns a discrete transition after the delay (if the run did not stop by violating an invariant)(2).

A controller strategy σ_c and an environment strategy σ_e can be combined into a global strategy $\sigma_{(\sigma_c,\sigma_e)}$ as follows. If both players try to take a transition, we consider that the controller cannot guarantee his transition will be taken, thus the environment chooses. If one player decides on a discrete transition while the other decides to wait, the discrete transition is taken. If both players decide to wait, we wait for the smallest delay.

Definition 15 (global strategy). *Let σ_c be a controller strategy and σ_e an environment strategy. For all $h \in \mathcal{H}$, the global strategy $\sigma_{(\sigma_c,\sigma_e)}$ is defined by:*

- $\sigma_e(h) = t_u \in T_u \implies \sigma_{(\sigma_c,\sigma_e)}(h) = t_u$
- $\sigma_c(h) = t_c \in T_c \wedge \sigma_e(h) = \delta \geq 0 \implies \sigma_{(\sigma_c,\sigma_e)}(h) = t_c$
- $\sigma_c(h) = \delta \geq 0 \wedge \sigma_e(h) = \delta' \geq 0 \implies \sigma_{(\sigma_c,\sigma_e)}(h) = min(\delta, \delta')$

Example 4. Let us look at possible strategies in location preparing_coffee of the running example. The machine can choose serve_coffee while the user can select ask_sugar. If both want to do an action, the strategy chooses ask_sugar, thus giving priority to the user. If only one of them wants to take an action and the other waits, the action is taken. Hence, the machine can do serve_coffee if the user is waiting. This is the expected behavior of a coffee machine and its user.

The global strategy induces a unique run, introducing null delays between two discrete transitions to guarantee the alternation with timed transitions.

Definition 16 (run induced by a global strategy). *Let an initial state s_0 and a global strategy σ be given. The* run induced by strategy σ *is the unique $r_\sigma = s_0 s_1 s_2 \ldots$ obtained by:*

- *If i is even, the next transition is a timed transition :*
  - *If $\sigma(\langle s_0, \ldots, s_i\rangle) = t \in T$ a delay 0 is added: $s_i \rightarrow^0 s_{i+1} \rightarrow^t s_{i+2}$*
  - *If $\sigma(\langle s_0, \ldots, s_i\rangle)$ returns a delay $\delta \geq 0$ and there is a unique state s such that $s_i \rightarrow^\delta s$ (invariant not violated), then $s_{i+1} = s$.*
  - *Otherwise, the invariant is violated and the run ends.*
- *If i is odd, the next transition is a discrete transition. By the properties of a strategy $\sigma(\langle s_0, \ldots, s_i\rangle)$ returns a transition t such that there is a unique state s where $s_i \rightarrow^t s$. Then, $s_{i+1} = s$.*

Definition 17 (winning strategy). *A controller strategy σ_c is said to be* winning *from a state $s \in \mathbb{S}$ w.r.t. a reachability objective R if and only if all runs starting in s and* adhering to *σ_c are winning w.r.t. the objective. State s is said to be winning if there exists a winning strategy from it.*

Run r is adhering to *a controller strategy σ_c if there exists an environment strategy σ_e, such that $r = r_{\sigma_{(\sigma_c,\sigma_e)}}$.*

The question we now aim to answer is: Given a Parametric Timed Game G and a Reachability Objective R, is there a winning controller strategy from the initial state? The question depends on the value of the parameters. So, more precisely, we are interested in the question: *For which parameter valuations is the corresponding initial state winning?*

3 Solving the Game

In this section, we introduce necessary elements for solving a game. We first describe the symbolic state space on which the algorithm operates. Then we characterize the set of winning states as a nested fixed point.

3.1 Parametric Zone Graph

Since clock valuations assign real numbers, the timed transition system of a PTG has an uncountable number of states. Zones (cf. Def. 5) are a practical tool to regroup these states in more manageable sets. Recall that zones (like guards and invariants) are conjunctions of simple constraints on valuations, and can be viewed as sets of valuations. Our algorithms operate on symbolic states $\xi = (\ell, Z)$, which consist of a location and a zone. We require that $Z \subseteq \mathit{Inv}(\ell)$. For instance, the set of initial states of a PTG ξ_0 (cf. Def. 10) can be described by the symbolic state $(\ell_0, \mathit{Inv}(\ell_0) \wedge \bigwedge_{\mathrm{x} \in X} \mathrm{x} = 0)$.

In the notation, we identify a symbolic state (ℓ, Z) with its semantics as the set of concrete states: $\{(l, v) \mid v \vDash Z\} \subseteq \mathbb{S}$. We will write $\xi.\ell$ to denote the (common) location of a symbolic state ξ. Zones are closed under the following operations, which we extend to symbolic states:

- Intersection between sets
- Temporal successors: $\xi^{\nearrow} = \{s' \in \mathbb{S} \mid \exists s \in \xi, s \to^{\delta} s'\}$
- Temporal predecessors: $\xi^{\swarrow} = \{s' \in \mathbb{S} \mid \exists s \in \xi, s' \to^{\delta} s\}$
- Discrete successors: $\mathit{Succ}(t, \xi) = \{s' \in \mathbb{S} \mid \exists s \in \xi, s \to^{t} s'\}$
- Discrete predecessors: $\mathit{Pred}(t, \xi) = \{s' \in \mathbb{S} \mid \exists s \in \xi, s' \to^{t} s\}$
- Projection onto parameters: $\xi{\downarrow_P} = \{v_P \mid \exists v_X, \ell,\ (\ell, (v_X, v_P)) \in \xi\}$

These operations can be implemented by standard operations on convex polyhedra [7]. We also use union, set complement and set difference, which can return non-convex shapes. These are represented as unions of zones, still denoting sets of concrete states. All previous operations are extended to unions of zones.

Our algorithms operate on the Parametric Zone Graph (PZG). The PZG of a PTG is not guaranteed to be finite, so our algorithms are in fact semi-algorithms.

Definition 18 (Parametric Zone Graph). *Given a PTA of the form $G = (L, X, P, \mathit{Act}, T_c, T_u, \ell_0, \mathit{Inv})$, its* Parametric Zone Graph *is defined as the tuple $(\Xi, \xi_0^{\nearrow}, \Rightarrow_c^t, \Rightarrow_u^t)$, where $\Xi \subseteq 2^{\mathbb{S}}$; $\forall \xi, \xi' \in \Xi$ we have $\xi \Rightarrow_c^t \xi'$ if $\xi' = \mathit{Succ}(t, \xi)^{\nearrow}$ and $t \in T_c$; and $\xi \Rightarrow_u^t \xi'$ if $\xi' = \mathit{Succ}(t, \xi)^{\nearrow}$ and $t \in T_u$.*

3.2 Alternating Fixed Point Property

The algorithm works by alternating between exploring new states and back-propagating winning-state information from discovered winning states, starting from target states. The exploration relies on a fixed point property of the set $\mathit{Reach}(\xi_0)$, defined as all symbolic states in some run from an initial state in ξ_0:

Lemma 1 (from [15]). *$Reach(\xi_0)$ is the smallest set S containing $\xi_0^{\nearrow}$ such that $\forall t \in T,\ Succ(t, S)^{\nearrow} \subseteq S$.*

Similarly, the set of winning states $W(R)$ of the PTG with reachability objective R can be computed as a fixed point. Intuitively, we can win the game if we can take a temporal transition (without being diverted by an uncontrollable action leading us to a non-winning state) to a state that is either directly winning, or has a controllable transition to a winning state. We formalize this with three operators on sets of states. Let W be the set of winning states of the game.

We call $\mathit{WinningMoves}(S) = \{s \in \mathbb{S} \mid \exists t \in T_c, s' \in S,\ s \rightarrow^t s'\}$ the set of states that have access to a controllable action leading to S. When applied to W, it gives us the states with a controllable action to reach W, from which we have a winning strategy. $\mathit{WinningMoves}(S)$ is increasing in S. It can be computed using the previous operators as $\mathit{WinningMoves}(S) = \bigcup_{t_c \in T_c} \mathit{Pred}(t_c, S)$.

We call $\mathit{Uncontrollable}(S) = \{s \in \mathbb{S} \mid \exists t \in T_u, s' \notin S,\ s \rightarrow^t s'\}$ the set of states where an uncontrollable action leads to a state outside S. When applied to W, it gives us the states where the environment can derail us into a state outside W, from which we have no winning strategy. $\mathit{Uncontrollable}(S)$ is decreasing in S. It can be computed from the operators from the previous subsection by $\mathit{Uncontrollable}(S) = \bigcup_{t_u \in T_u} \mathit{Pred}(t_u, \mathbb{S} \backslash S)$.

Finally, we call $\mathit{SafePred}(S_1, S_2)$ the set of states that can reach S_1 by a temporal transition while avoiding S_2. Since it aims to be applied to reach winning moves while avoiding uncontrollable actions, if a state is in the intersection of S_1 and S_2, priority is given to the environment and the state is not considered safe. $\mathit{SafePred}(S_1, S_2) = \{s \in \mathbb{S} \mid \exists s' \in S_1, s \rightarrow^{\delta} s' \wedge \mathit{Cover}(s \rightarrow^{\delta} s') \cap S_2 = \emptyset\}$. $\mathit{SafePred}(S_1, S_2)$ is increasing in S_1 and decreasing in S_2.

Thanks to the work of Cassez et al. [10], *SafePred* can be computed between zones using the precedent operations and extended to union of zones:

Lemma 2 (from [10] for TG, [17] for PTG).

$$\mathit{SafePred}(S_1, S_2) = (S_1^{\swarrow} \backslash S_2^{\swarrow}) \cup ((S_1 \cap (S_2^{\swarrow})) \backslash S_2)^{\swarrow}$$

$$\mathit{SafePred}(\bigcup_i S_{1i}, \bigcup_j S_{2j}) = \bigcup_i (S_{1i}^{\swarrow} \cap \bigcap_j \mathit{SafePred}(S_{1i}), S_{2j})$$

We can now formulate the fixed point property followed by W.

Lemma 3. *$W(R)$ is the smallest set S containing R such that $\mathit{SafePred}(S \cup \mathit{WinningMoves}(S), \mathit{Uncontrollable}(S)) \subseteq S$.*

Proof. See [12, App. B]

4 Algorithm and Correctness

We can now introduce the algorithm for parameter synthesis for PTG. Alg. 1 explores the state space and creates a map of symbolic states connected by

Algorithm 1 For PTG $G = (L, X, P, Act, T_c, T_u, \ell_0, Inv)$ and reachability objective R, returns the set of all parameter valuations that win the game.

```
$Explored, WaitingUpdate, WaitingExplore \leftarrow \emptyset, \emptyset, \{\xi_0^{\nearrow}\}$    ▷ Symbolic state sets
$Win := \{\}$    ▷ Map from symbolic states to unions of zones
$Depends := \{\}$    ▷ Map from symbolic states to sets of symbolic states
$WinningParam := False$

function SOLVEPTG
    while $\neg$TERMINATE() do
        Choos  ith r EXPLORE() or UPDATE()
    return $WinningParam$

procedure EXPLORE
    $\xi \leftarrow extract(WaitingExplore)$
    for $t$ transition  rom $\xi$ : do
        $\xi' := Succ(t, \xi)^{\nearrow}$
        $Depends[\xi'] \leftarrow Depends[\xi'] \cup \{\xi\}$
        if $\xi'$ not in $Explored$ then
            $WaitingExplore \leftarrow WaitingExplore \cup \{\xi'\}$
    if $\xi.\ell \in R$ then
        $Win[\xi] \leftarrow \xi$
        $WaitingUpdate \leftarrow WaitingUpdate \cup Depends[\xi]$
    $WaitingUpdate \leftarrow WaitingUpdate \cup \{\xi\}$
    $Explored \leftarrow Explored \cup \{\xi\}$

procedure UPDATE
    $\xi \leftarrow extract(WaitingUpdate)$
    $Uncontrollable \leftarrow \bigcup_{\{(\xi', t) \mid \xi \Rightarrow_u^t \xi'\}} Pred(t, \xi' \setminus Win[\xi'])$
    $WinningMoves \leftarrow \bigcup_{\{(\xi', t) \mid \xi \Rightarrow_c^t \xi'\}} Pred(t, Win[\xi'])$
    $NewWin := SafePred(Win[\xi] \cup WinningMoves, Uncontrollable) \cap \xi$
    if $NewWin \not\subseteq Win[\xi]$ then
        $WaitingUpdate \leftarrow WaitingUpdate \cup Depends[\xi]$
        $Win[\xi] \leftarrow Win[\xi] \cup NewWin$
    $WinningParam \leftarrow (Win[\xi] \cap \xi_0)\downarrow_P$

function TERMINATE
    return $WaitingExplore = \emptyset \wedge WaitingUpdate = \emptyset$
```

a discrete transition through the operation $Succ(t_s, \_)^{\nearrow}$. Simultaneously, any newly found winning states in a symbolic state ξ, starting from the target locations, are propagated by marking the predecessors of ξ for an update. To update a symbolic state ξ, we compute $\mathit{SafePred}(\mathit{Win} \cup \mathit{WinningMoves}(\mathit{Win}), \mathit{Uncontrollable}(\mathit{Win}))$ within ξ and add the result to $\mathit{Win}[\xi]$. If new winning states are found, we mark ξ predecessors for an update.

The algorithm is non-deterministic: it does not describe how we choose between explore and update, and which symbolic state in the waiting lists to explore or update. These choices are left abstract on purpose, as optimization opportunities. A fair strategy would be to join *WaitingExplore* and *WaitingUpdate* in a single queue, whose head determines which operation to apply next. In our implementation, we prioritized back-propagation from *WaitingUpdate*.

4.1 Invariants and Correctness

Recall that the algorithm works on a zone graph. We are looking for subsets of winning states within symbolic states. The same state may appear in different symbolic states and may not have the same status in each instance. Therefore, the set W_{temp}, the winning states found by the algorithm so far, and W, the set of all winning states, also take into account the symbolic state considered. Formally, W consists of all pairs (ξ, s) where s is a winning state contained in the symbolic state ξ, and $W_{temp} = \bigcup_{\xi \in Explored} \{\xi\} \times Win[\xi]$.

Theorem 1. *These invariants hold during the execution of the algorithm:*

1. $\xi_0^{\nearrow} \in \mathit{Explored}$.
2. $\forall \xi \in \mathit{Explored}, t \in T, \xi'$, *if* $\xi \Rightarrow^t \xi'$, *then* $\xi' \in \mathit{WaitingExplore} \cup \mathit{Explored}$
3. $\forall \xi \in \mathit{Explored}$, *if* $\xi.l \in R$, $\{\xi\} \times \xi \subseteq W_{temp}$.
4. $W_{temp} \subseteq W$.
5. $\forall \xi \in \mathit{Explored}$, *we have either* $\xi \in \mathit{WaitingUpdate}$ *or* $\mathit{SafePred}(W_{temp} \cup \mathit{WinningMoves}(W_{temp}), \mathit{Uncontrollable}(W_{temp})) \cap (\{\xi\} \times \xi) \subseteq W_{temp}$.

Proof. See [12, App. B].

Invariant 4 guarantees that even if the algorithm times out the winning states found by the algorithm are indeed winning. Furthermore, if the algorithm terminates and the waiting lists are empty, we can apply the fixed point properties of $\mathit{Reach}(\xi_0)$ and W, and W_{temp} corresponds exactly to W over the explored symbolic states that cover $\mathit{Reach}(\xi_0)$.

Theorem 2. *Alg. 1 is correct (when it terminates).*

Proof. See [12, App. B].

Example 5. For the coffee machine, the PZG is only finite after applying inclusion subsumption (Sec. 5). However, even on this finite PZG, Alg. 1 does not terminate, but keeps reporting solutions at Line 29. In fact, it produces increasingly more general solutions, including $n\,\mathrm{p}_2 \geq \mathrm{p}_1$ (for any $n > 0$). If we bound these parameters in the initial specification, for instance $\mathrm{p}_2 \geq 1 \wedge \mathrm{p}_1 \leq 5$, our algorithm synthesizes the extra constraints $\mathrm{p}_1 + \mathrm{p}_2 \leq \mathrm{p}_4 \wedge \mathrm{p}_3 < \mathrm{p}_4$, as expected.

Theorem 3. *Provided the waiting lists are treated fairly, any explored winning state is discovered as winning by the algorithm eventually.*

Proof (sketch). For a classical TG, we can represent the underlying TA as a finite classical automaton (e.g. the region graph). On this automaton, we can define a (finite) turn-based reachability game equivalent to the initial TG. Hence, we can use the notion of *discrete distance to target* in a reachability game, corresponding to the smallest number of discrete transitions in which a controller can ensure to reach a target. This is equivalent to solving a *Min-Cost Reachability game* as studied in [9] where delay transitions have weight 0 and discrete transitions have weight 1. The game graph is finite and the weights non-negative, so the discrete distance to target of a winning state is positive and finite.

While the same construction is not necessarily finite in a PTG, any state of a PTG is a state $(\ell, (v_P, v_X))$ of the TG, where all parametric linear terms in guards have been replaced by their valuation through v_P. Therefore, this result extends to winning states of a PTG.

Let s be an explored winning state of the PTG and n its distance to target. We only need to explore states reachable in n discrete transitions from s. By invariant 2 from Thm. 1, when all states reachable in k discrete transitions are explored, all states reachable in $k+1$ discrete transitions are either already explored or in the exploration waiting list. Assuming fairness of the waiting lists, at some time they have all been explored. Therefore, at some time, all states reachable from s in n discrete steps have been explored.

When all states reachable in n discrete steps have been explored, all target states within are discovered. Those are states with distance to target 0. For $0 \leq k < n$, when all winning states reachable in less than $n-k$ discrete transitions from s and with a distance to target less than k are discovered, then all winning states reachable in less than $n-(k+1)$ discrete transitions from s and with a distance to target less than $k+1$ are discoverable by update. Using the invariant (5) of Thm. 1, those states are either already discovered as winning or they are in the update waiting list. Assuming fairness of the waiting lists, at some time they have all been discovered winning. Applying this recurrence until $k = n$, we get that there is a time where s is discovered winning.

We can guarantee: (1) All winning parameter valuations reported in Line 29 are correct, since the algorithm satisfies the invariants of Thm. 1. (2) Every winning parameter valuation will eventually be reported, provided the waiting lists are treated fairly. Hence, Alg. 1 is “sound and complete in the limit” [5].

5 Optimizations

We present four optimizations to the algorithm presented in Section 4. All of them adapt optimizations from previous works, three of them (coverage pruning, inclusion checking and losing state propagation) from Cassez et al. [10] and one of them (cumulative pruning) from André et al. [5]. We start by updating the exploration procedure to include the optimizations, as shown in Alg. 2.

Algorithm 2 Adding optimizations to the explore procedure

```
1: **procedure** EXPLORE
2:     $\xi \leftarrow extract(WaitingExplore)$
3:     **if** $\xi{\downarrow_P} \subseteq WinningParam$ **then**                          ▷ *Cumulative Pruning*
4:         **return**
5:     **if** $\xi.\ell \in R$ **then**
6:         $Win[\xi] \leftarrow \xi$
7:         $WaitingUpdate \leftarrow WaitingUpdate \cup Depends[\xi]$
8:     **if** $controller\_deadlock(\xi) \ \wedge \ Win[\xi] \neq \xi$ **then**          ▷ *Losing state propagation*
9:         $WaitingUpdate_L \leftarrow WaitingUpdate_L \cup Depends[\xi]$
10:    **if** $Win[\xi] = \xi \ \vee \ controller\_deadlock(\xi)$ **then**              ▷ *Coverage Pruning*
11:        **return**
12:    **for** $t$ transition  rom $\xi$ : **do**
13:        $\xi' := Succ(t, \xi)^{\nearrow}$
14:        **if** $\exists \xi'' \in Explored : \xi' \subseteq \xi''$ **then**                 ▷ *Inclusion check*
15:            $Depends[\xi''] \leftarrow Depends[\xi''] \cup \{\xi\}$
16:        **else**
17:            $Depends[\xi'] \leftarrow Depends[\xi'] \cup \{\xi\}$
18:            $WaitingExplore \leftarrow WaitingExplore \cup \{\xi'\}$
19:    $WaitingUpdate_W \leftarrow WaitingUpdate_W \cup \{\xi\}$
20:    $WaitingUpdate_L \leftarrow WaitingUpdate_L \cup \{\xi\}$                        ▷ *Losing state propagation*
21:    $Explored \leftarrow Explored \cup \{\xi\}$
```

5.1 Pruning

First, we present some pruning techniques, as these only require slight modifications in the exploration procedure. To this end, we introduce the notion of a *controller deadlock* state. A state is a *controller deadlock* state if it has no controllable transitions. We define it as the following predicate on symbolic states:

$$controller\_deadlock(\xi) = \forall t, \xi' : \text{ if } \xi \Rightarrow^{t} \xi' \text{ then } t \in T_u$$

Now, we introduce the two kinds of pruning:

- **Cumulative Pruning:** If the projected parameters of a zone in a new symbolic state are included in the current set of winning parameters, we can safely prune the successors of this state. Indeed, if the only possible parameters in the zone already are determined to be winning, no new winning parameter can be found by exploring the successors of this state. This check can be seen in Lines 3 and 4 of Alg. 2.
- **Coverage Pruning:** If a symbolic state is either winning or a controllable deadlock state, its successors can safely be pruned. Indeed, if the symbolic state is winning, we gain nothing from exploring further. Dually, a controller deadlock state can never become winning, since the controller has no action to do. This check can be seen in Lines 10 and 11 of Alg. 2.

5.2 Inclusion checking

Originally, checking if a symbolic state ξ' has been explored already is done by checking if $\xi' \in \mathit{Explored}$. The optimization by inclusion checking instead checks if $\exists \xi'' \in \mathit{Explored} : \xi' \subseteq \xi''$. If this is the case, the newly discovered symbolic state can safely be discarded since its superset has already been explored. Of course, the new dependency that ξ depends on ξ'' still must be added. This optimization is done in the exploration procedure (Lines 14 to 18 of Alg. 2).

5.3 Losing state propagation

Losing state propagation is inspired by Cassez et al. [10] for TG. The idea is that instead of only discovering and propagating winning states, we will now also do the same for losing states, starting from controller deadlock states. A map *Lose* will maintain the currently known losing states for a given symbolic state. Thus, each symbolic state ξ can now be partitioned into three:

- Winning: $Win[\xi]$,
- Losing: $Lose[\xi]$
- Unknown: $\xi \setminus (Win[\xi] \cup Lose[\xi])$.

To initially mark a state as losing, we use the controller deadlock predicate again while also making sure that the state is not winning, as shown in Alg. 2, Lines 8 and 9. On Lines 19 and 20, we partition the *WaitingUpdate* list into two lists for propagating winning and losing states respectively.

While pruning and inclusion checking only required the modification of the exploration procedure, the propagation of losing states influences all of the procedures of the original algorithm. We go through them now.

Update procedure. We create a new procedure for updating losing states, which can be seen in Alg. 3. As the dual of the original update procedure, it is almost identical. Instead of *Uncontrollable*, we compute *Controllable*, i.e. the union of zones where the controller can lead to a non-losing state. Similarly, instead of *WinningMoves*, we compute *LosingMoves* which is the set of states where the environment can lead to a losing state. We then compute *NewLosing* which is the set of states where the environment can lead us to a losing state while avoiding states where *only* controllable transitions are enabled ($\mathit{Controllable} \setminus \mathit{LosingMoves}$). Finally, we update $Lose[\xi]$ and $\mathit{WaitingUpdate}_L$ accordingly.

Terminate function. The terminate function is modified to allow for early termination if all possible information is already known, i.e. $\xi_0^{\nearrow} \setminus (Win[\xi_0^{\nearrow}] \cup Lose[\xi_0^{\nearrow}]) = \emptyset$. Indeed, if for all valuations we have determined that we either win or lose, the algorithm can safely terminate. This is shown in Alg. 4.

The final algorithm is then modified to include the new procedures and data structures introduced. As a result, in the main loop we now have to choose between three waiting lists instead of two: *WaitingExplore*, $\mathit{WaitingUpdate}_W$ and $\mathit{WaitingUpdate}_L$.

Algorithm 3 Adding new update procedure for losing state propagation

```
procedure UPDATE_L
    $\xi \leftarrow extract(WaitingUpdate_L)$
    $Controllable \leftarrow \bigcup_{\{(\xi',t) \mid \xi \Rightarrow_c^t \xi'\}} Pred(t, \xi' \setminus Lose[\xi'])$
    $LosingMoves \leftarrow \bigcup_{\{(\xi',t) \mid \xi \Rightarrow_u^t \xi'\}} Pred(t, Lose[\xi'])$
    $NewLosing := SafePred(Lose[\xi] \cup LosingMoves,\ Controllable \setminus LosingMoves) \cap \xi$
    if $Lose[\xi] \subsetneq NewLosing$ then
        $WaitingUpdate_L \leftarrow WaitingUpdate_L \cup Depends[\xi]$; $Lose[\xi] \leftarrow NewLosing$
```

Algorithm 4 New TERMINATE with early termination if initial zone is covered

```
function TERMINATE
    $isE\ \ pty \leftarrow WaitingExplore = \emptyset \wedge WaitingUpdate = \emptyset$
    $initialZoneCovered \leftarrow (\xi_0^{\nearrow}) \subseteq (Win[\xi_0^{\nearrow}] \cup Lose[\xi_0^{\nearrow}])$
    return $isE\ \ pty \vee initialZoneCovered$
```

6 Implementation and Experimental Evaluation

To evaluate the termination behavior and efficiency of the semi-algorithm and the optimizations, we implement them in the IMITATOR toolset and measure the performance on some realistic case studies.

6.1 Implementation

We have implemented our proposed algorithm and optimizations in the IMITATOR model checker [4], which features a wide repertoire of synthesis algorithms for PTA. We have extended its input language to PTG and added our PTG parameter synthesis algorithm, including the optimizations described in Sec. 5. The source code (in OCaml) is available on github[3].

In IMITATOR, the user specifies a model consisting of parameters, clocks and a network of parametric timed automata. The user can analyse the model using an analysis or synthesis query. IMITATOR selects the corresponding algorithm to use, after which it outputs the result of the query.

Our extension enables the user to specify edges in a PTA as (un)controllable, effectively turning it into a PTG. Along with this we add a new property `Win` and a corresponding algorithm `AlgoPTG`. In order to synthesize parameters for a PTG one must use `property := #synth Win(state_predicate)`, using a predicate to define which states are winning. Usually, this predicate is simply `accepting`, meaning that any state in an accepting location of the PTG is winning.

In Alg. 1 we left the choice between exploration and back-propagation to be non-deterministic. In the implementation back-propagation is prioritized over exploration whenever possible (i.e. when *WaitingUpdate* is non-empty). This seems to yield the fastest results in practice.

[3] `https://github.com/imitator-model-checker/imitator`, branch: d v lop

6.2 Experiment Design

We selected two large case studies, one PTA and one TG, and extended them to PTGs by adding (un)controllable actions, and clock parameters, respectively. An artifact containing instructions to run all the experiments is available online [11].

Production Cell. This case study [19] has two conveyor belts (1 / 2), a robot with two arms (A / B) and a press. Plates arrive at conveyor belt 1 and are taken to the press by robot arm A, where they are processed for some time. Robot arm B takes processed plates and removes them through conveyor belt 2.

We model systems with 1–4 plates in Imitator. In the goal location, every plate made it safely to conveyor belt 2. If two plates collide before they are picked up by arm A, the game is lost immediately. We assume that the rotation speed of the robot arm, the speed of the conveyor belt and the time to press are known constants. The aim is to synthesize a parameter `MINWAIT`, the minimum time interval between two plates arriving at the conveyor belt. The maximum time interval between two plates is fixed by an additional constant `MAXWAIT`.

Our PTG model is largely inspired by the TG model of Cassez et al. [10]. Besides adding parameters, we check for collisions between plates rather than defining a maximum waiting time frame. For 2–4 plates, we create a winning and a losing configuration of the constants; for 1 plate a collision is not possible. The losing configurations are created by setting `MAXWAIT` too small, which will deadlock the system for any value of `MINWAIT`.

The Imitator model for the 1-plate configuration can be seen in [12, App. C].

Bounded Retransmission Protocol. The BRP provides reliable communication over an unreliable channel. We create a PTG from a PTA model of the BRP [6], in turn based on a TA model [13], by making message loss uncontrollable.

In the BRP, a sender sends message frames to a receiver, tagged with an alternating bit, through a lossy channel. The receiver acknowledges all frames. If the sender does not receive an acknowledgement in time, it retransmits the message at most k times, after which the sender gives up. The goal location indicates the successful transmission of the message, or the abort by the sender.

Experimental Setup. All experiments were run on a single core of a computer with an Intel Core i5-10400F CPU @ 2.90GHz with 16GB of RAM running Ubuntu 20.04.6 LTS. For each implementation (basic, inclusion checking, cumulative pruning, coverage pruning, losing state propagation) we run the experiments 5 times and report the average time and state space size. A timeout of 2 hours is used.

6.3 Experimental Results

We present the results of the experiments in Table 1. We do not include the runs without optimizations as they all timed out. This indicates that inclusion checking is the most vital optimization and should always be enabled.

Table 1. Exp rim ntal r sults or diff r nt optimizations: inclusion ch ck (inc), cumulativ pruning (cm), cov rag pruning (cv), losing stat propagation (lp). Running tim in s conds (s) and numb r o symbolic stat s (siz). Gr n indicat s th b st r sults.

		inc		inc+cm		inc+cm+cv		inc+cm+cv+lp	
		Tim	Siz	Tim	Siz	Tim	Siz	Tim	Siz
plates	Production C ll								
1	Win	0.06s	86	0.06s	86	0.06s	86	0.08s	86
2	Win	7.19s	746	7.56s	746	6.60s	701	7.22s	701
	Los	1.43s	439	1.44s	439	2.03s	517	2.17s	517
3	Win	36.7s	1900	37.3s	1900	24.0s	1539	34.2s	1539
	Los	13.4s	1372	13.9s	1372	9.53s	1251	14.2s	1251
4	Win	4903s	10755	4750s	10755	2394s	9350	3522s	9350
	Los	34.8s	2605	35.6s	2605	21.6s	2372	153s	2372
Bound d R transmission Protocol									
		34.3s	1042	32.2s	1042	7.1s	612	7.5s	612

Indicated in green cells are the best results for each row. We can clearly see that coverage pruning has the biggest effect of all the optimizations in our experiments. Losing state propagation seems to not provide much benefit in these experiments, as the overhead overshadows any positive effect it might have had.

7 Conclusion

We provide the first implementation of parameter synthesis for Parametric Timed Games with reachability objectives, based on an on-the-fly algorithm [10,16]. It appears that without additional pruning heuristics, the algorithm cannot handle the case studies, Bounded Retransmission Protocol and Production Cell. Inclusion subsumption is a minimal requirement to achieve any result.

Contrary to previous algorithms for PTA [5] and TG [10], the parameter synthesis algorithm does not terminate, even if the parametric zone graph is finite. But we found that in the limit all parameter values will be enumerated.

We added additional pruning techniques (coverage pruning and cumulative pruning) to further reduce the search space. These techniques generally increased the speed. We also experimented with propagating losing states, but in our examples the overhead of checking and propagating losing states was not compensated by any pruning potential. Future work could study under which circumstances the propagation of losing states could be beneficial, but also strengthen the detection of (partially) losing states. Another venue for future work is to study other objectives, like safety games or liveness conditions.

Acknowledgment. We thank Étienne André for his help with integrating our algorithm in the IMITATOR tool set.

References

1. Raj v Alur and David L. Dill. A th ory o tim d automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Raj v Alur, Thomas A. H nzing r, and Mosh Y. Vardi. Param tric r al-tim r asoning. In *STOC*, pag s 592–601. ACM, 1993.
3. Éti nn Andr´. What's d cidabl about param tric tim d automata? *Int. J. Softw. Tools Technol. Transf.*, 21(2):203–219, 2019.
4. Éti nn Andr´. IMITATOR 3: Synth sis o timing param t rs b yond d cidability. In *CAV (1)*, volum 12759 o *Lecture Notes in Computer Science*, pag s 552–565. Spring r, 2021.
5. Éti nn Andr´, Jaim Arias, Laur P trucci, and Jaco van d Pol. It rativ bound d synth sis or ffici nt cycl d t ction in param tric tim d automata. In *TACAS*, LNCS 12651, pag s 311–329. Spring r, 2021.
6. Éti nn Andr´, Dylan Marinho, and Jaco van d Pol. A b nchmarks library or xt nd d param tric tim d automata. In *TAP@STAF*, volum 12740 o *Lecture Notes in Computer Science*, pag s 39–50. Spring r, 2021.
7. Rob rto Bagnara, Patricia M. Hill, and En a Zaffan lla. Th parma polyh dra library: Toward a compl t s t o num rical abstractions or th analysis and v rification o hardwar and so twar syst ms. *Sci. Comp. Prog.*, 72(1-2):3–21, 2008.
8. G rd B hrmann, Agn`s Cougnard, Al xandr David, Emmanu l Fl ury, Kim Guldstrand Lars n, and Didi r Lim . Uppaal-tiga: Tim or playing gam s! In *CAV*, volum 4590 o *Lecture Notes in Computer Science*, pag s 121–125. Spring r, 2007.
9. Thomas Brihay , Gill s G ra rts, Ax l Haddad, and B njamin Monm g . To r ach or not to r ach? Effici nt algorithms or total-payoff gam s. In *CONCUR*, volum 42 o *LIPIcs*, pag s 297–310. Schloss Dagstuhl - L ibniz-Z ntrum ür In ormatik, 2015.
10. Franck Cass z, Al xandr David, Emmanu l Fl ury, Kim G. Lars n, and Didi r Lim . Effici nt on-th -fly algorithms or th analysis o tim d gam s. In Martín Abadi and Luca d Al aro, ditors, *CONCUR 2005 – Concurrency Theory*, pag s 66–80, B rlin, H id lb rg, 2005. Spring r B rlin H id lb rg.
11. Mika l B. Dahls n-J ns n, Jaco van d Pol, Laur P trucci, and Baptist Fi v t. Arti act or "On-Th -Fly Algorithm or R achability in Param tric Tim d Gam s". Z nodo, 10.5281/z nodo.10046945, Octob r 2023.
12. Mika l Bisgaard Dahls n-J ns n, Baptist Fi v t, Laur P trucci, and Jaco van d Pol. On-th -fly algorithm or r achability in param tric tim d gam s (xt nd d v rsion). arXiv, 10.48550/arxiv.2401.11287, 2024.
13. P. R. D'Arg nio, J. P. Kato n, T. C. Ruys, and J. Tr tmans. Th bound d r transmission protocol must b on tim ! In Ed Brinksma, ditor, *Tools and Algorithms for the Construction and Analysis of Systems*, pag s 416–431, B rlin, H id lb rg, 1997. Spring r B rlin H id lb rg.
14. Thomas Hun , Judi Romijn, Mari¨ll Sto linga, and Frits W. Vaandrag r. Lin ar param tric mod l ch cking o tim d automata. *J. Log. Algebraic Methods Program.*, 52-53:183–220, 2002.
15. Al ksandra Jovanovic, S´basti n Faucou, Didi r Lim , and Olivi r H. Roux. R al-tim control with param tric tim d r achability gam s. In *IFAC WODES*, pag s 323–330. Els vi rs, 2012.
16. Al ksandra Jovanovic, Didi r Lim , and Olivi r H. Roux. Synth sis o bound d int g r param t rs or param tric tim d r achability gam s. In *ATVA*, volum 8172 o *Lecture Notes in Computer Science*, pag s 87–101. Spring r, 2013.

17. Aleksandra Jovanović, Didier Lime, and Olivier Henri Roux. A game approach to the parametric control of real-time systems. *International Journal of Control*, pages 1–12, January 2018.
18. Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. In Ernst W. Mayr and Claude Puech, editors, *STACS 95*, pages 229–242, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
19. Helmut Melcher and Klaus Winkelmann. Controller synthesis for the "production cell" case study. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, page 24–33, New York, NY, USA, 1998. ACM.

Rabin Games and Colourful Universal Trees[⋆][⋆⋆]

Rupak Majumdar[1], Irmak Sağlam[1], and K. S. Thejaswini[2,3](✉)

[1] Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
{rupak,isaglam}@mpi-sws.org
[2] Department of Computer Science, University of Warwick, Coventry, UK
[3] Institute of Science and Technology, Klosterneuburg, Austria
thejaswini.k.s@ista.ac.at

Abstract. We provide an algorithm to solve Rabin and Streett games over graphs with n vertices, m edges, and k colours that runs in $\tilde{O}\left(mn(k!)^{1+o(1)}\right)$ time and $O(nk\log k\log n)$ space, where $\tilde{O}$ hides poly-logarithmic factors. Our algorithm is an improvement by a super quadratic dependence on $k!$ from the currently best known run time of $O\left(mn^2(k!)^{2+o(1)}\right)$, obtained by converting a Rabin game into a parity game, while simultaneously improving its exponential space requirement.

Our main technical ingredient is a characterisation of progress measures for Rabin games using *colourful trees* and a combinatorial construction of succinctly-represented, *universal* colourful trees. Colourful universal trees are generalisations of universal trees used by Jurdziński and Lazić (2017) to solve parity games, as well as of Rabin progress measures of Klarlund and Kozen (1991). Our algorithm for Rabin games is a progress measure lifting algorithm where the lifting is performed on succinct, colourful, universal trees.

Keywords: Rabin games· Parity games· Colourful trees

1 Introduction

A *Rabin game* is a two-player infinite-duration game played on a directed, coloured graph, where each vertex has a finite set of *good* colours and a finite set of *bad* colours associated with it [29]. The two players Controller and Environment take turns to move a token along an edge to form a *play*, an infinite path in the graph. Such a play is winning for Controller if there is a colour that is a good colour for some vertex seen infinitely often along the path and is not a bad colour for any vertex seen infinitely often. Rabin games lie at the core of reactive synthesis for omega-regular specifications and efficient algorithms for Rabin games are of practical interest in synthesis tools.

[⋆] This work is a part of the project VAMOS that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreements No 101020093. Rupak Majumdar was partially supported by the DFG project 389792660 TRR 248—CPEC.

[⋆⋆] The full version of the paper [25] is available on arXiv at http://arxiv.org/abs/2401.07548

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 213–231, 2024.
https://doi.org/10.1007/978-3-031-57256-2_11

Rabin automata already appear in McNaughton's solution of Church's synthesis problem [4,26] and in Rabin's proof of the decidability of SnS [29], where it was first defined in the setting of infinite trees. To solve Church's synthesis problem for ω-regular specifications, represented by non-deterministic Büchi automata, there are two well-studied (polynomial-time equivalent) approaches: either reduce it to the emptiness problem for Rabin tree automata or solve a Rabin game.

Rabin conditions are also known to be suitable specifications for *general fairness constraints* [15]. Klarlund and Kozen [20] defined Rabin measures over graphs and applied them to prove program termination under a general fairness constraint. Indeed, the acceptance condition that defines *strong fairness*, i.e., if a given set of actions (edges) is enabled infinitely often (the source vertex is seen infinitely often), it is taken infinitely often, is naturally expressed by the complement of the Rabin condition, called the Streett condition [30].

Algorithmically, the problem of solving Rabin games was shown to be NP-complete by Emerson and Jutla [11,13] in the late 1980s. In the same paper, Emerson and Jutla, and independently, Pnueli and Rosner [28], gave an algorithm that takes time $O\big((nk)^{3k}\big)$ time, where n is the number of vertices of the game graph and k the number of colours.

Steady progress was made to solve Rabin games, and within a decade, Kupferman and Vardi [23] reduced the cubic dependence on n^k to a quadratic one by giving an algorithm to check non-emptiness in a Rabin tree automata in time $O\big(mn^{2k}k!\big)$. Later, Horn [16] gave a different solution to solve Streett games—and therefore Rabin games—with the same running time.

A lot of progress was simultaneously made on *parity games* [12], a special case of Rabin games where colours are assigned to each subset of states in a chain of subsets. Inspired by fixpoint evaluation algorithms [12] and the small progress measure algorithm [17] of Jurdziński for parity games, Piterman and Pnueli [27] gave a fast $O\big(mn^{k+1}kk!\big)$-time, $O(nk)$-space, algorithm for Rabin games. This algorithm used a concept of a measure to solve Rabin games.

The work of Piterman and Pnueli remained state-of-the-art for Rabin games until the quasi-polynomial breakthrough for parity games by Calude, Jain, Khoussainov, Li, and Stephan [1]. They gave a fixed parameter tractable algorithm (FPT) for Rabin games on k colours by converting it to a parity game and using the quasi-polynomial algorithm.

A Rabin game with n vertices, m edges, and k colours, can be reduced to a parity game over $N = nk^2k!$ vertices, $M = nk^2k!m$ edges, and $K = 2k+1$ colours [12]. By combining the reduction from Rabin to parity games and state-of-the-art algorithms for parity games [18,8,14,9] in a "space-efficient" manner, say of Jurdziński and Lazić [18], one can solve Rabin games in time $O\big(\max\{MN^{2.38}, 2^{O(K\log K)}\}\big)$, but exponential space (since the parity game is exponentially bigger).

On substitution of the values of M and N, the algorithm of Jurdziński and Lazić would take time at least proportional to $m(nk^2 \cdot k!)^{3.38}$ for games with n vertices, m edges and k colours. However, observe that the parity game obtained from a Rabin game is such that the number of vertices $N = nk^2k!$ is much larger than the number of colours $K = 2k+1$. Indeed, this results in $K \in o\big(\log(N)\big)$. For cases where the number of vertices of the resulting parity game is much larger than the number of priorities, say the number of colours $(2k+1)$ is $o\big(\log(N)\big)$—which is the case above as k grows—Jurdziński and Lazić also give an analysis of their algorithm that would solve Rabin games in time $O\big(nmk!^{2+o(1)}\big)$. Closely matching this are the run times in the work of Fearnley et al. [14] who provide, among other bounds, a quasi-bi-linear bound of $O\big(MN\mathfrak{a}(N)^{\log\log N}\big)$, where $\mathfrak{a}$ is the inverse-Ackermann function. In either case above, this best-known algorithm has at least a $(k!)^{2+o(1)}$ dependence in its run time, and takes space proportional to $(nk^2k!)\log(nk^2k!)$, which has a $k!$ dependence again.

Our Contribution. Our result breaks through the $2+o(1)$ barrier, while simultaneously using polynomial space, to give a fixed-parameter tractable algorithm for Rabin games. We show a new algorithm for Rabin games on graphs that runs in time $\tilde{O}(mn(k!)^{1+o(1)})$ time and $O(nk\log k\log n)$ space, for a game on n vertices, m edges, and k colours. Our algorithm improves the quadratic $(k!)^2$ dependence in the number of colours in the best current algorithms, while simultaneously using only polynomial space.

Our first technical contribution is a characterisation of winning states in Rabin games using "*colourful trees,*" by generalizing previous work on Rabin measures on graphs by Klarlund and Kozen [20]. Using our characterisation, we provide an algorithm to compute winning states and strategies as a fixed point of a lifting function over the lattice of functions from vertices of a game to nodes of a colourful tree.

Our second contribution is the construction of a *universal* colourful tree that embeds any colourful tree with a given number of leaves and fixed set of colours. Universal trees are found underlying all the quasi-polynomial algorithms for parity games [18,6,19,8,21]. Our construction uses the theory of universal trees developed for parity games, especially that of Jurdziński and Lazić [18]. From our construction of universal colourful trees, we can also naturally construct an instance of universal graphs for Rabin objectives, where the definition of universal graph is as introduced by Colcombet and Fijalkow. Although constructing universal graphs directly give us a lifting algorithm, for the sake of completeness, we also provide a lifting algorithm that uses our construction of colourful universal trees. Therefore, we show how to construct a small universal colourful tree (our upper bound is tight up to a polynomial factor) that can be succinctly encoded and efficiently navigated.

By applying the lifting algorithm to our succinct universal colourful tree, we get our time and space bounds.

Just as Piterman and Pnueli's result generalized ranking techniques and progress measures for parity games, we generalize the notion of measures [20] and universal trees [18] central to the fastest algorithms for parity games to obtain our algorithm.

2 Preliminaries

We use $\mathbb{N}$ to denote the set of all natural numbers $\{0,1,2,\ldots\}$. A directed graph consists of a finite set of vertices V along with a binary relation E over the set of vertices called the *edge set*. We write $u \to v$ to denote an edge $(u, v) \in E$. A finite (resp. infinite) *path* in a directed graph is a finite (resp. infinite) sequence of vertices such that a tuple formed by any two consecutive vertices in this sequence is an edge in E.

(c_0, C)-Colourful Ordered Trees. Let C be a finite set of colours and let $c_0 \notin C$ be a distinguished *root colour*. Informally, a (c_0, C)-colourful ordered tree with root colour c_0 is an ordered tree of height at most $|C| + 1$ whose root is associated with the colour $c_0 \notin C$, and whose every other node has a colour from C associated to it. As an exception, we allow some leaves to be left uncoloured, denoted by a "dummy colour" $\perp \notin C$. We also require that along any path from the root to a leaf, each node must have a different colour.

Formally, for a finite set C, we recursively define (c_0, C)-colourful trees

- if $C = \emptyset$, $(c_0, \langle\rangle)$ and $(c_0, \langle(\perp, \langle\rangle), \dots, (\perp, \langle\rangle)\rangle)$ are $(c_0, \emptyset)$-colourful trees.
- if $C \neq \emptyset$, we say $\mathcal{T}$ is (c_0, C)-colourful tree if it is either
  - a (c_0, C')-colourful tree rooted at c_0 for some $C' \subsetneq C$; or
  - $\mathcal{T} = (c_0, \langle \mathcal{T}_1, \dots, \mathcal{T}_\ell \rangle)$, and for all $i \in \{1, \dots, \ell\}$, either there is a $c_i \in C$ and $\mathcal{T}_i$ is a $(c_i, C \setminus \{c_i\})$-colourful ordered tree, or $\mathcal{T}_i = (\perp, \langle\rangle)$. Note that these c_i need not be different from one another.

We define the *concatenation* of a (c_0, C_1)-colourful tree $\mathcal{T}_1 = \left(c_0, \left\langle \mathcal{T}_1^1, \dots, \mathcal{T}_1^m \right\rangle\right)$ and a (c_0, C_2)-colourful tree $\mathcal{T}_2 = \left(c_0, \left\langle \mathcal{T}_2^1, \dots, \mathcal{T}_2^\ell \right\rangle\right)$ as the $(c_0, C_1 \cup C_2)$-colourful tree denoted by $\mathcal{T}_1 \cdot \mathcal{T}_2$ as $\left(c_0, \left\langle \mathcal{T}_1^1, \dots, \mathcal{T}_1^m, \mathcal{T}_2^1, \dots, \mathcal{T}_2^\ell \right\rangle\right)$. For a root colour c_0, a number $\ell \in \mathbb{N}$, and a (c, C)-colourful ordered tree $\mathcal{T}$, we denote $\mathcal{T}^\ell$ to be the tree with ℓ many copies of $\mathcal{T}$, $(c_0, \langle \mathcal{T}, \mathcal{T}, \dots, \mathcal{T} \rangle)$. When (c_0, C) is clear from context, we simply say "colourful tree."

Embedding Colourful Trees. Given a (c_0, C)-colourful tree $\mathcal{U}$ and a (c_0, C')-colourful tree $\mathcal{T}$, such that $C' \subseteq C$, we say $\mathcal{U}$ *embeds* $\mathcal{T}$ if $\mathcal{T} = (c_0, \langle\rangle)$, or $\mathcal{T} = (c_0, \langle \mathcal{T}_1, \dots, \mathcal{T}_\ell \rangle)$ and $\mathcal{U} = (c_0, \langle \mathcal{U}_1, \dots, \mathcal{U}_m \rangle)$ for some ℓ, m, and there is some increasing sequence of indices $1 \leq i_1 < i_2 < \dots < i_\ell \leq m$ such that $\mathcal{U}_{i_j}$ embeds $\mathcal{T}_j$ recursively. Notice both $\mathcal{U}_{i_j}$ and $\mathcal{T}_j$ must be rooted at the same colour, say c_j and both are $\left(c_j, C \setminus \{c_j\}\right)$-colourful and $\left(c_j, C' \setminus \{c_j\}\right)$-colourful trees respectively.

Labelled Colourful Trees. In what follows, we shall additionally label colourful trees with labels from some linearly ordered set. It is more convenient to define such labelled colourful trees as prefix-closed sets of sequences, using the isomorphism between a (recursively defined) tree and its set of paths.

Let $\mathbb{L}$ be a set of labels with a linear ordering $<_{\mathbb{L}} \subseteq \mathbb{L} \times \mathbb{L}$. An $\mathbb{L}$-labelled (c_0, C)-colourful tree is a prefix-closed set of sequences over $\mathbb{L} \times (C \cup \{\perp\})$ where $\mathbb{L} \times (C \cup \{\perp\})$ is the Cartesian product of $\mathbb{L}$ and $(C \cup \{\perp\})$.

Given an element $\tau_0 \in \mathbb{L} \times (C \cup \{\perp\})$ and a sequence $\left(\tau_1, \tau_2, \dots, \tau_j\right)$ in $(\mathbb{L} \times (C \cup \{\perp\}))^*$, we use $\odot$ to denote concatenation to the tuple, where we say $\tau_0 \odot \left(\tau_1, \tau_2, \dots \tau_j\right) = (\tau_0, \tau_1, \tau_2, \dots \tau_j)$. We extend this notation to sets of sequences $\mathcal{L}$, by also defining $\tau_0 \odot \mathcal{L} = \left\{(\tau_0, \tau_1, \tau_2, \dots \tau_j) \mid (\tau_1, \tau_2, \dots \tau_j) \in \mathcal{L}\right\}$.

We say a prefix-closed set $\mathcal{L} \subseteq (\mathbb{L} \times (C \cup \{\perp\}))^*$ is an $\mathbb{L}$-labelling of a (c_0, C)-colourful ordered tree $\mathcal{T}$

- if $\mathscr{T} = (c_0, \langle(\bot, \langle\rangle)^m\rangle)$, and $\mathscr{L}$ is the prefix closure of the set $\{(\alpha_1, \bot), \ldots, (\alpha_m, \bot)\}$ for some $\alpha_1 <_{\mathbb{L}} \alpha_2 <_{\mathbb{L}} \cdots <_{\mathbb{L}} \alpha_m \in \mathbb{L}$,
- if $\mathscr{T} = (c_0, \langle \mathscr{T}_1, \ldots, \mathscr{T}_m\rangle)$ then $\mathscr{L}$ is the prefix closure of the set

$$(\alpha_1, c_1) \odot \mathscr{L}_1 \cup (\alpha_2, c_2) \odot \mathscr{L}_2 \cup \cdots \cup (\alpha_m, c_m) \odot \mathscr{L}_m$$

  for some $\alpha_1 \leq_{\mathbb{L}} \alpha_2 \leq_{\mathbb{L}} \cdots \leq_{\mathbb{L}} \alpha_m$ in $\mathbb{L}$, such that for all j,
  - $\mathscr{T}_j$ is a $C \setminus \{c_j\}$-colourful tree rooted at c_j and $\mathscr{L}_j$ is an $\mathbb{L}$-labeling of $\mathscr{T}_j$,
  - $c_j \in C \cup \{\bot\}$, and
  - whenever $\alpha_j = \alpha_{j+1}$, we have $c_j \neq c_{j+1}$

Note that the root colour c_0 of $\mathscr{T}$ does not appear in $\mathscr{L}$; instead of tracking c_0 along with $\mathscr{L}$ explicitly, we implicitly assume the root colour of the tree $\mathscr{L}$ above is c_0.

We refer to elements of the prefix-closed set $\mathscr{L}$ of a labelled tree as *nodes* of the tree. For two nodes n_1 and n_2 in $\mathscr{L}$, we define *the greatest common ancestor*, written $\mathsf{GCA}(n_1, n_2)$, as the longest common prefix of n_1 and n_2. We define n_1 to be an *ancestor* of n_2 if $n_1 = \mathsf{GCA}(n_1, n_2)$. In particular, n_1 is a parent of n_2, written $n_1 = \text{parent}(n_2)$, if n_1 is the largest node other than n_2 such that $n_1 = \mathsf{GCA}(n_1, n_2)$; we then say n_2 is a child of n_1.

The colouring of a node is defined to be the last colour occurring in the sequence: For the empty sequence $()$, we define $\mathsf{colour}(()) = c_0$, and $\mathsf{colour}((\alpha_1, c_{i_1}), \ldots, (\alpha_j, c_{i_j})) = c_{i_j}$. Furthermore we define $\mathsf{ColourSet} : \mathscr{L} \to 2^{C \cup \{c_0\}}$, which maps a node to the set of colours seen from the root to that node: $\mathsf{ColourSet}(n) = \left\{\mathsf{colour}(n') \mid n' = \mathsf{GCA}(n', n)\right\} \setminus \{\bot\}$.

Ordering. We define an ordering $\prec_{\mathscr{L}}$ on $\mathscr{L}$. First, we fix some arbitrary linear order on the set C and set colour $\bot$ to be larger than all the colours in C in the ordering. We compare elements by extending the linear order $<_{\mathbb{L}}$ over $\mathbb{L}$ and an arbitrary fixed order $<$ over C to a linear order over the set $\mathbb{L} \times (C \cup \{\bot\})$ lexicographically as follows: for two elements in $\mathbb{L} \times (C \cup \{\bot\})$, we declare $(\alpha_1, c_1) < (\alpha_2, c_2)$ if either $\alpha_1 <_{\mathbb{L}} \alpha_2$ or $\alpha_1 = \alpha_2$ and $c_1 < c_2$.

For two nodes $n_1, n_2 \in \mathscr{L}$, we define $n_1 \prec_{\mathscr{L}} n_2$ if either n_1 is a strict prefix of n_2, or if n_1 is lexicographically smaller than n_2 when viewed as sequences over $\mathbb{L} \times (C \cup \{\bot\})$.

Due to space constraints, the missing proofs can be found in the full version of the paper.

Example 1. Figure 1 depicts a (●, {🟡, 🔴, 🔵})-colourful tree, where the nodes denoted by ○ represents uncoloured nodes. A fixed ordering on the set of colours 🟡 < 🔵 < 🔴 < ○, a labelling of this tree over $\mathbb{L} = \{1, 2\} \subseteq \mathbb{N}$ is the prefix closure of the following set {(1🟡, 1🔵, 1○), (1🟡, 1🔵, 2🔴, 1○), (1🟡, 1🔵, 2🔴, 2○), (1🟡, 1🔴, 1🔵), (1🟡, 1🔴, 2🔵, 2○), (1○), (2🔵, 2🟡), (2🔴, 1🔵, 1🟡, 1○), (2🔴, 1🔵, 2○)}. The ordering $\prec_{\mathscr{L}}$, (represented by $\prec$) on some nodes is as follows: () ≺ (1🟡) ≺ (1🟡, 1🔴) ≺ (1○) ≺ (2🔵, 2🟡). The ordering in the nodes of the tree in the figure decreases when we go from a child to a parent, or we go “left” in the tree, but otherwise increases.

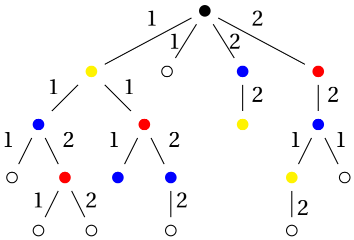


Fig. 1: A colourful tree.

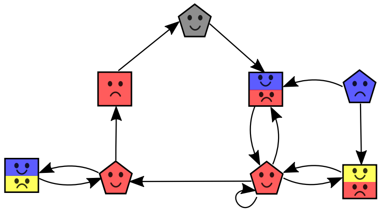


Fig. 2: A colourful Rabin graph $\mathcal{G}$ where all infinite paths satisfy the Rabin condition

3 Rabin measure and Colourful Decompositions

In this section, our aim is to understand the Rabin acceptance condition on graphs. We define such acceptance conditions and provide a local witness called a *Rabin measure* for graphs where all paths satisfy the Rabin condition.

A (c_0, C)-colourful *Rabin graph* $\mathcal{G}$ consists of (1) a directed graph (V, E), (2) a finite set C of *colours* and a special colour $c_0 \notin C$, and (3) for each vertex $v \in V$, a set of *good* colours $G_v \subseteq C \cup \{c_0\}$ for v and a set of *bad* colours $B_v \subseteq C$ for v. Observe that $c_0 \notin B_v$ for any v. We call each colour c in G_v a *good* colour for v, and each colour in B_v a *bad* colour for v.

We assume every vertex has some outgoing edge in the directed graph. An infinite path in $\mathcal{G}$ satisfies the *Rabin condition* if there is some colour c in $C \cup \{c_0\}$ such that c is a good colour for some v seen infinitely often along the path and c is not a bad colour for any v seen infinitely often along the path.

Example 2. Consider the (●, {🔴, 🔵, 🟡})-colourful Rabin game in Fig. 2. The colours that are in the good set of each vertex are represented with a smiley face in the same colour and those that are bad colours appear with a sad face. Although a vertex can have more than one colour assigned to it as a good colour (or a bad colour), we only consider at most one good and bad colour per vertex for this example. In our example, the leftmost vertex in the graph $\mathcal{G}$ in Fig. 2 has the singleton set {🔵} as the set of good colours and the set {🟡} as the set of bad colours. Similarly, the topmost vertex in Fig. 2 has the set {●} as the set of good colours and an empty set of bad colours. Observe that in the graph $\mathcal{G}$, any infinite path satisfies the Rabin condition. Indeed, for any infinite path there is some colour that is not a bad colour for any of the vertices that occur infinitely often and is a good colour for some vertex that occurs infinitely often. For example, if a path is such that all the vertices of $\mathcal{G}$ are visited infinitely often, then the colour ● is not a bad colour of any vertex and the same colour ● is a good colour of the topmost vertex.

As opposed to preexisting definition in literature of Rabin games that use Rabin pairs to represent the acceptance condition, we instead define two sets of colours associated to a vertex rather than a pair of subsets of vertices associated to a colour. This does not add more than a constant factor in terms of representation size.

A Measure for Rabin Graphs. We fix a (c_0, C)-colourful Rabin graph $\mathscr{G}$ with the underlying graph (V, E) with good colours for a vertex v denoted by G_v and the bad colours denoted by B_v. Let $\mathbb{L}$ be a linearly ordered set of labels, and let $\mathscr{L}$ be an $\mathbb{L}$ labelled (c_0, C) -coloured tree. We define $\mathscr{L}^\top = \mathscr{L} \cup \{\top\}$ by adjoining an element $\top$ to $\mathscr{L}$ and we extend the ordering $\prec_{\mathscr{L}}$ (denoted henceforth by $\prec$) to $\mathscr{L}^\top$, by declaring $t \prec \top$ for all $t \in \mathscr{L}$.

Consider a map $\mu : V \to \mathscr{L}^\top$. We call an edge $u \to v$ *consistent* with respect to μ, if either $\mu(u)$ is mapped to $\top$ or it satisfies the condition $(\mathsf{G}_\succ$ OR $\mathsf{G}_\downarrow)$ AND B; for $\mathsf{G}_\succ$, $\mathsf{G}_\downarrow$, and B defined below.

$(\mathsf{G}_\succ)$ $\mu(u) \succ \mu(v)$
$(\mathsf{G}_\downarrow)$ $\mathsf{GCA}(\mu(u), \mu(v)) = \mu(u)$ and $\mathsf{colour}(\mu(u)) \in G_u$.
(B) $\mathsf{ColourSet}(\mu(u)) \cap B_u = \emptyset$

In words, $\mathsf{G}_\succ$ conveys that the measure μ decreases along the edge $u \to v$ and $\mathsf{G}_\downarrow$ says that the measure can increase along an edge but only into a descendent node and only when the colour of the node that is currently mapped to is a good colour for u. The condition represented by B says that none of the colours assigned to any ancestor of u is a bad colour for it.

If the map μ is clear from the context, we call an edge or a vertex consistent without mentioning the mapping. We say the relation and function $\mathsf{GCA}(\cdot, \top)$ and $\mathsf{colour}(\top)$ are undefined, and the condition $\mathsf{G}_\downarrow$ or B are not satisfied when $\mu(v)$ is mapped to $\top$ and $\mu(u)$ is not mapped to $\top$.

We say the map μ is a (c_0, C)-colourful *Rabin measure* for a graph $\mathscr{G}$ if all edges in E are consistent with respect to μ. A mapping from the vertices of a Rabin graph to the nodes of a tree ensures that an infinite play corresponds to an infinite set of nodes in a tree. If a mapping is consistent, then such a mapping serves as a witness to the fact that an infinite path in the Rabin graph satisfies the Rabin condition.

Our definition is a modification of Klarlund and Kozen's [20] notion of Rabin measures, following recent approaches to faster algorithms for parity games [18,8].

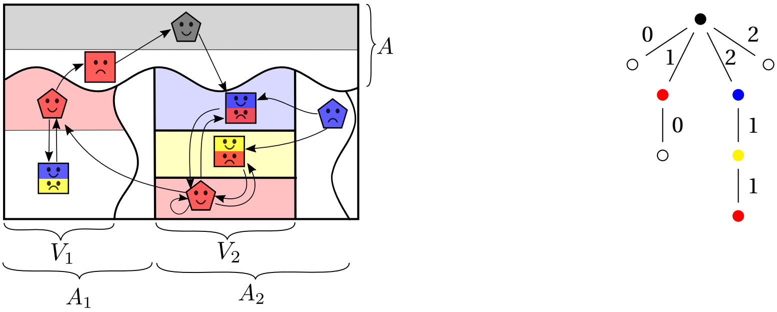
Colourful Decomposition. The Rabin measure, as with other progress measures, is based exclusively on local properties. Indeed, in the above case, we have a progress measure when each edge satisfies certain conditions. Before we show that Rabin measures capture winning sets of a graph, we define an intermediate structure, which we call *colourful decompositions*. These colourful decompositions of a Rabin graph highlight a recursive structure that captures the acceptance of all paths in a way which relates naturally to colourful trees. Colourful decompositions generalise attractor decompositions of parity games to Rabin games [7,19,8].

Consider a (c_0, C)-colourful Rabin graph $\mathscr{G}$. A (c_0, C)-colourful decomposition $\mathscr{D}$ of $\mathscr{G}$ is a recursive sub-division of vertices V of $\mathscr{G}$ into subsets of vertices defined as follows. If $C = \emptyset$, then we say $\mathscr{D} := \langle V \rangle$ is a (c_0, C)-colourful decomposition if and only if all infinite paths from all vertices in V visit a vertex v such that $c_0 \in G_v$. Else, if $C \neq \emptyset$ and if $|V| \geq 1$, and

$$\mathscr{D} = \left\langle A, (c_1, V_1, \mathscr{D}_1, A_1), \ldots, \left(c_j, V_j, \mathscr{D}_j, A_j\right)\right\rangle$$

satisfies the following conditions:

- A is the set of all vertices in V such that all infinite paths starting from A in $\mathcal{G}$ visit some vertex $v \in V$ such that $c_0 \in G_v$;
- Set $W_1 = V \setminus A$. For $i \in 1, \ldots, j$,
  - V_i is a set of vertices which has no path to $W_i \setminus V_i$ and $c_i \notin B_v$ for all $v \in V_i$;
  - $\mathcal{D}_i$ is a $(c_i, C \setminus \{c_i\})$-colourful decomposition of V_i.
  - A_i is the set of all vertices in W_i such that all infinite paths from A_i within W_i visits some vertex in V_i;
  - $W_{i+1} = W_i \setminus A_i$.
- $W_{j+1} = \emptyset$.



(a) A colourful decomposition of a Rabin graph $\mathcal{G}$ where all paths satisfy the Rabin condition

(b) A labelled colourful tree into which the graph $\mathcal{G}$ has a Rabin measure.

Fig. 3: A colourful decomposition and tree for Rabin measure

The crux of this section is Theorem 1 below which shows the equivalence between Rabin measure, the existence of a colourful decomposition and a Rabin graph where all paths satisfy the Rabin condition.

Theorem 1. *The following three statements are equivalent for a (c_0, C)-colourful Rabin graph $\mathcal{G}$.*

1. *All infinite paths in $\mathcal{G}$ satisfy the Rabin condition.*
2. *There is a (c_0, C)-colourful decomposition $\mathcal{D}$ of the vertices of $\mathcal{G}$.*
3. *There is an $\mathbb{L}$-labelled (c_0, C)-colourful Rabin measure for $\mathcal{G}$, where no vertex is mapped to $\top$ for some linearly ordered infinite set $\mathbb{L}$.*

The theorem above is proved by showing $1 \implies 2$, $2 \implies 3$, and finally $3 \implies 1$.

Proof Sketch $1 \implies 2$. If C is empty, then the decomposition is $\mathcal{D} = \langle V \rangle$ for a $(c_0, \emptyset)$-colourful graph where all paths satisfy the Rabin condition. If C is not empty, we first remove all vertices A from $\mathcal{G}$ that can visit a vertex for which c_0 is a good colour. In the decomposition of the graph into Strongly Connected Components (SSCs) induced by $V \setminus A$, each infinite path satisfies the Rabin condition, and therefore especially the infinite path which consists of all the vertices of some bottom SCC, V_1. Hence, there must be one colour c that is not a bad colour for any vertex and is a good colour for at least some of the vertices V_1. One can therefore inductively construct a $(c, C \setminus \{c\})$-colourful decomposition $\mathscr{D}_1$ for the vertices of V_1. Later, in the graph $\mathscr{G}$ without the vertices of A and V_1 and all vertices A_1 from which all paths lead to V_1, we again get an other graph where all infinite paths satisfy the Rabin condition. This graph, again by induction has a (c_0, C)-colourful Rabin decomposition $\mathscr{D}'$. We finally 'glue' together $\langle A, (c, V_1, \mathscr{D}_1, A_1)\rangle$ and $\mathscr{D}'$ obtained above.

Proof Sketch 2 $\Longrightarrow$ 3. The proof follows a recursive construction of an $\mathbb{L}$-labelled (c_0, C)-colourful tree where the recursion is based on the structure of the decomposition. An example of how such a mapping to a tree is obtained from a picture is exemplified in Fig. 3. The decomposition $\mathscr{D}$ of the game $\mathscr{G}$ is (A, (🔴, $V_1, \mathscr{D}_1, A_1$), (🔵, $V_2, \mathscr{D}_2, A_2$)). Some of the sets of the decomposition are indicated in Fig. 3a. The measure obtained from the decomposition into the given tree is intuitive. For example, the measure obtained from the given decomposition of the game $\mathscr{G}$ is such that the vertex for which the colour ● is a good colour is mapped to the root of the tree. Similarly, this measure maps the vertex in V_1 for which the colour 🔴 is a good colour to the node 1🔴 in the tree. The only vertex in $A_2 \setminus V_2$ is mapped to the node 2○.

Proof Sketch 3 $\Longrightarrow$ 1. If there is a Rabin measure, each edge in the infinite path satisfies B, as well as $\mathsf{G}_{\succ}$ or $\mathsf{G}_{\downarrow}$. For such an infinite path, we consider the infinite sequence of nodes of the colourful tree, obtained by taking the image of μ on the run. In this sequence obtained, consider the smallest node of the tree t that is visited infinitely often, and let $c = \mathsf{colour}(t)$. We show that t is a common ancestor for all elements of the sequence after a finite prefix. Since all edges satisfy $\mathsf{G}_{\succ}$ or $\mathsf{G}_{\downarrow}$, c is a colour such that $c \in G_v$ for some v visited infinitely often. As all edges satisfy B, we have $c \notin B_v$ for all vertices v in the run after some finite prefix.

Remark 1. A similar statement to the equivalence of item 1 and item 2 has been proved in the work of Klarlund and Kozen [20], however, a reader familiar with their work might have observed some differences in the definition of a measure as well as a colourful tree. Our definition of colourful trees is more restrictive than theirs. For instance, colourful trees in the work of Klarlund and Kozen have no restrictions about the colours along a path in a tree, i.e, in their definition, the trees can have the same colour along a path, and in fact only a partial colouring is required. However, an examination of their proof reveals that in the direction of the proof where they construct a Rabin measure, they inherently use a construction which produces a mapping into colourful trees as we have defined and therefore, it is enough to only consider such trees. We make this explicit and also prove Theorem 1 in the appendix to suit our situation.

4 A lifting algorithm

In this section, we define Rabin game formally and first show how such Rabin games also have a notion of a Rabin measure.

Inspired by the breakthrough algorithms to solve parity games, Colcombet, Fijalkow, Gawrychowski, and Ohlmann [5] proposed a formalism for algorithms that solve games where one player has a positional strategy. They showed that if there is a special kind of graph homomorphism into a graph with a total order on its vertices, then one can obtain a lifting algorithm for such games. From their work [5, Theorem 3.1] combined with Theorem 4, we can show that Rabin measures defined in our previous section can also be used to provide a lifting algorithm for Rabin games. However, to make this work self-contained and to provide an explicit space-efficient algorithm using our non-trivial totally ordered set, we show how such lifting is performed step-by-step in this section. We believe our following section would help future implementation of such algorithms.

A (c_0, C)-colourful *Rabin game* $\mathfrak{G}$ consists of an *arena* which is a (c_0, C)-colourful Rabin graph $\mathscr{G}$ with vertices V, a *start vertex* $v_0 \in V$, and a partition of V into V_c and V_e, the vertices of two players, whom we call Controller and Environment, respectively.

A *positional strategy* σ for Controller over the game graph is a subset of edges outgoing from each of Controller's set of vertices V_c. We denote the graph restricted to a strategy σ for the Controller by $\mathscr{G}|_\sigma$ and it is defined as the Rabin graph over the same vertex set with a new edge relation which contains exactly the edges in σ along with all the edges from all vertices belonging to Environment.

The Rabin game $\mathfrak{G}$ is *winning* for the Controller if and only if there exists a positional strategy σ for the Controller where, all infinite paths starting from v_0 in $\mathscr{G}|_\sigma$ satisfy the Rabin condition. We describe an algorithm that identifies whether a Rabin game $\mathfrak{G}$ is winning for Controller, using Rabin measures on graphs.

Remark 2. We only consider strategies of the Controller that are positional, but this is enough from the results of Emerson and Jutla [13], which shows that the Controller always has a positional winning strategy in Rabin games if there is any winning strategy at all.

Consistency in games. Consider a (c_0, C)-colourful Rabin game $\mathfrak{G}$. Let μ be a function from V, the vertices of the game graph to an $\mathbb{L}$-labelled (c_0, C)-colourful tree $\mathscr{L}$. We simply extend the definition of consistency from graphs to games by defining a vertex to be *consistent* with respect to μ in $\mathfrak{G}$ if either it belongs to the Environment and all outgoing edges from it are consistent in $\mathscr{G}$ or if it belongs to the Controller and there is at least one outgoing edge that is consistent in $\mathscr{G}$. A map μ from V to a $\mathscr{L}^\top$ is a *Rabin measure* for a (c_0, C)-colourful Rabin game $\mathfrak{G}$ if and only if *all* vertices are consistent with respect to μ.

An overview of the algorithm. We describe an algorithm that identifies whether a Rabin game $\mathfrak{G}$ is winning for Controller, using Rabin measures defined earlier for Rabin graphs. The basic principle in the algorithm is that given a colourful tree, the algorithm finds if there is a Rabin measure that maps vertices of the game into nodes of that tree. The algorithm does so by starting with the smallest map (all vertices are mapped to the root of this tree) and then at each step, if a vertex is not consistent, increase the value of this map just at this vertex which is not consistent. The value is modified (increased) until either all vertices are consistent, or the value cannot be increased anymore.

Toward our goal of formally defining this algorithm, we define monotonic, inflationary operators on the set of all maps from vertices of a game to a tree such that the simultaneous fixpoints of these operators exactly correspond to a Rabin measure.

Consider a Rabin measure μ which is a function mapping the vertices V of a (c_0, C)-colourful Rabin game $\mathfrak{G}$ into an $\mathbb{L}$-labelled (c_0, C)-colourful tree $\mathscr{L}$. We define a function lift_μ, which maps edges E of the arena of the game to $\mathscr{L}^\top$. For an edge $u \to v$ of $\mathscr{G}$, we define $\text{lift}_\mu(u, v)$ to be the smallest element t in $\mathscr{L}^\top$ such that (1) $t \succeq \mu(u)$ and (2) edge $u \to v$ is consistent with respect to the mapping $\mu[u := t]$, where we use the notation $\mu[u := t]$ to indicate the mapping μ' where $\mu'(x) = \mu(x)$ if $x \neq u$ and $\mu'(x) = t$ if $x = u$.

For each vertex v, we define an operator Lift_v on the lattice of all maps from V to $\mathscr{L}^\top$. The operator Lift_v only modifies an input map μ at v and nowhere else. We define

$$\text{Lift}_v(\mu)(u) = \begin{cases} \mu(u) & \text{for } u \neq v \\ \min_{(v,w)\in E} \left\{\text{lift}_\mu(v, w)\right\} & \text{if } u = v \in V_c \\ \max_{(v,w)\in E} \left\{\text{lift}_\mu(v, w)\right\} & \text{if } u = v \in V_e \end{cases}$$

Proposition 1. *The function* Lift_v *is monotonic for each* v.

The above proposition follows from our definition of the Lift_v function. Now that we know that each Lift_v is inflationary and monotonic. Therefore, the simultaneous least fixpoint of Lift_v on the map μ, which maps all vertices to the root of $\mathscr{L}$ exists (from the Knaster-Tarski theorem [31]). We can moreover state the following proposition that such fixpoints correspond to the Rabin measures, which almost follows from our definitions.

Proposition 2. *For a* (c_0, C)*-colourful Rabin game* $\mathfrak{G}$ *where the vertex set is* V *and a fixed* $\mathbb{L}$*-labelled* (c_0, C)*-colourful tree* $\mathscr{L}$*,*

- *any simultaneous fixpoint of the set of functions* Lift_v *for all* $v \in V$ *is a Rabin measure;*
- *any Rabin measure is a simultaneous fixpoint of* Lift_v *for all* $v \in V$.

Our algorithm, like any other progress-measure algorithm, computes a fixpoint and is described as follows. The correctness follows from Propositions 1 and 2.

Algorithm 1 The lifting algorithm on game (c_0, C)-colourful Rabin game $\mathfrak{G}$ with vertices V to tree $\mathscr{L}$

```
Require: For each $v \in V$, $\mu(v)$ is declared to be root in $\mathscr{L}$
while there is some vertex $v$ that is inconsistent with respect to $\mu$. do
    $\mu \leftarrow \text{Lift}_v(\mu)$.
end while
return $\mu$
```

Remark 3. If there is a (c_0, C)-colourful Rabin game $\mathfrak{G}$ and an $\mathbb{L}$-labelled (c_0, C)-colourful tree $\mathscr{L}'$, such that there is a Rabin measure μ' from V to $\mathscr{L}'$, and $\mathscr{L}$ embeds $\mathscr{L}'$, then there is also a Rabin measure μ to $\mathscr{L}$ such that all the elements that are not mapped to $\top$ by μ' are still not mapped to $\top$ by μ. This map is obtained by composing μ' with the embedding of $\mathscr{L}'$ into $\mathscr{L}$.

Runtime. For a finer analysis of the runtime, we need to understand the size of the lattice where the lifting algorithm takes place. In this section however, we restrict ourselves to analysing the runtime of our algorithm for a fixed $\mathscr{L}$. We write $|\mathscr{L}|$ to represent the number of nodes in the labelled tree $\mathscr{L}$. We write n to denote the number of vertices in a Rabin game, m to denote the number of edges, and $k = |C \cup \{c_0\}|$ to denote the number of colours.

Lemma 1. *Given a mapping from the vertices of a (c_0, C)-colourful Rabin game $\mathfrak{G}$ to an $\mathbb{L}$-labelled (c_0, C)-colourful tree $\mathscr{L}$, the value of* $\mathrm{Lift}_v(\mu)(v)$ *can be computed in time $O\left(\deg(v) \cdot T_{\mathsf{next}}\right)$, where* $\deg(v)$ *is the degree (number of outgoing edges) of v and T_{next} is defined as the maximum of the time taken to*

- *make a linear pass on a node in $\mathscr{L}$ (assuming the node is represented by a sequence of elements of $\mathbb{L} \times C$),*
- *compute the next node in $\mathscr{L}$, and*
- *find the next node that uses colours only from $C' \cup \{\perp\}$ for a given node $t \in \mathscr{L}$ and subset of colours $C' \subseteq C$ such that* $\mathsf{colour}(t) \in C'$.

Proof (sketch). The proof of the above lemma reduces to arguing carefully that using these above items as subroutines, we can find the node larger than $\mathrm{Lift}_v(\mu)(v)$ in the given tree that satisfies the conditions B along with at least one of $\mathsf{G}_{>}$ or $\mathsf{G}_{\downarrow}$. To satisfy condition B, we need to find the next node that does not use a bad colour of v (using item 3 of the lemma) and then find its first child larger than the current node value of $\mu(v)$ that either satisfies $\mathsf{G}_{>}$ or $\mathsf{G}_{\downarrow}$using the operations described above. The exact details of how the last step are done are provided in the full version of the paper.

Theorem 2. *For a (c_0, C)-colourful Rabin game $\mathfrak{G}$ with n vertices and m edges, and an $\mathbb{L}$-labelled (c_0, C)-colourful tree $\mathscr{L}$, Algorithm 1 on $(\mathfrak{G}, \mathscr{L})$ returns the smallest Rabin measure to $\mathscr{L}^{\top}$ in time $O(m|\mathscr{L}|T_{\mathsf{next}})$ where T_{next} is as defined in Lemma 1 and $|\mathscr{L}|$ denotes the number of nodes in $\mathscr{L}$.*

Proof. First, we observe that performing Lift_v on the mapping strictly increases the mapping for a vertex that is not consistent. Each operation of Lift_v also calls at most $\deg(v)$ many calls of $\mathrm{lift}_\mu(v, u)$ for some edge $v \to u$. Suppose each operation $\mathrm{lift}_\mu(v)$ takes time T_{next}, to find the value of $\mathrm{Lift}_v(\mu)(v)$ takes time at most $\deg(v) \cdot T_{\mathsf{next}}$. Since each non-trivial application of Lift_v strictly increases the value that v is mapped to, it can be called at most as many times as the number of nodes in tree $\mathscr{L}$, this ensures that the time taken is

$$\sum_{v \in V} \deg(v)|\mathscr{L}|(T_{\mathsf{next}}) \in O(m|\mathscr{L}|T_{\mathsf{next}})$$

where m denotes the number of edges.

5 Small Colourful Universal Trees

In the previous section, we concluded that our algorithm identifies correctly the smallest Rabin measure into a fixed labelled colourful tree $\mathscr{L}$. However, from Theorem 1, *there exists* a Rabin measure into an $\mathbb{L}$-labelled (c_0, C)-colourful tree $\mathscr{L}$ with at most n leaves. Observe that we only need to consider n leaves of $\mathscr{L}$ which correspond exactly to the image of the Rabin measure. Therefore, for a Rabin game, there is a Rabin measure into $\mathscr{L}^{\top}$ where all start vertices from which the game is winning for Controller are not mapped to $\top$. In order for the algorithm to successfully determine the winner of all (c_0, C)-colourful Rabin games with n vertices, we need to ensure that the tree $\mathscr{L}$ used in Algorithm 1 would be able to embed all (c_0, C)-colourful trees with n leaves. Since the runtime is linearly dependent on the tree size, smaller trees that satisfy the above property are desirable.

We now show that we can obtain colourful *universal* trees, i.e., colourful trees that are large enough to embed *any* (c_0, C)-colourful $\mathscr{L}$ with n-nodes. We also modify the technique of succinct universal trees of Jurdziński and Lazić [18] to encode each node of these colourful universal trees using polynomial space, which helps navigate these labelled colourful trees efficiently.

Colourful universal trees. A (c_0, C)-colourful tree $\mathscr{U}$ is n*-universal*, if it embeds *any* (c_0, C)-colourful tree $\mathscr{T}$ with at most n leaves. We henceforth assume that the set C consists exactly of the colours $c_1, \ldots, c_h$, with the fixed ordering $c_1 < c_2 < \cdots < c_h$ on the colours, and use k to denote $h+1$.

A näive attempt at constructing an n-universal (c_0, C)-colourful tree could be to take all possible (c_0, C)-colourful trees with at most n leaves with the root colour c_0 and concatenate them. Clearly, such an n-universal (c_0, C)-colourful tree can be created as there are only finitely many such trees up to isomorphism (for a fixed C and n). But of course, this tree is not only large, but can also be difficult to navigate. A more tractable attempt is to construct a tree that branches $n \cdot h$ many times at the root. The subtrees at the root that occur from this $n \cdot h$ branching have n repetitions of the h colours $c_1, c_2, \ldots, c_h$, in that order. Each of the children in-turn branch into $n \cdot (h-1)$ many times similarly, thus creating a tree of size bounded by $n^h h!$. We claim that indeed such a tree was exactly the one underlying the algorithm of Piterman and Pnueli [27], which led to their $O(mn^{k+1} kk!)$ algorithm.

Below, we give a more involved construction of a significantly smaller universal tree. In our construction, we inductively describe such a (c_0, C)-colourful n-universal tree, which we call $\mathscr{U}^{\ell}_{C}$, for a fixed $n \leq 2^{\ell}$.

- if $C = \emptyset$, then there is exactly one tree to embed, and therefore

$$\mathscr{U}^{\ell}_{(c_0, C)} = \left(c_0, \left\langle (\bot, \langle\rangle)^{2^{\ell}} \right\rangle\right)$$

- if $\ell = 0$, then the tree to be embedded has exactly one leaf and therefore, for each colour c_i in C, we have a child of colour c_i which hosts a subtree whose colour at the root is c_i. This is defined inductively as

$$\mathscr{U}^{0}_{(c_0, C)} = \left(c_0, \left\langle \mathscr{U}^{0}_{(c_1, C_1)}, \ldots, \mathscr{U}^{0}_{(c_h, C_h)}, (\bot, \langle\rangle) \right\rangle\right)$$

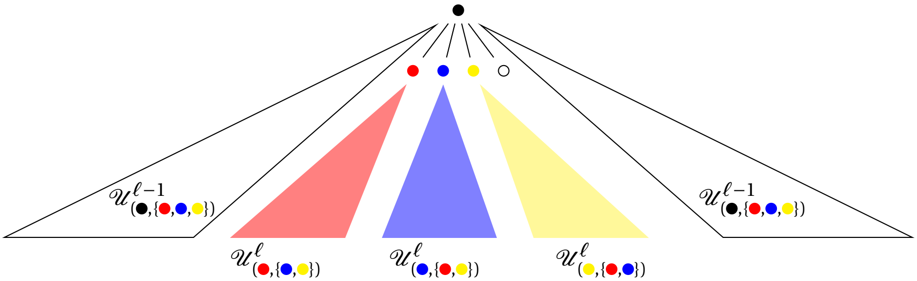

$\mathcal{U}^{\ell-1}_{(●,\{●,●,●\})}$
$\mathcal{U}^{\ell-1}_{(●,\{●,●,●\})}$
$\mathcal{U}^{\ell}_{(●,\{●,●\})}$
$\mathcal{U}^{\ell}_{(●,\{●,●\})}$
$\mathcal{U}^{\ell}_{(●,\{●,●\})}$

Fig. 4: Inductive construction of a smaller colourful n-universal tree

where C_i is $C \setminus \{c_i\}$.

- if $C \neq \emptyset$ and $\ell > 0$, then we define the coloured tree to be two copies of an $n/2$-universal tree, and h many copies of the n-universal tree where one colour is dropped each time. More formally,

$$\mathcal{U}^{\ell}_{(c_0,C)} = \mathcal{U}^{\ell-1}_{(c_0,C)} \cdot \left(c_0, \left\langle \mathcal{U}^{\ell}_{(c_1,C_1)}, \dots, \mathcal{U}^{\ell}_{(c_h,C_h)}, (\bot, \langle\rangle) \right\rangle\right) \cdot \mathcal{U}^{\ell-1}_{(c_0,C)}.$$

In Fig. 4, we demonstrate how the inductive construction is done if $c_0 = ●$ and the set of colours is $C = \{●,●,●\}$. To the left and right are the $(●, C)$-colorful $n/2$-universal trees and between them, there are $|C|$ many n-universal trees each of which uses one fewer colour and one node with just the dummy colour represented there by ○.

Theorem 3. *For $C \neq \emptyset$, and $k = |C| + 1$, $\mathcal{U}^{\ell}_{(c_0,C)}$ constructed is a (c_0, C)-colourful n-universal tree with at most*

$$nk!\left(\min\left\{n2^k, \binom{\ell+k}{k-1}\right\}\right)$$

many leaves, where $\ell = \lceil \log n \rceil$.

Proof. Firstly, we need to show that $\mathcal{U}^{\ell}_{(c_0,C)}$ is (c_0, C)-colourful n-universal tree. Then, we prove using induction that $\mathcal{U}^{\ell}_{(c_0,C)}$ has at most $2^k \cdot k! \cdot 4^{\ell}$ leaves and later show that it also has at most $\binom{\ell+k}{k-1} \cdot 2^{\ell} \cdot k!$ leaves, leading to the proof of our theorem.

In fact, we have a lower bound for n-universal (c_0, C)-colourful trees, which is within a polynomial factor of the upper bound.

It is known from the work of Calude et al., as well as from Casares et al. [1,2] that there are no algorithms that solve Rabin games in time $n^{O(1)} \cdot 2^{o(k \log k)}$. But observe that this does not exclude algorithms which is dependant on $k!$ by only a constant smaller than 1 in the exponent. We have improved the current state-of-the art from $2 + o(1)$ to $1 + o(1)$ in the exponent. A natural question to ask would be if the $k!$ component can be reduced further. We show below that we cannot improve our running time much further using our techniques.

Lemma 2 (Lower bound). *Any n-universal (c_0, C)-colourful tree must have size at least $\binom{\ell+k-1}{\ell}(k-1)!$ where $k = |C| + 1$ and $\ell = \lfloor \log n \rfloor$.*

Proof. Fix a permutation $c_{i_1}, \dots, c_{i_h}$ of the colours in C and consider any tree with n leaves where the order of colours from the root to the leaf is exactly the same as the given permutation. Moreover, we assume that the leaves all have the same depth from the root. This tree must have size at least the size of a 2^ℓ-universal tree (defined for ordered tree without colours). Such universal trees have size at least $\binom{\ell+h-1}{h-1}$ in the work of Czerwiński et. al [6]. For each choice of permutation, the universal tree restricted to that permutation must have size $\binom{\ell+h-1}{h-1}$. Furthermore, two universal trees obtained by fixing different permutations cannot share a leaf since distinct colours are assigned to some ancestor of such leafs. Therefore, we obtain a lower bound of $\binom{\ell+h-1}{h-1}h!$ on the size of any (c_0, C)-colourful n-universal trees.

This immediately gives us the bound $\binom{\ell+k-2}{\ell}(k-1)!$ for $k = |C| + 1$. Our lower bound also matches one of the upper bounds of our construction up to a polynomial factor in n and k.

Labelling Colourful Universal Trees. Here, we give a labelling of a universal colourful tree described in the previous section by giving an $\mathbb{W}$-labelling of any (c_0, C)-colourful tree where the set $\mathbb{W} = \{0,1\}^*$. We let ε denote the empty string in $\{0,1\}^*$. We define the ordering on $\{0,1\}^*$ as follows, similar to the succinct encoding of ordered trees [18]: $0 < \varepsilon < 1$ and for $b_1, b_2 \in \{0,1\}$ we have $b_1 \cdot w_1 < b_2 \cdot w_2$ if and only if $b_1 < b_2$ or $b_1 = b_2$ and $w_1 < w_2$.

Any node t in a $\mathbb{W}$-labelled (c_0, C)-colourful tree can be represented by a word generated by the following regular expression

$$\{0,1\}^* c_{i_1} \cdot \{0,1\}^* c_{i_2} \cdot \ldots \cdot \{0,1\}^* c_{i_m}$$

where $c_{i_j} \neq c_{i_k}$ if $j \neq k$ and $c_{i_j} = \perp$ if and only if $j = m$. We call the number of 0s and 1s occurring in the word, *the number of bits used to label t*. We show in the following lemma that it is possible to have a labelling of our colourful universal tree $\mathcal{U}^\ell_{(c_0,C)}$ such that the labelling of each node in it is 'short'.

Lemma 3. *There is a $\mathbb{W}$-labelling of the tree $\mathcal{U}^\ell_{(c_0,C)}$, denoted by $\mathcal{L}^\ell_C$ such that the number of bits used to label any node of $\mathcal{L}^\ell_C$ is at most ℓ.*

Proof (sketch). For $\ell > 0$, we have $\mathcal{U}^\ell_{(c_0,C)} = \mathcal{U}^{\ell-1}_{(c_0,C)} \cdot \left(c_0, \left\langle \mathcal{U}^\ell_{(c_1,C_1)}, \dots, \mathcal{U}_{(c_h,C_h)} \right\rangle\right) \cdot \mathcal{U}^{\ell-1}_{(c_0,C)}$. We obtain recursively a labelling of $\mathcal{U}^{\ell-1}_{(c_0,C)}$ and append the bit 0 for the copy on the left and append with 1 for the copy on the right. For all the labellings of $\mathcal{U}^\ell_{(c_i,C_i)}$, we add the element $\varepsilon \cdot c_i$ as a prefix.

We rigorously prove this in the full version of the paper, but only state here that the three operations defined in the statement of Lemma 1 can be computed in time $O(k\ell \log k)$ (denoted by T_{next}), where $k = |C| + 1$.

Theorem 4. *Finding the winner in a (c_0, C)-colourful Rabin game with n vertices, m edges, and $k = |C| + 1$, takes time*

$$\widetilde{O}\left(mnk^2 k! \min\left\{n2^k, \binom{\lceil \log n \rceil + k}{k-1}\right\}\right)$$

and $O(nk \log k \log n)$ space.

Proof. We know that the lifting Algorithm 1 for a (c_0, C)-colourful tree finds the Rabin measure into the tree $\mathscr{L}$ in time $O(m|\mathscr{L}|T_{\mathsf{next}})$ from Theorem 2. All that remains is to plug in the values of the size of the universal tree obtained from Theorem 3. For a game with n vertices, we instantiate the algorithm with $\mathscr{L}$ being the $\mathbb{W}$-labelling of the (c_0, C)-colourful 2^ℓ-universal tree $\mathscr{U}^{\ell}_{(c_0,C)}$ constructed, where $\ell = \lceil \log(n) \rceil$. The tree $\mathscr{L}$ therefore has at most $\left(nk! \min\left\{n2^k, \binom{\lceil \log n \rceil + k}{k-1}\right\}\right)$ many leaves from Theorem 3, and hence at most k times as many nodes, since each node has at most k ancestors. Moreover, the time taken to navigate the tree T_{next} is at most $O(k\ell \log k)$. The space required by the algorithm at each step is just the space required to store the map. To store a map, we need to store a node in the tree for each vertex. But from Lemma 3, storing each node only requires us to store a sequence of the k colours and at most $\log n$ bits. Since to store these k colours, we need $k \log k$ bits and, the total space complexity is $O(k \log k \log n)$ for each of the n vertices, giving us the desired space complexity.

6 Conclusions, Discussion and Future Work

We have shown an algorithm for Rabin games that requires almost quadratic space and takes time that is polynomial in n and $(k!)^{1+o(1)}$. Significantly more asymptotic improvement to the running time may be difficult, as it was shown in the work of Calude et al. [1,2] that there are no algorithms to solve Rabin games (as well as Muller games) in time $n^{O(1)} \cdot 2^{o(k \log k)}$ unless the Exponential Time Hypothesis fails (informally, it is the assumption that 3-SAT has no sub-exponential algorithms). However, improvements in the exponents of the parameter $k!$, which contributes to the majority of the running time would prove useful in any algorithm that solves Rabin games. We have shown that using colourful universal trees cannot provide a significant improvement bound because of the $k!^{1+o(1)}$ lowerbound on the size of such a tree. However, any technique that improves, even on a few targeted cases, this $1 + o(1)$ bound could lead to faster algorithms. For instance, the recent unpublished work of Liang, Khoussainov, and Xiao [24] improve the running time for specific values of k, where the size of k is large (comparable to n).

While we focus on the theoretical advance in this paper, an obvious future direction is to implement the algorithm. There are tools that convert LTL specifications to Rabin automata—such as Rabinizer 4 [22]. It will be interesting to see if solving the obtained Rabin games using our algorithms outperforms converting them instead to parity games and then using state-of-the-art parity game solvers such as Oink [10] framework. We believe improvement in state space of solving Rabin games through our paper might lead to more efficient algorithms for the problem of reactive synthesis of LTL formulas.

Our algorithm, like other progress measure algorithms, can display worst-case behaviour in certain asymmetric examples. To show a vertex is losing for Controller, the measure needs to increase until it reaches $\top$. This lack of symmetric treatment of the players by our algorithm might lead to worst case behaviour on several examples. But circumventing this problem by constructing similar measures for Environment in the hopes of finding a symmetric algorithm is not as straightforward, as Environment does not have a positional strategy in this game.

In a different direction, symbolic algorithms for parity games are either implicitly or explicitly guided by universal trees [3,19] constructed for both players. We believe with some effort, our small colourful universal trees can be exploited to make symbolic algorithms to solve Rabin games. One such algorithm would look like an asymmetric variation of the universal algorithm in the work of Jurdziński, Morvan, and Thejaswini [19] for parity games, combined with our construction of colourful universal trees. Indeed, we already have a definition of colourful decompositions which one might hope to obtain as an end-result of such a recursive symbolic algorithm.

Acknowledgements. We would like to thank Marcin Jurdziński and Anne-Kathrin Schmuck for valuable discussions and references. We also thank Aditya Prakash for his valuable comments and, in particular, for reading the section on colourful trees despite his colour blindness.

References

1. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasi-polynomial time. SIAM Journal on Computing **51**(2), STOC17–152–STOC17–188 (2022). https://doi.org/10.1137/17M1145288
2. Casares, A., Pilipczuk, M., Pilipczuk, M., Souza, U., Thejaswini, K.S.: Simple and tight complexity lower bounds for solving Rabin games (2023), accepted at SOSA 24.
3. Chatterjee, K., Dvořák, W., Henzinger, M., Svozil, A.: Quasipolynomial set-based symbolic algorithms for parity games. In: LPAR-22. EPiC Series in Computing, vol. 57, pp. 233–253. EasyChair, Awassa, Ethiopia (2018). https://doi.org/10.29007/5z5k
4. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Summaries of the Summer Institute of Symbolic Logic **1**, 3–50 (1957). https://doi.org/10.2307/2271310
5. Colcombet, T., Fijalkow, N., Gawrychowski, P., Ohlmann, P.: The theory of universal graphs for infinite duration games. Log. Methods Comput. Sci. **18**(3) (2022). https://doi.org/10.46298/lmcs-18(3:29)2022
6. Czerwiński, W., Daviaud, L., Fijalkow, N., Jurdziński, M., Lazić, R., Parys, P.: Universal trees grow inside separating automata: Quasi-polynomial lower bounds for parity games. In: Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019. pp. 2333–2349. SIAM (2019). https://doi.org/10.1137/1.9781611975482.142
7. Daviaud, L., Jurdziński, M., Lehtinen, K.: Alternating weak automata from universal trees. In: 30th International Conference on Concurrency Theory, CONCUR 2019. Leibniz International Proceedings in Informatics (LIPIcs), vol. 140, pp. 18:1–18:14.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Amsterdam, the Netherlands (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.18

8. Daviaud, L., Jurdziński, M., Thejaswini, K.S.: The Strahler number of a parity game. In: A. Czumaj, A.D., Merelli, A. (eds.) 47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference). LIPIcs, vol. 168, pp. 123:1–123:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ICALP.2020.123
9. Dell'Erba, D., Schewe, S.: Smaller progress measures and separating automata for parity games. Frontiers Comput. Sci. **4** (2022). https://doi.org/10.3389/fcomp.2022.936903
10. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: Tools and Algorithms for the Construction and Analysis of Systems, 24th International Conference, TACAS 2018. LNCS, vol. 10805, pp. 291–308. Springer, Thessaloniki, Greece (2018). https://doi.org/10.1007/978-3-319-89960-2_16
11. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs (extended abstract). In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 328–337. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21949
12. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: 32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991. pp. 368–377. IEEE Computer Society (1991). https://doi.org/10.1109/SFCS.1991.185392
13. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. SIAM Journal on Computing **29**(1), 132–158 (1999). https://doi.org/10.1137/S0097539793304741
14. Fearnley, J., Jain, S., de Keijzer, B., Schewe, S., Stephan, F., Wojtczak, D.: An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space. International Journal on Software Tools for Technology Transfer **21**(3), 325–349 (2019). https://doi.org/10.1007/s10009-019-00509-3
15. Francez, N., Kozen, D.: Generalized fair termination. In: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 46–53. POPL '84, Association for Computing Machinery, New York, NY, USA (1984). https://doi.org/10.1145/800017.800515
16. Horn, F.: Streett games on finite graphs. In: Games in Design and Verification (2005)
17. Jurdziński, M.: Small progress measures for solving parity games. In: 17th Annual Symposium on Theoretical Aspects of Computer Science. LNCS, vol. 1770, pp. 290–301. Springer, Lille, France (2000). https://doi.org/10.1007/3-540-46541-3_24
18. Jurdziński, M., Lazić, R.: Succinct progress measures for solving parity games. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017. pp. 1–9. IEEE Computer Society, Reykjavik, Iceland (2017). https://doi.org/10.1109/LICS.2017.8005092
19. Jurdziński, M., Morvan, R., Thejaswini, K.S.: Universal algorithms for parity games and nested fixpoints. In: Raskin, J.F., Chatterjee, K., Doyen, L., Majumdar, R. (eds.) Principles of Systems Design - Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13660, pp. 252–271. Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_12
20. Klarlund, N., Kozen, D.: Rabin measures and their applications to fairness and automata theory. In: [1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science. pp. 256–265 (1991). https://doi.org/10.1109/LICS.1991.151650
21. Koh, Z.K., Loho, G.: Beyond value iteration for parity games: Strategy iteration with universal trees. In: S. Szeider, R.G., Silva, A. (eds.) 47th International Symposium on Mathematical Foundations of Computer Science, MFCS 2022, August 22-26, 2022, Vienna, Austria.

LIPIcs, vol. 241, pp. 63:1–63:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.MFCS.2022.63

22. Kretínský, J., Meggendorfer, T., Sickert, S., Ziegler, C.: Rabinizer 4: From LTL to your favourite deterministic automaton. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 567–577. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_30
23. Kupferman, O., Vardi, M.Y.: Weak alternating automata and tree automata emptiness. In: Symposium on the Theory of Computing (1998). https://doi.org/10.1145/276698.276748
24. Liang, Z., Khoussainov, B., Xiao, M.: Two new algorithms for solving Muller games and their applications. CoRR **abs/2311.04655** (2023). https://doi.org/10.48550/ARXIV.2311.04655
25. Majumdar, R., Saglam, I., Thejaswini, K.S.: Rabin games and colourful universal trees. CoRR **abs/2311.04655** (2024). https://doi.org/10.48550/ARXIV.2401.07548
26. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Information and Control **9**(5), 521–530 (1966). https://doi.org/10.1016/S0019-9958(66)80013-X
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). https://doi.org/10.1109/LICS.2006.23
28. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. p. 179–190. POPL '89, Association for Computing Machinery, New York, NY, USA (1989). https://doi.org/10.1145/75277.75293
29. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the American Mathematical Society **141**, 1–35 (1969). https://doi.org/10.2307/1995086
30. Streett, R.S.: Propositional dynamic logic of looping and converse. In: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing. p. 375–383. STOC '81, Association for Computing Machinery, New York, NY, USA (1981). https://doi.org/10.1145/800076.802492
31. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**(2), 285 – 309 (1955). https://doi.org/10.2140/pjm.1955.5.285

Concurrency

Decidable Verification under Localized Release-Acquire Concurrency

Abhishek Kr Singh(✉) and Ori Lahav

Tel Aviv University, Tel Aviv, Israel
abhishek.uor@gmail.com, orilahav@tau.ac.il

Abstract. State reachability for finite state concurrent programs running under Release-Acquire (RA) semantics is known to be undecidable, while under a weaker variant, called Weak-Release-Acquire (WRA), the problem is decidable. However, WRA allows many counterintuitive behaviors not allowed under RA, in which threads locally oscillate between observed values. We propose a strengthening of WRA in the form of a new memory model, which we call Localized Release-Acquire (LRA), that prunes these oscillatory behaviors. We provide semantics for LRA and show that verification under LRA is decidable by extending the potential-based technique used to prove decidability under WRA. The LRA model is still weaker than RA, and thus our results can be used to soundly verify programs under RA.

Keywords: Relaxed Memory Concurrency · State Reachability · Release-Acquire Semantics

1 Introduction

The *Release-Acquire* memory model (RA), a prominent fragment of the C/C++ shared-memory concurrency specifications from 2011 [13, 16, 17, 27], has recently gained a lot of attention (see, e.g., [2, 7, 18, 23–25, 30]). For programmers, RA combines the essential guarantees of coherence [11] (a.k.a. "sequential consistency per-location") and causal consistency [10, 20], which enable the implementation of various concurrent algorithms and synchronization mechanisms with very few barriers. For implementors, RA is weaker than the Total Store Order model (TSO) [29, 31], which enables efficient mapping of memory accesses to Intel's x86 processors. Moreover, unlike TSO, RA is "monotone" [33], which, roughly speaking, means that replacing parallel composition with sequential composition can never introduce additional behaviors [26].

Unfortunately, the fundamental problem of state reachability in finite-state concurrent programs running under RA was recently shown to be undecidable [2]. This is in contrast with state reachability assuming the well-known model of sequential consistency (SC) [28], which amounts to standard reachability in a finite state system, as well as with state reachability assuming TSO, which was shown to be decidable [4, 5, 12] using the framework of well-structured transition systems (WSTS) [1, 15]. More recently, decidability of state reachability was

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 235–254, 2024.
https://doi.org/10.1007/978-3-031-57256-2_12

established for two variants of RA [21,22], called Strong Release-Acquire (SRA) and Weak Release-Acquire (WRA), which bound RA from above (every behavior allowed by SRA is allowed by RA) and below (every behavior allowed by RA is allowed by WRA). In particular, verification under WRA can be used to obtain sound (but incomplete) verification under RA, since any buggy program under RA is also buggy under WRA. The gap, however, between WRA and RA includes some dubious behaviors:

Example 1. The annotated behaviors in the three litmus tests below are allowed by WRA but disallowed by RA:

$$\begin{array}{c|c|c}
\text{(Oscillation 1)} & \text{(Oscillation 2)} & \text{(Oscillation 3)} \\
\mathtt{x} := 2 \; \left\| \; \begin{array}{ll} \mathtt{x} := 1 & \\ \mathtt{b} := \mathtt{x} & /\!/ \mathit{2} \\ \mathtt{c} := \mathtt{x} & /\!/ \mathit{1} \end{array} \right. &
\mathtt{x} := 2 \; \left\| \; \begin{array}{ll} \mathtt{a} := \mathtt{x} & /\!/ \mathit{1} \\ \mathtt{b} := \mathtt{x} & /\!/ \mathit{2} \\ \mathtt{c} := \mathtt{x} & /\!/ \mathit{1} \end{array} \; \right\| \; \mathtt{x} := 1 &
\mathtt{x} := 2 \; \left\| \; \begin{array}{ll} \mathtt{a} := & /\!/ \mathit{1} \\ \mathtt{b} := \mathtt{x} & /\!/ \mathit{2} \\ \mathtt{c} := \mathtt{x} & /\!/ \mathit{1} \end{array} \; \right\| \; \begin{array}{r} \mathtt{x} := 1 \\ := 1 \end{array}
\end{array}$$

Intuitively speaking, a thread in WRA can "change its mind" about the order of concurrent writes. In RA, every shared variable is governed by a "modification order" which dictates the (globally agreed upon) order of concurrent writes, and reads have to respect that order.

In this paper, we aim to narrow the gap between models with decidable reachability problem and RA by providing a model that lies between WRA and RA and still allows for decidable verification. More concretely, we propose to strengthen WRA in a way that eliminates the above oscillatory behaviors, while still (1) being weaker than RA and (2) inducing a decidable state reachability problem. The proposed model, which we call Localized Release-Acquire (LRA), is obtained by adding one constraint (a.k.a. axiom) to WRA's declarative consistency predicate. In turn, decidability is established similarly to [22], by carefully designing an operational "lossy" semantics based on maintaining *thread potentials*, so that it fits well in the framework of WSTS, and it is equivalent to LRA. Our proof establishes the equivalence of the lossy potential-based system with LRA using forward simulation in one direction and backward simulation in the converse.

The full version of this paper available in [32] contains detailed proofs for the claims of the paper.

2 Preliminaries

In this section we present the formal preliminaries for our results, including the representation of concurrent programs, memory systems, and declarative execution graphs. We employ the following *finite* domains (and metavariables ranging over them):

$$\begin{aligned}
\textit{thread identifiers } \tau, \pi &\in \mathsf{Tid} = \{\mathtt{T}_1, \mathtt{T}_2, ...\} \\
\textit{variables } x, y &\in \mathsf{Loc} \triangleq \{\mathtt{x}, \;\; , ...\} \\
\textit{values } v &\in \mathsf{Val} \triangleq \{0, 1, 2, ...\}
\end{aligned}$$

We represent concurrent programs as labeled transition systems. A *labeled transition system* (LTS, for short) A over an alphabet Σ is a triple $\langle Q, Q_0, T \rangle$, where

Q is a set of *states*, $Q_0 \subseteq Q$ is the set of *initial states*, and $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. We denote by $A.\mathtt{Q}$, $A.\mathtt{Q}_0$, and $A.\mathtt{T}$ the three components of an LTS A; we write $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q' \rangle \mid \langle q, \sigma, q' \rangle \in A.\mathtt{T}\}$ and $\rightarrow_A$ for $\bigcup_{\sigma \in \Sigma} \xrightarrow{\sigma}_A$. A state $q \in A.\mathtt{Q}$ is *reachable* in A if $q_0 \rightarrow_A^* q$ for some $q_0 \in A.\mathtt{Q}_0$. A sequence $\sigma_1, \ldots, \sigma_n$ is a *trace* of A if $q_0 \xrightarrow{\sigma_1}_A q_1 \xrightarrow{\sigma_2}_A \cdots q_{n-1} \xrightarrow{\sigma_n}_A q_n$ for some $q_0 \in A.\mathtt{Q}_0$ and $q_1, \ldots, q_n \in A.\mathtt{Q}$.

For brevity, we elide the definition of how concurrent programs in a programming language are interpreted as LTSs (see [22] for such definition), but only note that these LTSs are *finite-state* and they employ labels (a.k.a. "program transition labels") from the set $\mathsf{ProgLab} \triangleq \mathsf{Tid} \times (\mathsf{Lab} \cup \{\epsilon\})$, where Lab denotes the set of *action labels*, representing interactions that a program may have with the memory, and ϵ denotes a thread-internal transition. Action labels $l \in \mathsf{Lab}$ take one of the following forms: a read $\mathtt{R}(x, v_\mathtt{R})$, a write $\mathtt{W}(x, v_\mathtt{W})$, or a read-modify-write $\mathtt{RMW}(x, v_\mathtt{R}, v_\mathtt{W})$, where $x \in \mathsf{Loc}$ and $v_\mathtt{R}, v_\mathtt{W}, \in \mathsf{Val}$. The functions `t p`, `loc`, $\mathtt{val}_\mathtt{R}$, and $\mathtt{val}_\mathtt{W}$ respectively retrieve (when applicable) the type (R/W/RMW), variable (x), read value ($v_\mathtt{R}$), and written value ($v_\mathtt{W}$) of an action label. Furthermore, for a program transition label $\alpha \in \mathsf{ProgLab}$, the functions `tid` and `lab` respectively retrieve the thread identifier (τ) and the action label (or ϵ) of α, and the functions on action labels (`t p`, `loc`, ...) are lifted to program transition labels in the obvious way.

To represent concurrent programs running under a particular memory model, we synchronize the transitions of a program Pr with a memory system. A memory system is another LTS $\mathcal{M}$ (but, possibly infinite-state) whose set of transition labels consists of non-silent program transition labels (elements of $\mathsf{Tid} \times \mathsf{Lab}$) as well as a (disjoint) set $\mathcal{M}.\mathbf{\Theta}$ of memory-internal actions. Then, the composition of a program Pr and a memory system $\mathcal{M}$, denoted by $Pr \bowtie \mathcal{M}$, is the LTS whose transition labels are the elements of $\mathsf{ProgLab} \cup \mathcal{M}.\mathbf{\Theta}$; states are pairs $\langle \overline{p}, M \rangle \in Pr.\mathtt{Q} \times \mathcal{M}.\mathtt{Q}$; initial state is $\langle \overline{p}_{\mathsf{Init}}, \mathcal{M}.\mathtt{Q}_0 \rangle$; and transitions are given by:

$$\frac{\begin{array}{c}\alpha \in \mathsf{Tid} \times \mathsf{Lab} \\ \overline{p} \xrightarrow{\alpha}_{Pr} \overline{p}' \qquad M \xrightarrow{\alpha}_{\mathcal{M}} M'\end{array}}{\langle \overline{p}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}', M' \rangle} \qquad \frac{\begin{array}{c}\alpha \in \mathsf{Tid} \times \{\epsilon\} \\ \overline{p} \xrightarrow{\alpha}_{Pr} \overline{p}'\end{array}}{\langle \overline{p}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}', M \rangle} \qquad \frac{\begin{array}{c}\alpha \in \mathcal{M}.\mathbf{\Theta} \\ M \xrightarrow{\alpha}_{\mathcal{M}} M'\end{array}}{\langle \overline{p}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}, M' \rangle}$$

The state reachability problem for a memory system $\mathcal{M}$ receives as input a program Pr and a state $\overline{p} \in Pr.\mathtt{Q}$ and asks whether $\langle \overline{p}, M \rangle$ is reachable in $Pr \bowtie \mathcal{M}$ for some $M \in \mathcal{M}.\mathtt{Q}$.

Finally, we also need the notion of a *declarative* memory model, which accepts/rejects program behaviors based on constraints on the generated *execution graphs*.

Definition 1. An *execution graph* G is a pair $\langle E, rf \rangle$, where:

- E is a finite set of *events*. An *event* e is a tuple $\langle \tau, s, l \rangle$, where $\tau \in \mathsf{Tid}$, called the event's *thread identifier*; $s \in \mathbb{N}$, called the event's *serial identifier*, and $l \in \mathsf{Lab}$, called the event's *label*. The functions `tid`, `sn`, and `lab` return the thread identifier (τ), identifier (s), and action label (l) of an event. All functions on action labels ($\texttt{t p}$, $\texttt{loc}$, ...) are lifted to events in the obvious way. We denote by E the set of all events, and define the following subsets:

$$\mathsf{R} \triangleq \{e \in \mathsf{E} \mid \texttt{t p}(e) \in \{\texttt{R}, \texttt{RMW}\}\} \qquad \mathsf{W} \triangleq \{e \in \mathsf{E} \mid \texttt{t p}(e) \in \{\texttt{W}, \texttt{RMW}\}\}$$
$$\mathsf{RMW} \triangleq \mathsf{R} \cap \mathsf{W} \qquad \mathsf{E}^{\tau} = \{e \in \mathsf{E} \mid \texttt{tid}(e) = \tau\}$$

– rf is a *reads-from relation for* E, that is a relation on E satisfying:
  - If $\langle w, r\rangle \in \mathit{rf}$, then $w \in \mathsf{W}$ and $r \in \mathsf{R}$.
  - If $\langle w, r\rangle \in \mathit{rf}$, then $\texttt{loc}(w) = \texttt{loc}(r)$ and $\texttt{val}_{\texttt{W}}(w) = \texttt{val}_{\texttt{R}}(r)$.
  - $w_1 = w_2$ whenever $\langle w_1, r\rangle, \langle w_2, r\rangle \in \mathit{rf}$ (each read reads from at most one write).
  - For every $r \in E \cap \mathsf{R}$, there exists some $w \in E$ such that $\langle w, r\rangle \in \mathit{rf}$ (each read reads from some write).

We denote the components of G by $G.\texttt{E}$ and $G.\texttt{rf}$. For any set $E' \subseteq \mathsf{E}$, we write $G.E'$ for $G.\texttt{E} \cap E'$ (e.g., $G.\mathsf{W} = G.\texttt{E} \cap \mathsf{W}$). The *program order* induced by an execution graph G, denoted by $G.\mathsf{po}$, is defined as $G.\mathsf{po} \triangleq \{\langle e_1, e_2\rangle \in E \times E \mid \texttt{sn}(e_1) < \texttt{sn}(e_2) \wedge \texttt{tid}(e_1) = \texttt{tid}(e_2)\}$.

Given a set E of events, $\tau \in \mathsf{Tid}$, and $l \in \mathsf{Lab}$, $\mathsf{Ne\ tEvent}(E, \tau, l)$ denotes the event with thread identifier τ, label l, and a minimal fresh serial identifier w.r.t. E, i.e., $\mathsf{Ne\ tEvent}(E, \tau, l) \triangleq \langle \tau, s, l\rangle$, where $s = \min\{n \in \mathbb{N} \mid \langle \tau, n, l\rangle \notin E\}$.

Definition 2. An execution graph G is *generated by* a program Pr with final state $\overline{p} \in Pr.\texttt{Q}$ if $\langle \overline{p}_0, G_0\rangle \to^{*} \langle \overline{p}, G\rangle$ for some $\overline{p}_0 \in Pr.\texttt{Q}_0$, where G_0 denotes the empty execution graph (given by $G_0 \triangleq \langle \emptyset, \emptyset\rangle$) and $\to$ is defined by:

$$\frac{\overline{p} \xrightarrow{\tau, l}_{Pr} \overline{p}' \qquad E' = E \cup \{\mathsf{Ne\ tEvent}(E, \tau, l)\} \qquad \mathit{rf} \subseteq \mathit{rf}' \qquad \langle E', \mathit{rf}'\rangle \text{ is an execution graph}}{\langle \overline{p}, \langle E, \mathit{rf}\rangle\rangle \to \langle \overline{p}', \langle E', \mathit{rf}'\rangle\rangle} \qquad \frac{\overline{p} \xrightarrow{\tau, \varepsilon}_{Pr} \overline{p}'}{\langle \overline{p}, G\rangle \to \langle \overline{p}', G\rangle}$$

Using the above definitions, a declarative memory model can be identified with a set of so-called *consistent* execution graphs, and a program state $\overline{p}$ is 'emphreachable under a declarative memory model if some consistent execution graph G is generated by Pr with final state $\overline{p}$.

3 The Localized Release-Acquire Model

In this section we introduce the Localized Release-Acquire (LRA) model, starting with its declarative presentation. LRA is obtained by adding a single constraint, called "local-read-coherence", to WRA. We first briefly repeat the three constraints of WRA (see [20] for more details). Figure 1 summarizes the four constraints of LRA.

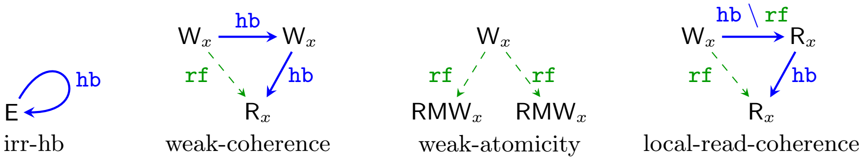


Fig. 1. Illustration of forbidden patterns in LRA

Notation for relations. Given a relation R, *dom*(R) denotes its domain; R and R^+ denote its reflexive and transitive closures; and R^{-1} denotes its inverse. The (left) composition of relations R_1, R_2 is denoted by $R_1 \,;\, R_2$. We denote by $[A]$ the identity relation on a set A (e.g., $[A] \,;\, R \,;\, [B] = R \cap (A \times B)$).

First, we need a derived "happens-before" relation. For a given execution graph G, we define $G.\mathtt{hb} \triangleq (G.\mathsf{po} \cup G.\mathtt{rf})^+$. We require that $G.\mathtt{hb}$ is a partial order, which results in our first constraint:

$$G.\mathtt{hb} \text{ is irreflexive} \qquad \text{(irr-hb)}$$

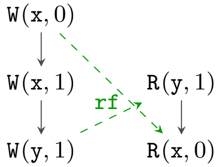
The next constraint intuitively makes sure that "a thread cannot read a value when it is aware of a later value written to the same location", where "aware" and "later" are interpreted using $G.\mathtt{hb}$. Formally, we define $G.\mathtt{hb}|_{\mathtt{loc}} \triangleq \{\langle e_1, e_2\rangle \in G.\mathtt{hb} \mid \mathtt{loc}(e_1) = \mathtt{loc}(e_2)\}$ (i.e., per-location restriction of the happens-before relation), and require the following:

$$G.\mathtt{hb}|_{\mathtt{loc}} \,;\, [\mathsf{W}] \,;\, G.\mathtt{hb} \,;\, G.\mathtt{rf}^{-1} \text{ is irreflexive} \qquad \text{(weak-coherence)}$$

In particular, the following annotated outcome of the message-passing (MP) test is forbidden:

(MP)

```
x := 0  ║
x := 1  ║  a :=      // 1
  := 1  ║  b := x    // 0
```



An execution graph justifying this outcome must have $\mathtt{rf}$-edges as depicted above. However, we have $\mathtt{hb}|_{\mathtt{loc}}$ from $\mathtt{W}(\mathtt{x}, 0)$ to $\mathtt{W}(\mathtt{x}, 1)$, $\mathtt{hb}$ from $\mathtt{W}(\mathtt{x}, 1)$ to $\mathtt{R}(\mathtt{x}, 0)$, and $\mathtt{rf}$ from $\mathtt{W}(\mathtt{x}, 0)$ to $\mathtt{R}(\mathtt{x}, 0)$, which is forbidden by weak-coherence.

The final condition that comes from WRA ensures that distinct RMW events never read from the same write event:

$$\forall \langle w_1, e_1\rangle, \langle w_2, e_2\rangle \in G.\mathtt{rf} \,;\, [\mathsf{RMW}].\; w_1 = w_2 \implies e_1 = e_2 \qquad \text{(weak-atomicity)}$$

This concludes the consistency constraints of WRA. As noted above, unlike RA, WRA admits behaviors in which threads oscillate between values that were concurrently written to the same location. Our proposed condition of LRA that prunes these behaviors is the following:

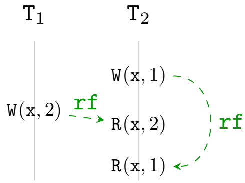
$$(G.\mathtt{hb}|_{\mathtt{loc}} \setminus G.\mathtt{rf}) \,;\, [\mathsf{R}] \,;\, G.\mathtt{hb} \,;\, G.\mathtt{rf}^{-1} \text{ is irreflexive} \qquad \text{(local-read-coherence)}$$

Intuitively, this constraint ensures that a thread cannot read from a certain write w if it is already aware of a read r' reading from the same location that is later than w and reads from some other write w'. Again, "aware" and "later" are interpreted using $G.\texttt{hb}$.

The following examples demonstrate "oscillations" between observed values that are allowed in WRA but forbidden in LRA.

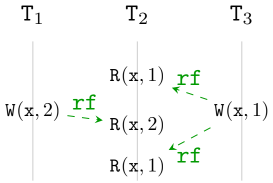
(Oscillation 1)

```
        ║ x := 1
x := 2  ║ b := x  // 2
        ║ c := x  // 1
```



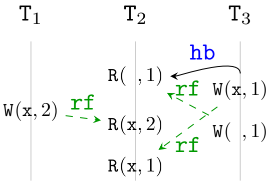
(Oscillation 2)

```
        ║ a := x  // 1 ║
x := 2  ║ b := x  // 2 ║ x := 1
        ║ c := x  // 1 ║
```



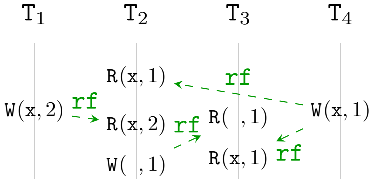
(Oscillation 3)

```
        ║ a :=      // 1 ║
x := 2  ║ b := x    // 2 ║ x := 1
        ║ c := x    // 1 ║   := 1
```



(Order-Propagation)

```
        ║ a := x  // 1 ║ d :=      // 1 ║
x := 2  ║ b := x  // 2 ║ c := x    // 1 ║ x := 1
        ║   := 1       ║                ║
```



It can be checked that local-read-coherence forbids these execution graphs: in all of them we have (1) $G.\texttt{hb}|_{\texttt{loc}} \setminus G.\texttt{rf}$ from $\texttt{W}(\texttt{x}, 1)$ to $\texttt{R}(\texttt{x}, 2)$; (2) $G.\texttt{hb}$ from $\texttt{R}(\texttt{x}, 2)$ to the read $\texttt{R}(\texttt{x}, 1)$ that represents the read to $\texttt{c}$; and (3) $\texttt{rf}$ from $\texttt{W}(\texttt{x}, 1)$ to that read.

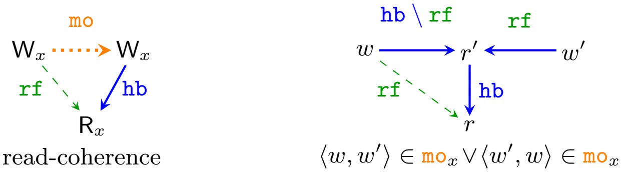
Next, we establish the relation between LRA and RA (see [22] for a definition of RA).

Proposition 1. LRA is weaker than RA, that is: if a program state is reachable under RA, then it is also reachable under LRA.

Proof. We establish this result by recalling the following "read-coherence" consistency constraint of RA (see Figure 2 and [20] for more details). Note the use of modification order $G.\texttt{mo}$ in RA to interpret one write being "later" than another, in the place of $G.\texttt{hb}|_{\texttt{loc}}$ in the "weak-coherence" in WRA. Here $G.\texttt{mo}$ is disjoint union of relations $\{G.\texttt{mo}_x\}_{x \in \mathsf{Loc}}$ where each $G.\texttt{mo}_x$ is a strict total order on W_x.

$$G.\texttt{mo} \mathbin{;} G.\texttt{hb} \mathbin{;} G.\texttt{rf}^{-1} \text{ is irreflexive} \qquad \text{(read-coherence)}$$

Since WRA is strictly weaker than RA, it suffices to show that the additional constraint "local-read-coherence" of LRA is also guaranteed in RA. The proof follows by contradiction. Assume otherwise, hence, for a given $x \in \mathsf{Loc}$, we have $w, w' \in \mathsf{W}_x$ and $r, r' \in \mathsf{R}_x$ where $\langle w, r' \rangle \in \texttt{hb} \setminus \texttt{rf}$, $\langle w', r' \rangle \in \texttt{rf}$, $\langle w, r \rangle \in \texttt{rf}$, and $w \neq w'$ (see right side of Figure 2). Since $\texttt{loc}(w) = x = \texttt{loc}(w')$, due to the RA semantics, we have one of the following cases:



read-coherence

$\langle w, w' \rangle \in \texttt{mo}_x \vee \langle w', w \rangle \in \texttt{mo}_x$

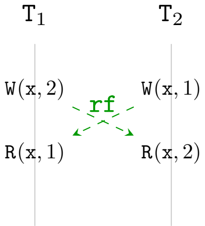
Fig. 2. Axiom read-coherence in RA and illustration for proof of Proposition 1

- $\langle w, w' \rangle \in \texttt{mo}_x$: In this case we have $\langle w, r \rangle \in \texttt{rf}$ while $\langle w, r \rangle \in \texttt{mo}_x \,;\texttt{hb}$, which contradicts the axiom read-coherence of RA.
- $\langle w', w \rangle \in \texttt{mo}_x$: In this case we have $\langle w', r' \rangle \in \texttt{rf}$ while $\langle w', r' \rangle \in \texttt{mo}_x \,;\texttt{hb}$, which again contradicts the axiom read-coherence of RA.

To see that LRA is *strictly* weaker than RA, we note that LRA does not provide full coherence. Indeed, as the next example shows, even programs with a single shared variable can exhibit weak behaviors:

(WW)

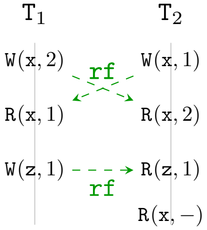
$$\begin{array}{l||l} \texttt{x} := 2 & \texttt{x} := 1 \\ \texttt{a} := \texttt{x} \quad /\!\!/\, \mathit{1} & \texttt{b} := \texttt{x} \quad /\!\!/\, \mathit{2} \end{array}$$



Interestingly, our final example shows the LRA model is possibly blocking: it may be the case that a thread simply cannot read from a certain location, since any option for reading would violate local-read-coherence.

(Blocking)

$$\begin{array}{l||l} \texttt{x} := 2 & \texttt{x} := 1 \\ \texttt{a} := \texttt{x} \quad /\!\!/\, \mathit{1} & \texttt{b} := \texttt{x} \quad /\!\!/\, \mathit{2} \\ \texttt{z} := 1 & \texttt{c} := \texttt{z} \quad /\!\!/\, \mathit{1} \\ & \texttt{d} := \texttt{x} \quad /\!\!/\, \textit{nothing can be read} \end{array}$$



Roughly speaking, the synchronization on z "joins" the threads and rules out both options. More formally, if the final read reads from $\texttt{W}(\texttt{x}, 1)$, we violate local-read-coherence due to $G.\texttt{hb}|_{\texttt{loc}} \setminus G.\texttt{rf}$ from $\texttt{W}(\texttt{x}, 1)$ to $\texttt{R}(\texttt{x}, 2)$ and $G.\texttt{hb}$ from $\texttt{R}(\texttt{x}, 2)$ to the final read. In turn, if the final read reads from $\texttt{W}(\texttt{x}, 2)$, we violate local-read-coherence due to $G.\texttt{hb}|_{\texttt{loc}} \setminus G.\texttt{rf}$ from $\texttt{W}(\texttt{x}, 2)$ to $\texttt{R}(\texttt{x}, 1)$ and $G.\texttt{hb}$ from $\texttt{R}(\texttt{x}, 1)$ to the final read.

It is important to note that the blocking aspect of LRA model does not affect the benefits of sound verification of the RA programs using LRA, since (due to Proposition 1) forbidden outcomes in LRA model (possibly due to a blocked run) are also forbidden in the RA model.

3.1 An Operational Presentation

Since LRA-consistency is “prefix-closed”, it is straightforward to “operationalize” LRA’s declarative presentation, which will help us below in relating the potential-model to LRA. To do so, we define a memory system, called opLRA, whose states are execution graphs, the only initial state is the empty execution graph, and the transitions are as follows:

$$\textsc{write}\quad \frac{\begin{array}{c} e = \mathsf{NextEvent}(G.\mathtt{E}, \tau, \mathtt{W}(x, v_{\mathtt{W}})) \\ G' = \langle G.\mathtt{E} \cup \{e\}, G.\mathtt{rf} \rangle \end{array}}{G \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\text{opLRA}} G'}$$

$$\textsc{read/rmw}\quad \frac{\begin{array}{c} l = \mathtt{R}(x, v_{\mathtt{R}}) \lor l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \\ e = \mathsf{NextEvent}(G.\mathtt{E}, \tau, l) \qquad G' = \langle G.\mathtt{E} \cup \{e\}, G.\mathtt{rf} \cup \{\langle w, e\rangle\}\rangle \\ w \in G.\mathsf{W}_x \qquad \mathtt{val}_{\mathtt{W}}(w) = v_{\mathtt{R}} \\ w \notin \mathit{dom}(G.\mathtt{hb}|_{\mathtt{loc}} \,;\, [\mathsf{W}] \,;\, G.\mathtt{hb} \;\; ; [\mathsf{E}^{\tau}]) \\ w \notin \mathit{dom}((G.\mathtt{hb}|_{\mathtt{loc}} \setminus G.\mathtt{rf}) \,;\, [\mathsf{R}] \,;\, G.\mathtt{hb} \;\; ; [\mathsf{E}^{\tau}]) \\ \mathtt{typ}(l) = \mathtt{RMW} \implies w \notin \mathit{dom}(G.\mathtt{rf} \,;\, [\mathsf{RMW}]) \end{array}}{G \xrightarrow{\tau, l}_{\text{opLRA}} G'}$$

These transitions are enforcing consistency on every step, which allows us to establish the following relation.

Proposition 2. LRA is equivalent to opLRA, that is: a program state is reachable under LRA iff it is reachable under opLRA.

4 Lossy semantics for LRA

In this section, we present loLRA, a potential-based memory system that is equivalent to LRA and well suited for verification in the framework of WSTS.

The memory states of loLRA maintain a collection of "read/write-option" lists for each thread, called the *potential* of the thread. Concretely, a state of loLRA is a *potential mapping* $\mathcal{B}$ which maps each thread $\tau \in \mathsf{Tid}$ to its potential $\mathcal{B}(\tau)$. Potentials are finite sets of *option lists*, where each option list stands for a sequence of possible future reads (*read options*) and writes (*write options*) that ascribe possible operations the thread may perform in the order it may perform them. For instance, a list $o_1 \cdot o_2$ consisting of two read options, o_1 and o_2, allows the thread to read $\mathtt{val}(o_1)$ from location $\mathtt{loc}(o_1)$ and then $\mathtt{val}(o_2)$ from location $\mathtt{loc}(o_2)$. Thread potentials are explicitly “lossy”—a thread can non-deterministically lose whatever parts of its potential at any point. Initially, the loLRA memory system non-deterministically starts in a state where all potentials consist solely of write options.

Next, we present the full definitions (which, except for loLRA’s transitions match precisely the definitions of the corresponding system for WRA in [22]).

Notation for sequences. We use ϵ to denote the empty sequence. The length of a sequence s is denoted by $|s|$ (in particular $|\epsilon| = 0$). We often identify a sequence s over Σ with its underlying function in $\{1, \ldots, |s|\} \to \Sigma$, and write $s(k)$ for the symbol at position $1 \leq k \leq |s|$ in s. We write $\sigma \in s$ if the symbol σ appears in s, that is if $s(k) = \sigma$ for some $1 \leq k \leq |s|$. We use "$\cdot$" for the concatenation of sequences, and lift it to concatenation of sets S_1 and S_2 of sequences in the obvious way ($S_1 \cdot S_2 \triangleq \{s_1 \cdot s_2 \mid s_1 \in S_1, s_2 \in S_2\}$). We identify symbols with sequences of length 1 or their singletons when needed (e.g., in expressions like $\sigma \cdot S$ for $\sigma \in \Sigma$ and a set S of sequences over Σ).

Definition 3. Options, option lists, potentials, and potential mappings are defined as follows:

1. An *option* o is either $\langle \tau, x, v, \pi_{\mathtt{RMW}} \rangle$ (*read option*) or $\mathtt{O}_{\mathtt{W}}(x)$ (*write option*), where $\tau, \pi_{\mathtt{RMW}} \in \mathsf{Tid}$, $x \in \mathsf{Loc}$, and $v \in \mathsf{Val}$. The functions `t p`, `tid`, `loc`, `val`, and `rmw-tid` return (when applicable) the type (R/W), thread identifier (τ), location (x), value (v), and RMW thread identifier ($\pi_{\mathtt{RMW}}$) of a given option.
2. An *option list* L is a finite sequence of (read or write) options. For a given option list L, we define $\mathtt{loc}(L) \triangleq \{\mathtt{loc}(o) \mid o \in L\}$.
3. A *potential* B is a finite non-empty set of option lists.
4. A *potential mapping* $\mathcal{B}$ is a function assigning a potential to every $\tau \in \mathsf{Tid}$.

We define a (well quasi) ordering on option lists that naturally extends to potentials and to potential mappings.

Definition 4. The (overloaded) relation $\sqsubseteq$ is defined by:

1. on option lists: $L \sqsubseteq L'$ if L is a (not necessarily contiguous) subsequence of L';
2. on potentials: $B \sqsubseteq B'$ if $\forall L \in B.\ \exists L' \in B'.\ L \sqsubseteq L'$ (a.k.a. "Hoare ordering");
3. on potential mappings: $\mathcal{B} \sqsubseteq \mathcal{B}'$ if $\mathcal{B}(\tau) \sqsubseteq \mathcal{B}'(\tau)$ for every $\tau \in \mathsf{Tid}$ (componentwise order).

The memory system loLRA is formally defined as follows.

Definition 5. The memory system loLRA is defined by:

- loLRA.$\mathtt{Q}$ is the set of potential mappings.
- loLRA.$\mathtt{Q}_0 = \{\mathcal{B} \mid \forall \tau \in \mathsf{Tid}, L \in \mathcal{B}(\tau), o \in L.\, \mathtt{t\ p}(o) = \mathtt{W}\}$.
- The transitions of loLRA are given in Figure 3.

The transitions of loLRA are informally understood as follows:

- READ: For a thread τ to read v from x, all lists of τ should start with an option o with $\mathtt{val}(o) = v$ and $\mathtt{loc}(o) = x$ (since it is the same option o in the head of all lists, all lists of τ also start with the same thread identifier, which is important for the equivalence result; see [22, Example 5.5]). The read step consumes these options by discarding the first element from each of τ's lists.

WRITE

$$\frac{\begin{array}{l} \qquad\qquad\qquad\qquad o = \langle \tau, x, v_{\mathtt{W}}, \pi_{\mathtt{RMW}} \rangle \\ \forall \pi \in \mathsf{Tid}, L' \in \mathcal{B}'(\pi). \\ \quad ((\pi = \tau \implies \mathtt{O}_{\mathtt{W}}(x) \cdot L' \in \mathcal{B}(\tau)) \land (\pi \neq \tau \implies L' \in \mathcal{B}(\pi))) \lor \\ \quad (\exists n \geq 1, L_0, \dots, L_n. \\ \qquad L' = L_0 \cdot (o \cdot L_1) \cdot (o \cdot L_2) \cdot \dots \cdot (o \cdot L_n) \land \\ \qquad \mathtt{O}_{\mathtt{W}}(x) \cdot (L_1 \cdot \dots \cdot L_{n-1}) \cdot \mathtt{O}_{\mathtt{W}}(x) \cdot L_n \in \mathcal{B}(\tau) \land \\ \qquad (\pi = \tau \implies \mathtt{O}_{\mathtt{W}}(x) \cdot L_0 \cdot \dots \cdot L_{n-1} \cdot \mathtt{O}_{\mathtt{W}}(x) \cdot L_n \in \mathcal{B}(\tau) \land x \notin \mathtt{loc}(L_0 \cdot \dots \cdot L_{n-1})) \land \\ \qquad (\pi \neq \tau \implies L_0 \cdot \dots \cdot L_{n-1} \cdot \mathtt{O}_{\mathtt{W}}(x) \cdot L_n \in \mathcal{B}(\pi) \land x \notin \mathtt{loc}(L_1 \cdot \dots \cdot L_{n-1}))) \end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\text{loLRA}} \mathcal{B}'}$$

READ

$$\frac{\begin{array}{c} \mathtt{loc}(o) = x \\ \mathtt{val}(o) = v_{\mathtt{R}} \\ \mathcal{B} = \mathcal{B}'[\tau \mapsto o \cdot \mathcal{B}'(\tau)] \end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\text{loLRA}} \mathcal{B}'}$$

RMW

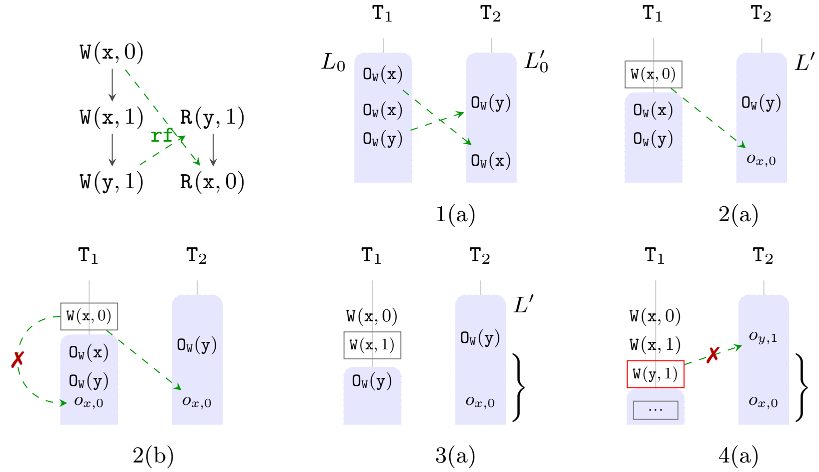
$$\frac{\begin{array}{c} \mathtt{loc}(o) = x \qquad \mathtt{val}(o) = v_{\mathtt{R}} \qquad \mathtt{rmw\text{-}tid}(o) = \tau \\ \mathcal{B} = \mathcal{B}_{\text{mid}}[\tau \mapsto o \cdot \mathcal{B}_{\text{mid}}(\tau)] \\ \mathcal{B}_{\text{mid}} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\text{loLRA}} \mathcal{B}' \end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})}_{\text{loLRA}} \mathcal{B}'}$$

LOWER

$$\frac{\mathcal{B}' \sqsubseteq \mathcal{B}}{\mathcal{B} \xrightarrow{\varepsilon}_{\text{loLRA}} \mathcal{B}'}$$

Fig. 3. Transitions of loLRA memory system

- WRITE: For a thread τ to write v to x, an option $\mathtt{O}_{\mathtt{W}}(x)$ must be the first in each of τ's lists. The WRITE consumes these options, discarding the first element from each of τ's lists. To allow future reads from the executed write, the write may add a read option o with $\mathtt{loc}(o) = x$, $\mathtt{val}(o) = v$, $\mathtt{tid}(o) = \tau$, and some $\mathtt{rmw\text{-}tid}(o)$ (possibly multiple times) in every existing list of every thread (including the writer itself). The WRITE step enforces carefully tailored conditions on *where* these new options are added:
  1. In the potential of the writer itself, a new option cannot be added after an existing write option to x (except for the write option that is consumed in this write step) and the last added read option should immediately precede an existing write option to x.
  2. In the potential of other threads the last added read option should immediately precede an existing write option to x that is to be consumed by the current write step.
  3. If more than one option is added, the added read options can never "surround" an existing read/write option with location x.
  4. New read options can be placed in a list L only if the suffix of L after the first occurrence of the newly added read options are present as an option list of the writing thread τ.
- RMW: The only additional requirement when performing an RMW compared to a non-interrupted execution of a read followed by a write is that two RMWs should never read from the same event. This is achieved by including *RMW thread identifiers* in read options, denoting the (unique) thread that may consume this option when executing an RMW. When a thread writes, it picks an (arbitrary) unique thread identifier ($\pi_{\mathtt{RMW}}$) for its added options; reads ignore this field; and RMWs by thread τ can only consume read options whose RMW thread identifier is τ.



1(a) 2(a)

2(b) 3(a) 4(a)

Fig. 4. This figure shows the loLRA transitions for MP program. Here the dashed line in 1(a) between $\mathtt{O_W}(x)$ of $\mathtt{T}_1$ and $\mathtt{O_W}(x)$ of $\mathtt{T}_2$ indicates that a future write $\mathtt{W}(x, 0)$ of $\mathtt{T}_1$ (see 2(a)) may replace the $\mathtt{O_W}(x)$ of $\mathtt{T}_2$ with a read option $o_{x,0}$. We follow a similar depiction in all the remaining diagrams of the paper.

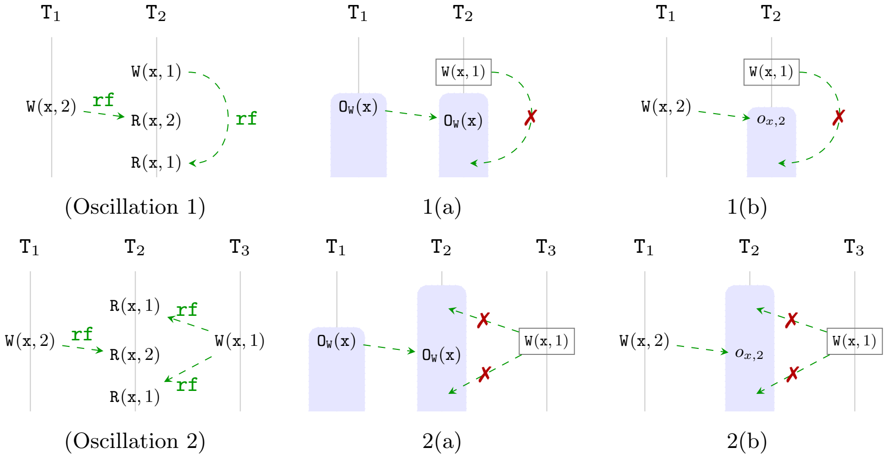
- LOWER: The step allows to remove read/write options as well as full option lists at any time.

We revisit the examples from §3 to illustrate that loLRA forbids those outcomes. In following discussions, shaded portions of the diagram for each thread correspond to its option lists. We write $o_{x,v}$ to represent a read option o with $\mathtt{loc}(o) = x$ and $\mathtt{val}(o) = v$.

Example 2. Recall the execution graph of MP from §3 (see Figure 4). Since no step in loLRA can introduce a write option, we observe the following facts about the option lists $L_0 \in \mathcal{B}_0(\mathtt{T}_1)$ and $L'_0 \in \mathcal{B}_0(\mathtt{T}_2)$ where $\mathcal{B}_0$ may lead to the annotated program state ($\mathtt{a} = 1$ and $\mathtt{b} = 0$) using a trace in which L_0 and L'_0 are not discarded by a LOWER step:

1. L_0 contains $\mathtt{O_W(x)} \cdot \mathtt{O_W(x)} \cdot \mathtt{O_W(\ \ )}$ as a sub-list to enable $\mathtt{W(x,0)}$, $\mathtt{W(x,1)}$, and $\mathtt{W(\ \ ,1)}$ in $\mathtt{T}_1$.
2. For the reads $\mathtt{R(\ \ ,1)}$ and $\mathtt{R(x,0)}$ to happen the corresponding writes $\mathtt{W(\ \ ,1)}$ and $\mathtt{W(x,0)}$ need to insert read options $o_{\mathrm{y},1}$ and $o_{\mathrm{x},0}$ at these locations (see READ step).
3. L'_0 contains $\mathtt{O_W(\ \ )}$ followed by $\mathtt{O_W(x)}$ to enable future insertions of read options $o_{\mathrm{y},1}$ and $o_{\mathrm{x},0}$ by the writes $\mathtt{W(\ \ ,1)}$ and $\mathtt{W(x,0)}$ respectively (see condition 2 of WRITE step).

Starting in the state $\mathcal{B}_0$ (1(a) in Figure 4), one can reach state 3(a) through state 2(a) in two successive steps corresponding to execution of the first two writes, $\mathtt{W(x,0)}$ and $\mathtt{W(x,1)}$ of $\mathtt{T}_1$, where the first write $\mathtt{W(x,0)}$ replaces $\mathtt{O_W(x)}$ in the option list of $\mathtt{T}_2$ with a read option $o_{\mathrm{x},0}$ resulting in $L' = L'_0[\mathtt{O_W(x)} \mapsto o_{\mathrm{x},0}]$.



(Oscillation 1) 1(a) 1(b)

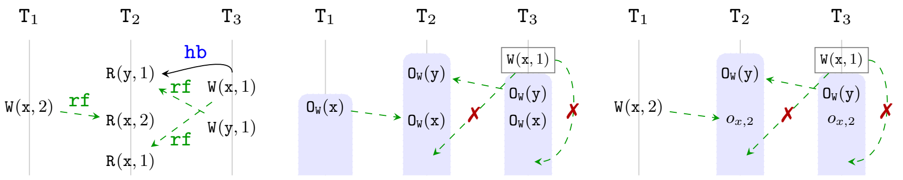
(Oscillation 2) 2(a) 2(b)

Fig. 5. loLRA transitions for Oscillation 1 and Oscillation 2 (Example 3).

In the next step (shown as 4(a)), we hope to perform the write $\mathtt{W}(\ \ ,1)$ in $\mathtt{T}_1$ and replace $\mathtt{O}_{\mathtt{W}}(\ \ )$ in $\mathtt{T}_2$ with the read option $o_{\mathrm{y},1}$. However, the current write step requires that the suffix of L' after $\mathtt{O}_{\mathtt{W}}(\ \ )$ (here, $o_{\mathrm{x},0}$) be present as an option list of thread $\mathtt{T}_1$ (due to condition 4 of the WRITE step). This is clearly not true and hence we can not continue with the current execution trace. To circumvent this blocking run the first write $\mathtt{W}(\mathtt{x},0)$ of $\mathtt{T}_1$ might want to non-deterministically insert a read option $o_{\mathrm{x},0}$ at the specified location (see 2(b)) in its option list. However, due to the presence of an earlier $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ in the option lists of $\mathtt{T}_1$ this is not allowed. Therefore, the loLRA semantics successfully forbids the annotated outcome of the message passing test.

Example 3. Recall the execution graphs of (Oscillation 1) and (Oscillation 2) from §3 (see Figure 5), where $\mathtt{T}_2$ oscillates between the observed values of $\mathtt{x}$. Consider following two cases (and the corresponding execution graphs) to observe a contradiction for each possible trace of loLRA:

- $\mathtt{W}(\mathtt{x},1)$ executes before $\mathtt{W}(\mathtt{x},2)$: For (Oscillation 1) and (Oscillation 2) this is depicted as 1(a) and 2(a) of Figure 5 respectively. Note the presence of $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ at the specified locations in the option lists of thread $\mathtt{T}_2$ to mark the end of new read options due to the future write $\mathtt{W}(\mathtt{x},2)$. In the current state of (Oscillation 1), the write $\mathtt{W}(\mathtt{x},1)$ of thread $\mathtt{T}_2$ is not allowed to put a read option in its own option list due to the presence of an earlier $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ (see condition 1 of WRITE step). Similarly in the current state of (Oscillation 2), the write $\mathtt{W}(\mathtt{x},1)$ of thread $\mathtt{T}_3$ cannot place new read options in the list of thread $\mathtt{T}_2$ because $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ appears between the new read options (see condition 3 of WRITE step).
- $\mathtt{W}(\mathtt{x},1)$ executes after $\mathtt{W}(\mathtt{x},2)$: For (Oscillation 1) and (Oscillation 2) this is depicted as 1(b) and 2(b) of Figure 5 respectively. Note the presence of $o_{\mathrm{x},2}$ (instead of $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ in the previous case) at the specified location in the option



(Oscillation 3) 3(a) 3(b)

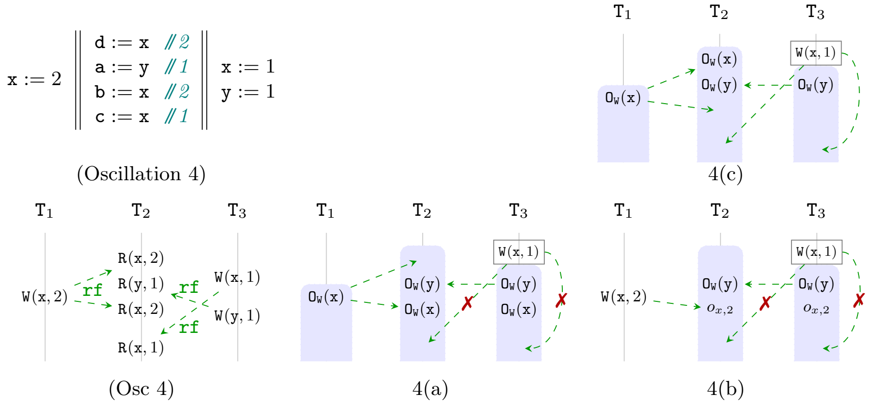
Fig. 6. The loLRA transitions for the program Oscillation 3 (Example 4).

lists of thread $\mathtt{T}_2$ to allow the read $\mathtt{R}(\mathtt{x}, 2)$ to read in future from the write $\mathtt{W}(\mathtt{x}, 2)$ of $\mathtt{T}_1$. Again in the states corresponding to 1(b) and 2(b), due to conditions 1 and 3 of WRITE step, $\mathtt{W}(\mathtt{x}, 1)$ is not allowed to put new read options at the specified locations.

Example 4. Recall the execution graph of (Oscillation 3) from §3, where $\mathtt{T}_2$ oscillates between the observed values of $\mathtt{x}$ (see Figure 6). We consider the following two cases and the resulting execution graphs, based on the order of execution between the write events $\mathtt{W}(\mathtt{x}, 1)$ and $\mathtt{W}(\mathtt{x}, 2)$, to observe a contradiction in each trace of loLRA:

- $\mathtt{W}(\mathtt{x}, 1)$ executes before $\mathtt{W}(\mathtt{x}, 2)$: This condition is depicted as 3(a). Note the presence of $\mathtt{O}_{\mathtt{W}}(\ )$ and $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ at the specified location in the option list of $\mathtt{T}_2$ to mark the end of new read options due to the future writes $\mathtt{W}(\ , 1)$ and $\mathtt{W}(\mathtt{x}, 2)$ of $\mathtt{T}_3$ and $\mathtt{T}_1$ respectively. Also note the presence of $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ in the option lists of $\mathtt{T}_3$. We claim that this $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ is needed as justification for the future write $\mathtt{W}(\ , 1)$ of $\mathtt{T}_3$ (when the write $\mathtt{W}(\ , 1)$ will be replacing the write option $\mathtt{O}_{\mathtt{W}}(\ )$ on $\mathtt{T}_2$ with the read option $o_{\mathrm{y},1}$). To justify the claim, assume otherwise (i.e., $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ is absent in the option list of $\mathtt{T}_3$), and we observe that $\mathtt{W}(\ , 1)$ of $\mathtt{T}_3$ can not continue in any of the following possible cases:
  - $\mathtt{W}(\mathtt{x}, 2)$ has not occurred when $\mathtt{W}(\ , 1)$ tries to execute: In this case $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ is still present in the option list of $\mathtt{T}_2$ and hence is required at the specified location in the option list of $\mathtt{T}_3$ as a justification for the current write $\mathtt{W}(\ , 1)$ (see condition 4 of the WRITE step). Therefore, the write $\mathtt{W}(\ , 1)$ can not continue in this case.
  - $\mathtt{W}(\mathtt{x}, 2)$ has occurred when $\mathtt{W}(\ , 1)$ tries to execute: In this case $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ on $\mathtt{T}_2$ has been replaced with a $o_{\mathrm{x},2}$ and hence $o_{\mathrm{x},2}$ is also expected in the option list of $\mathtt{T}_3$ (as justification for the current WRITE step $\mathtt{W}(\ , 1)$). However, the presence of $o_{\mathrm{x},2}$ in $\mathtt{T}_3$ can only be ensured (as insertion of new read option) by the corresponding write $\mathtt{W}(\mathtt{x}, 2)$. The write $\mathtt{W}(\mathtt{x}, 2)$ can not add a $o_{\mathrm{x},2}$ at the specified location due to the absence of $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ at the same location to mark the end of newly added read options (see condition 2 of WRITE step). Hence, in this case again the write $\mathtt{W}(\ , 1)$ can not continue.

  Assuming the presence of $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ in the option list of $\mathtt{T}_3$ (as shown in 3(a)) it is easy to see that $\mathtt{W}(\mathtt{x}, 1)$ of $\mathtt{T}_3$ can not put a read option in its own option



```
x := 2 || d := x  // 2 || x := 1
       || a := y  // 1 || y := 1
       || b := x  // 2 ||
       || c := x  // 1 ||
```

(Oscillation 4)

(Osc 4) 4(a) 4(b) 4(c)

Fig. 7. The loLRA transitions for the program Oscillation 4 (Example 5).

list (see condition 1 of WRITE step) which is necessary as justification for the future write W(, 1) of T_3 (again using similar arguments as discussed above). Therefore, the current case is forbidden by the lossy loLRA semantics.

– W(x, 1) executes after W(x, 2): This condition is depicted as 3(b), where the write W(x, 2) of T_1 has replaced the write option $O_W(x)$ in the option lists of T_2 with a read option $o_{x,2}$. Again, as discussed in the previous case (for justifying the future write W(, 1) of T_3), the write W(x, 2) of T_1 should also place a read option $o_{x,2}$ in the option lists of T_3 at the specified location. Now, as shown in 3(b), the write W(x, 1) of T_3 can not put a read option in its own option list (due to the presence of an earlier $o_{x,2}$) which is necessary as justification for the future write W(, 1) of T_3. Thus, the current case is also forbidden by the lossy loLRA semantics.

In the discussions so far (particularly related to cases 1(a), 2(a), and 3(a) of the previous examples), we observed that marking the end of newly added read options (using a pre-existing write option) is helpful in forbidding oscillations. In all of these cases it is easy to see (using exactly similar arguments) that we can also forbid these oscillatory behaviors by requiring (in conditions 1 and 2 of the WRITE step) that the beginning of newly added read options be marked using a pre-existing write option. Next example illustrates the distinctive advantage of marking the end over marking the beginning.

Example 5. Consider execution graph (Osc 4) corresponding to the annotated outcome of (Oscillation 4) shown in Figure 7. The constraint local-read-coherence forbids this execution graph since we have (1) $G.\mathtt{hb}|_{\mathtt{loc}} \setminus G.\mathtt{rf}$ from W(x, 1) to the third read R(x, 2) of T_2; (2) $G.\mathtt{hb}$ from the third read R(x, 2) of T_2 to the last read R(x, 1) of T_2; and (3) rf from W(x, 1) to the last read R(x, 1) of T_2.

Consider the following two possibilities (4(b) and 4(a) of Figure 7) corresponding to this outcome where: (1) W(x, 1) executes after W(x, 2); and (2) W(x, 1) executes before W(x, 2).

Assuming (1) and using arguments similar to Example 4, we land in configuration 4(b) which is not allowed by the lossy loLRA semantics. However, note that assuming (2) we get a contradiction only because $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ is present at the specified location in 4(a) to mark the end of new read options in the option list of $\mathtt{T}_2$ (by the write $\mathtt{W}(\mathtt{x}, 2)$ of thread $\mathtt{T}_1$). Instead, if we choose to mark the beginning (and not the end) of new read options in the option list of $\mathtt{T}_2$ we result in the configuration of 4(c) resulting in the absence of any pre-existing $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ at the end of the new entries. In this case, we observe that there is a trace of lossy loLRA (for the annotated outcome of (Oscillation 4)) in which $\mathtt{W}(\mathtt{x}, 1)$ and $\mathtt{W}(\ \ , 1)$ of $\mathtt{T}_3$ appears before $\mathtt{W}(\mathtt{x}, 2)$ of $\mathtt{T}_1$.

Next, we show that for a given program Pr, $Pr \bowtie$loLRA admits the required conditions of the WSTS framework that ensure decidability of the induced coverability problem (see, e.g., [9, 15]). In particular, the compatibility condition between the well-quasi-ordering on states and the transitions is trivial since we explicitly include the (LOWER) step in loLRA.

Lemma 1. *Given a program Pr, the LTS $Pr \bowtie lo$LRA equipped with the well-quasi-ordering $\sqsubseteq$ (lifted to states of $Pr \bowtie lo$LRA by defining $\langle \overline{p}, \mathcal{B} \rangle \sqsubseteq \langle \overline{p}', \mathcal{B}' \rangle$ iff $\overline{p} = \overline{p}'$ and $\mathcal{B} \sqsubseteq \mathcal{B}'$) is a WSTS that admits effective initialization and effective pred-basis.*

As a corollary, we obtain that state reachability under loLRA is decidable. We refer the reader to [32] where we give more details and proofs (which generally follow those in [22]).

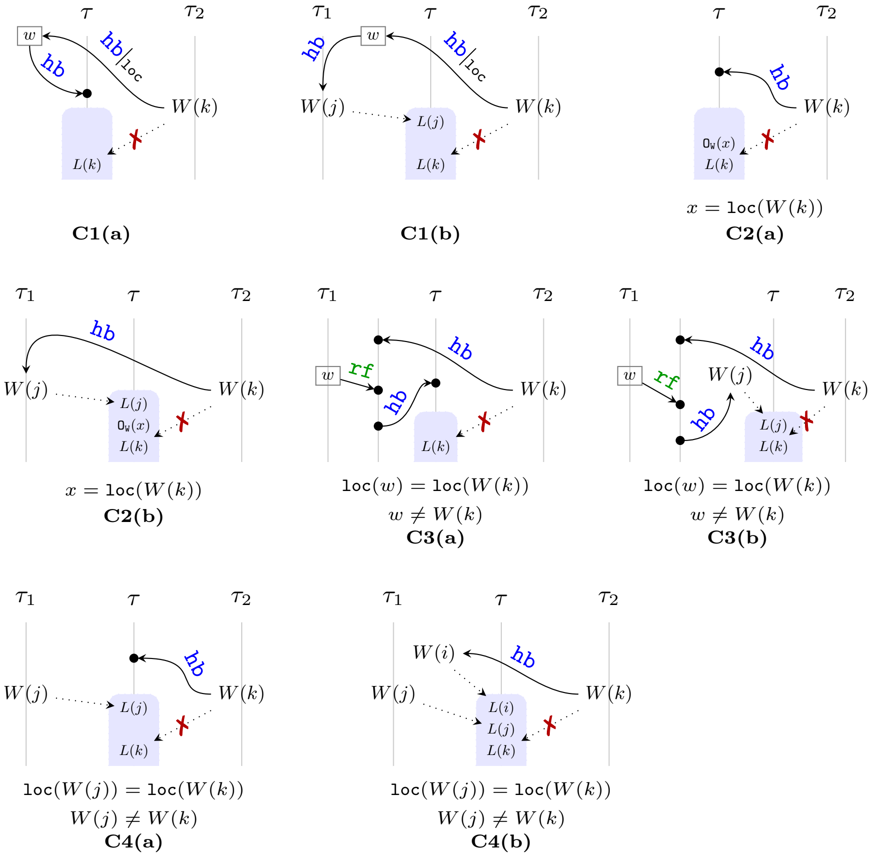
5 Equivalence of the Memory Systems for LRA

In this section we establish the equivalence between loLRA and opLRA by demonstrating a simulation between these systems. The states of loLRA and opLRA are related to each other using *write lists*, which match read options in loLRA's potentials with concrete write event in opLRA's execution graphs.

Definition 6. A *write list* is a sequence of write events and write options. Let G be an execution graph, L an option list, and $tid_{\mathtt{RMW}} : \mathsf{W} \rightharpoonup \mathsf{Tid}$. A write list W is a $\langle G, L, tid_{\mathtt{RMW}} \rangle$*-write-list* if $|L| = |W|$ and the following hold for every $1 \leq k \leq |W|$:

- If $L(k)$ is a write option, then $W(k) = L(k)$.
- If $L(k) = \langle \tau, x, v, \pi_{\mathtt{RMW}} \rangle$, then $W(k) \in G.\mathsf{W}$, $\mathtt{tid}(W(k)) = \tau$, $\mathtt{loc}(W(k)) = x$, $\mathtt{val}_{\mathtt{W}}(W(k)) = v$, and $tid_{\mathtt{RMW}}(W(k)) = \pi_{\mathtt{RMW}}$.

In addition to the above, we require that weak-coherence and local-read-coherence are maintained by any extension of the execution graph G with a sequence of reads and writes of thread τ that are obtained by following the write list W. This is formalized in the following notion of $\langle G, \tau \rangle$*-consistency* of a write list W.



C1(a)

C1(b)

$x = \texttt{loc}(W(k))$
C2(a)

$x = \texttt{loc}(W(k))$
C2(b)

$\texttt{loc}(w) = \texttt{loc}(W(k))$
$w \neq W(k)$
C3(a)

$\texttt{loc}(w) = \texttt{loc}(W(k))$
$w \neq W(k)$
C3(b)

$\texttt{loc}(W(j)) = \texttt{loc}(W(k))$
$W(j) \neq W(k)$
C4(a)

$\texttt{loc}(W(j)) = \texttt{loc}(W(k))$
$W(j) \neq W(k)$
C4(b)

Fig. 8. Illustration of conditions in Definition 7 for the $\langle G, \tau\rangle$-*consistency* of W. Each condition is split into two cases (e.g., C1 is summarized using C1(a) or C1(b)).

Definition 7. A write list W is $\langle G, \tau\rangle$-*consistent* if for every $1 \leq k \leq |W|$ with $W(k) \in \mathsf{E}$:

C1 $W(k) \notin dom(G.\texttt{hb}|_{\texttt{loc}} ; [\mathsf{W}] ; G.\texttt{hb} ; [\mathsf{E}^\tau \cup \{W(j) \mid 1 \leq j < k\}])$.
C2 If $W(i) = \texttt{O}_\texttt{W}(\texttt{loc}(W(k)))$ for some $i < k$,
then $W(k) \notin dom(G.\texttt{hb} ; [\mathsf{E}^\tau \cup \{W(j) \mid 1 \leq j < i\}])$.
C3 $W(k) \notin dom((G.\texttt{hb}|_{\texttt{loc}} \setminus G.\texttt{rf}) ; [\mathsf{R}] ; G.\texttt{hb} ; [\mathsf{E}^\tau \cup \{W(j) \mid 1 \leq j < k\}])$.
C4 If $\texttt{loc}(W(j)) = \texttt{loc}(W(k))$ and $W(k) \neq W(j)$ for some $j < k$,
then $W(k) \notin dom(G.\texttt{hb} ; [\mathsf{E}^\tau \cup \{W(i) \mid 1 \leq i < j\}])$.

Intuitively, for any future extension of execution graph with a sequence of events on τ, conditions C1 and C2 help in maintaining weak-coherence while C3 and C4 ensure that local-read-coherence is preserved. To assist readers, these conditions are depicted using diagrams in Figure 8 where the shaded area of τ represents a sequence of future events.

The simulation relation Υ is now defined as follows.

Definition 8. A state $\mathcal{B} \in \text{loLRA.}\mathtt{Q}$ *matches* an execution graph G, denoted by $\mathcal{B} \curlyvee G$, if there exists a function $tid_{\mathtt{RMW}} : \mathsf{W} \to \mathsf{Tid}$, such that: (1) for every $\tau \in \mathsf{Tid}$ and $L \in \mathcal{B}(\tau)$, there exists a $\langle G, \tau\rangle$-consistent $\langle G, L, tid_{\mathtt{RMW}}\rangle$-write-list, and (2) for every $\langle w, e\rangle \in G.\mathtt{rf} \mathbin{;} [\mathsf{RMW}]$, we have $\mathtt{tid}(e) = tid_{\mathtt{RMW}}(w)$.

Based on the simulation relation, we establish the equivalence of loLRA and opLRA. The proof, given in [32], shows that $\curlyvee$ constitutes a forward simulation from loLRA to opLRA, and $\curlyvee^{-1}$ constitutes a backward simulation from opLRA to loLRA.

Theorem 1. *The traces of lo*LRA *and the traces of op*LRA *coincide.*

6 Conclusion, Related and Future Work

We established the decidability of state reachability for finite-state programs under LRA, a memory model that lies strictly between WRA and RA. For that matter, we adapted the potential-based semantics of WRA from [22] to LRA, and showed that it meets the requirements for decidability of the WSTS framework.

In addition to the closely related work discussed in the introduction to this paper, the paper [14] studies the problem of verifying whether a given memory system provides causal consistency, which is a different verification problem than the one discussed in the current paper. The CC model in [14] (when restricted to single instruction transactions) is equivalent to (the RMW-free fragment of) WRA, whereas CCv from [14] is equivalent to SRA.

Another line of related work concerns *parametrized* programs, where one has an unknown number of threads but all of them run the same code. This arises a decidable verification problem under SC and TSO [5], but decidability of this problem is still unknown for WRA, SRA, and LRA. For the RMW-free fragment this problem is PSPACE for TSO [8] as well as for RA [19] (the latter result also allows a fixed number of distinguished threads running loop-free programs, possibly including RMWs).

An interesting direction for future work is to try to further close the gap between LRA and RA by introducing a restricted form of RA's modification order. A related problem that is still open (to the best of our knowledge) is whether the fragment of RA without RMWs induces a decidable verification problem. In addition, other models with undecidable reachability problems (such as the promising semantics [6] and the full POWER model [3]) may be bounded from below by decidable models.

Acknowledgements This work was supported by the Israel Science Foundation (grant number 814/22) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811).

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. The Bulletin of Symbolic Logic **16**(4), 457–515 (2010), http://www.jstor.org/stable/40961367
2. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.: Verification of programs under the release-acquire semantics. In: PLDI. pp. 1117–1132. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314649
3. Abdulla, P.A., Atig, M.F., Bouajjani, A., Derevenetc, E., Leonardsson, C., Meyer, R.: On the state reachability problem for concurrent programs under Power. In: NETYS. pp. 47–59. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-67087-0_4
4. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: The benefits of duality in verifying concurrent programs under TSO. In: CONCUR. LIPIcs, vol. 59, pp. 5:1–5:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), https://doi.org/10.4230/LIPIcs.CONCUR.2016.5
5. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: A load-buffer semantics for total store ordering. Logical Methods in Computer Science **Volume 14, Issue 1** (Jan 2018). https://doi.org/10.23638/LMCS-14(1:9)2018
6. Abdulla, P.A., Atig, M.F., Godbole, A., Krishna, S., Vafeiadis, V.: The decidability of verification under PS 2.0. In: ESOP. pp. 1–29. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_1
7. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA), 135:1–135:29 (Oct 2018). https://doi.org/10.1145/3276505
8. Abdulla, P.A., Atig, M.F., Rezvan, R.: Parameterized verification under TSO is PSPACE-complete. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371094
9. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: Algorithmic analysis of programs with well quasi-ordered domains. Information and Computation **160**(1), 109–127 (2000). https://doi.org/10.1006/INCO.1999.2843
10. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing **9**(1), 37–49 (1995). https://doi.org/10.1007/BF01784241
11. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752
12. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: ESOP. pp. 26–46. Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_2
13. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL. pp. 55–66. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1925844.1926394
14. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: POPL. pp. 626–638. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009888
15. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256**(1), 63 – 92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X
16. ISO/IEC 14882:2011: Programming language C++ (2011)
17. ISO/IEC 9899:2011: Programming language C (2011)

18. Kaiser, J.O., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: ECOOP. pp. 17:1–17:29. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.17
19. Krishna, S., Godbole, A., Meyer, R., Chakraborty, S.: Parameterized verification under release acquire is PSPACE-complete. In: PODC. pp. 482–492. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3519270.3538445
20. Lahav, O.: Verification under causally consistent shared memory. ACM SIGLOG News **6**(2), 43–56 (Apr 2019). https://doi.org/10.1145/3326938.3326942
21. Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: PLDI. pp. 211–226. ACM (2020). https://doi.org/10.1145/3385412.3385966
22. Lahav, O., Boker, U.: What's Decidable About Causally Consistent Shared Memory? ACM Trans. Program. Lang. Syst. **44**(2), 8:1–8:55 (2022), https://doi.org/10.1145/3505273
23. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL. pp. 649–662. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837643
24. Lahav, O., Margalit, R.: Robustness against release/acquire semantics. In: PLDI. pp. 126–141. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314604
25. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: ICALP. pp. 311–323. Springer-Verlag, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
26. Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: FM. LNCS, vol. 9995, pp. 479–495. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-48989-6_29
27. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI. pp. 618–632. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062352
28. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979)
29. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
30. Raad, A., Lahav, O., Vafeiadis, V.: On parallel snapshot isolation and release/acquire consistency. In: ESOP. pp. 940–967. Springer, Berlin, Heidelberg (2018). https://doi.org/10.1007/978-3-319-89884-1_33
31. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010). https://doi.org/10.1145/1785414.1785443
32. Singh, A.K., Lahav, O.: Decidable verification under localized release-acquire concurrency (extended version) (2024), https://www.cs.tau.ac.il/~orilahav/papers/tacas24full.pdf
33. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: POPL. pp. 209–220. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2676726.2676995

OxiDD

A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust[⋆]

Nils Husung[1](✉), Clemens Dubslaff[2], Holger Hermanns[1], and Maximilian A. Köhl[1]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
`{husung,hermanns,koehl}@cs.uni-saarland.de`
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
`c.dubslaff@tue.nl`

Abstract. *Decision diagrams* (DDs) are an important data structure in computer science with applications ranging from circuit design and verification to machine learning. Most prominently, *binary DDs* are commonly used to succinctly represent Boolean functions. Due to the practical importance of DDs, there is an ongoing quest for high-performance software libraries supporting the construction and manipulation of DDs. With OxiDD, we present a new framework for DDs that focuses on *safety*, *concurrency*, and *modularity*. Following a highly *modular design* we implement OxiDD in Rust, which facilitates the integration of various kinds of DDs such as MTBDDs, ZBDDs, and TDDs, all within safe code also in a concurrent setting. Already in its initial release, OxiDD does not compromise *performance*, which we show to be on par with or even better than established highly optimized DD libraries.

1 Introduction

Boolean functions play a central role in the design and analysis of computing systems. They frequently appear in different representations through logics, circuits, machine learning classifiers, or *binary decision diagrams* (BDDs) [1,12]. In particular, BDD representations are appealing as they are strongly normalizing and provide efficient operations such as applying Boolean operators, finding and counting satisfying assignments, or checking equivalence. Applications of BDDs encompass a wide range, including symbolic model checking and logic synthesis [13,15,26,20,16]. Much work on BDD research and implementations has been conducted during the first two decades after Bryant's seminal work [12]. This lead to various other types of *decision diagrams* (DDs) that extend the core

[⋆] OxiDD is open source and publicly available via `https://oxidd.net`.
This work was partially supported by the DFG under the projects TRR 248 (see `https://perspicuous-computing.science`, project ID 389792660) and EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy), as well as the NWO Veni grant VI.Veni.222.431.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 255–275, 2024.
https://doi.org/10.1007/978-3-031-57256-2_13

principles beyond Boolean functions or improving the efficiency for specific applications. Most prominently, *multi-terminal BDDs* (MTBDDs) [17,4] enable pseudo-Boolean function representations, *zero-suppressed BDDs* (ZBDDs) [32] usually provide more efficient representations for sparse sets than BDDs, or *list DDs* (LDDs) [9] efficiently encode transition vectors.

The most frequently used BDD libraries that are still considered state-of-the-art are BuDDy [27] and CUDD [40]. They originate from the 90s and do not fully exploit recent scientific advancements and modern design opportunities. Therefore, DDs and in particular BDDs gain more and more attention again, incorporating insights from satisfiability checking [8] but also providing advances in distributed and parallel computation and feature selection algorithms [18,39,7,23]. Sylvan [18] is a more recent BDD library that focuses on multithreaded operators, which however is also entirely written in C. Hence, it requires all memory management to be done manually, in particular challenging in the parallel setting. Manual resource management is one of the common sources for bugs that lead to *undefined behavior (UB)*, a situation where the programming language does not assign any semantics to the code. Consequences of UB are crashing programs or wrong results, the latter particularly being intolerable in verification tools or other critical applications where BDD libraries are commonly employed. Further, while existing libraries provide support for different kinds of BDDs such as MTBDDs or ZBDDs, the inherent lack of genericity in C required specifically tailored implementations. More elaborate extensions, e.g., towards *ternary decision diagrams* (TDDs) [38], would also require major internal changes in the library implementations.

In this paper, we develop a new DD framework, called *OxiDD*, to provide the basis for future developments in DD research and technology. As such, OxiDD focuses on easing the implementation of new DD types, providing reusable components commonly used in different kinds of DDs, and relying on modern technology. This leads to the following four major development goals for OxiDD: *safety*, *concurrency*, *modularity*, and *performance*.

By *safety*, we mean the absence of undefined behavior. *Concurrency* refers to thread-safety when used from multithreaded applications on the one hand. On the other hand, the framework itself should leverage multicore architectures for *performance*. *Modularity* should already be fulfilled by the nature of a framework, clearly separating concerns and enhancing extensibility. Here, clear interfaces should separate algorithms from data structures and allow to easily replace implementations of a component by another.

We tackle all the four development goals by implementing OxiDD in Rust, which is considered to be a safe programming language. Rust achieves safety via a rich type system but does not compromise performance: usually, Rust programs do not show any runtime overhead compared to C/C++. Furthermore, Rust allows us to define clear and generic interfaces, as well as efficient implementations of data structures. Also here, genericity does not come with any runtime overhead, as the compiler generates specialized code at compile time. For high performance, we opt into *Unsafe Rust*, a language syntactically separated from

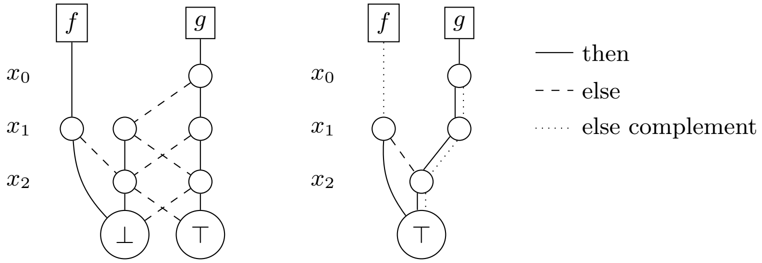
Library	Version	Last Release	Language	BDD	BCDD	MTBDD	ZBDD	LDD	TDD	Shared	Reordering	Thread Safe	Parallel Op.	Ext. Memory
Adiar [39]	1.2.2	2022/11	C++		✓		✓							✓
Biddy [31]	2.2.1	2022/12	C	✓	✓		✓			✓				
BeeDeeDee [28]	2.0	2017/09	Java	✓						✓		✓		
BuDDy [27]	2.4	2004	C	✓						✓	✓			
CAL [33]	2.1.1	2022/01	C		✓					✓	✓			✓
CUDD [40]	3.0.0	2015	C		✓	✓	✓			✓	✓			
JDD [41]	111	2019	Java	✓			✓			✓				
LibBDD [6]	0.5.10	2024/01	Rust	✓								✓		
PJBDD [7]	1.0.10	2021/07	Java	✓			✓			✓		✓	✓	
Sylvan [18]	1.8.1	2023/11	C		✓	✓		✓		✓		✓	✓	
OxiDD			Rust	✓	✓	✓	✓		✓	✓	(✓)	✓	✓	

Fig. 1: Popular DD libraries

Safe Rust using the `unsafe` keyword. Unsafe Rust enables a few additional operations whose safety cannot be checked by the compiler. Connecting Unsafe and Safe Rust requires safe abstractions upholding the central soundness property of Safe Rust: "No matter what, Safe Rust can't cause Undefined Behavior." [36] The art is to keep the portion of Unsafe Code as small as possible without violating the soundness property. One instance where we need Unsafe Rust is to support reordering of variables without node-wise locking. In this case, designing safe abstractions has been challenging. In the end, however, we gain both performance *and* implementations of all DD operations entirely in Safe Rust.

Contributions and Outline. We report on generic implementations of BDDs, MTBDDs, ZBDDs, and TDDs in OxiDD, focusing on implementation design and evaluating OxiDD's performance. Section 2 gives a detailed description of these DD types and enhancements. For working with these DDs from Rust, we provide high-level interfaces similar to those of existing libraries that—in contrast to those—cannot cause UB, and also provide C and C++ bindings. Section 3 goes into more detail about the framework's architecture and implementation details. We also point out some insights from tuning the data structures for performance. For this, we design safe abstractions, a highly non-trivial process we report on in Section 3.3. In Section 4, we finally evaluate the performance of OxiDD's BDD implementation. Our results show that OxiDD is on par with existing libraries, and even outperforms them in certain scenarios. This lets us conclude that in OxiDD, safety and modularity do not come at the expense of performance.

Further Related Work. For an overview comparing the features of popular and recently maintained BDD libraries, see Fig. 1. Here, BCDD refers to BDDs with complemented edges. The standard libraries BuDDy, CUDD, and Sylvan are widely used in several communities due to their manifold BDD manipulation operators and rich functionalities. Besides those, there are various other libraries that mostly provide specialized implementations. Biddy [31] mainly started as



(a) BDD **(b)** BCDD

Fig. 2: Example decision diagrams for Boolean functions $f, g\colon \mathbb{B}^3 \to \mathbb{B}$ where $f(x_0, x_1, x_2) = \neg(x_1 \vee x_2)$ and $g(x_0, x_1, x_2) = x_0 \leftrightarrow x_1 \leftrightarrow x_2$

an educational implementation but nowadays also supports a wide range of different BDD types such as *tagged BDDs* [19,14]. Java implementations such as JDD [41], BeeDeeDee [28], or PJBDD [7] provide better safety properties than C implementations, but usually cannot compete with performance. In case DDs grow beyond the size of the entire main memory, it becomes especially important to reduce the amount of random disk accesses. This is what the external memory libraries Adiar [39] and CAL [37] focus on. Development of CAL ceased back in 1996, but it was recently brought back to life in context of research on Adiar. Biodivine/LibBDD [6] is a notable BDD implementation in Rust and to the best of our knowledge the only Rust library besides OxiDD that supports existential and universal quantification. We are not aware of any DD implementation in the spirit of a modular framework that emphasizes safety as much as OxiDD does, while being concurrent and delivering high performance.

2 Background: Decision Diagrams and Rust

We recall kinds of DDs relevant for this paper, explain the role of variable orders and variable reordering, as well as preliminaries on safe abstractions in Rust.

2.1 Kinds of Decision Diagrams

Decision trees (DTs) are tree-like structures that represent functions through variable-labeled *decision nodes* and *terminal nodes* with function outcomes. Each path from the root to a terminal stands for assigning variables with values with the function outcome of the terminal. *Decision diagrams* (DDs) are rooted directed acyclic graphs that arise from DTs by merging isomorphic subtrees. We assume DDs to be *ordered*, i.e., variable occurrences follow a given total order on all paths in the DD. The order restriction may also be formulated by assigning each node a level, which we number from top to bottom. Then, a *variable order* σ is a bijection between the levels $0, \ldots, k-1$ and the k input variables.

```
fn apply_and(n: &Node, m: &Node) -> &Node {
  // "terminal cases"
  if n == m { return n; } if n == ⊥ || m == ⊥ { return ⊥; }
  if n == ⊤ { return m; } if m == ⊤ { return n; }
  if n.level < m.level { // n is above m
    level = n.level; t = apply_and(n.t, m);   e = apply_and(n.e, m);
  } else if n.level == m.level {
    level = n.level; t = apply_and(n.t, m.t); e = apply_and(n.e, m.e);
  } else { // n is below m
    level = m.level; t = apply_and(n, m.t);   e = apply_and(n, m.e);
  }
  return get_or_make_node(level, t, e);
}
```

Fig. 3: Apply algorithm for conjunctions (pseudocode)

Terminals are considered to be on a distinguished level ∞ at the bottom. Then, every node at level i can only have successor nodes at levels greater than i.

Binary DDs (BDDs). The most prominent kind of DDs are BDDs, used to represent Boolean functions $f\colon \mathbb{B}^k \to \mathbb{B}$ over $\mathbb{B} = \{\bot, \top\}$. They comprise terminal nodes $\top$ and $\bot$ as well as inner nodes n with outgoing “then” and “else” edges pointing to nodes n_t and n_e, respectively. By $n, m, ...$, we usually denote nodes and by $x_0, x_1, \ldots$ variables. BDDs are usually considered to be *reduced*, i.e., for any inner nodes n, m (1) $n_t \neq n_e$ and (2) if $\mathit{level}(n) = \mathit{level}(m)$, $n_t = m_t$, and $n_e = m_e$ then $n = m$. One major advantage of such BDDs is that they are strongly normalizing, i.e., they agree up to isomorphism for any Boolean function [21]. *Shared BDDs* associate function names with nodes, allowing for multiple functions to be represented in a single BDD structure. See Fig. 2a for an example of a (shared reduced) BDD with two functions f and g.

The semantics $[\![n]\!]$ of a BDD node n is recursively defined as a Boolean function. If n is a terminal, $[\![n]\!]$ is a constant function, mapping always to true if $n = \top$ or false if $n = \bot$, respectively. If n is an inner node at level i, then $[\![n]\!]$ is $(x_{\sigma(i)} \wedge [\![n_t]\!]) \vee (\neg x_{\sigma(i)} \wedge [\![n_e]\!])$, the Shannon decomposition of $[\![n]\!]$ w.r.t. $x_{\sigma(i)}$.

A BDD is typically created by successively applying Boolean connectives to already existing BDDs. As an example, the apply algorithm for conjunctions works as shown in Fig. 3. Here, it is assumed that the `get_or_make_node` function at the bottom also maintains reducedness, typically implemented using a hash table called *unique table* [11]. Note that the runtime of a naïve `apply_and` implementation is exponential in the number of variables of the functions represented by `n` and `m`. By applying memoization, the runtime can be reduced to $\mathcal{O}(|\mathtt{n}||\mathtt{m}|)$, where $|\cdot|$ denotes the count of descendant nodes. Memoization is typically implemented using a fixed-size cache called *apply cache* or *computed table*. The design of combining unique table and computed table towards an efficient BDD implementation was originally proposed by Brace et al. [11]. Besides apply algorithms based on recursion, there are also breadth-first apply algorithms implemented, e.g., in the BDD libraries CAL and Adiar [37,39].

Complement Edges. To reduce the number of nodes in a BDD and to support negation in shared BDDs in $\mathcal{O}(1)$, Brace et al. proposed *complement edges* as a new edge type in DDs [11]. We abbreviate BDDs that contain complement edges by BCDD. The semantics of a complemented edge pointing to a node n is just $\neg[\![n]\!]$. To recover a strong normal form, we remove the $\bot$ terminal node and impose the restriction that a "then" edge is never complemented. The latter forms, besides the two standard conditions on reduced BDDs, the third condition rendering BCDDs reduced. To ensure this condition, any node n whose "then" edge is complemented can be replaced by a node n' whose "then" edge is regular. The "else" edge of n' is the complement of the "else" edge of n such that $[\![n']\!] = \neg[\![n]\!]$. This means that all nodes that previously referred to n with a regular edge now have to use a complemented edge to n' and vice versa. This is the reason why—in contrast to the `apply_and` in Fig. 3—we formulate all algorithms based on edges (i.e., possibly tagged node references) rather than simple node references. Since functions f and $\neg f$ are represented by a single node, BCDDs may halve the number of nodes compared to BDDs.

Zero-Suppressed BDDs. A function $f\colon \mathbb{B}^k \to \mathbb{B}$ may also be interpreted as a characteristic function of a set $S = \{v \in \mathbb{B}^k \mid f(v) = 1\} \subseteq \mathbb{B}^k$. We can even view a Boolean vector as a subset of some "universe" U, so we also have $S \subseteq \mathcal{P}(U)$. For example, let $U = \{a, b\}$. The function a represents the set of all sets containing a, i.e., $\{\{a\}, \{a, b\}\}$. Conversely, the set $\{\{a\}\}$ is represented by the function $a \wedge \neg b$. This means that we can use BDDs to represent sets of Boolean vectors or sets of finite sets. If these sets are sparse, however, the corresponding BDD can be very large. Zero-suppressed BDDs (ZBDDs, ZDDs, or ZSDDs), which were introduced by Minato [32], are more apt for this use case. Like BDDs, ZBDDs have inner nodes with two outgoing edges we call `hi` and `lo` here. The terminal nodes are $\varnothing$ ("empty") and $\{\varnothing\}$ ("base"). Their semantics is just $[\![\varnothing]\!] = \varnothing$ and $[\![\{\varnothing\}]\!] = \{\varnothing\}$. For an inner node n at level i, we have $[\![n]\!] = [\![n_{\mathtt{lo}}]\!] \cup \{x_{\sigma(i)} \cup \alpha \mid \alpha \in [\![n_{\mathtt{hi}}]\!]\}$. To ensure reduced ZBDDs, a different first condition than BDDs is imposed: While for all nodes n in BDDs its children should represent different functions, i.e., (1) $n_t \neq n_e$, in ZBDDs we require that the node itself and the `lo`-node should represent different functions, i.e., (1') $n_{\mathtt{hi}} \neq \varnothing$.

Multi-Terminal BDDs (MTBDDs). While BDDs only contain two terminal nodes $\bot$ and $\top$, MTBDDs allow for arbitrary finitely many terminals [17]. Hence, MTBDDs can represent functions $\mathbb{B}^k \to S$, where S is an arbitrary set. A prominent application for MTBDDs is in symbolic probabilistic model checking [5] where $S = [0, 1]$. To allow such infinite sets, terminal nodes are usually created on demand, ensuring finiteness due to finitely many inner nodes of the MTBDD. MTBDDs are also known as *algebraic decision diagrams* (ADDs) [4].

Multivalued DDs (MDDs). Representing functions $D_0 \times \cdots \times D_{k-1} \to S$ imposes implementation challenges. For finite domains D_i we could rely on a binary encoding and resort to (MT)BDDs, then also called *finite domain decision diagram* (FDD). However, the properties of such FDDs heavily depend on the chosen bit-blasting encoding of the domains. As an alternative, MDDs directly encode multiple values as multiple outgoing edges [25]. Just like in MTBDDs, there is one terminal node per (used) value of S. *Ternary decision diagrams* (TDDs) may be viewed as one instance of MDDs, where $D_0 = \cdots = D_{k-1} = S = \{\bot, ?, \top\}$. That is, TDDs represent functions of three-valued logic [38].

2.2 Reordering

The size of a DD—no matter of which kind—may heavily depend on its variable order. There are functions $\mathbb{B}^k \to \mathbb{B}$ where different variable orders can lead to node counts in the class of $\Theta(2^k)$ but also $\Theta(k)$. Determining whether a variable order is suboptimal itself is an NP-complete problem [10], but there are heuristics to derive a good variable order from a (propositional) formula describing the function [34,2]. However, there are applications where such a formula is not available in advance. Furthermore, building the BDD for some intermediate result may require a different variable order than building the final BDD. In such cases, it is possible to reorder the existing DD, e.g., using Rudell's sifting algorithm [35]. The core of this algorithm is to pick a variable, try out all positions for it, and then move it to the best position. This procedure is repeated until no improvement is made.

There are various other reordering algorithms, but moving a variable to another position usually boils down to swapping all nodes of adjacent levels. Key characteristics of variable swap are that the semantics of nodes is preserved, and the operation can be performed in-place, i.e., locally. This is crucial, because nodes at levels i and $i + 1$ may be referenced by many nodes at higher levels. To explain the swap operator, we restrict ourselves to BDDs for simplicity. Let n be a node initially at level i where at least one of n_t and n_e is initially at level $i + 1$. The semantics of n then depends on both, the upper variable $x = \sigma(i)$ and the lower variable $y = \sigma(i + 1)$. Hence, n is essential at level i, redirecting the edge to n_t towards a node for $[\![n]\!][y := \top]$ (i.e., $[\![n]\!]$ with y set to true) and n_e towards a node for $[\![n]\!][y := \bot]$. If new children already existed and the old children have no incoming edges anymore, the node count decreases. Otherwise, it is well possible that the node count stays the same or even increases.

2.3 The Power of Safe Abstractions

Rust's central soundness property, "No matter what, Safe Rust can't cause Undefined Behavior" [36], is very powerful. In general, while software components may seem sound in isolation, their composition can still cause UB. This is because computer-checkable interface specifications, e.g., function types, are usually too limited to capture all conditions required to prevent UB.

For Safe Rust, the situation regarding UB—notably including data races—is different. Due to the soundness property, we can be sure that any composition of components either does not cause UB or is forbidden by the type system. While this translates to peace of mind for the user, it also requires a soundness argument for every piece of unsafe Rust code. For instance, the following unsafe code is unsound:

```
fn bad_deref(ptr: *const u32) -> u32 { unsafe { *ptr } }
```

Inside the `unsafe` block, we dereference a raw pointer, which is an unsafe operation. The unsafety arises from the fact that dereferencing a dangling pointer has no defined semantics. Now, we would need to argue why `ptr` cannot be dangling. But, any pointer can be passed to `bad_deref`, so the code is unsound.

To remedy this issue, the function must be marked `unsafe` as well, so that it cannot be called from Safe Rust. Note that this now requires the use of `unsafe` by the caller. To prevent the entire code base from becoming infected with `unsafe`, a *safe abstraction* is required. For instance, the `Box` type in Rust's standard library encapsulates a raw pointer and maintains the *safety invariant* that this pointer is always safe to dereference. As the pointer itself is inaccessible from the outside, this invariant cannot be violated and `Box` can thus provide a safe method for dereferencing it. The safety of this method is established entirely by *local reasoning* on the `Box` type and its safety invariant.

3 Architecture and Implementation

OxiDD's architecture is highly modular. In Rust, *crates* serve as counterparts to packages in languages such as Python, OCaml, or Haskell. OxiDD's implementation is split into multiple crates, to encapsulate functionality and expose a public versioned API. Fig. 4 shows how OxiDD is decomposed into separate crates and their dependencies on each other. Each crate has its own well-defined purpose. The architecture is centered around the `core` crate that mainly consists of trait definitions which formalize the key concepts of DDs. Traits are Rust's equivalent to interfaces or abstract classes in object-oriented programming. By using traits for abstracting from concrete implementations of key concepts, OxiDD achieves its high degree of modularity. Notably, there are no dependencies between algorithms and concrete implementations of data structures, all algorithms and data structures are written in a generic way. To provide end users with default implementations, e.g., towards the use of OxiDD as a BDD library, there is the `oxidd` crate, which assembles standards that have been shown useful in practice.

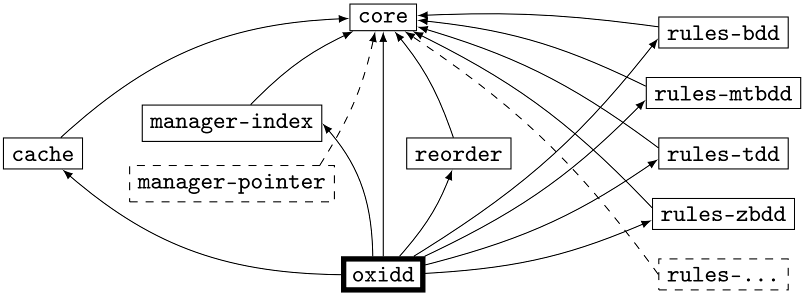


Fig. 4: OxiDD's architecture: dependency graph of the main crates.

3.1 The OxiDD Framework

Instead of being yet another DD library, its modular architecture is what makes OxiDD a *framework* (Fig. 4). Different implementations can be composed and swapped out for alternatives. All functionality has clear interfaces and can be separated into individually maintained and versioned crates. Third-party contributors can easily develop crates for novel kinds of DDs, core data structures, or reordering heuristics. Facilitated by OxiDD's abstractions, those crates will work seamlessly together, thereby making it ideal for future research on DDs. In this section, we provide further details on key concepts of this framework.

Manager. The *manager* is the data structure that stores all nodes of a DD and ensures their uniqueness via a *unique table* [11]. It also provides functionality for delayed *garbage collection* (GC), where the removal of nodes is delayed as far as possible. Early removal of nodes would lower performance, if nodes need to be recreated. An implementation of the manager trait also defines an *edge* type. An edge is a reference to a node, and may additionally have a *tag*. Tags are used, e.g., to mark edges as complemented in BCDDs. An *inner node* consists of its outgoing edges and optionally a level number. The latter is required for most kinds of DDs but can be omitted, e.g., in quasi-reduced BDDs.

OxiDD allows for different manager implementations. The `manager-index` crate contains a manager implementation that uses 32-bit unsigned integers to represent edges. These 32-bit are split into an index referencing a node and a tag. If 2^{32} nodes are too limiting for a use case, it is well possible to implement a different manager, e.g., one where nodes are referred to by pointers. In Fig. 4, this is indicated by the dashed `manager-pointer` box.

Cache. Typically, each manager has an associated *apply cache*, which is required by our recursive apply implementations for DD manipulation. Notably, the architecture of OxiDD is also open to other implementations, e.g., for a breadth-first apply algorithm (cf. [37,39]). The `cache` crate provides an apply cache as a fixed-size hash table. As with managers, alternative implementations of the apply cache are possible, and they can be freely composed with other implementations of the core infrastructure, e.g., managers.

Functions. Recall that shared DDs represent functions of various types (cf. Section 2), represented in a single data structure. In the graphical DD representation (cf. Fig. 2), functions correspond to the boxed fs and gs. From an implementation perspective, functions are an edge paired with a reference to the manager storing the respective node. For end users, functions provide a convenient interface for creating and manipulating DDs.

Support for Various Kinds of DDs. The apply algorithms for the different DD kinds are implemented in the crates starting with `rules`. Besides the reduction rules, these crates also define terminal node and edge-tag types. Depending only on the abstractions provided by the `core` crate, other kinds of DDs can easily be implemented. Notably, implementations are also shielded from UB as they can be implemented entirely in safe code.

```
1 let manager_ref = oxidd::bdd::new_manager(2048, 1024, 8);
2 let (x1, x2, x3) = manager_ref.with_manager_exclusive(|manager| {(
3       BDDFunction::new_var(manager).unwrap(),
4       BDDFunction::new_var(manager).unwrap(),
5       BDDFunction::new_var(manager).unwrap(),
6 )});
7 let res = x1.and(&x2)?.or(&x3)?;
8 println!("{}", res.satisfiable());
```

Fig. 5: Constructing a BDD for $(x_1 \wedge x_2) \vee x_3$ with OxiDD's API.

Reordering. OxiDD provides the fundamental mechanism of swapping levels in DDs for variable reordering (cf. Section 2). Currently, the `reorder` crate implements functionality to establish a given variable order, e.g., harmonize variable orders of different DDs or impose a static variable order heuristic. Support for dynamic reordering, e.g., via sifting [35], is planned for OxiDD's next release.

End User Ergonomics. While achieving a high degree of modularity through abstraction, this does not come at the expense of developer ergonomics for end users. Fig. 5 shows an example for constructing a manager, creating three variables, building the expression $(x_1 \wedge x_2) \vee x_3$, and then checking satisfiability. Here, 2048 and 1028 are the capacities of the manager for nodes and the apply cache, respectively, and 8 is the number of threads to use (see Line 1).

The method `with_manager_exclusive` is used to obtain exclusive access to the manager, required for creating variables. As existing libraries, OxiDD offers functions for applying operators (Line 7) or checking satisfiability (Line 8). Note that the interfaces provided by OxiDD shield from UB, whether caused by memory mismanagement or data races. Therefore, Fig. 5 does not contain a single line of unsafe code. The question marks `?` are part of Rust's mechanism for handling errors, which may happen, e.g., when running out of memory.

3.2 Design Choices and Defaults

Implementing OxiDD, we also focused on providing a good set of default implementations, selected and tuned for performance.

Node Store. The `manager-index` implements a store for inner nodes as an array, consisting of an initialized part followed by an uninitialized part. Each element of this array may either be a node along with a reference counter, a free slot with a reference to the next free slot, or uninitialized (see Fig. 6).

When creating a new node, we first check if the linked list of free slots contains an element. If yes, this element is removed from the list and the node is stored there. Otherwise, the first uninitialized slot is used. Should there be no uninitialized slot in the array, then we return an out-of-memory error. When deleting a node, we prepend the node's slot to the list of free slots.

In a concurrent setting, both the first-uninitialized index and the free slot list head are shared state requiring synchronization. To prevent contention, every worker thread gets its own first-uninitialized index and free slot list. Instead of incrementing the shared first-uninitialized index by 1, the worker pre-allocates the slots until the next multiple of 2^{16}. The free slot list is then split into multiple lists of (approximately) 2^{16} elements. The shared state maintains an array of these lists, while the workers have just one of these lists. If GC reaches 2^{16} nodes for a worker, the local list is moved to the shared state. The large lists avoid frequent synchronization with the shared state and thus contention.

Terminal nodes are managed independently of inner nodes. To distinguish between inner and terminal nodes, we split the 32-bit "address space" into two parts. The first N node IDs are used for terminal nodes, the remaining ones for inner nodes. The actual array index is the node ID minus N. For example, we set $N = 2$ in case of BDDs, where ID 0 is used for $\bot$, and ID 1 for $\top$. Determining the value of a terminal node does not require any memory operation here. For MTBDDs, however, we have to store terminal nodes in a separate array, similar to the inner node store described above.

Reference Counting. For GC, we use reference-counting instead of a mark-and-sweep method. One reason for this design decision is in level-local GC used for reordering. Iterating through the entire DD for mark-and-sweep GC is very expensive. It would be possible to only materialize reference counters during reordering and use mark-and-sweep GC otherwise (implemented, e.g., in BuDDy [27]). However, this does not resolve the following issue: GC must not remove any objects that are referenced by local variables. In languages like C, C++, and Rust, we cannot simply inspect the program stack. BuDDy resolves this issue using a second stack to register all locally referenced objects there. The problem is that accidentally forgetting the registration may lead to use-after-free bugs and ultimately UB. This would imply that apply algorithms need to be written in Unsafe Rust, which is undesirable. Some solutions to this problem have been discussed [22], but have no advantage over plain reference counting in case of DDs. Our preliminary benchmarks indicate that the amount of runtime spent on reference counting is in the order of 5 %. Given that mark-and-sweep GC would probably not be zero cost either, this seems acceptable.

Unique Table. The unique table is split into multiple hash tables, one per level. This split is useful for reordering, where we need to iterate over all nodes

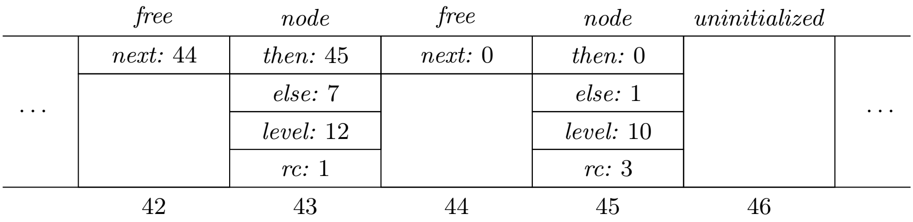


Fig. 6: Node store array (binary nodes with level).

on a level. Since we need to grow these tables on demand, we protect each table with a lock. The hash tables in use are designed with cache locality in mind. In particular, we use linear probing to resolve hash collisions. For space efficiency, the tables only contain IDs of the respective nodes, and not the nodes' outgoing edges. To improve performance when resizing the hash table, which normally requires rehashing all nodes, we store the hash next to the node ID. Thus, we can avoid rehashing any nodes. We further truncate the hash to 31 bits, so we can use the same 32-bit integer to mark the bucket as empty or as tombstone.

Apply Cache. For the apply cache, we use a fixed-size hash table. Each entry consists of the operator ID, a fixed-size array of operands, and the result of the operation. To synchronize accesses on the table, we use a spinlock per bucket. On usual lookups, we do not wait in case another thread has the lock, we rather recompute the entry. When inserting a new entry, we always replace a previously present entry in the bucket. We also experimented with bucket sizes larger than one entry and replacement policies such as *first in, first out* (FIFO), and *least frequently used* (LFU), but these turned out to be slower than the direct-mapped apply cache. One reason might be that in our benchmarks, we generally observed rather few cache hits (in the order of 20-30 %). Larger bucket sizes would require checking more entries before concluding that an entry is not contained in the cache. In addition, FIFO and LFU do not account for the different costs of operations. Ideally, the apply cache would merely keep those entries that take more time to recompute and are also used frequently. We plan to investigate such a strategy in more detail in future work. Notably, such experiments are facilitated by the modular architecture of OxiDD.

A particular important optimization is to elide reference counter updates when inserting or removing entries from the apply cache. This is due to referenced nodes rarely being in the CPU cache. Eliding reference counter updating implies that we must ensure that no nodes are deleted while referenced from the apply cache. Nodes can only be deleted during GC and reordering. Since a GC may run in background, we lock and empty all buckets of the apply cache prior to the GC. Only after the GC, we unlock the buckets again.

Concurrent Apply Algorithms. OxiDD has recursive apply algorithms, both in a single-threaded and a concurrent version. The concurrent version uses task-based parallelism with work-stealing, similarly to Sylvan [18]. The idea is to execute the recursive calls (cf. Fig. 3) concurrently. For the implementation, we use the `rayon` crate [30]. As splitting the work into tasks comes with a runtime overhead (in the order of +35 %), we only split the tasks until a certain recursion depth. From then on, we use the single-threaded apply algorithm.

3.3 Safe Abstraction for Modifying Nodes

A challenge when designing OxiDD was to find a safe abstraction for modifying nodes, e.g., during reordering, as it requires synchronization. A lock per node would lead to incorrect results when accessing nodes subject to a level swap, and

```
1  func_a.with_manager_shared(|manager, edge_a /* &Edge<'1> */| {
2      let edge_b /* &Edge<'1> */ = func_b.as_edge(manager);
3      let edge_res /* Edge<'1> */ = apply_and(manager, edge_a, edge_b);
4      BDDFunction::from_edge(manager, edge_res)
5  })
6  // Mixing branded types leads to a compiler error.
7  func_a.with_manager_shared(|manager_a, edge_a /* &Edge<'1> */| {
8      func_b.with_manager_shared(|manager_b, edge_b /* &Edge<'2> */| {
9          let edge_res = apply_and(manager_a, edge_a, edge_b); // <-- Error
10         BDDFunction::from_edge(manager_a, edge_res)
11     })
12 })
```

Fig. 7: Usage of branded types.

moreover be diametral for performance. Instead, we use a single read/write lock to coordinate exclusive access to the entire DD. A shared append-only view is sufficient for apply algorithms and most other operations such as model counting or satisfiability checking. Reordering requires exclusive access.

Once exclusive access is acquired, we must ensure that all nodes we modify actually belong to the respective manager we have exclusive access on. To this end, a safety invariant is required: All descendants of a node are stored in the same manager. This is a very natural assumption, also needed for correctness, avoiding a node in manager A to reference a node of manager B. As this invariant is needed for safety, there must not be a way to violate it from Safe Rust. The challenge is that when creating a node, there is no efficient way to check the invariant. After all, we only work with edges here, and edges do not (necessarily) provide any information about the manager the node belongs to. Only the function type stores both a node reference and a reference to the respective manager. So, before actually starting an apply operation, we must ensure that the operands (of function type) belong to the same manager, and the entire code in between needs to uphold the invariant. In a naïve implementation, without a proper abstraction, this would require a lot of unsafe code.

We can drastically reduce the amount of unsafe code if every manager has its own edge and node types, as this prevents mixing edges from different managers. To realize this idea without fixing the number of managers upfront, we use branded types as presented by Yanovski et al. [42]. Branded types leverage Rust's lifetimes. In Rust, a reference is essentially a pointer with the invariant that it is always safe to dereference. As references may point to stack variables, the compiler needs to make sure that the referenced variables do not go out of scope as long as the reference is live. This is done by adding a lifetime to reference types. The lifetime corresponds to the referenced variable's scope.

As an example, computing the conjunction of functions `func_a` and `func_b` works as in Lines 1-5 of Fig. 7. The `with_manager_shared` method acquires the lock (for shared access) of the manager referenced by `func_a`. Further, it takes a closure to which it passes the manager reference and edge. This is the place where the new brand/lifetime is introduced. We denote it as `'1` in the comment. When converting func_b into its underlying edge in Line 2, we check that it belongs to manager. If this is not the case, we abort the execution with an appropriate error message. Otherwise, we obtain an edge of the same branded type as edge_a. This means that when calling the recursive apply_and function, it can safely assume that the nodes referenced by edge_a and edge_b, as well as all their descendants are stored in the same manager. This simply follows from type safety. As the branded type is only valid inside the closure, we convert the resulting edge back into a function in Line 4. Notably, if we nest with_manager_shared calls as shown in Lines 6-12 of Fig. 7, we get a compile time error because the types of edge_a and edge_b have different brands. This safe abstraction enables the implementation of apply algorithms entirely within safe code.

4 Evaluation

OxiDD is designed not only for modularity and safety, but also with performance in mind. We (mostly) use zero cost abstractions and eliminate runtime checks via type invariants. Our evaluation is driven by two research questions:

RQ1 How does the single-threaded runtime of OxiDD compare to other popular BDD libraries?
RQ2 Can OxiDD achieve similar speed as Sylvan in the multithreaded setting?

As the set of libraries we compare against, we choose BuDDy 2.4, CUDD 3.0.0, and Sylvan 1.8.0 since these are the most popular libraries. Furthermore, we compare against LibBDD 0.5.10, a relatively mature Rust library, and Adiar (commit ca4f7351), which apparently is the most performant external memory library in the large scale. The version of OxiDD corresponds to commit 8113c12. Among this set of libraries, Sylvan and OxiDD are the only multithreaded libraries. For a fair comparison, we integrated OxiDD into the bdd-benchmark framework[3] initially developed by Steffan Sølvsten for the evaluation of Adiar [39]. It contains the following set of combinatorial and verification benchmarks:

- N-Queens: Given $N \in [12, 15]$, how many ways are there to place N queens on an $N \times N$ chess board without threatening each other?
- Tic-Tac-Toe: Given $N \in [20, 24]$, how many ways are there for player 1 to place N crosses in a 3D $4 \times 4 \times 4$ cube and tie if player 2 places naughts in all remaining positions?
- Picotrav: Given a hierarchical circuit, a BDD is constructed for each output. We use this to verify the equality of two circuits. In our case, the circuits are a subset of the EPFL combinational benchmark suite [3].

Input sizes and files are selected based on preliminary experiments regarding resource consumption. Note that complement edges are not beneficial for N-Queens and Tic-Tac-Toe: Negations occur on variables only, the remaining operations are just conjunctions and disjunctions. This is different for Picotrav.

[3] github.com/SSoelvsten/bdd-benchmark, our version is available at Zenodo [24].

`bdd-benchmark` is designed in a way that is generic over the respective BDD library. All benchmarks are written against an abstract adapter that provides operations such as conjunction, disjunction, and negation in case of BDDs. This means that the same operations are executed with the same variable order, regardless of the DD library in use. In particular, dynamic reordering is disabled. Note that `bdd-benchmark` is written in C++, so OxiDD's adapter makes use of the C++ bindings. All libraries except BuDDy use complemented edges. Only OxiDD implements both BDDs and BCDDs, as the genericity easily allows us to do so. Since both implementations are based on the same data structures, we also get a relatively good estimate of the performance impact complemented edges have. For the remainder of this section, we use "OxiDD" to refer to the BDD implementation and explicitly add "BCDD" otherwise.

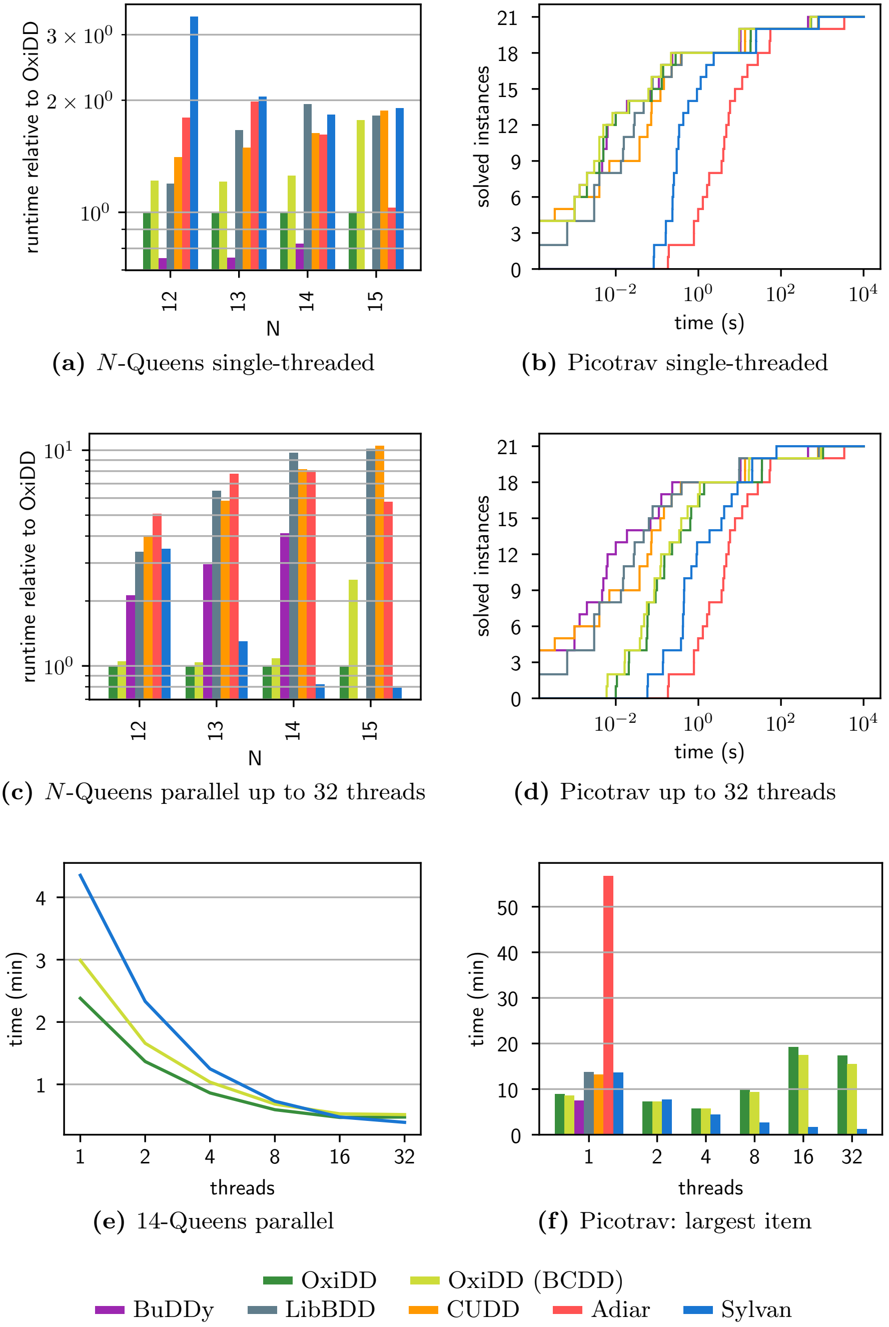
We executed the benchmarks on a 16 core / 32 thread *AMD Ryzen 9 5950X* CPU with 128 GiB of RAM and approximately 800 GiB free SSD space, running Ubuntu 22.04 (Linux kernel 5.15). The libraries were compiled using Clang 16.0.6 or rustc 1.71.1, which are both based on LLVM 16. We set a timeout of 3 hours. To reduce the number of TLB misses during execution, we enabled transparent hugepages by setting `/sys/kernel/mm/transparent_hugepage/enabled` to `always`. The default on many systems is that programs have to issue respective `madvise` calls. OxiDD is the only library that does this to some extent. The performance impact of this setting is quite large: In preliminary experiments we observed a 1.6× speedup for 14-Queens with BuDDy. We ran each benchmark three times and report the average running times.

4.1 RQ1: Single-thread Performance

Overall, our benchmarks show that for single-threaded execution, BuDDy performs best. OxiDD is slightly slower and faster than all other libraries. In Fig. 8a, we show the runtimes on the N-Queens benchmark relative to OxiDD. For $N = 12$, OxiDD takes 4.2 s, 24.6 s for $N = 13$, 2.4 min for $N = 14$, and 16.1 min for $N = 15$. On 15-Queens, OxiDD performs best. BuDDy runs out of memory, mainly due to its limitation to $2^{31} - 1$ nodes. As the BDD construction produces more than 2^{31} nodes, this only works with sufficiently many GCs. OxiDD (BCDD) is restricted to 2^{31} nodes (the last bit is needed for complement edges), and the GCs cause OxiDD (BCDD) to be much slower than OxiDD in this specific benchmark instance. Still, OxiDD (BCDD) is faster than CUDD, Sylvan, and LibBDD. For this problem size, breadth-first apply algorithms also start to shine. Adiar is only 1.03× slower than OxiDD.

The situation is very similar for the Tic-Tac-Toe problem. For Picotrav, however, complement edges may have a notable impact on the node count. On many instances, the BCDD variant of OxiDD performs slightly better than its BDD variant and BuDDy, see Fig. 8b. All libraries solved the smallest 21 out of 23 instances, the remaining two timed out or ran out of memory.

So with respect to RQ1, we can say that OxiDD is among the best libraries. However, a manager implementation that is not restricted to 2^{31} or 2^{32}, respectively, might be interesting for some use cases.



(a) N-Queens single-threaded

(b) Picotrav single-threaded

(c) N-Queens parallel up to 32 threads

(d) Picotrav up to 32 threads

(e) 14-Queens parallel

(f) Picotrav: largest item

Fig. 8: N-Queens and Picotrav benchmark statistics

4.2 RQ2: Multi-thread Performance

From Fig. 8e, we observe that OxiDD's parallelization is already effective in its initial release. However, for increasing number of threads, Sylvan performs better. This is probably due locking on each level in the unique table of OxiDD leads to contention. 14-Queens has 196 variables/levels, so it is not that unlikely that two out of 32 threads try to acquire the same lock. Notably, OxiDD's performance for 32 threads is slightly worse than for 16 threads. Especially for the smaller Picotrav instances (cf. Fig. 8d), we also observe a significant slowdown using 32 threads. Sylvan shows a slowdown as well, but not as serious as OxiDD. Only for the largest solved instance, Sylvan has a significant speedup of 10.5× for 32 threads (cf. Fig. 8f).

Regarding RQ2, we conclude that Sylvan's highly optimized parallel engine leads to better performance on a high numbers of threads. In large combinatorial problems with at most 16 threads, OxiDD's parallelization outperforms Sylvan's. For the verification problems we tested, the current implementation does not achieve parallel speedups. Still, we remark that OxiDD in single-threaded execution outperforms the multithreaded Sylvan significantly in all but one Picotrav instance. Note that OxiDDs parallelization can still be optimized, e.g., by using concurrent hash tables countering contention issues mentioned (cf. Section 3.2).

5 Conclusion

In this paper, we have presented OxiDD, a new decision diagram framework in Rust. OxiDD emphasizes on modularity, which eases extension on functionalities and new kinds of decision diagrams. Our implementations benefit from high performance and can safely be used in concurrent contexts. Depending on the workload, there may also be significant speedups in multithreaded execution. We demonstrated this by comparing OxiDD's B(C)DD implementations to other popular BDD libraries. Moreover, we showed how we can leverage Rust's type system to ensure that edges from different managers cannot accidentally be mixed up. This allowed us to implement the building blocks for dynamic reordering while keeping the apply algorithms entirely in Safe Rust.

Aiming at the basis for future research and developments, there are plenty of opportunities. First, OxiDD's B(C)DD, MTBDD, and ZBDD implementations are not yet as feature-rich as matured BDD packages such as CUDD. Adding the remaining operations is, however, facilitated by our modular design. Second, we pointed out that the current unique table is likely to be a bottleneck for concurrent performance. Recently, there have been interesting developments on growing concurrent hash tables [29], which we plan to further investigate. Third, we plan to implement dynamic reordering heuristics relying on our reordering building blocks presented here. Last but not least, the argument that our unsafe code upholds Rust's invariant is currently informal. Formally verifying OxiDD would be a challenging but rewarding avenue to pursue.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Transactions Computers **27**(6), 509–516 (Jun 1978). https://doi.org/10.1109/TC.1978.1675141
2. Aloul, F.A., Markov, I.L., Sakallah, K.A.: MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation. Journal of Universal Computer Science **10**(12), 1562–1596 (2004). https://doi.org/10.3217/jucs-010-12-1562
3. Amarú, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. In: Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS) (2015), https://infoscience.epfl.ch/record/207551
4. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design. pp. 188–191. ICCAD '93, IEEE Computer Society Press, Washington, DC, USA (1993). https://doi.org/10.1109/ICCAD.1993.580054
5. Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M., Ryan, M.: Symbolic model checking for probabilistic processes. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) Automata, Languages and Programming. pp. 430–440. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
6. Benes, N., Brim, L., Kadlecaj, J., Pastva, S., Šafránek, D.: AEON: attractor bifurcation analysis of parametrised boolean networks. 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 I. Lecture Notes in Computer Science, vol. 12224, pp. 569–581. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_28
7. Beyer, D., Friedberger, K., Holzner, S.: PJBDD: A BDD library for Java and multi-threading. In: Hou, Z., Ganesh, V. (eds.) Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12971, pp. 144–149. Springer (2021). https://doi.org/10.1007/978-3-030-88885-5_10
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam, The Netherlands (2009)
9. Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigün, H. (eds.) Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5160, pp. 81–95. Springer (2008). https://doi.org/10.1007/978-3-540-85762-4_6
10. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Transactions on Computers **45**(9), 993–1002 (1996). https://doi.org/10.1109/12.537122
11. Brace, K., Rudell, R., Bryant, R.: Efficient implementation of a BDD package. In: 27th ACM/IEEE Design Automation Conference. pp. 40–45 (1990). https://doi.org/10.1109/DAC.1990.114826
12. 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

13. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (Sep 1992). https://doi.org/10.1145/136035.136043
14. Bryant, R.E.: Chain reduction for binary and zero-suppressed decision diagrams. Journal of Automated Reasoning **64**(7), 1361–1391 (2020). https://doi.org/10.1007/s10817-020-09569-6
15. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inform. and Comp. **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A
16. Cimatti, R., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, O.: NuSMV 2: An opensource tool for symbolic model checking. In: Proceedings of the 14th Conference on Computer Aided Verification (CAV). vol. LNCS:2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
17. Clarke, E.M., Fujita, M., McGeers, P.C., McMillan, K.L., Yang, J.C., Zhao, X.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In: Proc. International Workshop on Logic & Synthesis (1993)
18. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer **19**(6), 675–696 (2017). https://doi.org/10.1007/s10009-016-0433-2
19. van Dijk, T., Wille, R., Meolic, R.: Tagged BDDs: Combining reduction rules from different decision diagram types. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017. pp. 108–115. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102248
20. Drechsler, R., Becker, B.: Binary Decision Diagrams - Theory and Implementation. Springer (1998). https://doi.org/10.1007/978-1-4757-2892-7
21. Fortune, S., Hopcroft, J., Schmidt, E.M.: The complexity of equivalence and containment for free single variable program schemes. In: Ausiello, G., Böhm, C. (eds.) Automata, Languages and Programming. pp. 227–240. Springer Berlin Heidelberg, Berlin, Heidelberg (1978). https://doi.org/10.1007/3-540-08860-1_17
22. Goregaokar, M.: Designing a GC in Rust (2015), http://web.archive.org/web/20230714074109/https://manishearth.github.io/blog/2015/09/01/designing-a-gc-in-rust/
23. Harder, H., Jantsch, S., Baier, C., Dubslaff, C.: A unifying formal approach to importance values in Boolean functions. In: Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China. pp. 2728–2737. ijcai.org (2023). https://doi.org/10.24963/ijcai.2023/304
24. Husung, N., Dubslaff, C., Hermanns, H., Köhl, M.A.: OxiDD: Artifact (2024). https://doi.org/10.5281/zenodo.10578461
25. Kam, T., Villa, T., Brayton, R.K.: Multi-valued decision diagrams: theory and applications (1998), https://api.semanticscholar.org/CorpusID:53828281
26. Lai, Y.T., Sastry, S., Pedram, M.: Boolean matching using binary decision diagrams with applications to logic synthesis and verification. In: Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors. pp. 452–458 (1992). https://doi.org/10.1109/ICCD.1992.276313
27. Lind-Nielsen, J.: BuDDy: A binary decision diagram package, version 2.4 (2004), https://buddy.sourceforge.net/manual/

28. Lovato, A., Macedonio, D., Spoto, F.: A thread-safe library for binary decision diagrams. In: Giannakopoulou, D., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 35–49. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_4
29. Maier, T., Sanders, P., Dementiev, R.: Concurrent hash tables: Fast and general(?)! ACM Trans. Parallel Comput. **5**(4), 16:1–16:32 (2019). https://doi.org/10.1145/3309206
30. Matsakis, N., Stone, J.: Rayon (2023), https://docs.rs/rayon/1.8.0/rayon/
31. Meolic, R.: The Biddy BDD package. Journal of Open Source Software **4**(34), 1189 (2019). https://doi.org/10.21105/joss.01189
32. Minato, S.i.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of the 30th International Design Automation Conference. pp. 272–277. DAC '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/157485.164890
33. Ranjan, R., Gosti, W., Brayton, R., Sangiovanni-Vincenteili, A.: Dynamic reordering in a breadth-first manipulation based bdd package: challenges and solutions. In: Proceedings International Conference on Computer Design VLSI in Computers and Processors. pp. 344–351 (1997). https://doi.org/10.1109/ICCD.1997.628893
34. Rice, M., Kulhari, S.: A survey of static variable ordering heuristics for efficient bdd/mdd construction. University of California, Tech. Rep p. 130 (2008)
35. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD). pp. 42–47 (1993). https://doi.org/10.1109/ICCAD.1993.580029
36. Rust Foundation: The Rustonomicon (2023), http://web.archive.org/web/20230918045612/https://doc.rust-lang.org/nomicon/intro.html
37. 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
38. Sasao, T.: Ternary decision diagrams: Survey. In: 27th IEEE International Symposium on Multiple-Valued Logic, ISMVL 1997, Antigonish, Nova Scotia, Canada, May 28-30, 1997, Proceedings. pp. 241–252. IEEE Computer Society (1997). https://doi.org/10.1109/ISMVL.1997.601404
39. Sølvsten, S.C., van de Pol, J.: Adiar 1.1 - zero-suppressed decision diagrams in external memory. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13903, pp. 464–471. Springer (2023). https://doi.org/10.1007/978-3-031-33170-1_28
40. Somenzi, F.: CUDD: CU decision diagram package. Tech. rep., University of Colorado at Boulder (2015)
41. Vahidi, A.: JDD: A pure Java BDD and Z-BDD library (2003), https://bitbucket.org/vahidi/jdd
42. Yanovski, J., Dang, H., Jung, R., Dreyer, D.: GhostCell: separating permissions from data in Rust. Proc. ACM Program. Lang. **5**(ICFP), 1–30 (2021). https://doi.org/10.1145/3473597

Verification under TSO with an infinite Data Domain

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Florian Furbach[2](✉), and Shashwat Garg[3]

[1] Uppsala University, Uppsala, Sweden
[2] Technical University of Denmark, Kongens Lyngby, Denmark
fwafu@dtu.dk
[3] Indian Institute of Technology Bombay, Mumbai, India

Abstract. We examine verification of concurrent programs under the total store ordering (TSO) semantics used by the x86 architecture. In our model, threads manipulate variables over infinite domains and they can check whether variables are related for a range of relations. We show that, in general, the control state reachability problem is undecidable. This result is derived through a reduction from the state reachability problem of lossy channel systems with data (which is known to be undecidable). In the light of this undecidability, we turn our attention to a more tractable variant of the reachability problem. Specifically, we study context bounded runs, which provide an under-approximation of the program behavior by limiting the possible interactions between processes. A run consists of a number of contexts, with each context representing a sequence of steps where a only single designated thread is active. We prove that the control state reachability problem under bounded context switching is PSPACE complete.

1 Introduction

Over the years, research on concurrent verification has been chiefly conducted under the premise that the threads run according to the classical Sequential Consistency (SC) semantics. Under SC, the threads operate on a set of shared variables through which they communicate *atomically*, i.e., read and write operations take effect immediately. In particular, a write operation is visible to all the threads as soon as the writer thread carries out its operation. Therefore, the threads always maintain a uniform view of the shared memory: they all see the latest value written on any given variable and we can interpret program runs as interleavings of sequential thread executions. Although SC has been immensely popular as an intuitive way of understanding the behaviours of concurrent threads, it is not realistic to assume computation platforms guarantee SC anymore. The reason is that, due to hardware and compiler optimizations, most modern platforms allow more relaxed program behaviours than those permitted under SC, leading to so-called *weak memory models*. Weakly consistent platforms are found at all levels of system design such as multiprocessor architectures (e.g.,

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 276–295, 2024.
https://doi.org/10.1007/978-3-031-57256-2_14

[33,32]), Cache protocols (e.g., [31,19]), language level concurrency (e.g., [24]), and distributed data stores (e.g., [17]). Program behaviours change dramatically when moving from the SC semantics to weaker semantics. Therefore, in recent years, research on the verification of concurrent programs under weak memory models have started to become popular. A classical example of weak memory models is the Total Store Ordering (TSO) semantics which is a formalization of the Intel x86 processor architecture [29]. The TSO semantics inserts an unbounded FIFO buffer, called the *store buffer*, between each thread and the main memory. When a thread performs a write instruction, the corresponding operation is appended to the end of the buffer, and hence it is not immediately visible to other threads. The write messages are non-deterministically propagated from the store buffer of a given thread to the shared memory. Verification of programs that contain data races needs to take the underlying memory model into account. This is crucial in hardware-close programming, especially in concurrent libraries or kernels. Such applications are inherently racy; exploiting racy WMM operations for efficiency is standard practice. Our work serves as a foundation for ensuring the correctness of such systems, which often rely on these intricate memory models to achieve optimal performance.

In a parallel development, significant research has been done on extending model checking frameworks to programs with infinite state spaces. There are two main reasons why a program might have an infinite state space. The first is that the program has unbounded control structures, which means it can have an unbounded number of threads. Examples include parameterized systems, in which correctness of the system is checked regardless of the number of threads, and programs that allow dynamic thread creation through spawning [11]. Secondly, the program may operate on unbounded data structures, such as clocks [12], stacks [16], and queues ([10,1]). These works, including their extensions, have been done under the SC assumption. Although recent works have started to explore parameterized verification for weak memory models [6,4,22], the verification of programs that operate on a shared unbounded data structure with weak memory semantics has remained unexplored until now.

In this paper, we combine infinite-state programs with weak memory models: we study the decidability and complexity of the reachability problem for programs operating on unbounded data structures under the TSO semantics. While the TSO semantics has been extensively studied (e.g., [15,5]), it has been assumed that the data domain is finite. This means that the possible values of a shared variable or a register are bounded. In contrast, our model allows for an infinite domain such as natural numbers $\mathbb{N}$ or real numbers $\mathbb{R}$. It contains register assignments, an operator that may assign an arbitrary value to a register, and a set of relations that act as guards. We focus on relations equality and "greater than" on totally ordered sets and combinations, negations and inversions of them. Our model finds practical utility in continuously running concurrent protocols. A prime example is the bakery ticket protocol used in various scenarios. It is presented in Section 4. Here, an unbounded number of requests occur, each assigned increasing numbers and the lowest-numbered request is serviced. This presents a scenario with inherent races that requires an infinite domain which our model can effectively capture. Note that our model is infinite in multiple dimensions: the threads are infinite-state as they operate on unbounded data domains, the store buffers are unbounded, and they carry write-messages over an unbounded domain.

In order to perform safety verification, we need to decide whether there is an execution that can reach some undesirable control state. We study the control state reachability problem and show that for many domains and relations, it is undecidable. Therefore, we propose an alternative approach by introducing an under-approximation schema using context-bounding [30,28,25,23,14]. Context-bounding has been proposed in [30] as a suitable approach for efficient bug detection in multithreaded programs. Indeed, for concurrent programs, a bounding concept that provides both good coverage and scalability must be based on aspects related to the interactions between concurrent components. It has been shown experimentally that concurrency bugs usually show up after a small number of context switches [28]. In this work, we study a context bounded analysis where only the active thread may perform an operation and update the memory. We show that in this case, the state reachability problem is not only decidable, but even PSPACE complete. To this end, we perform a two-step abstraction that employs insights about context bounded runs of TSO semantics as well as the structure of reachable configurations.

In the first step of our abstraction process, we refine the methods introduced by [14]. Their construction introduces a code-to-code translation that abstracts the buffer, simplifying the problem to state reachability under SC. Our approach leverages the fact that this abstraction does not explicitly depend on variable values. In our case, the abstraction step yields a register machine where the register values are integers or real numbers, and the transitions are conditioned by “gap-constraints" [9,18,27]. Gap constraints serve to identify, within each system configuration, (i) the variables with identical values and (ii) the gaps (differences) between variable values. Notably, these gaps can be arbitrarily large. The papers [9,18,27] analyze programs with gap constraints within the framework of well-structured systems [8,20]. As a result, they do not provide upper bounds on the complexity.

As another key contribution of this paper, we propose a method to achieve PSPACE completeness. The fundamental idea behind our algorithm is that for any system execution, there is an alternative execution with larger gaps among the variables. This implies that we do not need to explicitly track the gaps between variables, as is the case in [9,18,27]. Instead, we implement a second (precise) abstraction step, focusing solely on the order of variables. For any pair of variables x and y, we record whether $x = y$, $x < y$, or $x > y$.

2 Related Work

Not much current work considers the complexity and decidability of infinite-state state programs on weak memory models. Furthermore, most existing works consider parameterized verification rather than programs with infinite data domains. The paper [6] considers parameterized verification of programs running under TSO, and shows that the reachability problem is PSPACE complete. However, the work assumes that the threads are finite-state and, in particular, the threads do not manipulate unbounded data domains. The paper [22] shows PSPACE completeness when the underlying semantics is the Release-Acquire fragment of C11. The latter semantics gives rise to a different semantics compared to TSO. The paper also considers finite-state threads.

In [2], parameterized verification of programs running under TSO is considered. However, the paper applies the framework of well-structured systems where the buffers of the threads are modelled as lossy channels, and hence the complexity of the algorithm is non-primitive recursive. In particular, the paper does not give any complexity bounds for the reachability problem (or any other verification problems). The paper [15] considers checking the robustness property against SC for parameterized systems running under the TSO semantics. However, the robustness problem is entirely different from reachability and the techniques and results developed in this work cannot be applied in our setting.

The paper [4] considers parameterized verification under the TSO semantics when the individual threads are infinite-state. However, the authors study a *restricted* model, where it assumes that (i) all threads are identical and (ii) the threads do not use atomic operations. Generally, parameterized verification for the restricted model is easier than non-parameterized verification. For instance, in the case of TSO where the threads are finite-state, the restricted parameterized verification problem is in PSPACE [6] while the non-parameterized problem has a non-primitive recursive complexity [13].

The are many works on extending infinite-state systems with unbounded data domains. Well studied examples are Petri nets with data tokens [27], stacks with unbounded stack alphabets [7], and lossy channel systems with unbounded message alphabets [1]. All these works assume the SC semantics and are hence orthogonal to this work.

3 Total Store Order (TSO)

Let $\mathbb{B} = \{true, false\}$. Given a function $f : A \to B$ with $a \in A, b \in B$, $f[a \leftarrow b]$ is defined as follows: $f[a \leftarrow b](a) := b$, $f[a \leftarrow b](a') := f(a')$ for any $a' \in A$ with $a' \neq a$. We write $x \in w$ for letter $x \in \Sigma$ occurring in word $w \in \Sigma^*$ and $w' \leq w$ for $w' \in \Sigma^*$ being a subsequence of w.

Let x and y be two natural (real) numbers. Let $n \in \mathbb{N}$, we use $x <_n y$ (resp. $\leq_n y$) to denote that $x + n < y$ (resp. $x + n \leq y$). A data theory is defined by a pair $(\mathtt{D}, \mathsf{Rl})$ where $\mathtt{D}$ is an infinite data domain and $\mathsf{Rl} \subseteq \mathtt{D} \times \mathtt{D} \to \mathbb{B}$ is a finite set of relations over $\mathtt{D}$. In this paper, we restrict ourselves to the set of natural/real numbers as data domain, and the set of relations Rl to be a subset of $\mathsf{Rl}_{\leq n} = \{=, \neq, <, \leq, <_n, \leq_n | \; n \in \mathbb{N}\}$. We assume w.l.o.g. that $0 \in \mathtt{D}$.

Transition Systems A labelled transition system is a tuple $\mathcal{TS} = (\Gamma, \mathcal{L}, \mathcal{T}, \gamma_{\mathsf{init}})$ that consists of a set of *configurations* Γ, a finite set of labels $\mathcal{L}$, a labelled transition relation $\mathcal{T} \subseteq \Gamma \times \mathcal{L} \times \Gamma$, and an initial configuration $\gamma_{\mathsf{init}} \in \Gamma$. We write $\gamma \xrightarrow{\ell} \gamma'$ for $\langle \gamma, \ell, \gamma' \rangle \in \mathcal{T}$. We say that $\pi = t_1 \ldots t_n \in \mathcal{T}^*$ is a run of $\mathcal{TS}$ if there is a sequence of configurations $\gamma_1, \gamma_2, \ldots, \gamma_{n+1}$ such that $t_i = \gamma_i \xrightarrow{\ell_i} \gamma_{i+1}$ for $i \leq n$ and $\gamma_1 = \gamma_{\mathsf{init}}$. The run π ends in configuration γ_{n+1}. We say that γ is reachable if there is a run π of $\mathcal{TS}$ that ends in γ.

Programs A concurrent program Prog consists of finite set of threads $\mathcal{T}$. Each thread $t \in \mathcal{T}$ is a finite state machine that works on its own set of local registers $\mathcal{R}_t$. The local registers of different threads are disjoint. Let $\mathcal{R} = \cup_{t \in \mathcal{T}} \mathcal{R}_t$. The threads communicate over a finite set of shared variables $\mathcal{X}$. The registers and the shared variables take their values from a data theory $(\mathtt{D}, \mathsf{Rl})$. Formally, a thread is a tuple $t = \langle \mathcal{Q}_t, \mathcal{R}_t, \Delta_t, q^t_{\mathsf{init}} \rangle$ where $\mathcal{Q}_t$ is a finite set of states of thread t, $q^t_{\mathsf{init}} \in \mathcal{Q}_t$ is the initial state of t, and $\Delta_t \subseteq \mathcal{Q}_t \times \mathsf{Op} \times \mathcal{Q}_t$ is a finite set of transitions that change the state and execute an operation $\mathsf{op} \in \mathsf{Op}$. Let $x \in \mathcal{X}, r_1, r_2 \in \mathcal{R}_t$. A transition $\delta \in \Delta_t$ is a tuple $\delta = \langle q, \mathsf{op}, q' \rangle$ where the operation $\mathsf{op} \in \mathsf{Op}$ has one of the following forms: (1) $r_1 := r_2$ assigns the value of register r_2 to register r_1, (2) $r_1 := \circledast$ non-deterministically assigns a value to register r_1, (3) $\mathsf{rl}(r_1, r_2)$ checks if the values of the two registers r_1 and r_2 satisfy the relation $\mathsf{rl} \in \mathsf{Rl}$, (4) $\mathsf{rd}(x, r_1)$ reads the value of shared variable x and stores it in register r_1, (5) $\mathsf{wt}(x, r_1)$ writes the value of register r_1 to shared variable x, and (6) $\mathsf{arw}(x, r_1, r_2)$ is the atomic read write operation which atomically executes a read followed by a write operation.

TSO Semantics The TSO memory model [33] is used by the x86 processor architecture. Each thread has its own FIFO write buffer. Write operations $\mathsf{wt}(x, r)$ in a thread t do not update the memory immediately; if $d \in \mathtt{D}$ is the value of r, then (x, d) is appended to the buffer of t. The buffer contents are updated to the shared memory non-deterministically. A read operation $\mathsf{rd}(x, r)$ in t accesses the latest write in the buffer of t. In case there is no such write, it accesses the shared memory. For the atomic read write operation $\mathsf{arw}(x, r_1, r_2)$ in thread t, the buffer of t must be empty (ϵ), and the value of x in the memory must be same as the value of r_1. Then x is set to the value of r_2.

Formally, the TSO memory model is a labelled transition system. A configuration γ is defined as a tuple $\gamma = \langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle$ where $\mathsf{St} : \mathcal{T} \to \bigcup_{t \in \mathcal{T}} \mathcal{Q}_t$ maps each thread to its current state, $\mathsf{RVal} : \mathcal{R} \to \mathtt{D}$ maps each register in a thread to its current value, $\mathsf{Buf} : \mathcal{T} \to (\mathcal{X} \times \mathtt{D})^*$ maps each thread buffer to its content, which is a sequence of writes. Finally, $\mathsf{Mem} : \mathcal{X} \to \mathtt{D}$ maps each shared variable to its current value in the memory. The initial configuration of Prog is defined by a tuple $\gamma_{\mathsf{init}} = \langle \mathsf{St}_{\mathsf{init}}, \mathsf{RVal}_{\mathsf{init}}, \mathsf{Buf}_{\mathsf{init}}, \mathsf{Mem}_{\mathsf{init}} \rangle$ where $\mathsf{St}_{\mathsf{init}}$ maps each thread t to its initial states q^t_{init}, $\mathsf{RVal}_{\mathsf{init}}$ and $\mathsf{Mem}_{\mathsf{init}}$ assign all registers

$$\frac{\langle q, r_1 := r_2, q' \rangle \in \Delta_t}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle \xrightarrow{t, r_1 := r_2} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow \mathsf{RVal}(r_2)], \mathsf{Buf}, \mathsf{Mem} \rangle} \text{ assign}$$

$$\frac{\langle q, r_1 := \circledast, q' \rangle \in \Delta_t \quad d \in \mathtt{D}}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle \xrightarrow{t, r_1 := \circledast} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow d], \mathsf{Buf}, \mathsf{Mem} \rangle} \text{ new value}$$

$$\frac{\langle q, \mathsf{rl}(r_1, r_2), q' \rangle \in \Delta_t \quad \mathsf{rl}(R(r_1), R(r_2))}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle \xrightarrow{t, \mathsf{rl}(r_1, r_2)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle} \text{ relation}$$

$$\frac{\langle q, \mathsf{wt}(x, r_1), q' \rangle \in \Delta_t}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle \xrightarrow{t, \mathsf{wt}(x, r_1)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}, \mathsf{Buf}[t \leftarrow (x, \mathsf{RVal}(r_1)).\mathsf{Buf}(t)], \mathsf{Mem} \rangle} \text{ write}$$

$$\frac{\langle q, \mathsf{rd}(x, r_1), q' \rangle \in \Delta_t \quad \nexists d \in \mathtt{D} : (x, d) \in \mathsf{Buf}(t)}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle \xrightarrow{t, \mathsf{rd}(x, r_1)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow \mathsf{Mem}(x)], \mathsf{Buf}, \mathsf{Mem} \rangle} \text{ global read}$$

$$\frac{\langle q, \mathsf{rd}(x, r_1), q' \rangle \in \Delta_t \quad \mathsf{Buf}(t) = \alpha.(x, d).\beta \quad \alpha, \beta \in (\mathcal{X} \cdot \mathtt{D})^* \quad \nexists d' \in \mathtt{D} : (x, d') \in \alpha}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle \xrightarrow{t, \mathsf{rd}(x, r_1)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow d], \mathsf{Buf}, \mathsf{Mem} \rangle} \text{ local read}$$

$$\frac{\langle q, \mathsf{arw}(x, r_1, r_2), q' \rangle \in \Delta_t \quad \mathsf{Buf}(t) = \epsilon \quad \mathsf{RVal}(r_1) = \mathsf{Mem}(x)}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle \xrightarrow{t, \mathsf{arw}(x, r_1, r_2)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}[x \leftarrow \mathsf{RVal}(r_2)] \rangle} \text{ atomic read write}$$

$$\frac{}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}[t \leftarrow \mathsf{Buf}(t).(x, d)], \mathsf{Mem} \rangle \xrightarrow{t, u} \langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}[x \leftarrow d] \rangle} \text{ memory update}$$

Fig. 1. The transition relation of TSO. We assume that $\mathsf{St}(t) = q$.

and shared variables the value 0, and $\mathsf{Buf}_{\mathsf{init}}$ initializes all thread buffers to the empty word ϵ. We formally define the labelled transition relation $\xrightarrow{\ell}$ on configurations in Figure 1 where the label ℓ is either of the form t, op (to denote a thread operation) or t, u (to denote an update operation) with $t \in \mathcal{T}$ is a thread and $\mathsf{op} \in \mathsf{Op}$ is an operation.

The Reachability Problem ***Reach*** Given a concurrent program Progand a state $q_{final} \in \mathcal{Q}_t$ of thread t, Reach asks, if a configuration $\gamma = \langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle$ with $\mathsf{St}(t) = q_{final}$ is reachable by the transition system given by the TSO semantics of Prog. In this case, we say that the state q_{final} is reachable by Prog. We use $\mathtt{Reach}[\mathtt{D}, \mathsf{Rl}]$ to denote the reachability problem for a concurrent program with the data theory $(\mathtt{D}, \mathsf{Rl})$.

4 Lamport's Bakery Algorithm

To demonstrate the practical application of our model, we use it to model Lamport's Bakery Algorithm [26]. Created by Leslie Lamport in 1974, it is a cornerstone solution for achieving mutual exclusion in concurrent systems. Picture threads as patrons entering a bakery, each is handed a unique ticket upon arrival. These tickets, representing the order of entry, dictate the sequence for accessing critical sections. They ensure an orderly execution flow and preventing race conditions in a critical section.

Each thread is assigned a unique number that is larger then the numbers currently assigned to other threads. The thread possessing the lowest number is granted entry to the critical section. This thread may access the critical section an unbounded number of times. This means the assigned tickets keep increasing and thus an infinite domain is required. Note that the algorithm does not rely on precise tickets values, we only need to compare the tickets to each other. This makes the protocol well suited to our program model.

The protocol contains n threads where each thread $i \leq n$ is associated with two variables: The ticket number $ticket_i$ and the flag $chosen_i$ which signals whether the thread has chosen a ticket number. We assume r_{TRUE} and r_{FALSE} are initialized with different values that represent the boolean values of a flag and that $ticket_i$ is initially the same as r_{FALSE} for all $i \leq n$.

The algorithm for thread i is given in Algorithm 1. For the sake of simplicity and compactness we present the transition system as pseudocode. This is equivalent to a program definition since the code only accesses variables and registers using operations Op with relations $\mathsf{Rl}_<$. The remaining instructions only affect the finite control flow and can be expressed using transitions.

Algorithm 1 Lamport Bakery Protocol

```
 1: $\mathsf{wt}(chosen_i, r_{FALSE})$ {Begin choosing}
 2: $r_i := \circledast$ {Pick random ticket}
 3: for all $1 \leq j \leq n$ do
 4:    $\mathsf{rd}(ticket_j, r_j)$
 5:    if $(r_i < r_j)$ then
 6:       goto line 1 {New ticket needed.}
 7:    end if
 8: end for
 9: $\mathsf{wt}(ticket_i, r_i)$ {Ticket accepted}
10: $\mathsf{wt}(chosen_i, r_{TRUE})$ {Choosing finished}
11: for all $1 \leq j \leq n$ do
12:    $\mathsf{rd}(chosen_j, r_j)$
13:    if $(r_j \neq r_{TRUE})$ then
14:       goto line 12 {Thread $j$ is still choosing}
15:    end if
16:    $\mathsf{rd}(ticket_j, r_j)$
17:    if $(r_j \neq r_{FALSE} \ \& \ r_j < r_i)$ then
18:       goto line 16 {Lower ticket $j$ found}
19:    end if
20: end for
21: CRITICAL Section
22: $r_i := r_{FALSE}$
23: goto line 1 {Back to NON-CRITICAL}
```

5 State Reachability for TSO with (Dis)-Equality Relation

We show that the reachability problem for concurrent programs under TSO is undecidable when $\{=,\neq\} \subseteq \mathsf{Rl}$. The proof is achieved through a reduction from the state reachability problem of Lossy Channel Systems with Data (DLCS) [1], which is already known to be undecidable. To simulate the lossy channel, we employ write buffers, as both are implemented as first-in-first-out queues. However, there are three main distinctions that must be considered: (i) write buffers do not contain letters, (ii) write buffers are not lossy, and (iii) the semantics of reads differ from receives.

We address these distinctions as follows: (i) We encode the letters as variables. (ii) We model writes being lost by avoiding to read them. (iii) To prevent buffer reads, we transfer the writes into a write buffer of a second thread with a different variable. We ensure that every write is accessed only once by overwriting them immediately with a different value.

Theorem 1. *Reach*$[\mathtt{D}, \mathsf{Rl}]$ *is undecidable for* $\{=,\neq\} \subseteq \mathsf{Rl}$.

The rest of this section is devoted to the proof of the above theorem. We first recall the definition of Lossy Channel Systems with Data (DLCS) [1]. Then, we present the reduction from state reachability problem of DLCS to $\mathtt{Reach}[\mathtt{D}, \mathsf{Rl}]$.

$$\frac{\langle q, x := y, q'\rangle \in \Delta_{\mathcal{L}}}{\langle q, \mathsf{XVal}, w\rangle \xrightarrow{x:=y} \langle q', \mathsf{XVal}[x \leftarrow \mathsf{XVal}(y)], w\rangle}\ \text{assign}$$

$$\frac{\langle q, x := \circledast, q'\rangle \in \Delta_{\mathcal{L}} \quad d \in \mathtt{D} \setminus \{\mathsf{XVal}(y) \mid y \in \mathcal{X}_{\mathcal{L}}\}}{\langle q, \mathsf{XVal}, w\rangle \xrightarrow{x:=\circledast} \langle q', \mathsf{XVal}[x \leftarrow d], w\rangle}\ \text{new value}$$

$$\frac{\langle q, x = y, q'\rangle \in \Delta_{\mathcal{L}} \quad \mathsf{XVal}(x) = \mathsf{XVal}(y)}{\langle q, \mathsf{XVal}, w\rangle \xrightarrow{x=y} \langle q', \mathsf{XVal}, w\rangle}\ \text{equality}$$

$$\frac{\langle q, x \neq y, q'\rangle \in \Delta_{\mathcal{L}} \quad \mathsf{XVal}(x) \neq \mathsf{XVal}(y)}{\langle q, \mathsf{XVal}, w\rangle \xrightarrow{x\neq y} \langle q', \mathsf{XVal}, w\rangle}\ \text{disequality}$$

$$\frac{\langle q, !\langle a, x\rangle, q'\rangle \in \Delta_{\mathcal{L}}}{\langle q, \mathsf{XVal}, w\rangle \xrightarrow{!\langle a,x\rangle} \langle q', \mathsf{XVal}, (a, \mathsf{XVal}(x)).w\rangle}\ \text{send}$$

$$\frac{\langle q, ?\langle a, x\rangle, q'\rangle \in \Delta_{\mathcal{L}}}{\langle q, \mathsf{XVal}, w.(a, d)\rangle \xrightarrow{?\langle a,x\rangle} \langle q', \mathsf{XVal}[x \leftarrow d], w\rangle}\ \text{receive}$$

$$\frac{w' \leq w}{\langle q, \mathsf{XVal}, w\rangle \xrightarrow{loss} \langle q, \mathsf{XVal}, w'\rangle}\ \text{lossiness}$$

Fig. 2. The transition relation of DLCS

Lossy Channel Systems with Data A DLCS $\mathcal{L} = \langle \mathcal{Q}_\mathcal{L}, \mathcal{X}_\mathcal{L}, \Sigma_\mathcal{L}, \Delta_\mathcal{L}, q_{\mathsf{init}} \rangle$ consists of a finite set of states $\mathcal{Q}_\mathcal{L}$, a finite number of variables $\mathcal{X}_\mathcal{L}$ ranging over an infinite domain $\mathtt{D}$, a finite channel alphabet $\Sigma_\mathcal{L}$, $q_{\mathsf{init}} \in \mathcal{Q}$ is the initial state, and a finite set of transitions $\Delta_\mathcal{L}$. The set $\Delta_\mathcal{L}$ of transitions is a subset of $\mathcal{Q}_\mathcal{L} \times \mathsf{Op}_\mathcal{L} \times \mathcal{Q}_\mathcal{L}$. Let $x, y \in \mathcal{X}_\mathcal{L}$. The set $\mathsf{Op}_\mathcal{L}$ consists of the following operations (1) $x := y$ which assigns the value of y to x, (2) $x := \circledast$, which assigns a fresh value from $\mathtt{D}$ that is different from the existing values of all variables[4], (3) $x = y$ ($x \neq y$) which compares the value of variables x and y, (4) $!\langle a, x \rangle$ which appends letter $a \in \Sigma_\mathcal{L}$ together with the value of x to the channel, (5) $?\langle a, x \rangle$ which deletes the head of the channel $\langle a, d \rangle$ and stores the value d in x, and (6) *loss* which removes elements in the channel.

A configuration γ of DLCS is defined by the tuple $\langle q, \mathsf{XVal}, w \rangle$ where $q \in \mathcal{Q}_\mathcal{L}$ is the current state, $\mathsf{XVal} : \mathcal{X}_\mathcal{L} \to \mathtt{D}$ is the current valuation of the variables, and $w \in (\Sigma \times \mathtt{D})^*$ is the content of the lossy channel. The system is lossy, which means any element in the channel may disappear anytime. The initial configuration γ_{init} of $\mathcal{L}$ is defined by $(q_{\mathsf{init}}, \mathsf{XVal}_{\mathsf{init}}, \epsilon)$ where $\mathsf{XVal}_{\mathsf{init}}(x) = 0$ for all $x \in \mathcal{X}_\mathcal{L}$. The transition relation of DLCS is given in Figure 2.

The state reachability problem for $\mathcal{L}$ asks whether, for a given final state $q_{final} \in \mathcal{Q}$, there is a reachable configuration γ of the form $\gamma = \langle q_{final}, \mathsf{XVal}, w \rangle$. In this case, we say that the state q_{final} is reachable by $\mathcal{L}$.

Theorem 2 ([1]). *The state reachability problem for DLCS is undecidable.*

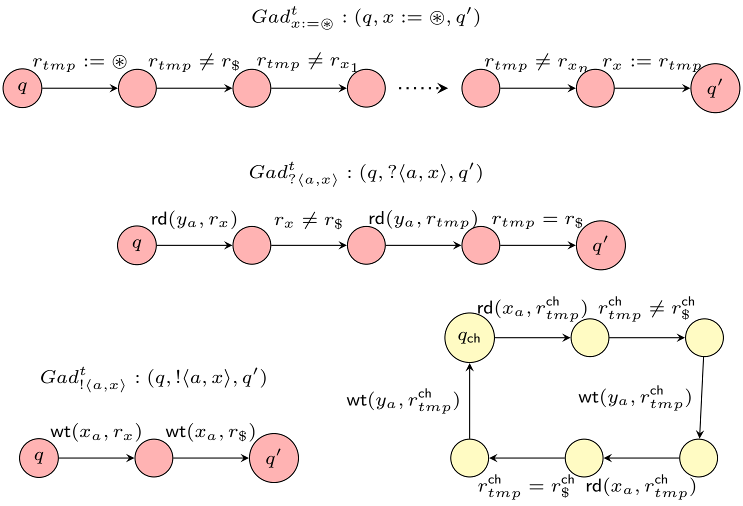

$Gad^t_{x:=\circledast} : (q, x := \circledast, q')$
$r_{tmp} := \circledast$
$r_{tmp} \neq r_\$$
$r_{tmp} \neq r_{x_1}$
$r_{tmp} \neq r_{x_n}$
$r_x := r_{tmp}$
q
q'
$Gad^t_{?\langle a,x \rangle} : (q, ?\langle a, x \rangle, q')$
$\mathsf{rd}(y_a, r_x)$
$r_x \neq r_\$$
$\mathsf{rd}(y_a, r_{tmp})$
$r_{tmp} = r_\$$
q
q'
$Gad^t_{!\langle a,x \rangle} : (q, !\langle a, x \rangle, q')$
$\mathsf{wt}(x_a, r_x)$
$\mathsf{wt}(x_a, r_\$)$
q
q'
q_{ch}
$\mathsf{rd}(x_a, r^{\mathsf{ch}}_{tmp})$
$r^{\mathsf{ch}}_{tmp} \neq r^{\mathsf{ch}}_\$$
$\mathsf{wt}(y_a, r^{\mathsf{ch}}_{tmp})$
$\mathsf{rd}(x_a, r^{\mathsf{ch}}_{tmp})$
$r^{\mathsf{ch}}_{tmp} = r^{\mathsf{ch}}_\$$
$\mathsf{wt}(y_a, r^{\mathsf{ch}}_{tmp})$

Fig. 3. $\mathsf{Prog}(\mathcal{L})$ with threads t (pink states) and t_{ch} (yellow states).

[4] This differs from the $\circledast$ in TSO where the value $d \in \mathtt{D}$ assigned by the operation $x := \circledast$ can be anything.

Reduction from DLCS reachability Given a DLCS $\mathcal{L} = \langle \mathcal{Q}_{\mathcal{L}}, \mathcal{X}_{\mathcal{L}}, \Sigma_{\mathcal{L}}, \Delta_{\mathcal{L}}, q_{\mathsf{init}} \rangle$ over data domain $\mathtt{D}$ with $\mathcal{X}_{\mathcal{L}} = \{x_1 \ldots x_n\}$, we reduce the state reachability of $\mathcal{L}$ to the reachability problem $\mathtt{Reach}[\mathtt{D}, \{=, \neq\}]$ of a concurrent program $\mathsf{Prog}(\mathcal{L})$, with two threads t, t_{ch}. The thread t simulates the operations of $\mathcal{L}$, while thread t_{ch} simulates the lossy channel of $\mathcal{L}$ using its write buffer. Let $\mathcal{R}_t = \{r_\$, r_{tmp}\} \cup \{r_x \mid x \in \mathcal{X}_{\mathcal{L}}\}$, $\mathcal{R}_{t_{\mathsf{ch}}} = \{r^{\mathsf{ch}}_\$, r^{\mathsf{ch}}_{tmp}\}$ be the local registers of threads t and t_{ch}. Corresponding to each $x \in \mathcal{X}_{\mathcal{L}}$, we have the register r_x in thread t, which stores the current values of x. Registers r_{tmp} and r^{ch}_{tmp} are used to temporarily store certain values. The shared variables of $\mathsf{Prog}(\mathcal{L})$ are $\mathcal{X} = \{x_a, y_a \mid a \in \Sigma_{\mathcal{L}}\}$, they help in simulating the behavior of the lossy channel of $\mathcal{L}$.

Simulating the DLCS. The transitions of $\mathsf{Prog}(\mathcal{L})$ are sketched in Figure 3. The initialization of the program is omitted in the figure and goes as follows. The thread t_{ch} starts by assigning a non-deterministic value (say \$) to the register $r^{\mathsf{ch}}_\$$ (i.e., $r^{\mathsf{ch}}_\$:= \circledast$), then checks that the new value \$ is different from 0 (i.e., by checking that $r^{\mathsf{ch}}_\$ \neq r^{\mathsf{ch}}_{tmp}$), and finally performs an atomic read write operation $\mathsf{arw}(x, r^{\mathsf{ch}}_{tmp}, r^{\mathsf{ch}}_\$)$ on each variable $x \in \mathcal{X}$. The thread t starts by reading the value of each shared variable $x \in \mathcal{X}$ (i.e., performing $\mathsf{rd}(x, r_\$)$) and checks if its value is different from 0 (i.e., $r_\$ \neq r_{tmp}$). At the end of this initialization phase, all the shared variables have the new value \$, the registers r_{tmp} and r^{ch}_{tmp} have the value 0 and the registers $r_\$$ and $r^{\mathsf{ch}}_\$$ have the value \$. The current state of thread t is the initial state q_{init} of $\mathcal{L}$ while the thread t_{ch} is in a state q_{ch}.

Every transition $\langle q, x := y, q' \rangle \in \Delta_{\mathcal{L}}$ is simulated in $\mathsf{Prog}(\mathcal{L})$ by threat t with a gadget—a sequence of transitions that starts in q and ends in q'. The transitions $(q, x := y, q')$, $(q, x = y, q')$ and $(q, x \neq y, q')$ in the DLCS are simulated by the thread t as gadgets with single transitions $(q, r_x := r_y, q'), (q, r_x = r_y, q')$ and $(q, r_x \neq r_y, q')$, respectively. We omit their description in Figure 3.

To simulate $x := \circledast$, we load the new value in register r_{tmp} and ensure that it is different from the values in registers $r_\$$ and $r_{x_1} \ldots r_{x_n}$. This is depicted by the gadget $Gad^t_{x:=\circledast}$ in thread t. The send operation $!\langle a, x \rangle$ in the DLCS is simulated by the gadget $Gad^t_{!\langle a,x \rangle}$. In the DLCS, the send appends the letter a and the value of x to the channel. This is simulated by the write $\mathsf{wt}(x_a, r_x)$, thereby appending $(x_a, val(r_x))$ to the buffer of t. To simulate reads of the DLCS, we first make note of a crucial difference in the way reads happen in DLCS and TSO. In DLCS, a read happens from the head of the channel, and the head is deleted immediately after the read. In TSO however, we can read from the latest write in the shared memory multiple times. In order to simulate the "read once" policy of the DLCS, we follow each $\mathsf{wt}(x_a, r_x)$ with another write $\mathsf{wt}(x_a, r_\$)$.

Thread t_{ch} is a loop from the state q_{ch} which continuously reads from x_a a value from a simulated send followed by the separator \$. It copies these values to y_a using local register r^{ch}_{tmp}. The first time it reads from x_a, it reads the value d of x from a simulated send $!\langle a, x \rangle$. It ensures that this is not the \$ symbol ($r^{\mathsf{ch}}_{tmp} \neq r^{\mathsf{ch}}_\$$), and writes this value from r^{ch}_{tmp} into variable y_a, thus appending (y_a, d) in the buffer of t_{ch}. It then reads again the value of x_a into r^{ch}_{tmp}. This time, it makes sure to read \$ with the check $r^{\mathsf{ch}}_{tmp} = r^{\mathsf{ch}}_\$$. The receive $?\langle a, x \rangle$ of the DLCS is simulated by $Gad^t_{?\langle a,x \rangle}$. First, we read from y_a and store it in r_x, ensuring this value d is not \$. Then, we read \$ from y_a. This ensures that the earlier value d is overwritten in the memory and is not read twice.

A loss in the channel of the DLCS results in losing some messages $\langle a, d \rangle$. This is accounted for in $\mathsf{Prog}_{\mathcal{L}}$ in two ways. Thread t_{ch} may not pass on a value written from x_a to y_a since the loop may not execute for every value. Thread t may not read a value written by t_{ch} in y_a since it was already overwritten by some later writes.

Lemma 1. *The state q_{final} is reachable by $\mathcal{L}$ if and only if q_{final} is reachable by* $\mathsf{Prog}(\mathcal{L})$*.*

The formal proof is given in Appendix A of the full version [3]. Theorem 1 extends to any set of relations that we can use to simulate equality and disequality. For instance $\leq, \not\leq \in \mathsf{Rl}$.

6 Context Bounded Analysis

In the light of this undecidability, we turn our attention to a variant of the reachability problem which is tractable. We study context bounded runs, an under-approximation of the program behavior that limits the possible interactions between processes. A run consists of a number of *contexts*. A context is a sequence of steps where only a certain fixed thread t is *active*. We say that $\pi \in \mathsf{CB}(k)$ if and only if there is a partitioning $\pi = \pi_1 \dots \pi_k$ such that for all contexts $i \leq k$ there is an active thread $t_i \in \mathcal{T}$ such that only the active thread updates the memory and performs operations: If $\gamma \xrightarrow{\ell} \gamma' \in \pi_i$, then $\ell \in \{t_i\} \times (\mathsf{Op} \cup \{u\})$.

In the following, we show PSPACE completeness of $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathtt{D}, \mathsf{Rl}_{\leq n}]$ for relations such as (dis) equality, "greater than" or even "greater by at least n" for $n \in \mathbb{N}$ (see Theorem 4). Our approach begins with a proof of PSPACE hardness through a reduction from the non-emptiness problem of the intersection of regular languages [21].

Next, we demonstrate PSPACE membership by reducing the problem to state reachability of a finite transition system which we solve in polynomial space. This reduction faces challenges from two main sources, namely, (i) the unbounded size of the write buffers, and (ii) the infinite data domain $\mathtt{D}$. In this section, we show how to construct a finite transition system while preserving state reachability in two key steps.

Following [14], we first perform a buffer abstraction. An in-depth analysis of the TSO semantics within context bounded runs reveals a critical insight: Even though the buffer may contain an unbounded number of writes, only a bounded number of these writes can be read later on. This allows us to non-deterministically identify and store the necessary writes using variables.

Finally, we implement a domain abstraction. A popular approach is to abstract the values into equivalence classes based on the supported relations. This reveals our next challenge: (iii) the set of relations $\mathsf{Rl}_{\leq n}$ is infinite. We conduct an analysis of the reachable configurations and discover the following: If a configuration is reachable, then any configuration that is the same except with greater distances between differing values is reachable as well. It follows that, for control state reachability, the abstraction does not require the precise distances between variables; their relative order is sufficient.

6.1 Lower-bound

We establish PSPACE hardness by polynomially reducing the problem of checking non-emptiness of the intersection of regular languages to $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{D}, \mathsf{Rl}_{\leq n}]$. Given a set of finite automata $\mathcal{A}_1 \ldots \mathcal{A}_n$ with $\mathcal{A}_i = \langle \mathcal{Q}_i, \Delta_i, q_i^{\mathsf{init}} \mathcal{Q}_i^F \rangle$, where $\Delta_i \subseteq \mathcal{Q}_i \times \Sigma \times \mathcal{Q}_i$, $q_i^{\mathsf{init}} \in \mathcal{Q}_i$, and $\mathcal{Q}_i^F \subseteq \mathcal{Q}_i$ for $i \leq n$, the problem asks whether there is a word $w \in \Sigma^*$ that is accepted by each automaton $\mathcal{A}_i$ with $i \leq n$. This is known to be PSPACE hard[21].

We construct a program $\mathsf{Prog}(\mathcal{A}_1 \ldots \mathcal{A}_n)$ that consists of a single thread and reaches a state $q_{fi\ al}$ if and only if there is such a word. The idea of the construction is that we assign each state $q_i \in \mathcal{Q}_i$ a unique value stored in a register r_{q_i} and we store the value of the current state of each automaton $\mathcal{A}_i$ in a register r_i. To begin, we ensure that the current states are the initial ones. This means $r_i = r_{q_i^{\mathsf{init}}}$ holds for each $i \leq n$. Then, we choose a letter $a \in \Sigma$ and simulate some transition $q_i \xrightarrow{a} q_i' \in \Delta_i$ for each automaton. This is done by ensuring that the current state is q_i with $r_i = r_{q_i}$ and then updating the current state with $r_i := r_{q_i'}$. We repeat this step until each current state is a final state. At this point, we know we have simulated runs for each automaton that accept the same word and we reach $q_{fi\ al}$.

The formal definition of the construction as well as the proof of correctness is given in Appendix B of [3]. This is a polynomial reduction of non-emptiness of the intersection of regular languages to $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{D}, \mathsf{Rl}_{\leq n}]$. Observe that we only need test for equality and disequality. The disequalitiy checks are necessary to ensure that each register r_{q_i} has been assigned a different value.

Theorem 3. *$\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{D}, \mathsf{Rl}_{\leq n}]$ is* PSPACE *hard.*

6.2 PSPACE Upper-bound

Assume that we are given a program Prog and a context bound k. As an intermediary step towards finite state space we construct a finite state machine $\mathtt{AB}(\mathsf{Prog}, k)$ with variables, over the infinite data domain $\mathtt{D}$. The name $\mathtt{AB}$ stands for *abstract buffer* as it abstracts from the unbounded write buffers using a finite number of variables. We show that $\mathtt{AB}(\mathsf{Prog}, k)$ is state reachability equivalent with the TSO semantics of Prog bound by $\mathsf{CB}(k)$.

While abstracting away the buffers, the main challenge is to simulate read operations. Recall from Section 3 that each read operation in a thread accesses either a write from its own buffer or from the shared memory. A buffer read always reads from the threads latest write on the same variable. Since only the active thread may interact with the memory during the context, we can assume w.l.o.g. that all memory updates occur at the end of a context. This means a memory read accesses the last write on the same variable that updated the memory in an earlier context, and hence we do not need to store the whole buffer content. For *memory reads*, we need the latest writes leaving the buffer at the end of each context for each variable. For *buffer reads*, we only require the latest writes on each variable that are issued by each thread.

Construction of the abstract machine The abstract machine $\mathsf{AB}(\mathsf{Prog}, k)$ is defined by the tuple $\langle \mathcal{Q}_{\mathsf{AB}}, \mathcal{X}_{\mathsf{AB}}, \Delta_{\mathsf{AB}}, q^{\mathsf{AB}}_{\mathsf{init}} \rangle$ where $\mathcal{Q}_{\mathsf{AB}}$ is the finite set of states, $\mathcal{X}_{\mathsf{AB}}$ is the finite set of variables, Δ_{AB} the transition relation, and $q^{\mathsf{AB}}_{\mathsf{init}}$ is the initial state. A control state $q_{\mathsf{AB}} \in \mathcal{Q}_{\mathsf{AB}}$ is a tuple $(\mathsf{St}, act, j, c, u)$ where: (i) the current state of every thread is stored using function $\mathsf{St} : \mathcal{T} \to \mathcal{Q}$; (ii) function $act : \{1 \dots k\} \to \mathcal{T}$ assigns to each context an active thread; (iii) the current context is stored in variable $j \in \{1 \dots k\}$; (iv) the function $c : \mathcal{X} \times \mathcal{T} \to \{0, 1 \dots k\}$ assigns to each variable $x \in \mathcal{X}$ and thread $t \in \mathcal{T}$, the (future) context j' in which the latest write on x will leave the write buffer of t. This determines when t can access the shared memory on that variable again; and (v) function $u : \{1 \dots k\} \to 2^{\mathcal{X}}$ assigns each context j the set of variables that are updated during j. Additionally, we will introduce some helper states with the transitions relation. We omit them from the definition of $\mathcal{Q}_{\mathsf{AB}}$. The initial state $q^{\mathsf{AB}}_{\mathsf{init}}$ is such a helper state.

The set of variables $\mathcal{X}_{\mathsf{AB}}$ contains: (i) the set of variables $\mathcal{X}$ in Prog, (ii) the set of registers $\mathcal{R}$, (iii) for each each context $j \leq k$ and each variable $x \in \mathcal{X}$, we introduce a variable x_j, which stores the value of the last write on x that leaves the write buffer in context j, (iv) for each thread t and each variable $x \in \mathcal{X}$, we introduce a variable x_t which stores the value of the newest write of t on x that is still in the buffer of t. Notice that this is the write that t accesses when reading x (if such a write exists).

We define the transition relation Δ_{AB} in Figure 4. Let $c_{\mathsf{init}}(x, t) = 0$ for all $x \in \mathcal{X}$ and $t \in \mathcal{T}$, and $u_{\mathsf{init}}(i) = \emptyset$ for all $i \in \{1, \dots, k\}$. The outgoing transitions of state $q^{\mathsf{AB}}_{\mathsf{init}}$ are the outgoing transitions of $(\mathsf{St}_{\mathsf{init}}, act, 0, c_{\mathsf{init}}, u_{\mathsf{init}})$ for every possible function act. This means the construction guesses a function act and behaves as if the other elements in the tuple have the initial values. Local transitions are adapted in a straightforward manner. A read on x from the buffer occurs if there is a write on x in the buffer. This means the latest write on x leaves the buffer in a context $c(x, t)$ after (or in) the current context j. In such a case, we access x_t which holds the latest write on x in the buffer of t. If there is no such write on x in the buffer, i.e. $c(x, t) < j$ holds, then the read fetches the value of x from the shared memory.

A write operation on x overwrites the latest entry in the write buffer on that variable x_t and determines a future (or current) context j' with $j' \geq j$ in which it leaves the buffer. This is recorded in the variable $x_{j'}$ and x is added to the set $u(j')$ which holds the variables that are updated in context j'. Note that j' cannot be smaller then any other context in which a write on a variable y leaves the buffer of t. This information is obtained from the function c. Also, j' must be a context in which t is active.

$$\frac{\langle q'_{\mathtt{AB}}, op, q''_{\mathtt{AB}}\rangle \in \Delta_{\mathtt{AB}} \quad q'_{\mathtt{AB}} = (\mathsf{St}_{\mathsf{init}}, act, 0, c_{\mathsf{init}}, u_{\mathsf{init}})}{\langle q^{\mathtt{AB}}_{\mathsf{init}}, op, q''_{\mathtt{AB}}\rangle}\ \text{init} \qquad \frac{op \in \{r_1 := r_2, r_1 := \circledast, \mathsf{rl}(r_1, r_2)\}}{\langle q_{\mathtt{AB}}, op, q_{\mathtt{AB}}\,[\mathsf{St}(t) \leftarrow q_b]\rangle}\ \text{local}$$

$$\frac{op = \mathsf{rd}(x, r_1) \quad c(x,t) \geq j}{\langle q_{\mathtt{AB}}, r_1 := x_t, q_{\mathtt{AB}}\,[\mathsf{St}(t) \leftarrow q_b]\rangle}\ \text{buffer read} \qquad \frac{op = \mathsf{rd}(x, r_1) \quad c(x,t) < j}{\langle q_{\mathtt{AB}}, r_1 := x, q_{\mathtt{AB}}\,[\mathsf{St}(t) \leftarrow q_b]\rangle}\ \text{memory read}$$

$$\frac{op = \mathsf{wt}(x, r_1) \quad j' \geq j \quad act(j') = t \quad j' \geq max\{c((y,t)) \mid y \in \mathcal{X}\}}{\langle q_{\mathtt{AB}}, x_t := r_1, q_\delta\rangle, \langle q_\delta, x_{j'} := r_1, q_{\mathtt{AB}}[\mathsf{St}(t) \leftarrow q_b,\ c(x,t) \leftarrow j', u(j') \leftarrow u(j') \cup \{x\}]\rangle}\ \text{write}$$

$$\frac{op = \mathsf{arw}(x, r_1, r_2) \quad j = c(x,t) = max\{c((y,t)) \mid y \in \mathcal{X}\}}{\langle q_{\mathtt{AB}}, x_t = r_1, q_{\delta,1}\rangle\langle q_{\delta,1}, x_j := r_2, q_{\delta,2}\rangle, \langle q_{\delta,2}, x_t := r_2, q_{\mathtt{AB}}[\mathsf{St}(t) \leftarrow q_b, u(j) \leftarrow u(j) \cup \{x\}]\rangle}\ \text{buffer arw}$$

$$\frac{op = \mathsf{arw}(x, r_1, r_2) \quad j > c(x,t) \quad j \geq max\{c((y,t)) \mid y \in \mathcal{X}\}}{\langle q_{\mathtt{AB}}, x = r_1, q_\delta\rangle, \langle q_\delta, x := r_2, q_{\mathtt{AB}}[\mathsf{St}(t) \leftarrow q_b]\rangle}\ \text{memory arw}$$

$$\frac{q_{\mathtt{AB}} \in \mathcal{Q}_{\mathtt{AB}} \qquad j < k \quad u(j) = \{x^1, \ldots, x^n\}}{\langle q_{\mathtt{AB}}, x^1 := x^1_j, q_{new,1}\rangle \ldots \langle q_{new,n-1}, x^n := x^n_j, q_{\mathtt{AB}}\,[j \leftarrow j+1]\rangle}\ \text{context switch}$$

Fig. 4. The transition relation $\Delta_{\mathtt{AB}}$ of $\mathtt{AB}(\mathsf{Prog}, k)$. Let $\delta = \langle q_a, op, q_b\rangle \in \Delta_t$ and $q_{\mathtt{AB}} = (\mathsf{St}, act, j, c, u)$ with $\mathsf{St}(t) = q_a$ and $act(j) = t$.

At any time, the run can switch from a context j with $j < k$ to $j + 1$. Let $u(j) = \{x^1 \ldots x^n\}$. These are the variables that are updated during context j. The values of the last updates on these variables in the context, stored in $x^1_j \ldots x^n_j$, are written to the corresponding variables in the shared memory. Since $\mathtt{AB}(\mathsf{Prog}, k)$ only performs memory updates at the end of a context, an atomic read write $\mathsf{arw}(x, r_1, r_2)$ requires that the current buffer content leaves the buffer in the current context. This is ensured by using the condition $j \geq max\{c((y,t)) \mid y \in \mathcal{X}\}$. If there is a write on x in the buffer of t, then $j = c(x,t)$. This is covered by the *buffer arw* rule in Figure 4. Here, the current value of x is stored in x_t, so we first check that it equals r_1 and update x_t as well as x_j with r_2. If $j > c(x,t)$ holds, then there is no write on x in the buffer of t (*memory arw* rule) and we compare the value of x in the shared memory with r_1 and update it to r_2.

A configuration $\gamma = (q_{\mathtt{AB}}, \mathsf{Mem})$ in the induced LTS of $\mathtt{AB}(\mathsf{Prog}, k)$ consists of a state $q_{\mathtt{AB}} \in \mathcal{Q}_{\mathtt{AB}}$ along with a variable assignment Mem. Let $\gamma_{\mathsf{init}} = (q^{\mathtt{AB}}_{\mathsf{init}}, \mathsf{Mem}_{\mathsf{init}})$ be the initial configuration of $\mathtt{AB}(\mathsf{Prog}, k)$. Given the transitions $\Delta_{\mathtt{AB}}$, we can define the transitions in the induced LTS in a straightforward manner. A state $q_{final} \in \mathcal{Q}_t$ of thread t is said to be reachable by $\mathtt{AB}(\mathsf{Prog}, k)$ if and only if there is a reachable configuration of the form $((\mathsf{St}, act, j, c, u), \mathsf{Mem})$ such that $\mathsf{St}(t) = q_{final}$ holds.

Lemma 2. *A state of* Prog *is reachable under TSO by a run* $\pi \in CB(k)$ *if and only if it is reachable by* $\mathtt{AB}(\mathsf{Prog}, k)$.

The proof of Lemma 2 is given in Appendix C of [3]. Next, we abstract away the infinite data domain from $\mathtt{AB}(\mathsf{Prog}, k)$. We remove this last source of infinity by constructing a finite state machine $\mathsf{RI}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ from $\mathtt{AB}(\mathsf{Prog}, k)$.

$$\frac{\langle q_{\mathtt{AB}}, x := x', q'_{\mathtt{AB}}\rangle \in \varDelta_{\mathtt{AB}} \quad x =_{\mathsf{Rl}'} x' \quad \forall \mathsf{rl} \in \mathsf{Rl}_{<}, \forall z, y \in \mathcal{X}_{\mathtt{AB}} \setminus \{x\} : \mathsf{rl}_{\mathsf{Rl}}(y,z) \Leftrightarrow \mathsf{rl}_{\mathsf{Rl}'}(y,z)}{\langle (q_{\mathtt{AB}}, \mathsf{Rl}), x := x', (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \varDelta} \text{ assign}$$

$$\frac{\langle q_{\mathtt{AB}}, x := \circledast, q'_{\mathtt{AB}}\rangle \in \varDelta_{\mathtt{AB}} \quad \forall \mathsf{rl} \in \mathsf{Rl}_{<}, \forall z, y \in \mathcal{X}_{\mathtt{AB}} \setminus \{x\} : \mathsf{rl}_{\mathsf{Rl}}(y,z) \Leftrightarrow \mathsf{rl}_{\mathsf{Rl}'}(y,z)}{\langle (q_{\mathtt{AB}}, \mathsf{Rl}), x := \circledast, (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \varDelta} \text{ new value}$$

$$\frac{\langle q_{\mathtt{AB}}, \mathsf{rl}''(x,y), q'_{\mathtt{AB}}\rangle \in \varDelta_{\mathtt{AB}} \quad \mathsf{rl}'' \in \mathsf{Rl}_{<} \quad \mathsf{Rl} = \mathsf{Rl}' \quad \mathsf{rl}''_{\mathsf{Rl}}(x,y)}{\langle (q_{\mathtt{AB}}, \mathsf{Rl}), \mathsf{rl}''(x,y), (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \varDelta} \ \mathsf{Rl}_{<} \text{ relation}$$

$$\frac{\langle q_{\mathtt{AB}}, \mathsf{rl}''(x,y), q'_{\mathtt{AB}}\rangle \in \varDelta_{\mathtt{AB}} \quad \mathsf{rl}'' \notin \mathsf{Rl}_{<} \quad \mathsf{Rl} = \mathsf{Rl}' \quad x <_{\mathsf{Rl}} y}{\langle (q_{\mathtt{AB}}, \mathsf{Rl}), \mathsf{rl}''(x,y), (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \varDelta} \ \mathsf{Rl}_{\leq n} \text{ relation}$$

Fig. 5. The transition relation of $\mathsf{Rl}_{<}{-}\mathtt{AB}(\mathsf{Prog}, k)$. Sets Rl and Rl' satisfy (i) equality is an equivalence relation; (ii) disequality holds iff equality does not hold; (iii) " $<$ " is a total order on variables that are not equal.

Domain Abstraction We use domain abstraction to solve $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathtt{D}, \mathsf{Rl}_{\leq n}]$ by reducing state reachability of $\mathtt{AB}(\mathsf{Prog}, k)$ to reachability of a finite state machine. We introduce the set of relations $\mathsf{Rl}_{<} = \{=, \neq, <\}$. To abstract away the infinite data domain, we abstract from the exact values of the variables. Instead of storing actual values, we store which relations from $\mathsf{Rl}_{<}$ holds between which pairs of variables, which is finite information. This way, we reduce the infinite domain $\mathtt{D}$ to the finite Boolean domain $\mathbb{B}$. For example, $(q_{\mathtt{AB}}, x = y)$ is an abstraction of a configuration $(q_{\mathtt{AB}}, \mathsf{Mem}(x) = 1, \mathsf{Mem}(y) = 1)$. Given a variable assignment Mem and a relation rl, we define $\mathsf{rl}_{\mathsf{Mem}}(x,y) := \mathsf{rl}(\mathsf{Mem}(x), \mathsf{Mem}(y))$. Any variable assignment Mem induces a set of relations $\mathsf{Rl}_{\mathsf{Mem}} = \{\mathsf{rl}_{\mathsf{Mem}} \mid \mathsf{rl} \in \mathsf{Rl}_{<}\}$ over the variables $\mathcal{X}_{\mathtt{AB}}$. When considering multiple sets of relations we denote a relation $\mathsf{rl} \in \mathsf{Rl}$ as $\mathsf{rl}_{\mathsf{Rl}}$. For a variable assignment Mem, we say set of relations Rl over variables is consistent with Mem if $\mathsf{Rl} = \mathsf{Rl}_{\mathsf{Mem}}$.

Given $\mathtt{AB}(\mathsf{Prog}, k) = \langle \mathcal{Q}_{\mathtt{AB}}, \mathcal{X}_{\mathtt{AB}}, \varDelta_{\mathtt{AB}}, q^{\mathtt{AB}}_{\mathsf{init}}\rangle$, we now construct the finite state machine $\mathsf{Rl}_{<}{-}\mathtt{AB}(\mathsf{Prog}, k) = \langle \mathcal{Q}, \varDelta, q_{\mathsf{init}}\rangle$ as follows: $\mathcal{Q} := \mathcal{Q}_{\mathtt{AB}} \times \{\mathsf{rl}_{\mathcal{X}_{\mathtt{AB}}} : \mathcal{X}_{\mathtt{AB}} \times \mathcal{X}_{\mathtt{AB}} \to \mathbb{B} \mid \mathsf{rl} \in \mathsf{Rl}_{<}\}$. We abstract from a variable assignment by storing in the states which relations are satisfied. The initial state is $q_{\mathsf{init}} = (q^{\mathtt{AB}}_{\mathsf{init}}, \mathsf{Rl}_{\mathsf{Mem}_{\mathsf{init}}})$. We define the transitions of $\mathsf{Rl}_{<}{-}\mathtt{AB}(\mathsf{Prog}, k)$ in Figure 5. We construct the transitions such that they abstract from the transitions of the LTS induced by the semantics of $\mathtt{AB}(\mathsf{Prog}, k)$. Where the semantics on transitions of $\mathtt{AB}(\mathsf{Prog}, k)$ require that certain values in the configurations before and after the operation are the same, the transitions of $\mathsf{Rl}_{<}{-}\mathtt{AB}(\mathsf{Prog}, k)$ only require that the relations between variables before and after the relation are the same. For instance, the assign rule for operation $x := x'$ requires that Rl and Rl' are the same for all variables except x and $x =_{\mathsf{Rl}'} x'$ must hold after the operation. Conditions (i)-(iii) in Figure 5 reflect the properties of $\mathsf{Rl}_{<}$ on values. They ensure that Rl and Rl' have consistent variable assignments. Note that for any operation $<_n$ (or $\leq_n$), we soften the condition to $x <_{\mathsf{Rl}} y$. We will show that this still results in an abstraction precise enough to be state reachability equivalent.

Since $\mathsf{Rl}_{<}{-}\mathtt{AB}(\mathsf{Prog}, k)$ is a finite state machine, it induces the obvious LTS where a configuration consists of a state. The following lemma shows that the construction is indeed an abstraction of $\mathtt{AB}(\mathsf{Prog},k)$. We assume Prog uses $\mathsf{Rl}_{\leq n}$.

Lemma 3. *If $q_{\mathtt{AB}}$ is reachable by $\mathtt{AB}(\mathsf{Prog},k)$, then a state $(q_{\mathtt{AB}},\mathsf{Rl})$ is reachable by $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$.*

Proof. Assume $\langle(q_{\mathtt{AB}},\mathsf{Mem}) \xrightarrow{op} (q'_{\mathtt{AB}},\mathsf{Mem}')\rangle$. We argue that $\langle(q_{\mathtt{AB}},\mathsf{Rl}_{\mathsf{Mem}}),op,(q'_{\mathtt{AB}},\mathsf{Rl}_{\mathsf{Mem}'})\rangle \in \Delta$ holds as well. The lemma follows immediately. We show this for operation $x := \circledast$. For all other operations, the proof is analogue and we omit it.

It follows from the semantics of $x := \circledast$, that $\mathsf{Mem}(y) = \mathsf{Mem}'(y)$ for any $y \in \mathcal{X}_{\mathtt{AB}} \setminus \{x\}$ holds. This means $\mathsf{Rl}_{\mathsf{Mem}}$ and $\mathsf{Rl}_{\mathsf{Mem}'}$ satisfy the new value rule. The equality relations in $\mathsf{Rl}_{\mathsf{Mem}}$ and $\mathsf{Rl}_{\mathsf{Mem}'}$ are consistent with the equality relations on values of Mem and Mem'. The equality relation given by the values is an equivalence relation and thus Condition (i) is satisfied. Similarly, Condition (ii) is satisfied since values are obviously not equal if and only if they are not related by equality. Condition (iii) is satisfied since relation $<$ on values forms a total order. All conditions are satisfied. This means $\langle(q_{\mathtt{AB}},\mathsf{Rl}_{\mathsf{Mem}}),x := \circledast,(q'_{\mathtt{AB}},\mathsf{Rl}_{\mathsf{Mem}'})\rangle \in \Delta$.

Lemma 4. *If a state $(q_{\mathtt{AB}},\mathsf{Rl})$ is reachable by $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$, then $q_{\mathtt{AB}}$ is reachable by $\mathtt{AB}(\mathsf{Prog},k)$.*

We prove this by performing an induction over runs of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$ and constructing equivalent runs of $\mathtt{AB}(\mathsf{Prog},k)$. In order to do this, we construct configurations with consistent variable assignments. The main challenge is that these variable assignments may not have large enough distances between the values. Take the operation $x <_n y$, for instance. Here, $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$ only requires $x < y$. Note that any value other than 0 was created by an $x := \circledast$ operation. We can modify a run so that some of these operations assign larger values. This way, we can increase the distances of variable assignments of reachable configurations without changing their consistency with respect to relations. The formal proof of this is given in Appendix E of [3].

Theorem 4. *$\mathit{CB}(k)$-$\mathtt{Reach}[\mathsf{D},\mathsf{Rl}_{\leq n}]$ is* PSPACE *complete.*

Proof. While $\mathsf{Rl}_{\leq n}$ is an infinite set, $\mathsf{Rl}_{<}$ has only 3 relations. This means $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$ is a finite transition system where state reachability is decidable. According to Lemma 2, Lemma 3 and Lemma 4, deciding state reachability of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$ is equivalent to solving $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{Rl}_{\leq n}]$.

We non-deterministically solve the state reachability of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$ by guessing a run that is length-bounded by the size of the state space and checking whether it reaches $q_{\mathit{fi\ al}}$. We store the current state $((\mathsf{St},\mathit{act},j,c,u),\mathsf{Rl})$ together with a binary encoding of the current length of the run. Note that the state only requires polynomial space. The number of states of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog},k)$ is exponential in the program size as well as k, which means the binary encoding also requires polynomial space.

We extend the run by choosing to either perform a context switch or an operation. We begin with the initial state $q^{\mathtt{AB}}_{\mathsf{init}}$, which is a special case since we first need to guess a function act according to the init rule in Figure 4. To perform an operation, we look at the current state of the active thread $\mathsf{St}(act(j))$, pick an outgoing transition from the program, and update the state according to the corresponding rules given in Figure 4 and Figure 5.

We illustrate this on the new-value operation. Assume we pick the outgoing transition $\langle q_a, x := \circledast, q_b\rangle \in \varDelta_{act(j)}$. In this case, we update the state according to the local rule in Figure 4. Then we update the set Rl according to the new-value rule in Figure 5. We leave all relations that do not include x unchanged, and we non-deterministically choose x to be either equal to some variable, or to be between two other adjacent variables, or to be the largest or smallest variable. We update the relations to x accordingly. For any other operation, the changes to Rl are uniquely determined. For writes, we additionally need to non-deterministically pick some future context j' of the update according to the write rule in Figure 4. In the case of a context switch, we perform a series of variable assignments according to the context switch rule.

Note that we do not explicitly construct the entire $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ transition system; the program and the rules given in Figure 4 and Figure 5 are sufficient to guess a run. Each step can be performed in polynomial space. Once $\mathsf{St}(act(j)) = q_{fi\ al}$ holds, we know $q_{fi\ al}$ is reachable. The complexity of this process is in PSPACE. According to Theorem 3, the problem is PSPACE hard as well.

7 Conclusion

We examined safety verification of concurrent programs running under TSO that operate on variables ranging over an infinite domain. We have shown that this is undecidable even if the program can only check the variables for equality and non-equality. We studied a context bounded variant of the problem as well. Here, we solved the problem for programs using relations in $\mathsf{Rl}_{\leq n}$ and showed that it is PSPACE complete.

As future work, we plan to examine more expressive under-approximations of the program behaviour than the presented context bounded analysis and how these under-approximations affect decidability and complexity of the problem. We also intend to explore the problem for additional relations and/or operations a program may perform.

References

1. Abdulla, P.A., Aiswarya, C., Atig, M.F.: Data communicating processes with unreliable channels. In: LICS. pp. 166–175. ACM (2016). `https://doi.org/10.1145/2933575.2934535`, `https://doi.org/10.1145/2933575.2934535`
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: A load-buffer semantics for total store ordering. LMCS **14**(1) (2018)
3. Abdulla, P.A., Atig, M.F., Furbach, F., Garg, S.: Verification under TSO with an infinite data domain (2024)

4. Abdulla, P.A., Atig, M.F., Furbach, F., Godbole, A.A., Hendi, Y.G., Krishna, S.N., Spengler, S.: Parameterized verification under TSO with data types. In: TACAS 2023. LNCS, vol. 13993, pp. 588–606. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_30, https://doi.org/10.1007/978-3-031-30823-9_30
5. Abdulla, P.A., Atig, M.F., Phong, N.T.: The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In: ESOP 2015. LNCS, vol. 9032, pp. 308–332. Springer (2015). https://doi.org/10.1007/978-3-662-46669-8_13, https://doi.org/10.1007/978-3-662-46669-8_13
6. Abdulla, P.A., Atig, M.F., Rezvan, R.: Parameterized verification under TSO is PSPACE-complete. Proc. ACM Program. Lang. **4**(POPL), 26:1–26:29 (2020). https://doi.org/10.1145/3371094, https://doi.org/10.1145/3371094
7. Abdulla, P.A., Atig, M.F., Stenman, J.: Dense-timed pushdown automata. In: LICS. pp. 35–44. IEEE Computer Society (2012). https://doi.org/10.1109/LICS.2012.15, https://doi.org/10.1109/LICS.2012.15
8. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comput. **160**(1-2), 109–127 (2000). https://doi.org/10.1006/inco.1999.2843, https://doi.org/10.1006/inco.1999.2843
9. Abdulla, P.A., Delzanno, G.: On the coverability problem for constrained multiset rewriting. In: Proc. AVIS'06, The fifth Int. Workshop on on Automated Verification of Infinite-State Systems (2006)
10. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. In: LICS. pp. 160–170. IEEE Computer Society (1993). https://doi.org/10.1109/LICS.1993.287591, https://doi.org/10.1109/LICS.1993.287591
11. Abdulla, P.A., Sistla, A.P., Talupur, M.: Model checking parameterized systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 685–725. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_21, https://doi.org/10.1007/978-3-319-10575-8_21
12. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8, https://doi.org/10.1016/0304-3975(94)90010-8
13. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: SIGPLAN-SIGACT. pp. 7–18. ACM (2010). https://doi.org/10.1145/1706299.1706303, https://doi.org/10.1145/1706299.1706303
14. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: CAV. LNCS, vol. 6806, pp. 99–115. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_9, https://doi.org/10.1007/978-3-642-22110-1_9
15. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_29, https://doi.org/10.1007/978-3-642-37036-6_29
16. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: CONCUR. LNCS, vol. 1243, pp. 135–150. Springer (1997). https://doi.org/10.1007/3-540-63141-0_10, https://doi.org/10.1007/3-540-63141-0_10
17. Burckhardt, S.: Principles of eventual consistency. FTPL **1**(1-2), 1–150 (2014). https://doi.org/10.1561/2500000011, https://doi.org/10.1561/2500000011
18. Cerans, K.: Deciding properties of integral relational automata. In: ICALP94 Proceedings. LNCS, vol. 820, pp. 35–46. Springer (1994). https://doi.org/10.1007/3-540-58201-0_56, https://doi.org/10.1007/3-540-58201-0_56

19. Elver, M., Nagarajan, V.: TSO-CC: consistency directed cache coherence for TSO. In: HPCA. pp. 165–176. IEEE Computer Society (2014). https://doi.org/10.1109/HPCA.2014.6835927, https://doi.org/10.1109/HPCA.2014.6835927
20. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. **256**(1-2), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X, https://doi.org/10.1016/S0304-3975(00)00102-X
21. Kozen, D.: Lower bounds for natural proof systems. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977). pp. 254–266 (1977). https://doi.org/10.1109/SFCS.1977.16
22. Krishna, S.N., Godbole, A., Meyer, R., Chakraborty, S.: Parameterized verification under release acquire is PSPACE-complete. In: Milani, A., Woelfel, P. (eds.) PODC. pp. 482–492. ACM (2022). https://doi.org/10.1145/3519270.3538445, https://doi.org/10.1145/3519270.3538445
23. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: CAV. LNCS, vol. 5643, pp. 477–492. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_36, https://doi.org/10.1007/978-3-642-02658-4_36
24. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: SIGPLAN-SIGACT. pp. 649–662. ACM (2016). https://doi.org/10.1145/2837614.2837643, https://doi.org/10.1145/2837614.2837643
25. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. FMSD **35**(1), 73–97 (2009). https://doi.org/10.1007/s10703-009-0078-9, https://doi.org/10.1007/s10703-009-0078-9
26. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (aug 1974). https://doi.org/10.1145/361082.361093, https://doi.org/10.1145/361082.361093
27. Lazic, R., Newcomb, T.C., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with tokens which carry data. Fundam. Informaticae **88**(3), 251–274 (2008), http://content.iospress.com/articles/fundamenta-informaticae/fi88-3-03
28. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI. pp. 446–455. ACM (2007). https://doi.org/10.1145/1250734.1250785, https://doi.org/10.1145/1250734.1250785
29. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. LNCS, vol. 5674, pp. 391–407. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_27, https://doi.org/10.1007/978-3-642-03359-9_27
30. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. LNCS, vol. 3440, pp. 93–107. Springer (2005)
31. Ros, A., Kaxiras, S.: Racer: TSO consistency via race detection. In: MICRO. IEEE Computer Society (2016). https://doi.org/10.1109/MICRO.2016.7783736, https://doi.org/10.1109/MICRO.2016.7783736
32. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: ACM SIGPLAN, PLDI. pp. 175–186. ACM (2011). https://doi.org/10.1145/1993498.1993520, https://doi.org/10.1145/1993498.1993520
33. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010). https://doi.org/10.1145/1785414.1785443, https://doi.org/10.1145/1785414.1785443

13th Competition on Software Verification—SV-Comp 2024

State of the Art in Software Verification and Witness Validation: SV-COMP 2024

Dirk Beyer ✉

LMU Munich, Munich, Germany

Abstract. The 13th edition of the Competition on Software Verification (SV-COMP 2024) was the largest competition of its kind so far: A total of 76 tools for verification and witness validation were compared. The competition evaluated 59 verification systems and 17 validation systems from 34 teams from 12 countries. This yields a good overview of the state of the art in tools for software verification. The competition was executed on a benchmark set with 30 300 verification tasks for C programs and 587 verification tasks for Java programs. The specifications again included reachability, memory safety, overflows, and termination. This year was the second time that the competition had an extra competition track on witness validation. We introduced a new witness format 2.0, and a new scoring schema for the validation track. All meta data about the verification and validation tools are available in the FM-Tools repository.

Keywords: Formal Verification · Program Analysis · Competition · Software Verification · Verification Tasks · Benchmark · Specification · Java Language · C Language · SV-COMP · SV-Benchmarks · BenchExec · CoVeriTeam

1 Introduction

This report describes the results of the 2024 edition of SV-COMP, and is an extension of the series of competition reports (see footnote). We also list important processes and rules, and give insights into some aspects of the competition. The 13th Competition on Software Verification (`https://sv-comp.sosy-lab.org/2024`) is again the largest comparative evaluation ever in this area. The objectives of the competitions were discussed earlier (1-4 [22]) and extended over the years (5-6 [23]):

1. provide an overview of the state of the art in software-verification technology and increase visibility of the most recent software verifiers,
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,

This report extends previous reports on SV-COMP [16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27].
Reproduction packages are available on Zenodo (see Table 3).

✉ `dirk.beyer@sosy-lab.org`

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 299–329, 2024.
https://doi.org/10.1007/978-3-031-57256-2_15

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,
6. provide research teams that do not have sufficient computing resources with the opportunity to obtain experimental results on large benchmark sets, and
7. conserve tools for formal methods for later reuse by using a standardized format to announce archives (via DOIs), default options, contacts, competition participations, and other meta data in a central repository.

The SV-COMP 2020 report [23] discusses the achievements of the SV-COMP competition so far with respect to these objectives.

Related Competitions. SV-COMP is one of many competitions that measure progress of research in the are of formal methods [15]. Competitions can lead to fair and accurate comparative evaluations because of the involvement of the developing teams. The competitions most related to SV-COMP are RERS [80], VerifyThis [65], Test-Comp [28], and TermCOMP [73]. A previous report [23] provides a more detailed discussion.

Quick Summary of Changes. While we try to keep the setup of the competition stable, there are always improvements and developments. For the 2024 edition, the following changes were made:

- New verification tasks were added, with an increase in C from 23 805 in 2023 to 30 300 in 2024.
- Tool archives are now uploaded to Zenodo, instead of GitLab, and the meta data about the tools are hosted and maintained in the Repository for Formal-Methods Tools (`https://gitlab.com/sosy-lab/benchmarking/fm-tools`).
- The improved witness format version 2.0 [7] (which is based on YAML instead of GraphML) was used for the first time.
- The scoring schema for the witness validators [44] was changed based on the 2023 community meeting in Paris.

2 Organization, Definitions, Formats, and Rules

Procedure. The overall organization of the competition did not change in comparison to the earlier editions [16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27]. 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

Table 1: Scoring schema for SV-COMP 2024 (unchanged from 2021 [24])

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)

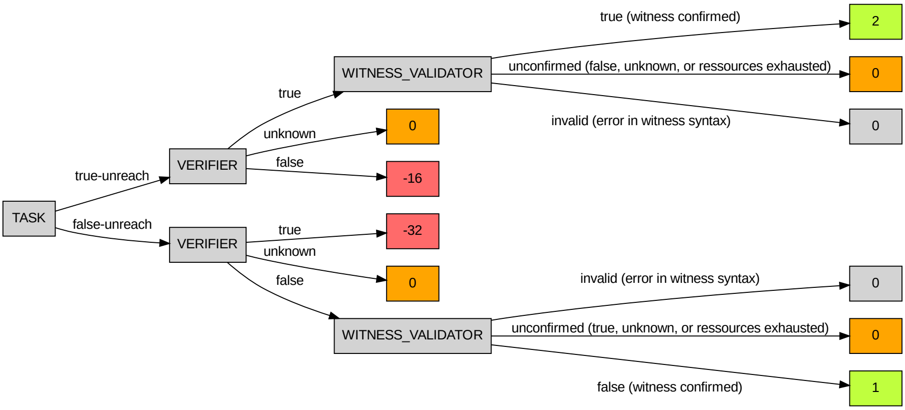

TASK
true-unreach
false-unreach
VERIFIER
VERIFIER
true
unknown
false
true
unknown
false
0
-16
-32
0
WITNESS_VALIDATOR
WITNESS_VALIDATOR
true (witness confirmed)
unconfirmed (false, unknown, or ressources exhausted)
invalid (error in witness syntax)
invalid (error in witness syntax)
unconfirmed (true, unknown, or ressources exhausted)
false (witness confirmed)
2
0
0
0
0
1

Fig. 1: Visualization of the scoring schema for the reachability property (unchanged from 2021 [24])

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 (cf. competition report of SV-COMP 2013 [17]). The tasks of the jury were described in more detail in the report of SV-COMP 2022 [26]. The team representatives of the competition jury are listed in Table 5.

Scoring Schema and Ranking. The scoring schema of SV-COMP 2024 was the same as for SV-COMP 2021. Table 1 provides an overview and Fig. 1 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 a correct verification result. *Opt-out from Categories* and *Score Normalization for Meta Categories* was done as described previously [17, page 597].

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),

Table 2: Publicly available components for reproducing SV-COMP 2024

Component	Fig. 3	Repository	Version
Verification Tasks	(a)	gitlab.com/sosy-lab/benchmarking/sv-benchmarks	svcomp24
Benchmark Definitions	(b)	gitlab.com/sosy-lab/sv-comp/bench-defs	svcomp24
Tool-Info Modules	(c)	github.com/sosy-lab/benchexec	3.21
Verifiers	(d)	gitlab.com/sosy-lab/benchmarking/fm-tools	svcomp24
Benchmarking	(e)	github.com/sosy-lab/benchexec	3.21
Witness Format	(f)	gitlab.com/sosy-lab/benchmarking/sv-witnesses	2.0.2
Continuous Integration		gitlab.com/sosy-lab/software/coveriteam	1.1

Table 3: Artifacts published for SV-COMP 2024

Content	DOI	Reference
Verification Tasks	10.5281/zenodo.10669723	[31]
Competition Results	10.5281/zenodo.10669731	[30]
Verifiers and Validators	10.5281/zenodo.10669735	[29]
Verification Witnesses	10.5281/zenodo.10669737	[32]
BenchExec	10.5281/zenodo.10671136	[122]
CoVeriTeam	10.5281/zenodo.10843666	[45]

(ii) does not restrict the usage of the verifier output (log files, witnesses), and
(iii) allows (re-)distribution of the unmodified verifier archive via SV-COMP repositories and archives.

Task-Definition Format 2.0. SV-COMP 2024 used the task-definition format in version 2.0. More details can be found in the report for Test-Comp 2021 [25].

Properties. Please see the 2015 competition report [19] for the definition of the properties and the property format. All specifications used in SV-COMP 2024 are available in the directory `c/properties/` of the benchmark repository.

Categories. The community significantly extended the benchmark set for SV-COMP 2024. The (updated) category structure of SV-COMP 2024 is shown in Fig. 2. We refer to the previous reports for a description and mention only the changes here: Compared to SV-COMP 2023, we added two new sub-categories *ReachSafety-Hardness* and *ReachSafety-Fuzzle* to main category *ReachSafety*. We restructured main category *SoftwareSystems* as follows: We removed sub-categories *SoftwareSystems-BusyBox-ReachSafety*, *SoftwareSystems-BusyBox-MemSafety*, and *SoftwareSystems-OpenBSD-MemSafety*, and added sub-categories *SoftwareSystems-coreutils-MemSafety*, *SoftwareSystems-coreutils-NoOverflows*, *SoftwareSystems-Other-ReachSafety*, and *SoftwareSystems-Other-MemSafety*. 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/2024/benchmarks.php`).

Reproducibility. SV-COMP results must be reproducible, and consequently, all major components are maintained in public version-control repositories. The

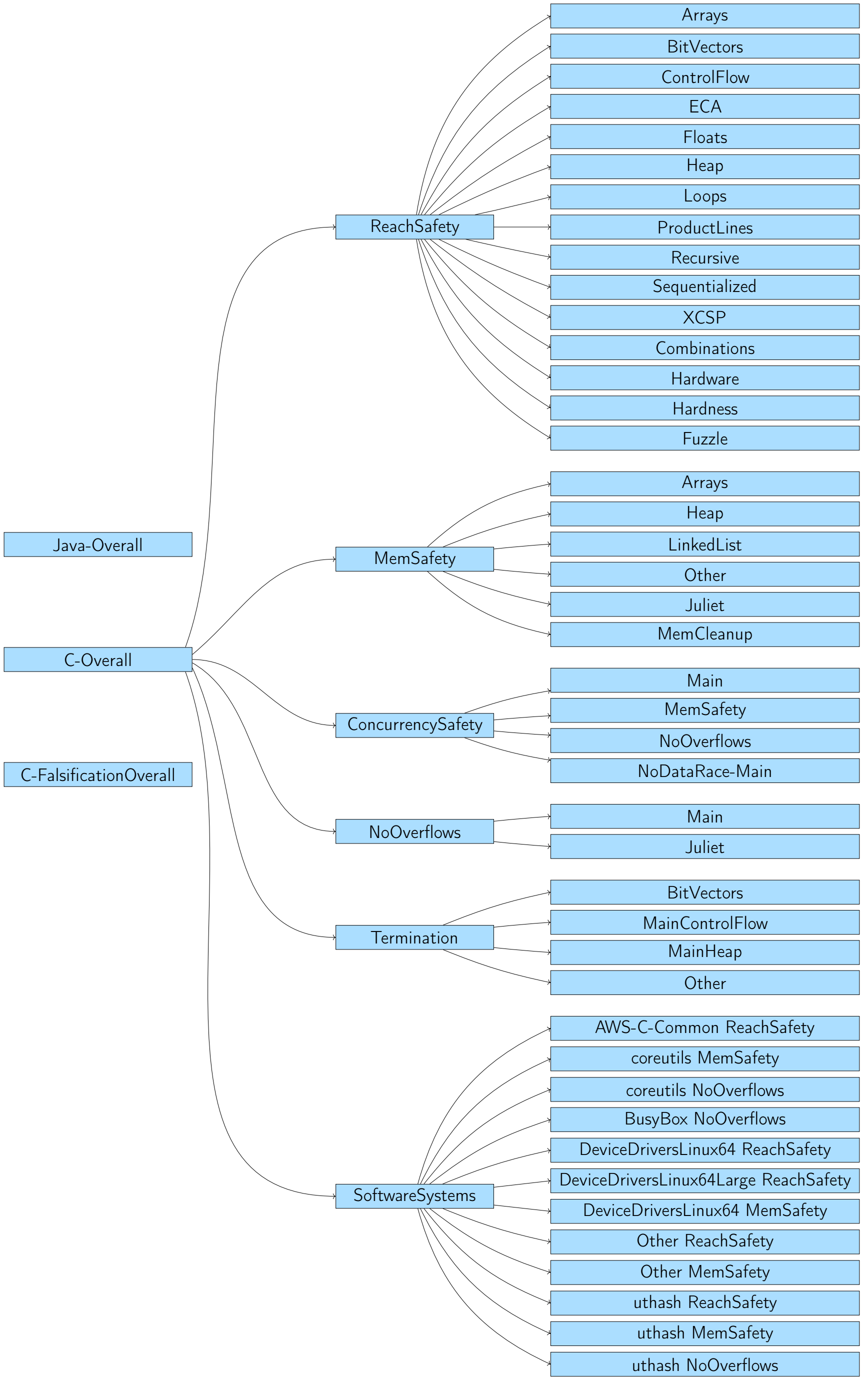


Fig. 2: Category structure for SV-COMP 2024 (changed from 2023)

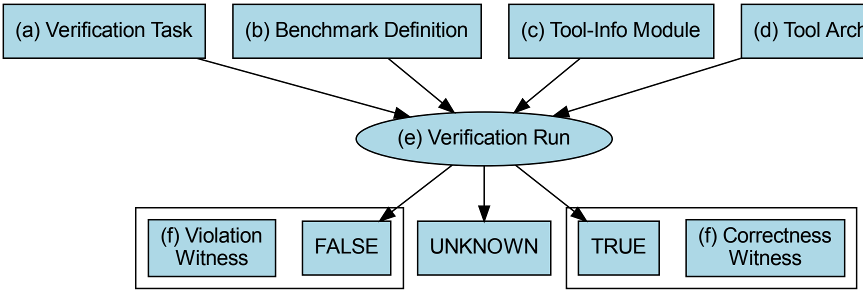


Fig. 3: Benchmarking components of SV-COMP and competition's execution flow (same as for SV-COMP 2020, except that we now download the tool archives from Zenodo instead of GitLab)

Table 4: Validation: Witness validators and witness linter

Validator	**Reference**	**Jury Member**	**Affiliation**
ConcurWitness2Test $^{\text{new}}$	[13]	L. Bajczi	BME Budapest, Hungary
CPAchecker	[33, 34, 36]	D. Baier	LMU Munich, Germany
CPA-witness2test	[35]	T. Lemberger	LMU Munich, Germany
Dartagnan	[106]	H. Ponce de León	Huawei Dresden, Germany
CProver-witness2test $^{\varnothing}$	[35]	(hors concours)	–
Goblint $^{\text{new}}$	[112]	S. Saan	U. of Tartu, Estonia
GWIT	[81]	F. Howar	TU Dortmund, Germany
JCWIT $^{\text{new}}$		Z. Cheng	U. of Manchester, UK
LIV $^{\text{new}}$	[43]	M. Spiessl	LMU Munich, Germany
MetaVal	[41]	M. Spiessl	LMU Munich, Germany
Mopsa $^{\text{new}}$	[99]	R. Monat	Inria and U. of Lille, France
NITWIT	[124]	J. (P.) Berger	RWTH Aachen, Germany
Symbiotic-Witch	[8]	P. Ayaziová	Masaryk U., Brno, Czechia
UAutomizer	[33, 34]	M. Heizmann	U. of Freiburg, Germany
WIT4JAVA $^{\varnothing}$	[123]	(hors concours)	–
Witch $^{\text{new}}$	[7, 9]	P. Ayaziová	Masaryk U., Brno, Czechia
WitnessLint	[7]	M. Lingsch-Rosenfeld	LMU Munich, Germany

overview of the components is provided in Fig. 3, and the details are given in Table 2. We refer to the SV-COMP 2016 report [20] for a description of all components of the SV-COMP organization. There are competition artifacts at Zenodo (see Table 3) to guarantee their long-term availability and immutability.

Competition Workflow. The workflow of the competition is described in the report for Test-Comp 2021 [25] (SV-COMP and Test-Comp use a similar workflow). For a description of how to reproduce single verification runs and a trouble-shooting guide, we refer to the 2022 report [26, Sect. 3].

Table 5: Verification: Participating verifiers with tool references and representing jury members; $^{\text{new}}$ for first-time, $^{\varnothing}$ for hors-concours (RELAY-SV$^{\text{new}}$ was not able to qualify)

Participant	Ref.	Jury member	Affiliation
2LS	[46, 96]	V. Malík	BUT, Czechia
AISE$^{\text{new}}$	[121]	Z. Chen	NUDT, China
BRICK	[47]	L. Bu	Nanjing U., China
Bubaak	[49]	M. Chalupa	ISTA, Austria
Bubaak-SpLit$^{\text{new}}$	[50]	M. Chalupa	ISTA, Austria
CBMC$^{\varnothing}$	[54, 91]	(h. c.)	–
COASTAL$^{\varnothing}$	[118]	(h. c.)	–
CoVeriTeam-AlgSel$^{\varnothing}$	[37, 38]	(h. c.)	–
CoVeriTeam-ParPort$^{\varnothing}$	[37, 38]	(h. c.)	–
CPAchecker	[10, 39]	D. Baier	LMU Munich, Germany
CPALockator$^{\varnothing}$	[5, 6]	(h. c.)	–
CPA-BAM-BnB$^{\varnothing}$	[4, 120]	(h. c.)	–
CPA-BAM-SMG$^{\varnothing}$		(h. c.)	–
CPV$^{\text{new}}$	[53]	P.-C. Chien	LMU Munich, Germany
Crux$^{\varnothing}$	[64, 113]	(h. c.)	–
CSeq$^{\varnothing}$	[59, 85]	(h. c.)	–
Dartagnan	[71, 105]	H. Ponce de León	Huawei Dresden, Germany
Deagle	[76]	F. He	Tsinghua U., China
DIVINE$^{\varnothing}$	[14, 92]	(h. c.)	–
EBF	[3]	F. Aljaafari	U. of Manchester, UK
EmergenTheta$^{\text{new}}$	[11]	L. Bajczi	BME Budapest, Hungary
ESBMC-incr$^{\varnothing}$	[55, 58]	(h. c.)	–
ESBMC-kind	[70, 97]	F. Brauße	U. Manchester, UK
Frama-C-SV	[42, 60]	M. Spiessl	LMU Munich, Germany
Gazer-Theta$^{\varnothing}$	[1, 75]	(h. c.)	–
GDart	[101]	F. Howar	TU Dortmund, Germany
GDart-LLVM$^{\varnothing}$		(h. c.)	–
Goblint	[111, 119]	S. Saan	U. Tartu, Estonia
Graves-CPA$^{\varnothing}$	[93]	(h. c.)	–
Graves-Par$^{\varnothing}$		(h. c.)	–
Infer$^{\varnothing}$	[48, 89]	(h. c.)	–
Java-Ranger$^{\varnothing}$	[82, 115]	(h. c.)	–
JayHorn	[88, 114]	H. Mousavi	U. Tehran, TIAS, Iran
JBMC	[56, 57]	P. Schrammel	U. Sussex / Diffblue, UK
JDart$^{\varnothing}$	[95, 100]	(h. c.)	–
Korn	[67, 68]	G. Ernst	LMU Munich, Germany
Lazy-CSeq$^{\varnothing}$	[83, 84]	(h. c.)	–
LF-checker$^{\varnothing}$		(h. c.)	–
Locksmith$^{\varnothing}$	[107]	(h. c.)	–
MLB		L. Bu	Nanjing U., China
Mopsa	[87, 99]	R. Monat	Inria and U. Lille, France

(continues on next page)

Table 5: Competition candidates (continued)

Participant	Ref.	Jury member	Affiliation
PeSCo-CPA$^{\varnothing}$	[109, 110]	(h. c.)	–
PIChecker$^{\varnothing}$	[116]	(h. c.)	–
Pinaka$^{\varnothing}$	[52]	(h. c.)	–
PredatorHP	[79, 104]	V. Šoková	BUT, Czechia
Proton$^{\text{new}}$	[98]	R. Metta	TCS, India
SPF$^{\varnothing}$	[102, 108]	(h. c.)	–
sv-sanitizers$^{\text{new}}$		S. Saan	U. of Tartu, Estonia
SWAT$^{\text{new}}$	[94]	N. Loose	U. of Luebeck, Germany
Symbiotic	[51, 86]	M. Jonáš	Masaryk U., Czechia
Theta	[12, 117]	L. Bajczi	BME Budapest, Hungary
UAutomizer	[77, 78]	M. Heizmann	U. Freiburg, Germany
UGemCutter	[69, 90]	D. Klumpp	U. Freiburg, Germany
UKojak	[66, 103]	F. Schüssele	U. Freiburg, Germany
UTaipan	[63, 74]	D. Dietsch	U. Freiburg, Germany
VeriAbs	[2, 61]	P. Darke	TCS, India
VeriAbsL	[62]	P. Darke	TCS, India
VeriOover$^{\varnothing}$		(h. c.)	–

Table 6: Algorithms and techniques that the participating verification systems used; $^{\text{new}}$ for first-time participants, $^{\varnothing}$ 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				✓	✓			✓	✓		✓						✓			
aise$^{\text{new}}$			✓																	
BRICK	✓		✓	✓				✓								✓				
Bubaak			✓								✓					✓	✓			✓
Bubaak-SpLit$^{\text{new}}$			✓		✓						✓				✓	✓	✓		✓	✓
CBMC$^{\varnothing}$				✓							✓					✓				
COASTAL$^{\varnothing}$			✓																	
CVT-AlgoSel$^{\varnothing}$	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓
CVT-ParPort$^{\varnothing}$	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓
CPAchecker	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓
CPALockator$^{\varnothing}$	✓	✓					✓				✓	✓	✓	✓		✓				

(continues on next page)

Table 6: Algorithms and techniques (continued)

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
CPA-BAM-BnB$^{\varnothing}$	✓	✓					✓				✓	✓	✓	✓						
CPA-BAM-SMG$^{\varnothing}$																				
CPV$^{\text{new}}$				✓	✓	✓								✓						
Crux$^{\varnothing}$			✓																	
CSeq$^{\varnothing}$				✓							✓					✓				
Dartagnan				✓							✓					✓				
Deagle				✓												✓				
DIVINE$^{\varnothing}$			✓				✓				✓					✓			✓	✓
EBF				✓																
EmergenTheta$^{\text{new}}$				✓	✓						✓	✓		✓					✓	✓
ESBMC-incr$^{\varnothing}$				✓	✓						✓					✓				
ESBMC-kind				✓	✓		✓	✓			✓					✓				
Frama-C-SV								✓												
Gazer-Theta$^{\varnothing}$	✓	✓		✓			✓				✓	✓	✓	✓						✓
GDart			✓								✓									✓
GDart-LLVM$^{\varnothing}$			✓								✓									
Goblint								✓								✓			✓	
Graves-CPA$^{\varnothing}$	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓
Graves-Par$^{\varnothing}$																				
Infer$^{\varnothing}$								✓	✓	✓										✓
Java-Ranger$^{\varnothing}$			✓								✓									
JayHorn	✓	✓				✓		✓					✓	✓						
JBMC				✓							✓					✓				
JDart$^{\varnothing}$			✓								✓									✓
Korn		✓	✓				✓													✓
Lazy-CSeq$^{\varnothing}$				✓							✓					✓				
LF-checker$^{\varnothing}$																				
Locksmith$^{\varnothing}$																✓				
MLB			✓								✓									✓
Mopsa								✓												
PeSCo-CPA$^{\varnothing}$	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓
PIChecker$^{\varnothing}$	✓	✓									✓	✓		✓		✓			✓	
Pinaka$^{\varnothing}$			✓	✓							✓									
PredatorHP									✓											

(continues on next page)

Table 6: Algorithms and techniques (continued)

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
Proton new				✓																
SPF ∅			✓						✓							✓				
sv-sanitizers new																✓				
SWAT new			✓																	
Symbiotic			✓		✓			✓	✓		✓					✓				✓
Theta	✓	✓					✓				✓	✓		✓		✓			✓	✓
UAutomizer	✓	✓									✓		✓	✓	✓	✓	✓		✓	✓
UGemCutter	✓	✓									✓		✓	✓	✓	✓			✓	✓
UKojak	✓	✓									✓		✓	✓						
UTaipan	✓	✓					✓	✓			✓		✓	✓	✓	✓			✓	✓
VeriAbs	✓			✓	✓		✓	✓										✓	✓	✓
VeriAbsL	✓			✓	✓		✓	✓										✓	✓	✓
VeriOover ∅																				

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	Apron
2ls		✓									✓	
aise new												
BRICK										✓	✓	
Bubaak										✓		
Bubaak-SpLit new												
CBMC ∅		✓									✓	
COASTAL ∅				✓								
CVT-AlgoSel ∅	✓	✓	✓		✓	✓	✓				✓	
CVT-ParPort ∅	✓	✓	✓		✓	✓	✓				✓	
CPAchecker	✓					✓	✓					✓

(continues on next page)

Table 7: Solver libraries and frameworks (continued)

Verifier	CPAchecker	CProver	Esbmc	Jpf	Ultimate	JavaSMT	MathSAT	Cvc4	SMTInterpol	z3	MiniSAT	Apron
CPALockator$^{\varnothing}$	✓					✓	✓					
CPA-BAM-BnB$^{\varnothing}$	✓					✓	✓					
CPA-BAM-SMG$^{\varnothing}$	✓					✓	✓					
Crux$^{\varnothing}$										✓		
CSeq$^{\varnothing}$		✓									✓	
Dartagnan						✓						
Deagle											✓	
DIVINE$^{\varnothing}$												
EBF			✓				✓					
EmergenTheta$^{\text{new}}$												
ESBMC-incr$^{\varnothing}$			✓				✓					
ESBMC-kind			✓				✓					
Frama-C-SV												
Gazer-Theta$^{\varnothing}$												
GDart								✓		✓		
GDart-LLVM$^{\varnothing}$										✓		
Goblint												✓
Graves-CPA$^{\varnothing}$	✓					✓	✓					
Graves-Par$^{\varnothing}$												
Infer$^{\varnothing}$												
Java-Ranger$^{\varnothing}$				✓								
JayHorn												
JBMC		✓									✓	
JDart$^{\varnothing}$				✓				✓		✓		
Korn										✓		
Lazy-CSeq$^{\varnothing}$		✓									✓	
LF-checker$^{\varnothing}$												
Locksmith$^{\varnothing}$												
MLB												
Mopsa												✓
PeSCo-CPA$^{\varnothing}$	✓					✓	✓					
PIChecker$^{\varnothing}$	✓					✓	✓		✓			
Pinaka$^{\varnothing}$												
PredatorHP												
Proton$^{\text{new}}$												
SPF$^{\varnothing}$				✓								
sv-sanitizers$^{\text{new}}$												

(continues on next page)

Table 7: Solver libraries and frameworks (continued)

Verifier	CPAchecker	CProver	Esbmc	Jpf	Ultimate	JavaSMT	MathSAT	Cvc4	SMTInterpol	z3	MiniSAT	Apron
SWAT new												
Symbiotic										✓		
Theta												
UAutomizer					✓		✓	✓	✓	✓		
UGemCutter					✓		✓	✓	✓	✓		
UKojak					✓				✓	✓		
UTaipan					✓		✓	✓	✓	✓		
VeriAbs	✓	✓								✓	✓	
VeriAbsL	✓	✓								✓	✓	
VeriOover∅												

3 Participating Verifiers and Validators

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/2024/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.

Validation of Verification Results. The validation of the verification results was done by 17 validation tools (16 proper witness validators, and one witness linter for syntax checks), which are listed in Table 4, including references to literature. The ten witness validators are evaluated based on all verification witnesses that were produced in the verification track of the competition.

Hors-Concours Participation. As in previous years, we also included verifiers to the evaluation that did not actively compete or that should not occur in the rankings for some reasons (e.g., meta verifiers based on other competing tools, or tools for which the submitting teams were not sure if they show the full potential of the tool). These participations are called *hors concours*, as they cannot participate in rankings and cannot “win” the competition. Those verifiers are marked as ‘hors concours’ in Table 5 and others, and the names are annotated with a symbol (∅).

4 Results of the Verification Track

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

Table 8: Verification: Quantitative overview over all regular results; empty cells are used for opt-outs; new for first-time participants; the number of tasks includes invalid tasks that were excluded from scoring by the jury (details available on web site or in artifact)

Participant	**ReachSafety** 17746 points 11305 tasks	**MemSafety** 3216 points 2135 tasks	**ConcurrencySafety** 5672 points 3259 tasks	**NoOverflows** 13044 points 8188 tasks	**Termination** 4000 points 2354 tasks	**SoftwareSystems** 5251 points 3813 tasks	**FalsificationOverall** 8817 points 28700 tasks	**Overall** 49097 points 31054 tasks	**JavaOverall** 828 points 587 tasks
2LS	6000	224	0	5976	**1584**	10	1311	10564	
AISE new									
BRICK									
Bubaak	3788	1890	11	6465	1481	-1082	-617	12206	
Bubaak-SpLit new	4692	1312	7	-41374	661	**872**	1959	-18177	
CPAchecker	**10084**	1897	2029	**8603**	1195	**784**	**4812**	**21568**	
CPV new	6330								
Dartagnan			**3547**						
Deagle									
EBF			636						
EmergenTheta new	1178								
ESBMC-kind	8364	2077	1853	8272	1048	-1063	2394	17896	
Frama-C-SV				1098					
GDart									**616**
Goblint	2289	1304	2583	7059	890	536		15458	
JayHorn									325
JBMC									**618**
Korn									
MLB									**676**
Mopsa	2241	1516		8063		**2197**			
PredatorHP		**2321**							
Proton new					**3526**				
sv-sanitizers new		290							
SWAT new									566
Symbiotic	7052	**2156**	238	7370	1258	687	**4050**	17192	
Theta	2119		2354						
UAutomizer	6320	**2110**	**3079**	**9497**	**3248**	261	3139	**26396**	
UGemCutter			**3189**						
UKojak	4869	1400	0	7363	0	233	2291	10593	
UTaipan	5751	2014	2655	**9231**	0	351	**3157**	**18042**	
VeriAbs	**10541**								
VeriAbsL	**10735**								

Table 9: Verification: Quantitative overview over all hors-concours results; empty cells represent opt-outs, $^{\varnothing}$ for hors-concours participation; the number of tasks includes invalid tasks that were excluded from scoring by the jury (details available on web site or in artifact)

Participant	**ReachSafety** 17746 points 11305 tasks	**MemSafety** 3216 points 2135 tasks	**ConcurrencySafety** 5672 points 3259 tasks	**NoOverflows** 13044 points 8188 tasks	**Termination** 4000 points 2354 tasks	**SoftwareSystems** 5251 points 3813 tasks	**FalsificationOverall** 8817 points 28700 tasks	**Overall** 49097 points 31054 tasks	**JavaOverall** 828 points 587 tasks
CBMC$^{\varnothing}$	1269	1330	1229	5771	1125	-2569	-3764	8391	
COASTAL$^{\varnothing}$									-2752
CVT-AlgoSel$^{\varnothing}$	2635		41						
CVT-ParPort$^{\varnothing}$	-6152	1655	911	-17812	1289	-1297	-9118	-7545	
CPA-BAM-BnB$^{\varnothing}$						-2439			
CPA-BAM-SMG$^{\varnothing}$		2039				-2804			
CPALockator$^{\varnothing}$			-4924						
Crux$^{\varnothing}$	2066			490					
CSeq$^{\varnothing}$			-12478						
DIVINE$^{\varnothing}$	4655	298	390	0	0	76	256	3576	
ESBMC-incr$^{\varnothing}$			542						
Gazer-Theta$^{\varnothing}$									
GDart-LLVM$^{\varnothing}$									
Graves-CPA$^{\varnothing}$	3831					-322	-1538	5470	
Graves-Par$^{\varnothing}$	876	1627	53	-17650	1256	-2037	-9024	-6731	
Infer$^{\varnothing}$	-99128		-8289	-73312		-24917			
Java-Ranger$^{\varnothing}$									398
JDart$^{\varnothing}$									382
Lazy-CSeq$^{\varnothing}$			-15024						
LF-checker$^{\varnothing}$			772						
Locksmith$^{\varnothing}$									
PeSCo-CPA$^{\varnothing}$	5814					-76	3247	17315	
PIChecker$^{\varnothing}$			521						
Pinaka$^{\varnothing}$	2418			1337	855				
SPF$^{\varnothing}$									182
VeriOover$^{\varnothing}$									

and correctness of the results, as accumulated in the score) and the efficiency (resource consumption in terms of CPU time). The results are presented in the same way as in last years, such that the improvements compared to the last years are easy to identify. The results presented in this report were inspected and approved by the participating teams.

Table 10: Verification: Overview of the top-three verifiers for each category; values for CPU time rounded to two significant digits; $^{\text{new}}$ for first-time participants

Rank	**Verifier**	**Score**	**CPU**	**Solved**	**Unconf.**	**False**	**Wrong**
			Time	**Tasks**	**Tasks**	**Alarms**	**Proofs**
			(in h)				
ReachSafety							
1	**VeriAbsL**	**10735**	190	7 075	1 138		**2**
2	VeriAbs	10541	190	6 720	1 032		**1**
3	CPAchecker	10084	200	6 468	286	2	
MemSafety							
1	**PredatorHP**	**2321**	1.2	1 823	3	3	
2	Symbiotic	2156	0.77	1 855	0		**5**
3	UAutomizer	2110	62	1 637	4		
ConcurrencySafety							
1	**Dartagnan**	**3547**	14	2 086	0		**5**
2	UGemCutter	3189	32	1 851	4	1	
3	UAutomizer	3079	28	1 791	3		**1**
NoOverflows							
1	**UAutomizer**	**9497**	62	4 532	2		
2	UTaipan	9231	66	4 420	11		**1**
3	CPAchecker	8603	18	5 596	192		
Termination							
1	**Proton** $^{\text{new}}$	**3526**	19	1 888	126	1	
2	UAutomizer	3248	18	1 631	11		
3	2ls	1584	4.2	1 167	201		
SoftwareSystems							
1	**Mopsa**	**2197**	15	2 030	0		
2	Bubaak-SpLit $^{\text{new}}$	872	0.42	480	163	8	
3	CPAchecker	784	43	1 756	71		
FalsificationOverall							
1	**CPAchecker**	**4812**	91	4 920	218	10	
2	Symbiotic	4050	27	4 281	191	11	
3	UTaipan	3157	33	1 602	34	1	
Overall							
1	**UAutomizer**	**26396**	290	13 617	114	3	**7**
2	CPAchecker	21568	320	17 968	698	16	**1**
3	UTaipan	18042	240	11 524	71	1	**13**
JavaOverall							
1	**MLB**	**676**	0.93	484	34		
2	JBMC	618	0.44	424	80		
3	GDart	616	2.6	453	9		

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

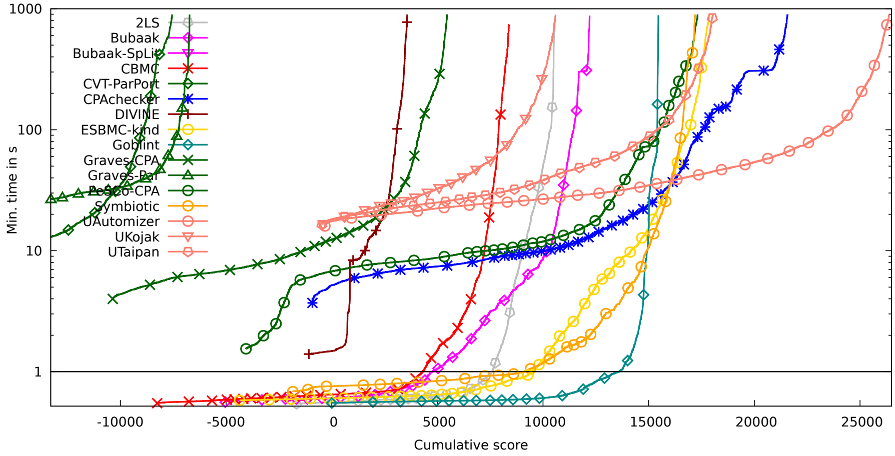


Fig. 4: Quantile functions for category *C-Overall*. Each quantile function illustrates the quantile (x-coordinate) of the scores obtained by correct verification runs below a certain run time (y-coordinate). More details were given previously [17]. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s.

tasks. The verification tasks consist of tasks with expected verdict True, expected verdict False, and tasks that are Void (tasks that were excluded from scoring by the jury). 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; Deagle was disqualified by the jury, with details on the web site). 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/2024/results`) and in the results artifact (see Table 3).

Table 10 reports the top three verifiers for each category. The run time (column 'CPU Time') refers 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.

Score-Based Quantile Functions for Quality Assessment. We use score-based quantile functions [17, 40] because these visualizations make it easier to understand the results of the comparative evaluation. The results archive (see Table 3) and the web site (`https://sv-comp.sosy-lab.org/2024/results`) include such a plot for each (sub-)category. As an example, we show the plot for category

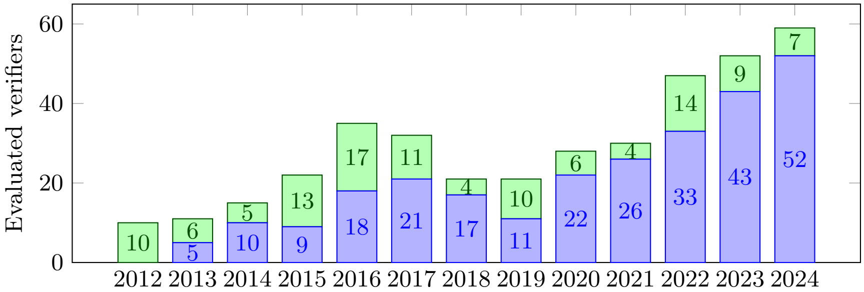


Fig. 5: Number of evaluated verifiers for each year (first-time participants on top)

Table 11: New verifiers in SV-COMP 2023 and SV-COMP 2024; column 'Sub-categories' gives the number of executed categories; $^{\text{new}}$ for first-time participants in 2024; $^{\varnothing}$ for those that were hors-concours participants in 2024

Verifier	Language	First Year	Sub-categories
AISE$^{\text{new}}$	C	2024	1
Bubaak-SpLit$^{\text{new}}$	C	2024	45
CPV$^{\text{new}}$	C	2024	20
EmergenTheta$^{\text{new}}$	C	2024	15
Proton$^{\text{new}}$	C	2024	5
sv-sanitizers$^{\text{new}}$	C	2024	13
SWAT$^{\text{new}}$	Java	2024	1
Bubaak	C	2023	40
GDart-LLVM$^{\varnothing}$	C	2023	1
Graves-Par$^{\varnothing}$	C	2023	40
LF-checker$^{\varnothing}$	C	2023	3
Mopsa	C	2023	32
PIChecker$^{\varnothing}$	C	2023	1
VeriAbsL	C	2023	13
VeriOover$^{\varnothing}$	C	2023	1
MLB	Java	2023	1

C-Overall (all verification tasks) in Fig. 4. A total of 16 verifiers participated in category *C-Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [17]). 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 [17, 20].

The winner of the competition, UAutomizer, achieves the best cumulative score (the graph for UAutomizer has the longest width from its left to its right end; the graph starts left from $x = 0$ because the verifier produced 7 wrong proofs and 4 false alarms and therefore received some negative points). Also other verifiers whose graphs start with a negative cumulative score produced wrong results.

New Verifiers. To acknowledge the verification systems that participate for the first or second time in SV-COMP, Table 11 lists the new verifiers (in SV-COMP 2023 or SV-COMP 2024). Figure 5 shows the growing interest in the competition over the years.

Computing Resources. The CPU time and memory limits were the same as in the previous competitions [20] (15 GB of memory and 15 min of CPU time), but we reduced the number of processing units per run from 8 to 4 processing units. This has the disadvantage that the measurements are more imprecise due to shared resources in the machine, but it roughly doubles the throughput. This change was necessary because of the ever increasing number of participating systems and the continuously increasing benchmark set. Witness validation was again 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 for running the experiments are part of a compute cluster at the SoSy-Lab at LMU that consists of 168 machines, where each machine has 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 22.04 with Linux kernel 5.15). We used BenchExec [40] to measure and control computing resources (CPU time, memory) and VCloud to distribute, install, run, and clean-up verification runs, and to collect the results. The values for the time are accumulated over all cores of the CPU.

To give an impression of the overall computation work, we report some statistics: One complete verification execution of the competition consisted of 787 779 verification runs (each verifier on each verification task of the selected categories according to the opt-outs), consuming 2 104 days of CPU time (without validation). This is almost double the CPU time spent for the previous edition of SV-COMP. Witness-based result validation required 13.6 million validation runs in 21 243 run sets (each validator on each verification task for categories with witness validation, and for each verifier), consuming 2290 days of CPU time. Each tool was executed several times, in order to make sure no installation issues occur during the execution.

5 Results of the Witness-Validation Track

The validation of verification results, in particular, verification witnesses, becomes more and more important for various reasons: verification witnesses justify and help to understand and interpret a verification result, they serve as exchange object for intermediate results, and they allow to make use of imprecise verification techniques (e.g., via machine learning). A case study on the quality of the results of witness validators [44] suggested that validators for verification results should also undergo a periodical comparative evaluation and proposed a scoring schema for witness-validation results. SV-COMP 2024 evaluated a total of 17 validators on 100 998 correctness and 71 577 violation witnesses in format 1.0, and 45 614 correctness and 27 561 violation witnesses in format 2.0. Figure 6 shows the growing importance of evaluating witness validators.

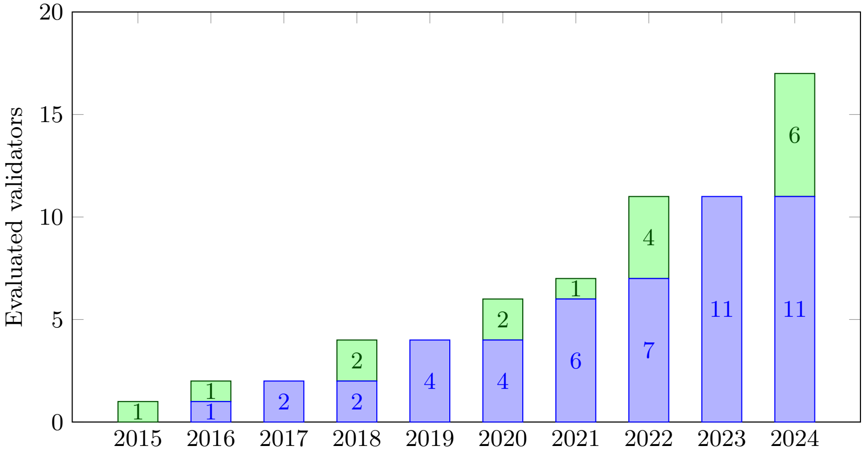


Fig. 6: Number of witness validators for each year (first-time participants on top)

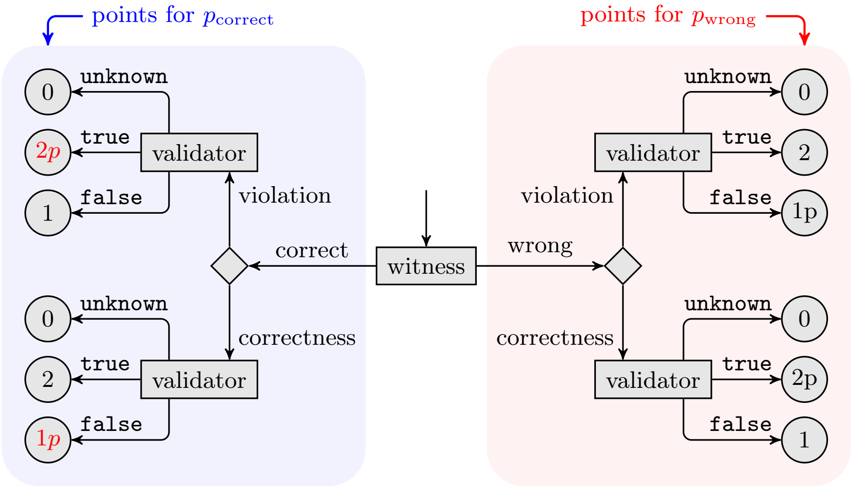


Fig. 7: Scoring schema for evaluation of validators; $p = -16$ for SV-COMP 2024; figure adopted from [44]; changed scores compared to 2023 are highlighted in red

Scoring Schema for Validation Track. The score of a validator in a subcategory is computed as

$$score = \left(\frac{p_{\text{correct}}}{|\text{correct}|} + \frac{p_{\text{wrong}}}{|\text{wrong}|} \right) \cdot \frac{|\text{correct}| + |\text{wrong}|}{2}$$

where the points in p_{correct} and p_{wrong} are determined according to the schema in Fig. 7 and then normalized using the normalization schema that SV-COMP uses for meta categories [17, page 597] (note that the factor q is removed in comparison to last year [27, page 513] from the formula, because it is not necessary to give a higher

Table 12: Validation of correctness witnesses (version 2.0): Overview of the top-three validators for each category; values for CPU time rounded to two significant digits

Rank	Validator	Score	CPU	Solved	False	Wrong
			Time	Tasks	Alarms	Proofs
			(in h)			
ReachSafety						
1	**UAutomizer**	**4545**	69	3 830		
2	Mopsa $^{\text{new}}$	3284	17	2 816		
3	CPAchecker	2872	23	2 784		
MemSafety						
1	**UAutomizer**	**5213**	190	5 701		
2	Mopsa $^{\text{new}}$	5015	2.5	5 658		
3	Goblint $^{\text{new}}$	4225	0.26	5 677		
ConcurrencySafety						
1	**Goblint** $^{\text{new}}$	**0**	0	0		
2	missing validator	0	0	0		
3	missing validator	0	0	0		
NoOverflows						
1	**UAutomizer**	**25441**	220	17 913		
2	Mopsa $^{\text{new}}$	23601	7.8	17 333		
3	Goblint $^{\text{new}}$	17143	0.77	14 125		
Termination						
1	**Goblint** $^{\text{new}}$	**0**	0	0		
2	missing validator	0	0	0		
3	missing validator	0	0	0		
SoftwareSystems						
1	**Mopsa** $^{\text{new}}$	**3521**	23	6 102		
2	Goblint $^{\text{new}}$	2793	9.2	4 636		
3	UAutomizer	1258	90	5 963		**14**
Overall						
1	**UAutomizer**	**20919**	570	33 407		**14**
2	Mopsa $^{\text{new}}$	20889	50	31 909		
3	Goblint $^{\text{new}}$	16186	11	26 224		

weight to wrong witnesses anymore). Witnesses that do not agree with the expected verification verdict are classified as *wrong*. Witnesses that agree with the expected verification verdict can be wrong although they agree with the expected version, for example, if a violation witness has a wrong path to the violation, or a correctness witness has an invariant that does not hold. Therefore, we use the information from the majority of the validators: a witness that agrees with the expected verification result is classified as *correct* if at least 75 % of the true/false results from validators confirm the result, and as *wrong* if at least 75 % of the true/false results from validators refute this result (and there must be at least 2 true/false results). Further details are given in the proposal [44]. This schema relates to

Table 13: Validation of correctness witnesses (version 1.0): Overview of the top-three validators for each category; values for CPU time rounded to two significant digits

Rank	Validator	Score	CPU Time (in h)	Solved Tasks	False Alarms	Wrong Proofs
ReachSafety						
1	**UAutomizer**	**28020**	540	21 331		
2	CPAchecker	25183	250	29 082		
3	LIV new	-44060	3.7	2 527		**31**
MemSafety						
1	**UAutomizer**	**259**	4.8	366		
2	LIV new	87	0.22	186		**6**
3	MetaVal	0	0	0		
ConcurrencySafety						
1	**UAutomizer**	**70**	2.6	120		
2	missing validator	0	0	0		
3	missing validator	0	0	0		
NoOverflows						
1	**CPAchecker**	**57309**	170	45 618		**9**
2	UAutomizer	56467	320	42 011		**2**
3	MetaVal	0	0	0		
Termination						
1	**missing validator**	**0**	0	0		
2	missing validator	0	0	0		
3	missing validator	0	0	0		
SoftwareSystems						
1	**CPAchecker**	**3275**	28	5 812		
2	UAutomizer	2211	240	13 916		**18**
3	LIV new	0	0	0		
Overall						
1	**UAutomizer**	**47571**	1 100	77 744		**20**
2	CPAchecker	35095	450	80 512		**9**
3	MetaVal	-38172	1 300	44 296		**504**

each base category from the verification track a meta category that consists of two sub-categories, one with the correct and one with the wrong witnesses.

Tables 12, 13, and 14 show the rankings of the validators. Violation witnesses in format version 2.0 were not yet ranked, because the jury decided that in SV-COMP 2024, this is a demonstration track. The score results for all validators and all categories are available on the SV-COMP web site [1] and in the artifact [30]. Wrong proofs in Tables 12 and 13 are claims of a validator that the program is correct according to invariants in a given correctness witness although the program contains a bug (the validator confirms a wrong correctness witness). False alarms in Table 14 are claims of a validator that the program contains

[1] https://sv-comp.sosy-lab.org/2024/results/results-validated/

Table 14: Validation of violation witnesses (version 1.0): Overview of the top-three validators for each category; values for CPU time rounded to two significant digits

Rank	Validator	Score	CPU Time (in h)	Solved Tasks	False Alarms	Wrong Proofs
ReachSafety						
1	**UAutomizer**	**24390**	120	15 932	4	
2	CProver-w2t	22251	18	16 970	2	
3	CPAchecker	15686	83	16 602	71	
MemSafety						
1	**Symbiotic-Witch**	**799**	0.59	1 723		
2	CPAchecker	626	3.7	1 570	6	
3	UAutomizer	472	5.2	809		
ConcurrencySafety						
1	**Dartagnan**	**9186**	37	8 674		
2	UAutomizer	6742	72	6 533		
3	CPAchecker	2110	14	3 061	28	
NoOverflows						
1	**UAutomizer**	**20030**	63	10 236	5	
2	CPAchecker	18892	81	14 323		
3	CProver-w2t	18400	7.7	13 679	18	
Termination						
1	**UAutomizer**	**692**	7.0	1 004		
2	MetaVal	0	0	0		
3	CPAchecker	-1496	5.3	993	26	
SoftwareSystems						
1	**UAutomizer**	**2633**	26	3 036	2	
2	Symbiotic-Witch	1696	0.59	1 113		
3	CPAchecker	1359	15	2 474		
Overall						
1	**UAutomizer**	**43235**	290	37 550	11	
2	Symbiotic-Witch	20980	42	27 484	4	
3	CProver-w2t	19651	27	32 936	178	

a bug described by a given violation witness although the program is correct (the validator confirms a wrong violation witness).

The adoption rate of the new witness format version 2.0 is discussed in the article that defines the format [7]. Tables 12 and 13 shows that there are categories that are supported still by less than three validators (‘missing validators’ for categories *ConcurrencySafety* and *Termination*).

While there are six new validators in SV-COMP 2024 (Fig. 6), and while there is a great adoption rate of the new witness format 2.0 (Table 12), there is still a remarkable gap in software-verification research: There are verification results that can not yet be independently confirmed.

6 Conclusion

The 13th edition of the Competition on Software Verification (SV-COMP 2024) again compared automatic tools for software verification and the validation of the produced verification witnesses. SV-COMP again had a record number of 59 participating verification systems (incl. 7 new verifiers and 19 hors-concours; see Fig. 5 for the participation numbers and Table 5 for the details). Furthermore, the validation track compared 17 validation tools; the validation tools were assessed in a similar manner as in the verification track, using a community-agreed scoring schema. The number of verification tasks in SV-COMP 2024 was significantly increased to 30 300 in the C category. Table 10 shows that the top verification tools have an extremely low number of wrong results. However, there are still wrong results, and validation of the verification results is absolutely necessary. We hope that this overview and the competition leads to a broader adoption of software verification, and in particular, that more and better validation tools are developed in the near future.

Data-Availability Statement. The verification tasks and results of the competition are published at Zenodo, as described in Table 3. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Table 2. For easy access, the results are presented also online on the competition web site https://sv-comp.sosy-lab.org/2024. The main results of last year's competition were reproduced in an independent reproduction study [72].

Funding Statement. Some participants of this competition were funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY).

Acknowledgments. We thank the verification community for contributing their tools to the evaluation, the jury for their work on improving the quality of the verification tasks and for their advice in refining and applying to the competition rules, Philipp Wendler for maintaining and improving BenchExec, Matthias Kettl for his help with the competition scripts, and the VCloud team for keeping the scheduling system up to speed.

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. IEEE (2019). https://doi.org/10.1109/ASE.2019.00121
3. Aljaafari, F., Shmarov, F., Manino, E., Menezes, R., Cordeiro, L.: EBF 4.2: Black-Box cooperative verification for concurrent programs (competition contribution). In: Proc. TACAS (2). pp. 541–546. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_33
4. 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

5. 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
6. Andrianov, P.S.: Analysis of correct synchronization of operating system components. Program. Comput. Softw. **46**, 712–730 (2020). https://doi.org/10.1134/S0361768820080022
7. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. Springer (2024)
8. Ayaziová, P., Strejček, J.: Symbiotic-Witch 2: More efficient algorithm and witness refutation (competition contribution). In: Proc. TACAS (2). pp. 523–528. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
9. Ayaziová, P., Strejček, J.: Witch 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Proc. TACAS. LNCS. pp. 341–346. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_18
10. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS. LNCS. pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
11. Bajczi, L., Szekeres, D., Mondok, M., Ádám, Z., Somorjai, M., Telbisz, C., Dobos-Kovács, M., Molnár, V.: EmergenTheta: Verification beyond abstraction refinement (competition contribution). In: Proc. TACAS. LNCS. pp. 371–375. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_23
12. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Mondok, M., Molnár, V.: Theta: Abstraction based techniques for verifying concurrency (competition contribution). In: Proc. TACAS. LNCS. pp. 412–417. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_30
13. Bajczi, L., Ádám, Z., Micskei, Z.: ConcurrentWitness2Test: Test-harnessing the power of concurrency (competition contribution). In: Proc. TACAS. LNCS. pp. 330–334. LNCS 14752, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_16
14. 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
15. 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
16. 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
17. 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
18. 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
19. 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
20. 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

21. 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
22. 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
23. 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
24. 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
25. 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
26. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
27. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
28. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Proc. FASE. pp. 309–323. LNCS 13991, Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_17
29. Beyer, D.: Fm-tools data set of metadata about verifiers and validators (SV-COMP 2024). Zenodo (2024). https://doi.org/10.5281/zenodo.10669735
30. Beyer, D.: Results of the 13th Intl. Competition on Software Verification (SV-COMP 2024). Zenodo (2024). https://doi.org/10.5281/zenodo.10669731
31. Beyer, D.: SV-Benchmarks: Benchmark set for software verification (SV-COMP 2024). Zenodo (2024). https://doi.org/10.5281/zenodo.10669723
32. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2024). Zenodo (2024). https://doi.org/10.5281/zenodo.10669737
33. 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
34. 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
35. 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
36. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multithreaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_26
37. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
38. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Proc. FASE. pp. 49–70. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_3

39. 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
40. 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
41. 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
42. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 429–434. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_26
43. Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00214
44. Beyer, D., Strejček, J.: Case study on verification-witness validators: Where we are and where we go. In: Proc. SAS. pp. 160–174. LNCS 13790, Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_8
45. Beyer, D., Wachowitz, H.: Coveriteam Release 1.1. Zenodo (2024). https://doi.org/10.5281/zenodo.10843666
46. 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
47. 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 (2). pp. 408–412. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_22
48. 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
49. Chalupa, M., Henzinger, T.: Bubaak: Runtime monitoring of program verifiers (competition contribution). In: Proc. TACAS (2). pp. 535–540. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
50. Chalupa, M., Richter, C.: Bubaak-SpLit: Split what you cannot verify (competition contribution). In: Proc. TACAS. LNCS. pp. 353–358. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_20
51. 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
52. 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
53. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS. LNCS. pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
54. 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
55. 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

56. 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
57. 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
58. 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
59. Coto, A., Inverso, O., Sales, E., Tuosto, E.: A prototype for data race detection in CSeq 3 (competition contribution). In: Proc. TACAS (2). pp. 413–417. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_23
60. 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
61. 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
62. Darke, P., Chimdyalwar, B., Agrawal, S., Venkatesh, R., Chakraborty, S., Kumar, S.: VeriAbsL: Scalable verification by abstraction and strategy prediction (competition contribution). In: Proc. TACAS (2). pp. 588–593. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_41
63. Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Taipan 2023 (competition contribution). In: Proc. TACAS (2). pp. 582–587. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_40
64. 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
65. 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
66. 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
67. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. arXiv:2010.05812v2, arXiv (January 2020). https://doi.org/10.48550/arXiv.2010.05812
68. Ernst, G.: Korn: Horn clause based verification of C programs (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_36
69. Farzan, A., Klumpp, D., Podelski, A.: Sound sequentialization for concurrent program verification. In: Proc. PLDI. pp. 506–521. ACM (2022). https://doi.org/10.1145/3519939.3523727
70. 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
71. 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

72. Gerhold, M., Hartmanns, A.: Reproduction report for SV-COMP 2023. Tech. rep., University of Twente (2023). https://doi.org/10.48550/arXiv.2303.06477
73. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_6
74. 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
75. 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
76. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). pp. 424–428. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25
77. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate automizer and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS. LNCS. pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
78. 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
79. 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
80. 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
81. Howar, F., Mues, M.: GWit (competition contribution). In: Proc. TACAS (2). pp. 446–450. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_29
82. Hussein, S., Yan, Q., McCamant, S., Sharma, V., Whalen, M.: Java Ranger: Supporting string and array operations (competition contribution). In: Proc. TACAS (2). pp. 553–558. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_35
83. 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
84. 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
85. 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
86. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Proc. TACAS. LNCS. pp. 406–411. LNCS (2024). https://doi.org/10.1007/978-3-031-57256-2_29
87. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. VSTTE. pp. 1–18. LNCS 12031, Springer (2019)

88. 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
89. Kettl, M., Lemberger, T.: The static analyzer Infer in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 451–456. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_30
90. 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 (2). pp. 479–483. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_35
91. 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
92. 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
93. Leeson, W., Dwyer, M.: Graves-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
94. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT: Modular dynamic symbolic execution for java applications using dynamic instrumentation (competition contribution). In: Proc. TACAS. LNCS. pp. 399–405. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_28
95. 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
96. Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and loop unwinding (competition contribution). In: Proc. TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_31
97. Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.: ESBMC v7.4: Harnessing the power of intervals (competition contribution). In: Proc. TACAS. LNCS. pp. 376–380. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_24
98. Metta, R., Karmarkar, H., Madhukar, K., Venkatesh, R., Chakraborty, S.: Proton: Probes for non-termination and termination (competition contribution). In: Proc. TACAS. LNCS. pp. 393–398. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_27
99. Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In: Proc. TACAS. LNCS. pp. 387–392. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_26
100. 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
101. Mues, M., Howar, F.: GDart (competition contribution). In: Proc. TACAS (2). pp. 435–439. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_27
102. 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

103. 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
104. 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
105. 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
106. Ponce-De-Leon, H., Haas, T., Meyer, R.: Dartagnan: Smt-based violation witness validation (competition contribution). In: Proc. TACAS (2). pp. 418–423. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_24
107. 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
108. 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
109. 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
110. 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
111. Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination (competition contribution). In: Proc. TACAS. LNCS. pp. 381–386. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_25
112. Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint Validator: Correctness witness validation by abstract interpretation (competition contribution). In: Proc. TACAS. LNCS. pp. 335–340. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_17
113. 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
114. 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
115. 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
116. Su, J., Yang, Z., Xing, H., Yang, J., Tian, C., Duan, Z.: PIChecker: A POR and interpolation-based verifier for concurrent programs (competition contribution). In: Proc. TACAS (2). pp. 571–576. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_38
117. 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

118. 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
119. 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
120. 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
121. Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: Proc. TACAS. LNCS. pp. 347–352. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_19
122. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.21. Zenodo (2024). https://doi.org/10.5281/zenodo.10671136
123. Wu, T., Schrammel, P., Cordeiro, L.: Wit4Java: A violation-witness validator for Java verifiers (competition contribution). In: Proc. TACAS (2). pp. 484–489. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_36
124. J. Švejda, 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

ConcurrentWitness2Test: Test-Harnessing the Power of Concurrency (Competition Contribution)

Levente Bajczi(✉)[⋆], Zsófia Ádám, and Zoltán Micskei

Department of Measurement and Information Systems,
Budapest University of Technology and Economics, Budapest, Hungary
bajczi@mit.bme.hu

Abstract. ConcurrentWitness2Test is a violation witness validator for concurrent software. Taking both nondeterminism of data and interleaving-based nondeterminism into account, the tool aims to use the metadata described in the violation witnesses to synthesize an executable test harness. While plagued by some initial challenges yet to overcome, the validation performance of ConcurrentWitness2Test corroborates the usefulness of the proposed approach.

Funding. This research was partially funded by the ÚNKP-23-{2,3}-I New National Excellence Program; and the Doctoral Excellence Fellowship Programme (funded by the NRDI Fund of Hungary and the BME University).

1 Validation Approach

There are multiple violation witness validators in the ReachSafety category of SV-COMP that are based on test harness generation [3]. However, none take part in the category for concurrent programs, presumably due to the increased complexity in orchestrating the different thread interleavings prescribed by the witness files. ConcurrentWitness2Test aims to fill this gap, by providing an enhanced test harness that takes not only the data-nondeterminism into account, but also the nondeterminism caused by concurrency. In this paper we concentrate on solving the latter, as the former is already well documented by the implementing tools [3].

The current witness format for concurrent software defines two edge data fields that we can extract information from [3]:

`createThread`: The unique ID of the new thread that results from the execution of the containing edge

`threadId`: Which thread is currently active when the containing edge is executed. Valid values have at least one `createThread` entry in the witness automaton that must be executed prior to the current edge

[⋆] Jury member representing ConcurrentWitness2Test at SV-COMP 2024.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 330–334, 2024.
https://doi.org/10.1007/978-3-031-57256-2_16

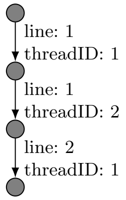


Fig. 1: Wi ness

```
    Thread 1:

    yield(0);
1   a = 0;
    release(0);
    yield(2);
2   asser (a == 0);
    release(2);

    Thread 2:

    yield(1);
1   a = 1;
    release(1);
```

Fig. 2: Sour e

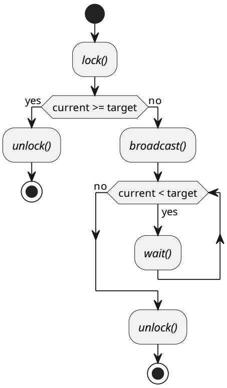


Fig. 3: `yield(target)`

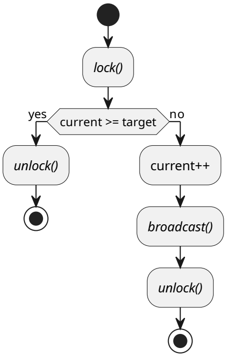


Fig. 4: `release(target)`

Using these pieces of information, we insert a `yield` and `release` call around each action (as seen in the example in Figure 2, based on the metadata from Figure 1), with the parameter `target` increasing at every encountered edge. These functions are shown in Figure 3 and Figure 4, respectively. They rely on a shared variable `current` denoting the next value where the functions need to take effect (to handle revisited locations in the source, e.g., in a loop), alongside a mutex and a condition variable. *Locking* and *unlocking* in the figures refer to operations on the mutex variable; while *broadcasting* and *waiting* refer to operations on the condition variable.

One of the main obstacles to overcome is the resolution of the `threadID` metadata. In our experience, none of the tools produce fully specified witnesses in terms of interleavings, i.e., not every action is totally ordered in the program. While this is acceptable according to the witness format [3], a certain level of nondeterminism might remain in the program after applying the witness. To overcome this problem we rely on statistics, i.e., we execute the resulting harness multiple times, and classify the results as *always observable*, *sometimes observable* and *never observable*. Observability refers to that of the error state, tested by inspecting the exit code of the program. At SV-COMP'24 we opted to only refuse witnesses with *never observable* verdicts.

2 Software Architecture

ConcurrentWitness2Test is a Python project, relying on `pycparser`[1] for parsing C files, and `networkx`[2] for parsing GraphML-based witnesses. As opposed to the harness-only solutions of other witness-to-test validators [3], Con-

[1] https://github.com/eliben/pycparser
[2] https://networkx.org/

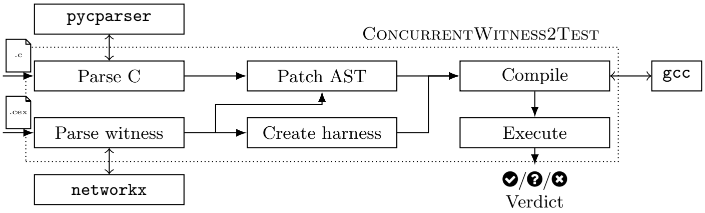


Fig. 5: Architecture of ConcurrentWitness2Test

currentWitness2Test also needs to modify the AST of the C file to insert the function calls to `yield` and `release`, therefore the intermediate output of ConcurrentWitness2Test consists of a patched C file and a separate test harness. We use `gcc`[3] to compile these resulting files to an executable. We run this executable at most 100 times, with an option for early termination if the error becomes observable. See Figure 5 for an overview of this workflow.

3 Discussion of Strengths and Weaknesses of the Approach

As seen in Table 1[4], ConcurrentWitness2Test lacks support for some tools' witnesses. Since then, this limitation has been mostly rectified, but not in time for SV-COMP. The main shortcoming of the competition version of ConcurrentWitness2Test was the handling of cases where edge attributes were given for complex syntactic elements, such as loops, and we tried to insert the function calls into the heads of loops instead of their body. This was an easy fix, and we hope to further the support for various tools even more for next year's SV-COMP.

Despite these temporary shortcomings, ConcurrentWitness2Test still correctly confirmed 1197 results[2]. In contrast, the validator was wrong only 239 times: 2 witnesses were confirmed and 237 witnesses were refused erroneously[5]. These numbers highlight the strength of our approach.

We also note that ConcurrentWitness2Test confirmed 932 results with only a *sometimes observable* verdict. This means that multiple tools produce nondeterministic witnesses, where some interleaving leads the execution to an error state, but not all. We suggest tool developers to concentrate on providing better, deterministic witnesses in order for their results to always be validated. We will aim to constrain our acceptance criteria to *always observable* in future competitions.

[3] https://gcc.gnu.org/

[4] Unofficial results, since no official results were published at the time of writing.

[5] Here, *erroneous* covers all cases when the tool could not reproduce the bug. Therefore, this might not be our tool's shortcoming, but the result of bad witnesses.

Table 1: Results per supported tool, results for wrong verdicts in parentheses

	Dartagnan	Divine	Theta	UAutomizer	UGemCutter	UTaipan
Confirmed	178	179 (2)	191	186	235	228
Refused	79	25 (1)	8	74	22	29
rror	193	111	96	168	194	170

4 Tool Setup and Configuration

The binary archive available at Zenodo [1] contains all required dependencies in the form a virtual environment except for the python 3 interpreter, which needs to be installed separately (e.g., via the `python3` package on Ubuntu 22.04).

The tool can be started either directly via the `main.py` file, or the convenience script in `start.sh`. Either way, the tool expects two inputs: an argument providing the (preprocessed) C file, and the witness file with the `--witness <file>` flag. Upon success, the tool always outputs a single line starting with the string `Verdict:`, with the verdict `SOMETIMES/ALWAYS/NEVER` directly afterward. Some handled exceptions also appear as verdicts.

Up-to-date badges on verification tool support can be seen on the main GitHub page[6]. Tool support has been significantly enhanced since the version nominated for the competition, in preparation for next year's SV-COMP, and for tools to use that may want to improve their witnesses in the meantime.

5 Software Project and Data Availability

ConcurrentWitness2Test is a validation tool maintained by the Critical Systems Research Group[7] of the Budapest University of Technology and Economics. The project is available open-source on GitHub[8] under an Apache 2.0 license. The version (`1.0.0`) used in the competition is available at [1].

References

1. Bajczi, L., Ádám, Z., Micskei, Z.: ConcurrentWitness2Test - SV-COMP'24 Validator Archive (Nov 2023). https://doi.org/10.5281/zenodo.10184336
2. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)

[6] https://github.com/ftsrg/ConcurrentWitness2Test#tool-support

[7] https://ftsrg.mit.bme.hu/en/

[8] https://github.com/ftsrg/ConcurrentWitness2Test

3. 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

Goblint Validator: Correctness Witness Validation by Abstract Interpretation
(Competition Contribution)

Simmo Saan[1](✉)⋆, Julian Erhard[2,3], Michael Schwarz[2],
Stanimir Bozhilov[2], Karoliine Holter[1], Sarah Tilscher[2,3],
Vesal Vojdani[1], and Helmut Seidl[2]

[1] University of Tartu, Tartu, Estonia
`{simmo.saan,karoliine.holter,vesal.vojdani}@ut.ee`
[2] Technische Universität München, Garching, Germany
`{julian.erhard,m.schwarz,stanimir.bozhilov,`
`sarah.tilscher,helmut.seidl}@tum.de`
[3] Ludwig-Maximilians-Universität München, Munich, Germany

Abstract. Goblint is an abstract interpretation framework for C programs with a specialty in concurrency. Using a novel approach, we turn it into a validator of YAML correctness witnesses for all SV-COMP categories. We describe its results at SV-COMP 2024 which includes the first large-scale evaluation of our validator.

1 Validation Approach

Goblint Validator is an extension of the Goblint verifier [14–16] for validation of correctness witnesses in the YAML format [1], consisting of location and loop invariants. The extension involves two related but independent components: witness invariants are checked for correctness and unassumed for speedup. We present here a high-level overview of our recently-published approach to abstract-interpretation–powered witness validation [17].

Correctness of witness invariants is determined by treating them as additional proof obligations. However, instead of inserting assert statements into the program, the validator uses the Goblint verifier as a black box to check whether its computed abstract states satisfy the witness invariants. Hence, invalid witness invariants cannot undermine soundness of the verification process via refinement.

Speedup from witness invariants is attained by incorporating novel *unassume* statements with the invariants into the program. As opposed to refining the abstract state like *assume* operations, these relax the state instead. Doing so in a controlled manner, fixpoint iteration can converge faster, i.e., in fewer iterations. In the best case, the witness invariant precisely characterizes the fixpoint, avoiding further iteration. Unassuming can also make the abstract interpreter more precise, without requiring more expressive abstract domains, by leading the solver to a more precise fixpoint, which widening would otherwise extrapolate over [17].

⋆ Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 335–340, 2024.
https://doi.org/10.1007/978-3-031-57256-2_17

Sound unassume operators must preserve all reaching concrete states, thus preserving soundness of the entire analysis. Goblint Validator implements two different unassume operators:

1. For non-relational domains (e.g., numeric intervals or points-to sets), a classic propagating algorithm for assume operators [4, 7] is adapted with minimal modifications. This admits relaxing abstract values in dynamically allocated memory through pointers.
2. For relational domains (e.g., octagons), dual-narrowing [8] is employed to retain more relations than a generic unassume operator definition [17].

2 Software Architecture

Goblint Validator builds on the Goblint verifier [14–16] which is implemented in OCaml, uses an updated fork of CIL [12] as its frontend and Apron [9] for relational domains.

Instead of altering the control-flow graphs, unassume statements are inserted implicitly as events that activated analyses can handle. In the modular architecture of Goblint [2] the unassume analysis is responsible for emitting these events after transfer functions corresponding to witness invariants. Widening tokens [10] are used to delay widening and allow the invariants to be incorporated without immediate precision loss. The solution of a side-effecting constraint system [3, 18] is post-processed to validate witness invariants and determine the verdict.

3 Strengths and Weaknesses

Overall, Goblint Validator inherits the strengths and weaknesses of Goblint, which are described in its tool papers [14–16]. Thanks to the generic validation approach, the validator works in *all* SV-COMP categories as the Goblint verifier, including those that are currently excluded from correctness witness validation, e.g., concurrency. Due to over-approximation, the verifier can only prove the absence of bugs, but not their presence. Consequently, the validator can currently only *confirm* correctness witnesses. However, it could be extended to *reject* violation witnesses in the future.

We evaluate our validator according to the same three aspects considered by Beyer et al. [6]: same-framework consistency, content-effort dependence and cross-framework validation. The first two only focus on witnesses produced by the Goblint verifier.

Regarding same-framework consistency, table 1 lists how many tasks with each property it can verify and how many of those witnesses Goblint Validator can confirm. The overall average confirmation rate of 78% is lower than the 90% Beyer et al. [6] report for CPAchecker and UAutomizer with GraphML witnesses. Reasons for unconfirmed witnesses range from excessive precision loss by unassuming to validator crashes. In some cases, the validator exceeds resource limits, likely due to large witnesses with many unhelpful invariants. A

Table 1. Number of tasks verified by Goblint and their witness validation verdicts by Goblint Validator, grouped by property.

Property	Correct tasks	Goblint verified	Goblint Validator Confirmed	Goblint Validator Unconfirmed
unreach-call	11,351	1,894	1,064 (56%)	830
no-overflow	5,562	3,932	3,416 (87%)	516
termination	1,536	619	297 (48%)	322
no-data-race	781	695	510 (73%)	185
valid-memsafety	2,796	1,963	1,801 (92%)	162
valid-memcleanup	2	0	–	–
Total	22,028	9,103	7,088 (78%)	2,015

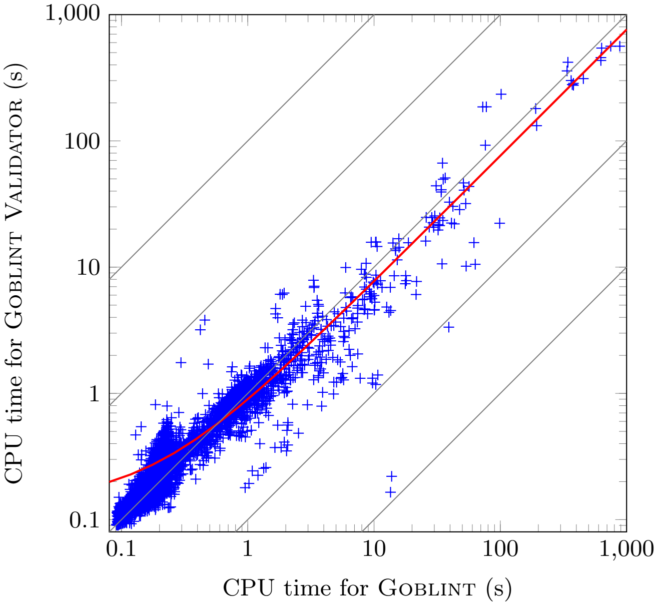


Fig. 1. CPU time scatter plot where each mark (in blue) indicates a task verified by Goblint and whose witness was confirmed by Goblint Validator. Ordinary least squares (OLS) regression (in red) follows $y = 0.76x + 0.14$ ($r^2 = 0.94$).

Table 2. Percentage of witnesses from other verifiers confirmed by Goblint Validator.

Verifier	CPAchecker	CPV	Mopsa	Ultimate Automizer	Ultimate GemCutter	Ultimate Kojak	Ultimate Taipan
Confirmed	8%	6%	78%	46%	60%	57%	51%

handful of instances indicate mismatches between witness generation and their interpretation due to implementation errors in either the verifier or the validator. Fixing such issues could improve the overall quality of the framework [6].

Regarding content-effort dependence, fig. 1 plots the corresponding verification and validation times in the 7,088 confirmed cases. While the results at the low end (< 1 s) are noisy, the results at the high end (> 5 s) show the benefit of witness validation, with up to 10× improvements. Regression analysis estimates an average speedup of 24%, which matches our previous results [17], albeit with greater variance. This is unlike CPAchecker and UAutomizer for which no general performance improvement from consuming witnesses was observed [6].

Regarding cross-framework validation, table 2 presents the confirmation rate of Goblint Validator of correctness witnesses from other tools. For the Ultimate tool family, the percentage is between 46% and 60%, which is similar to what Beyer et al. [6] observed. We have a high ratio for the Mopsa abstract interpreter [11], although it only produces trivial witnesses containing no invariants, on which Goblint Validator effectively reduces to the Goblint verifier. Nevertheless, overwhelming success of Mopsa in the *SoftwareSystems* category warrants independent validation of abstract interpretation results.

4 Tool Setup and Configuration

Goblint Validator version `svcomp24-0-gc2e9465a7` took part in *all* categories except *FalsificationOverall* of SV-COMP 2024 [5, 13]. It is available in both binary (Ubuntu 22.04) and source code form at our GitHub repository.[4] Instructions for building from source can be found in the `README`.

The tool-info module for BenchExec is named `goblint` and the benchmark definition for SV-COMP is `goblint-validate-correctness-witnesses-2.0`. They correspond to running the tool as follows:

```
./goblint --conf conf/svcomp24-validate.json \
          --set witness.yaml.unassume witness.yml \
          --set witness.yaml.validate witness.yml \
          --set ana.specification property.prp input.c
```

5 Software Project and Contributors

Goblint Validator development takes place alongside Goblint on GitHub, while related publications are listed on its website.[5] It is an MIT-licensed project initiated by Technische Universität München and the University of Tartu.

Acknowledgments. This work was supported by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 ConVeY 2. We would like to thank everyone who has contributed to the Goblint framework over the years, laying the foundation for our validator.

[4] https://github.com/goblint/analyzer/releases/tag/svcomp24

[5] https://github.com/goblint/analyzer and https://goblint.in.tum.de

Data Availability Statement. All data of SV-COMP 2024 are archived as described in the competition report [5] and available on the competition website. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of Goblint as used in the competition is archived on Zenodo [13].

Bibliography

[1] Format for correctness witnesses, version 2.0 (2023), URL https://sosy-lab.gitlab.io/benchmarking/sv-witnesses/yaml/correctness-witnesses.html

[2] Apinis, K.: Frameworks for analyzing multi-threaded C. Ph.D. thesis, Technische Universität München (2014)

[3] Apinis, K., Seidl, H., Vojdani, V.: Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In: APLAS '12, pp. 157–172, Springer (2012), doi: 10.1007/978-3-642-35182-2_12

[4] Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: Logic Programming, p. 230–244, The MIT Press (1999), doi: 10.7551/mitpress/4304.003.0024

[5] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: TACAS '24, Springer (2024)

[6] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: FSE '16, pp. 326–337, ACM (2016), doi: 10.1145/2950290.2950351

[7] Cousot, P.: The calculational design of a generic abstract interpreter. In: Calculational System Design, NATO ASI Series F. IOS Press, Amsterdam (1999), URL https://www.di.ens.fr/~cousot/COUSOTpapers/publications.www/Cousot-Marktoberdorf98.pdf.gz

[8] Cousot, P.: Abstracting induction by extrapolation and interpolation. In: VMCAI '15, pp. 19–42, Springer (2015), doi: 10.1007/978-3-662-46081-8_2

[9] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV '09, pp. 661–667, Springer (2009), doi: 10.1007/978-3-642-02658-4_52

[10] Mihaila, B., Sepp, A., Simon, A.: Widening as abstract domain. In: NASA Formal Methods, pp. 170–184, Springer (2013), doi: 10.1007/978-3-642-38088-4_12

[11] Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa-C: Improved verification for C programs, simple validation of correctness witnesses. In: TACAS '24, Springer (2024)

[12] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC '02, pp. 213–228, Springer (2002), doi: 10.1007/3-540-45937-5_16

[13] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint at SV-COMP 2024 (Nov 2023), doi: 10.5281/zenodo.10202867, tool artifact

[14] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination (competition contribution). In: TACAS '24, Springer (2024)

[15] Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints. In: TACAS '21, pp. 438–442, Springer (2021), doi: 10.1007/978-3-030-72013-1_28

[16] Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: Goblint: Autotuning thread-modular abstract interpretation. In: TACAS '23, vol. 2, pp. 547–552, Springer (2023), doi: 10.1007/978-3-031-30820-8_34

[17] Saan, S., Schwarz, M., Erhard, J., Seidl, H., Tilscher, S., Vojdani, V.: Correctness witness validation by abstract interpretation. In: VMCAI '24, pp. 74–97, Springer (2024), doi: 10.1007/978-3-031-50524-9_4

[18] Seidl, H., Vogler, R.: Three improvements to the top-down solver. Math. Struct. Comput. Sci. **31**(9), 1090–1134 (2021), doi: 10.1017/S0960129521000499

Witch 3: Validation of Violation Witnesses in the Witness Format 2.0⋆ (Competition Contribution)

Paulína Ayaziová(✉)⋆⋆ and Jan Strejček

Masaryk University, Brno, Czech Republic
{xayaziov,strejcek}@fi.muni.cz

Abstract. Witch 3 is a new validator of violation witnesses in the witness format 2.0. Note that our previous tool, Symbiotic-Witch 2, can validate only violation witnesses in the old GraphML format. Witch 3 validates witnesses of reachability of an error function, overflows, and invalid dereferences and deallocations. Similarly to Symbiotic-Witch 2, the tool is based on symbolic execution and uses parts of the Symbiotic framework. Support of the witness format 2.0 in Witch 3 includes features not supported by Symbiotic-Witch 2, such as constraints on the program variables and function return values, specifying statements by column, and providing the concrete statement in which the violation occurs. These additional features can further restrict the explored state space, and, more importantly, allow for much more precise validation.

1 Witness Validation Approach

Witch 3 is a new validator of violation witnesses based on symbolic execution. It extends the line of validators Symbiotic-Witch [1] and Symbiotic-Witch 2 [2], which are used to validate violation witnesses in the GraphML witness format [6] (now called 1.0). The main difference of Witch 3 is that it processes witnesses in the witness format 2.0[1] [3] (also known as "the YAML format"). Since this format is based on witness segments and waypoints as opposed to witness automata from the GraphML format, there are large differences in the validation process compared to Symbiotic-Witch 2.

Since the tool performs symbolic execution on the llvm IR [9] of the input program and some information may be lost during the compilation, we first preprocess both the witness and the input program. The preprocessing entails wrapping the branching conditions in the program with a special function so that the condition is not decomposed or flipped during compilation. This ensures that the conditions in the branching statements and the corresponding branches are correctly mapped to those described in the witness. Another crucial step is adjusting the witness so that the identifiers of the waypoints will be preserved

⋆ This work has been supported by the Czech Science Foundation grant GA23-06506S.

⋆⋆ Jury member of SV-COMP 2024

[1] Description is available at https://gitlab.com/sosy-lab/benchmarking/sv-witnesses.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 341–346, 2024.
https://doi.org/10.1007/978-3-031-57256-2_18

in the debug information in the LLVM program. In this phase we also inject the constraints from the assumption waypoints into the input program as calls to a special function `__VALIDATOR_assume` which will be handled later. After this preprocessing, the tool compiles the program into LLVM IR and runs the internal validator Witch-Klee on the LLVM program and the preprocessed witness.

The validator begins symbolic execution in the entry point of the program, associating this state with the first segment of the witness. Throughout the process, each state of symbolic execution is associated with one witness segment.

Whenever the tool executes an instruction that could be associated with a waypoint of type `function_enter`, `function_return`, or `branching` (i.e. a function call, function return, or a branching instruction), it is checked whether this instruction matches a waypoint of the associated segment and the corresponding constraint is enforced on the state. More precisely, if the instruction matches the type and location of an avoid waypoint in the segment, the negation of the constraint in the witness is added to the path condition of the state to guarantee that the waypoint is avoided. If the path condition is not satisfiable, the current state of symbolic execution is terminated. Note that this is always the case for waypoints of type `function_enter`, as their fixed constraint *true* is negated into *false*. If the instruction matches the follow waypoint of a segment, we add the given constraint to the path condition and the witness traversal moves to the next segment.

The `assumption` waypoints are handled slightly differently. Since the constraints are already injected in the program, what remains is to enforce them at the right time. Hence, whenever a `__VALIDATOR_assume` call is executed, the tool checks whether the current state of symbolic execution is associated with the corresponding segment. If it is not, the call is ignored and symbolic execution continues normally. Otherwise, for a follow waypoint, the tool adds the constraint to the path condition of the state and moves to the next segment. For an avoid waypoint, we enforce the negation of the constraint in a similar manner. If the resulting path condition is not satisfiable, the state is terminated.

If the symbolic executor detects a property violation, the tool investigates whether the violating instruction matches the `target` waypoint, which is the last waypoint of the violation witness. If the segment associated with the violating state is not the last, the tool terminates the current state but continues exploring other states of symbolic execution. This is also the case if the associated segment is the last but the `target` waypoint of the segment does not match the instruction violating the property. Otherwise, i.e., if the witness traversal reached the target waypoint, Witch 3 confirms the witness by reporting `false`.

If the exploration ends without confirming the witness, there are two possible results. Normally, Witch 3 outputs `true` to refute the witness. However, the symbolic executor used by Witch 3 may replace a symbolic value by a concrete one due to an unsupported feature (for example, it does not support symbolic floats). This substitution can cause that not all possible execution paths are explored and thus a valid witness can be refuted. Hence, in such instances, witness refutation is suppressed and Witch 3 reports `unknown`.

2 Strengths and Weaknesses

The main strong point of Witch 3 is the support of all features of the format 2.0. This includes enforcing constraints on the values of program variables. These constraints can be included also in the GraphML witnesses, but they are ignored by both Symbiotic-Witch and Symbiotic-Witch 2 with the exception of the equality constraints on the return values of `__VERIFIER_nondet_*` functions. These tools also ignore the attribute `offset` (replaced by `column` in the witness format 2.0) specifying the exact location of an instruction on a given program line. Such shortcomings of our older validators can lead to incorrectly validated witnesses and more `unknown` results. In contrast, full support of the new format allows Witch 3 to produce much more reliable results. Moreover, even in the cases where our older validators produce a correct result without using the contraints provided in the witness, Witch 3 can use the constraints to reduce the explored state-space and thus speed up the validation.

On the negative side, the witness format 2.0 currently supports only witnesses of reachability of an error function, overflows, and invalid dereferences and deallocations. Hence, Witch 3 can only be used in these categories. Once the format is extended for more properties, we plan to implement their support.

Another shortcoming is that the tool currently requires the exact location of a waypoint, including the optional column number. This does not cause any incorrect results since the validation fails in the case of missing information. Moreover, as of SV-COMP 2024, all tools which produced violation witnesses in the format 2.0 included this information. Despite this, we consider it a weakness and plan to fix it in future versions of the tool.

Witch 3 also inherits weaknesses from the technology that it uses. The fact that the symbolic executor works with programs in llvm requires more preprocessing on the program and the witness to ensure that no crucial information is lost during the translation. For this reason, there are cases in which the validation process may be slower. Additionally, the program may contain some inner nondeterminism, such as an unspecified order of evaluation, which is resolved during the compilation. If this order is different to that prescribed by the witness, the witness may be incorrectly refuted. However, we have not yet found any such incorrect result in practice. Most incorrect results stem from technical errors such as missing models of library functions — these functions are then treated as nondeterministic and pure, which may not be the case.

3 Software Architecture

Witch 3 can be divided into two components: Symbiotic [8], which is used as a wrapper for the second component, and Witch-Klee, a witness validator for llvm programs.

For the purposes of this tool, we extended Symbiotic 9 with scripts for preprocessing the witness and the program as previously described. It also compiles the program into llvm, links necessary function models, and parses the output of the internal validator, Witch-Klee.

Witch-Klee takes the program in the llvm IR and the preprocessed witness and performs the validation. The tool is based on the symbolic executor JetKlee, a fork of Klee [7] developed for the purposes of Symbiotic. Witch-Klee uses the yaml-cpp[2] library to parse the witness in the YAML format and Z3 [10] as the SMT solver in symbolic execution.

Both components of Witch 3 use llvm 10.0.1.

4 Tool Setup and Configuration

The archive containing Witch 3 as it participated in SV-COMP 2024 is available on Zenodo [4]. The validation is invoked by the command

```
./symbiotic [--prp <prop>] [--32 | --64] --witness-check <witness> <prg>
```

where `<prop>` is the considered property, the switches `--32` and `--64` specify the considered architecture, `<witness>` is a violation witness in the YAML format, and `<prg>` is the input C program. The property can be provided either as a `.prp` file or one of the shortcuts `no-overflow` and `valid-memsafety`. The default setting is the property of unreachability of the function `reach_error` and the 64-bit architecture.

The version of Symbiotic used by Witch 3, as well as the internal validator Witch-Klee, are available on GitHub (see below) under the tag `svcomp24`. To build Witch 3 from its sources, build each of the components separately. To run the validator, add the location of the Witch-Klee executable to `$PATH` and use the command as presented above.

5 Software Project and Contributors

Both components of Witch 3 are available on GitHub. The source code of the validator Witch-Klee is available at

https://github.com/ayazip/witch-klee

and the source code of the version of Symbiotic used by Witch 3 can be found at

https://github.com/ayazip/symbiotic/tree/witch-klee.

The tool has been developed at the Faculty of Informatics of Masaryk University by Paulína Ayaziová under the supervision and with advice of Jan Strejček. It is available under the MIT license and all internally used tools and libraries (llvm, JetKlee, Z3, yaml-cpp, Symbiotic) are available under open-source licenses that comply with SV-COMP's policy for the reproduction of results.

[2] https://github.com/jbeder/yaml-cpp

Data Availability Statement. All data of SV-COMP 2024 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 Witch 3 used in the competition is archived on Zenodo [4].

References

1. Ayaziová, P., Chalupa, M., Strejček, J.: Symbiotic-Witch: A Klee-based violation witness checker (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 468–473. Springer (2022), https://doi.org/10.1007/978-3-030-99527-0_33
2. Ayaziová, P., Strejček, J.: Symbiotic-Witch 2: More efficient algorithm and witness refutation (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 523–528. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
3. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. Submitted to SPIN 2024.
4. Ayaziová, P., Strejček, J.: Witch 3. Zenodo (2023). https://doi.org/10.5281/zenodo.10064512
5. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Finkbeiner, B., Kovács, L. (eds.) TACAS 2024. LNCS, vol. 14572, pp. xx–yy. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57256-2_15
6. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579, https://doi.org/10.1145/3477579
7. 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
8. Chalupa, M., Mihalkovič, V., Řechtáčková, A., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 462–467. Springer (2022), https://doi.org/10.1007/978-3-030-99527-0_32
9. 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
10. 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

AISE: A Symbolic Verifier by Synergizing Abstract Interpretation and Symbolic Execution (Competition Contribution)

Zhen Wang[1,2] and Zhenbang Chen[1,2](✉) *

[1] College of Computer, National University of Defense Technology, Changsha, China
[2] State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China
{wz,zbchen}@nudt.edu.cn

Abstract. AISE is a static verifier that can verify the safety properties of C programs. The core of AISE is a program verification framework that synergizes abstract interpretation and symbolic execution in a novel manner. Compared to the individual application of symbolic execution or abstract interpretation, AISE has better efficiency and precision. The implementation of AISE is based on KLEE and CLAM.

Keywords: Abstract Interpretation · Symbolic Execution · Program Verification.

1 Verification Approach

Given a program $\mathcal{P}$ and a property φ, a software verification technique or tool verifies whether $\mathcal{P}$ satisfies φ, *i.e.*, all the behavior (*e.g.*, the program paths) of $\mathcal{P}$ satisfies φ. If $\mathcal{P}$ does not satisfy φ, a counter-example (*e.g.*, a program input) will be given to demonstrate the violation of φ. Until now, many software verification techniques and tools have been developed and applied in different areas to result in successful stories [3,18,20,7,17].

AISE is a software verifier that verifies C programs with respect to reachability properties [5]. AISE's key idea is to synergize *symbolic execution* (SE) [4,21] and *abstract interpretation* (AI) [10,11]. In the main loop, our tool performs symbolic execution to analyze the program under verification. However, SE faces path explosion problem [23,16,9] when the program contains loops, which makes it infeasible for sound verification. AI can abstract a program in an over-approximation manner and automatically infer the program invariants at different program locations, which can be used to verify the property. However, the imprecision caused by over-approximation may result in false positives. AISE aims to combine these two techniques in a synergic manner to improve the verification's scalability as much as possible while ensuring precision. When doing SE, AISE carries out AI online to verify a part of the program, which can be used to prune the safe paths. On the other hand, SE can also improve the precision of AI. AISE only reports the violations detected by SE.

* Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 347–352, 2024.
https://doi.org/10.1007/978-3-031-57256-2_19

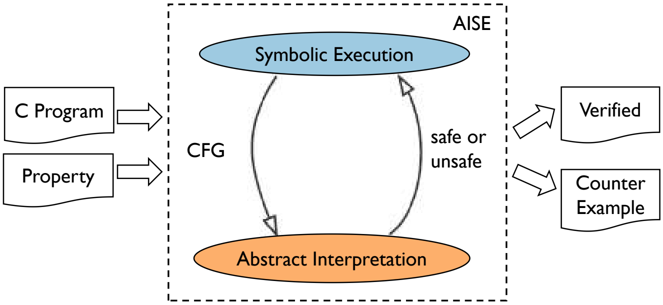


Fig. 1: `AISE`'s verification framework

2 Framework

Figure 1 shows `AISE`'s framework, which contains an AI module and a SE module. The two modules communicate by delivering control-flow graph (CFG) and verification results to help each other. On the one hand, SE constructs the sub-CFG on which AI is carried out; On the other hand, the verification results of AI are returned to SE to prune the redundant paths, *i.e.*, the paths that are guaranteed to satisfy the property.

2.1 Symbolic Execution Module

The SE module takes a C program as input and then executes the program with symbolic inputs. The SE procedure is a state-forking procedure [4]. The whole process is as follows. At the beginning of execution, the SE module constructs an initial symbolic state for the input program. As the state is executed, the data of the state is changed by executing instructions one by one. When the state encounters a branch instruction, a new state is forked based on the original state. A global state pool containing all forked states is maintained. After executing an instruction, the current state is paused, and another state is selected from the state pool to execute. When a state is terminated (*i.e.*, a state after executing the program exit instruction), `AISE` constructs a sub-CFG that contains all the instructions and the edges of the execution path that led to the state and carries out AI on the sub-CFG. Based on the AI's verification result, the state pool is updated, *i.e.*, adding the newly forked states or removing the pruned states. When SE finds a violation of an assertion, `AISE` reports the violation.

2.2 Abstract Interpretation Module

The AI module takes a sub-CFG as input and outputs safe or unsafe. Given the abstract domain [11], the AI module analyses the CFG to produce an invariant at each program location. The invariant describes the constraints of variables at the program location. Then, based on the invariant I, we can check the property φ by checking the validity of $I \Rightarrow \varphi$. If all assertions are checked, `AISE` can prune states that can only reach the edges in the sub-CFG. Intuitively, all the possible paths start from these states are contained in the sub-CFG, so they are all safe paths. Therefore, we can prune all states from which only the nodes and edges of the sub-CFG can be reached. Pruning states reduces the path space of SE and improves the scalability of verification.

2.3 Example

Figure 2 gives an example[3] to illustrate the idea of AISE. This program contains a loop adding y to x. AI using interval domain [11] fails to verify this assertion because y's invariant at Line 11 is $(-\infty, 1000000]$, which is not sufficient to prove the assertion. SE can verify this program by exploring all paths, but SE needs a long time as the paths of this program are numerous.

```
1  int main() {
2      int x=__VERIFIER_nondet_int();
3      int y=__VERIFIER_nondet_int();
4      if (!(y <= 1000000))
5          return 0;
6      if (y>0) {
7          while(x<100) {
8              x=x+y;
9          }
10     }
11     assert(y<=0||(y>0 && x>=100));
12     return 0;
13 }
```

Fig. 2: C code segment

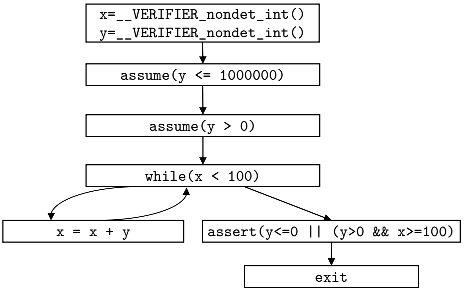

x=__VERIFIER_nondet_int()
y=__VERIFIER_nondet_int()
assume(y <= 1000000)
assume(y > 0)
while(x < 100)
x = x + y
assert(y<=0 || (y>0 && x>=100)
exit

Fig. 3: CFG based on a path

AISE can verify this program successfully in a short time. The SE module only needs to explore a few paths because many can be pruned. After SE module explores the following path: 2→3→4→6→7→8→7→11→12, it constructs the sub-CFG in Figure 3 based on this path. For this sub-CFG, the AI module successfully verifies the assertion. Then, AISE framework updates the state pool in the SE module, killing all the states forked from line 7. These states are forked when encountering the loop head. Then, there are no more states in the pool and SE terminates, *i.e.*, a *safe* result. This also demonstrates that SE can improve the precision of AI by considering the sub-CFG of a symbolic path.

3 Implementation, Results and Discussion

AISE's implementation is based on the AI framework CLAM [2,17] and the SE tool KLEE [7]. STP [15] is the SMT solver of SE. AISE accepts the input in

[3] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/raw/main/c/loops/terminator_03-2.i

LLVM [1] intermediate representation. The AI module of AISE uses the polyhedron abstract domain [12], and we use the implementation in Apron library [19]. The search strategy of the SE module is *nurs:covnew*. Besides, AISE also integrates ESBMC [22] to handle floating-point programs because the SE's module does not support the analysis of floating-point programs.

AISE participants in the *ReachSafety-Loops* category of SV-COMP 2024 [6]. Table 1 shows AISE's results. AISE achieved 847 points in this category, and there were 4 tools ranked ahead of it: Bubbaak [8], Symbiotic [20], VeriAbs [13], VeriAbsL [14]. The figure[4] shows the score-based quantile plots in this category. When the time is less than about 100s, AISE achieved the highest score among all the tools. If the pruning method works, AISE can verify a program in a short time; otherwise, AISE may fail to finish the job. Many of the AISE's failed cases are the programs with non-linear expressions. AI is limited for non-linear polynomials. Besides, AISE is also not efficient at handling large arrays. For example, AISE does not support symbolic size array, which is an inherited shortage from KLEE.

Table 1: AISE's results

	number	time(s)
total tasks	790	
total correct	491	9400
correct true	356	6200
correct false	135	3200
total incorrect	0	0
score	847	

4 Software Project, Setup and Contributors

AISE only participats in the *ReachSafety-Loops* category of SV-COMP benchmarks. The usage of AISE is as follows.

```
./bin/aise <program>
```

The <program> is the input program. AISE only needs the input program as the parameter because all the properties in the *ReachSafety-Loops* benchmarks are the same, *i.e.*, (unreach-call, ILP32), and these properties are built in AISE.

AISE can be found at https://github.com/zbchen/aise-verifier. AISE is a prototype project developed by National University of Defense Technology. The license of AISE is GPL 3.0. People involved in the project are fully listed as the authors of this paper.

Data-Availability Statement

AISE's artifact is available at Zenodo: https://doi.org/10.5281/zenodo.10201159.

Acknowledgement This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Program (No. 62172429).

[4] https://sv-comp.sosy-lab.org/2024/results/results-verified/quantilePlot-ReachSafety-Loops.svg

References

1. LLVM. https://llvm.org, accessed 2023-12-17
2. CLAM repository. https://github.com/seahorn/clam (2022)
3. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker with strategy selection (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
4. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (may 2018). https://doi.org/10.1145/3182657, https://doi.org/10.1145/3182657
5. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., Mckenzie, P.: Reachability Properties, pp. 79–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). https://doi.org/10.1007/978-3-662-04558-9_6, https://doi.org/10.1007/978-3-662-04558-9_6
6. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
7. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
8. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 535–540. Springer Nature Switzerland, Cham (2023)
9. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proceedings of the 38th International Conference on Software Engineering. p. 144–155. ICSE '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884843, https://doi.org/10.1145/2884781.2884843
10. Cousot, P.: Abstract interpretation. ACM Comput. Surv. **28**(2), 324–328 (jun 1996). https://doi.org/10.1145/234528.234740, https://doi.org/10.1145/234528.234740
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 84–96. POPL '78, Association for Computing Machinery, New York, NY, USA (1978). https://doi.org/10.1145/512760.512770, https://doi.org/10.1145/512760.512770
13. Darke, P., Agrawal, S., Venkatesh, R.: Veriabs: A tool for scalable verification by abstraction (competition contribution). In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 458–462. Springer International Publishing, Cham (2021)
14. Darke, P., Chimdyalwar, B., Agrawal, S., Kumar, S., Venkatesh, R., Chakraborty, S.: Veriabsl: Scalable verification by abstraction and strategy prediction (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and

Algorithms for the Construction and Analysis of Systems. pp. 588–593. Springer Nature Switzerland, Cham (2023)

15. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. pp. 519–531. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
16. Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. p. 23–33. ISSTA '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2001420.2001424, https://doi.org/10.1145/2001420.2001424
17. Gurfinkel, A., Navas, J.A.: Abstract interpretation of LLVM with a region-based memory model. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13124, pp. 122–144. Springer (2021). https://doi.org/10.1007/978-3-030-95561-8_8, https://doi.org/10.1007/978-3-030-95561-8_8
18. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Automizer 2024 (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
19. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 661–667. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
20. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
21. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (jul 1976). https://doi.org/10.1145/360248.360252, https://doi.org/10.1145/360248.360252
22. Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.: ESBMC v7.4: Harnessing the power of intervals (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
23. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. p. 225–236. ISSTA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572299, https://doi.org/10.1145/1572272.1572299

Bubaak-SpLit: Split what you cannot verify (Competition contribution)

Marek Chalupa[1(✉)][⋆] and Cedric Richter[2]

[1] Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
mchalupa@ist.ac.at

[2] Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
cedric.richter@uol.de

Abstract. Bubaak-SpLit is a tool for dynamically splitting verification tasks into parts that can then be analyzed in parallel. It is built on top of Bubaak, a tool designed for running combinations of verifiers in parallel. In contrast to Bubaak, that directly invokes verifiers on the inputs, Bubaak-SpLit first starts by splitting the input program into multiple modified versions called *program splits*. During the splitting process, Bubaak-SpLit utilizes a *weak* verifier (in our case symbolic execution with a short timelimit) to analyze each generated program split. If the weak verifier fails on a program split, we split this program split again and start the verification process again on the generated program splits. We run the splitting process until a predefined number of *hard-to-verify* program splits is generated or a splitting limit is reached. During the main verification phase, we run a combination of Bubaak-Lee and Slowbeast in parallel on the remaining unsolved parts of the verification task.

1 Verification approach

Bubaak [7] is a program analysis tool that runs multiple verifiers at the same time, and uses ideas from runtime monitoring and enforcement [5,10] to mediate the communication of useful information between the verifiers, such as invariants or already explored parts of the program. As of this year, the verifiers can be executed in an arbitrary combination of sequential and parallel portfolio, fully dynamically based on the information learned during the verification process.

With Bubaak-SpLit, we explore *program splitting* [12,13] as a way to improve the scalability of the verification process. The main idea behind program splitting is to split a given program P into multiple subprograms $P_1, \ldots, P_n$ which then can be analyzed in parallel. As a result, Bubaak-SpLit can verify multiple subprograms with multiple verifier instances at the same time.

[⋆] Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 353–358, 2024.
https://doi.org/10.1007/978-3-031-57256-2_20

```
1  int main(void) {
2    int x = nondet();
3    if( x >= 1000 ) abort();
4
5    if( x <= 10 ){
6      hard_to_verify_1(x);
7    } else {
8      hard_to_verify_2(x);
9    }
10 }
```

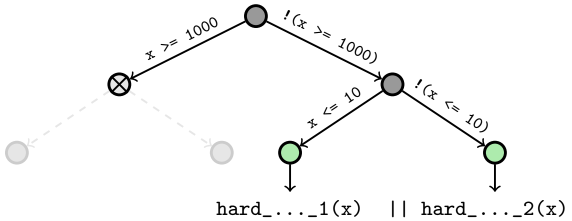


Fig. 1. Overview over the verification process of BUBAAK-SpLit for the given example. BUBAAK-SpLit splits program that are too hard to be verified by a weak verifier (gray nodes), stops for easy-to-verify nodes (crossed out nodes) and it proceeds until n hard-to-verify splits are found (green nodes).

Control-flow Splitting. BUBAAK-SpLit adopts *control-flow splitting*[3] [13] for splitting programs into subprograms. Control-flow splitting splits a program P at the first branching point B creating two subprograms P^+ and P^-. P^+ and P^- each represent the program P when assuming that the branching condition at B is evaluated to `true` or `false` respectively. For example, Figure 2 depicts P^+ and P^- when splitting the program in Figure 1 at the first branching point in Line 3. Syntactically splitting a program might result in suboptimal splits [12] where one part of the split is *easy-to-verify* and the other remains *hard-to-verify*. To mitigate the problem of suboptimal splits, BUBAAK-SpLit implements a dynamic splitting strategy: (1) we first check if the given program (or split) is hard-to-verify by running a weak verifier, (2) if it is hard-to-verify we split the program and repeat the process on the generated splits, (3) if it is not hard-to-verify we record the result of the weak verifier and continue with the other splits (if any). We continue this process until a fixed number of hard-to-verify splits is generated or a splitting limit is reached. If the problem is solved during the splitting process, we report the results of the weak verifiers.

Figure 1 provides an example of the splitting process. After splitting two times, BUBAAK-SpLit identifies two hard-to-verify splits which are then verified by two verifiers in parallel in the main verification phase. Existing static splitting strategies for C programs [12] might stop after the first split, resulting in a suboptimal split (with little to no benefits for the verification process).

Verification technology. BUBAAK-SpLit in SV-COMP 2024 utilizes verifiers based on *forward* and *backward symbolic execution*.

(Forward) symbolic execution (SE) [14] systematically explores program's executions from the initial location. Backward symbolic execution (BSE) [8] explores executions that reach a given (error) location and it does so by analyzing the program backwards from the locations. We employ a variant of BSE with

[3] Our variant of control-flow splitting was mainly inspired by Mooly Sagiv's invited talk "Scaling Formal Verification to Realistic Code with Applications to DeFi" at ETAPS 2023. Our implementation however splits C programs, not Solidity contracts.

```
int main(void) { // P+
  int x = nondet();
  assume( x >= 1000 );
  abort();
}
```

```
int main(void) { // P-
  int x = nondet();
  assume( !(x >= 1000) );
  if( x <= 10 ) ...
}
```

Fig. 2. Result of splitting the program from Figure 1 at the first branching point.

loop folding (BSELF) [8] which allows us to generate loop invariants and prove programs correct.

SE can very quickly identify easy-to-verify problems, so we use it with a short timeout as the weak verifier during splitting. Strong verifiers in the main verification phase are selected based on the property. For the property *unreach-call*, we use BSELF and SE (with no timeout) in parallel – BSELF to prove programs correct and SE to (mainly) find bugs. Other properties are not supported by BSELF. For checking termination properties, we run SE and *termination with inductive invariants with progress* (TIIP) [7]. For checking memory safety, we use only SE. Note that the splitting phase is executed for all properties.

2 Software architecture

Bubaak runs verification tools in a combination of sequential and parallel portfolio. The verifiers are not composed into a fixed scheme, but they are invoked dynamically based on the information gathered during the verification process. In a bit more detail, the architecture of Bubaak is inspired by *process algebras* [4] and is centered about *tasks* and their *rewriting*. The tool starts with the execution of a set of initial tasks; upon finishing, each task either yields a result, or rewrites itself into a new task or a set of new tasks. Whenever a task rewrites itself into a set of new tasks, it also specifies how the results of the new tasks should be aggregated into a single result. The important feature is that generating new tasks is not fixed in a static scheme: a task can rewrite itself into new tasks based on the context and information hitherto gathered about the program during the verification process.

What tasks are executed and how they are being rewritten is defined by a selected *workflow*. The workflow for splitting in SV-COMP 2024 is depicted in Figure 3. It defines the task *Split(P)* that takes program P and splits it into two parts as described in Section 1. This task is invoked as the initial task on the input program. After splitting the program, *Split* rewrites itself into two identical tasks *CCAndCheckWeak* that are invoked on those two splits. As the name suggests, the input split is compiled (into LLVM [1]) and the weak verifier is ran on it to check if the split is easy to solve. If a split is not easy to solve, the task *Split* is invoked on the split recursively, and this process continues until a pre-defined depth is reached, at which point instead of splitting further the workflow invokes the strong verifier.

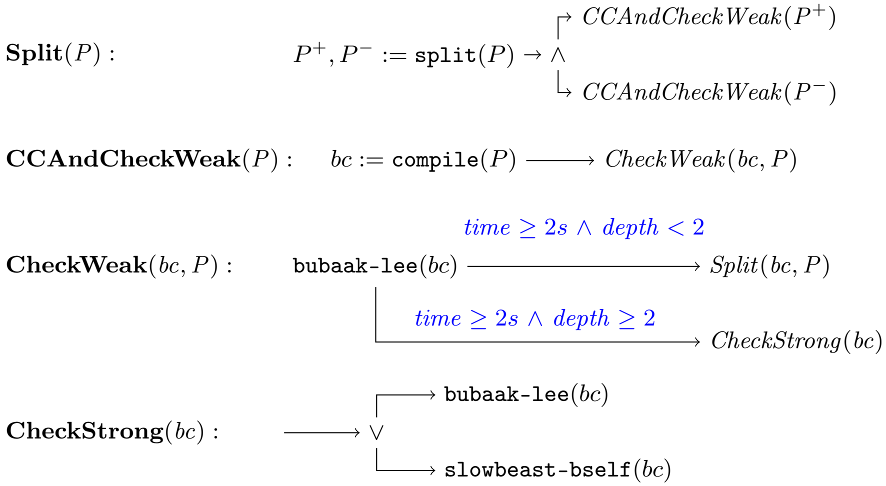


Fig. 3. The workflow of Bubaak-SpLit for SV-COMP 2024. For brevity, the scheme does not show errors handling and the result propagation.

Workflows are only an abstraction: internally, task execution and rewriting is implemented using an event loop that handles events coming from tasks, task creation and destruction, and the results aggregation.

The weak verifier is based on Bubaak-Lee and we run a combination of Bubaak-Lee and Slowbeast during the main verification phase. Bubaak and Slowbeast are implemented in Python, Bubaak-Lee is in C++. Both Slowbeast and Bubaak-Lee use Z3 [9] as the SMT solver.

3 Strengths and Weaknesses

Program splitting has been shown to improve the *runtime efficiency* [15] and *verification effectiveness* [11] of symbolic execution engines. By splitting the program into several parts, SE and BSE can analyze different parts of the program at the same time, which can lead to results being decided more quickly. In SV-COMP 2024 [6], Bubaak-SpLit was able to solve 60 benchmarks that Bubaak was not able to solve and 456 benchmarks were solved faster, often significantly. In comparison to Bubaak, Bubaak-SpLit misses several violations on the *ReachSafety* benchmarks. Most of them are due to the fact that we severely limit the execution time of SE during verification. Another problem is the scalability of our approach in the restricted setting of the SV-COMP. By splitting the program up to n times, we currently run up to 2^n verifiers at the same time. While in praxis this might significantly reduce the walltime, it also significantly reduces the cputime available to each verifier. Overall, Bubaak-SpLit inherits the strengths of the underlying analyses which allows the tool to perform well in the categories *ReachSafety* and *SoftwareSystems*. After SV-COMP 2024, we have found and fixed several bugs in the implementation of Bubaak-SpLit which might have severely limited its performance.

Acknowledgements This work was partially supported by the ERC-2020-AdG 10102009 grant.

Data-Availability Statement The submitted version of our tool contribution is archived and available at Zenodo [2]. The source code is also available on GitLab [3].

References

1. llvm.org. https://llvm.org, accessed: 2023-12-21
2. BUBAAK-SpLit artifact (2023). https://zenodo.org/records/10202207
3. BUBAAK-SpLit repository (2023), https://gitlab.com/mchalupa/bubaak
4. Baeten, J.C., Weijland, W.P.: Process algebra. Cambridge university press (1991)
5. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_1
6. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
7. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers - (competition contribution). In: TACAS 2023. LNCS, vol. 13994, pp. 535–540. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
8. Chalupa, M., Strejcek, 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. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS 2008. pp. 337–340. Springer (2008)
10. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 103–134. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_4
11. Haltermann, J., Jakobs, M., Richter, C., Wehrheim, H.: Parallel program analysis via range splitting. In: FASE 2023. LNCS, vol. 13991, pp. 195–219. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_11
12. Haltermann, J., Jakobs, M., Richter, C., Wehrheim, H.: Ranged program analysis via instrumentation. In: SEFM 2023. LNCS, vol. 14323, pp. 145–164. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5_9
13. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: SAS 1998. LNCS, vol. 1503, pp. 200–214. Springer (1998). https://doi.org/10.1007/3-540-49727-7_12
14. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
15. Siddiqui, J.H., Khurshid, S.: Scaling symbolic execution using ranged analysis. In: OOPSLA 2012. pp. 523–536. ACM (2012). https://doi.org/10.1145/2384616.2384654

CPAchecker 2.3 with Strategy Selection
(Competition Contribution)

Daniel Baier[*], Dirk Beyer, Po-Chun Chien,
Marek Jankola, Matthias Kettl, Nian-Ze Lee,
Thomas Lemberger, Marian Lingsch-Rosenfeld,
Martin Spiessl, Henrik Wachowitz, and Philipp Wendler

http://cpachecker.sosy-lab.org

LMU Munich, Munich, Germany

Abstract. CPAchecker is a versatile framework for software verification, rooted in the established concept of *configurable program analysis*. Compared to the last published system description at SV-COMP 2015, the CPAchecker submission to SV-COMP 2024 incorporates new analyses for reachability safety, memory safety, termination, overflows, and data races. To combine forces of the available analyses in CPAchecker and cover the full spectrum of the diverse program characteristics and specifications in the competition, we use *strategy selection* to predict a sequential portfolio of analyses that is suitable for a given verification task. The prediction is guided by a set of carefully picked program features. The sequential portfolios are composed based on expert knowledge and consist of bit-precise analyses using k-induction, data-flow analysis, SMT solving, Craig interpolation, lazy abstraction, and block-abstraction memoization. The synergy of various algorithms in CPAchecker enables support for all properties and categories of C programs in SV-COMP 2024 and contributes to its success in many categories. CPAchecker also generates verification witnesses in the new YAML format.

1 Software Architecture

CPAchecker [10] is a flexible framework for automatic software verification based on the concept of *Configurable Program Analysis* (CPA) [9]. Abstract domains needed by a verification approach are represented as CPAs, and multiple CPAs can be combined in a modular fashion to achieve synergy. CPAchecker provides basic functionalities for program analysis, such as tracking the control flow or callstack, as standalone CPAs, which facilitate the implementation of new analyses. Through its modular architecture, a rich collection of verification algorithms [7, 12, 14, 24] has been implemented in CPAchecker, and its flexibility and extensibility have been evidenced by many research projects.

Operating Platform. CPAchecker is platform-independent as it is written in Java. However, its default SMT solver MathSAT5 [17] is bundled only for Linux. Thanks to the versatility of the used library JavaSMT [23], a different SMT solver can be chosen on other platforms.

[*] Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 359–364, 2024.
https://doi.org/10.1007/978-3-031-57256-2_21

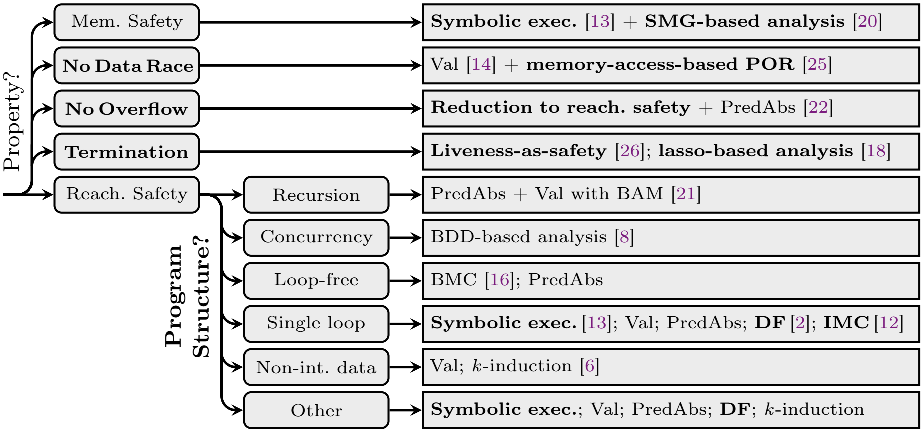

Property?
Mem. Safety
No Data Race
No Overflow
Termination
Reach. Safety
Program Structure?
Recursion
Concurrency
Loop-free
Single loop
Non-int. data
Other
Symbolic exec. [13] + SMG-based analysis [20]
Val [14] + memory-access-based POR [25]
Reduction to reach. safety + PredAbs [22]
Liveness-as-safety [26]; lasso-based analysis [18]
PredAbs + Val with BAM [21]
BDD-based analysis [8]
BMC [16]; PredAbs
Symbolic exec. [13]; Val; PredAbs; DF [2]; IMC [12]
Val; k-induction [6]
Symbolic exec.; Val; PredAbs; DF; k-induction

Fig. 1: Strategy selection based on the property to verify and program structure (New components since the last published system description [19] are marked in boldface. '+' and ';' denote component composition and sequential execution, respectively.)

Witnesses. CPAchecker produces correctness and violation witnesses for all properties where the corresponding witness type is already defined by the community. These are exported in the established GraphML format [4, 5] as well as in the new YAML format that is introduced with SV-COMP 2024.

2 Verification Approaches

To effectively solve the verification tasks from the heterogeneous benchmark set used in the competition, we need different verification strategies. Given a verification task, we select a suitable strategy with a two-level approach according to the property of the task and the structure of the program. A strategy could be a sequential portfolio of different verification techniques, each of which is assigned a time limit that is determined with expert knowledge. Figure 1 shows the selection procedure. The first-level selection is based on the property of the verification task. If the property is among *memory safety*, *no-dataraces*, *no-overflows*, or *termination*, a dedicated strategy is immediately assigned to solve the task. If the property is *reachability safety*, we further distinguish the program structure of a task into six classes by a set of carefully picked features, and a tailored strategy is invoked for each class. The details for each property and program structure are given below.

Memory Safety. Memory safety is checked by an unbounded analysis based on symbolic memory graphs (SMGs) [20]. It utilizes symbolic execution [13] to reason over non-concrete values, enabling us to verify the safety of low-level memory operations. The graph-based approach allows us to not only represent heap memory efficiently, but also to abstract linked memory structures (e.g., linked lists) that are created with low-level memory operations.

No Data Race. Data races are checked with a combination of value analysis (Val) [14], the thread handling from our concurrency analysis [8], and a CPA that

tracks read and write accesses to memory locations. We perform partial order reduction (POR) [25] over thread-local memory accesses to improve performance.
No Overflow. Overflows are checked with a CPA that adds additional constraints for overflow detection and a bit-accurate predicate abstraction (PredAbs) [7]. For recursive tasks we add block-abstraction memoization (BAM) [21, 27], which summarizes the input-output behavior of recursive functions.
Termination. Our termination strategy consists of two techniques. The first technique transforms liveness to a safety property [26]. With a combination of predicate and value analyses we check whether there exists some program state at a loop head that can be visited twice. If the program is recursive or the analysis reaches a time limit of 300 s, we switch to the second techniques, which uses ideas first implemented in Terminator [18]: We apply CPAchecker's predicate-based reachability analysis to detect potentially non-terminating program executions, called candidate *lassos*. A lasso consists of a *stem* (a finite program path) that is followed by a *loop* (a finite program path that describes a syntactic cycle in the program). Found candidate lassos are analyzed with the library LassoRanker [24] to synthesize termination and non-termination arguments. If a non-termination argument is found for at least one candidate lasso, violation of the termination property is reported. Otherwise, the analysis claims the program as terminating.
Reachability Safety. For the reachability of an error location, we tailor our verification strategy based on the structure of the program. If the program contains a *recursive* function, we apply block-abstraction memoization [21, 27] in combination with value analysis (Val) and predicate abstraction (PredAbs). If the program is *multi-threaded*, a concurrency analysis [8] that relies on binary decision diagrams (BDD) is applied. We set an upper limit of five threads for the analysis, and if this threshold is surpassed, the analysis is aborted. For non-recursive and single-threaded programs, we assign one of the four verification strategies in Fig. 1 according to the following structural features: the number of loops and whether the program contains non-integer data types, such as floating-point variables, arrays, or composite data structures [3]. The four strategies are all based on sequential combinations [19] of various bit-precise analyses with different time limits. For *loop-free* programs, we apply bounded model checking (BMC) [16] with a fallback to PredAbs [22]. For programs with a *single loop*, we apply a sequence of symbolic execution [13], Val [14], PredAbs [11], interval-based data-flow analysis (DF) [2], and interpolation-based model checking (IMC) [12]. For programs with multiple loops and *non-integer data types*, we apply Val and k-induction [6]. For all *other* programs, i.e., those with multiple loops but without non-integer data types, we apply a sequential portfolio of symbolic execution, Val, PredAbs, DF, and k-induction.

3 Strengths and Weaknesses

CPAchecker with strategy selection performed well in SV-COMP 2024 [1], winning the second place in category *Overall* and the first place in category *FalsificationOverall*. Notably, it produced 17 968 correct and confirmed results, more than any other participant, and outperformed the winner in category *Overall* by 32 %. CPAchecker is also robust: More than 96 % of its correct results were confirmed by witness validators, and it produced only 17 wrong results (0.06 % of all tasks).

CPAchecker won the third place in category *ReachSafety* by using various analyses orchestrated by strategy selection. For programs with non-integer data types, k-induction was the most effective analysis. In programs with loops, most alarms were found by symbolic execution, and most proofs were delivered by value analysis and predicate abstraction.[1]

The only categories without a medal for CPAchecker were *Termination*, *ConcurrencySafety*, and *MemSafety*. In particular, all wrong results in the category *MemSafety* are due to imprecise abstractions of nested lists. To alleviate them, we intend to improve the precision of our list abstraction and incorporate SMT-based array abstraction, which would make CPAchecker more effective in this category. To improve the termination analysis, we plan to make the analyses more cooperative and carry over partial proofs in the sequential combination. Additionally, CPAchecker needs improvements for finding invariants with quantifiers, which mainly affects verification tasks with large arrays.

4 Setup and Configuration

SV-COMP 2024 ran CPAchecker version 2.3 [15] on all categories with C programs. It runs on a standard GNU/Linux system with a Java 17 compatible runtime environment. To start CPAchecker, execute the following command:

```
scripts/cpa.sh -svcomp24 -benchmark -heap 10000M -timelimit 900s
    -spec property.prp program.i
```

For programs assuming a 64-bit memory model, append the argument `-64` to the command line. At the end of the execution, the verification result is printed to the console output and the witnesses are written to the files `witness.graphml` and `witness.yml` in the directory `output/`.

Note that the configuration `-svcomp24` is optimized specifically for the resource limits used in SV-COMP (15 GB of RAM and 15 min CPU time per task). For other use cases (e.g., with less RAM or a different time limit), please apply a different configuration (e.g., `-default`) and adjust the memory consumption with the command-line option `-heap` as described in the documentation.

5 Project and Contributors

More than 100 developers have contributed to CPAchecker, mainly from LMU Munich, TU Darmstadt, U Paderborn, U Passau, TU Prague, U Oldenburg, TU Vienna, ISP RAS, and several other universities and institutes. We would like to thank all contributors for their investment in CPAchecker. A complete list and more information about the project is available at https://cpachecker.sosy-lab.org. A list of bugs that CPAchecker found in the Linux kernel is also available.

[1] Note that the observations are specific to our sequential portfolios and influenced by the orders of analyses in the combination.

Data-Availability Statement. The tool is available at https://cpachecker.sosy-lab.org and the version used in SV-COMP 2024 is archived at Zenodo [15].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 496588242 (IDEFIX), and the LMU Postdoc Support Fund.

References

1. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
2. Beyer, D., Chien, P.C., Lee, N.Z.: CPA-DF: A tool for configurable interval analysis to boost program verification. In: Proc. ASE. pp. 2050–2053. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00213
3. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features: A simple but effective approach. In: Proc. ISoLA. pp. 144–159. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11
4. 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
5. 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
6. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
7. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
8. Beyer, D., Friedberger, K.: A light-weight approach for verifying multi-threaded programs with CPAchecker. In: Proc. MEMICS. vol. 233, pp. 61–71. EPTCS (2016). https://doi.org/10.4204/EPTCS.233.6
9. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
10. 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
11. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010). https://dl.acm.org/doi/10.5555/1998496.1998532
12. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. arXiv/CoRR **2208**(05046) (July 2022). https://doi.org/10.48550/arXiv.2208.05046
13. Beyer, D., Lemberger, T.: CPA-SymExec: Efficient symbolic execution in CPAchecker. In: Proc. ASE. pp. 900–903. ACM (2018). https://doi.org/10.1145/3238147.3240478
14. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11

15. Beyer, D., Wendler, P.: CPAchecker release 2.3 (unix) (2023). https://doi.org/10.5281/zenodo.10203297
16. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
17. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
18. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Proc. CAV. pp. 415–418. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_37
19. 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
20. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Proc. SAS. pp. 215–237. LNCS 7935, Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_13
21. Friedberger, K.: CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 912–915. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_58
22. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
23. Karpenkov, E.G., Friedberger, K., Beyer, D.: JavaSMT: A unified interface for SMT solvers in Java. In: Proc. VSTTE. pp. 139–148. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_11
24. Leike, J., Heizmann, M.: Ranking templates for linear loops. Logical Methods in Computer Science **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:16)2015
25. Peled, D.: Ten years of partial order reduction. In: Proc. CAV. pp. 17–28. Springer (1998). https://doi.org/10.1007/BFb0028727
26. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electr. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006). https://doi.org/10.1016/j.entcs.2005.11.018
27. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Proc. ICFEM. pp. 332–347. LNCS 7635, Springer (2012). https://doi.org/10.1007/978-3-642-34281-3_24

CPV: A Circuit-Based Program Verifier
(Competition Contribution)

Po-Chun Chien(✉)* and Nian-Ze Lee(✉)

LMU Munich, Munich, Germany
{po-chun.chien,nian-ze.lee}@sosy.ifi.lmu.de

Abstract. We submit to SV-COMP 2024 CPV, a circuit-based software verifier for C programs. CPV utilizes sequential circuits as its intermediate representation and invokes hardware model checkers to analyze the reachability safety of C programs. As the frontend, it uses Kratos2, a recently proposed verification tool, to translate a C program to a sequential circuit. As the backend, state-of-the-art hardware model checkers ABC and AVR are employed to verify the translated circuits. We configure the hardware model checkers to run various analyses, including IC3/PDR, interpolation-based model checking, and k-induction. Information discovered by hardware model checkers is represented as verification witnesses. In the competition, CPV achieved comparable performance against participants whose intermediate representations are based on control-flow graphs. In the category *ReachSafety*, it outperformed several mature software verifiers as a first-year participant. CPV manifests the feasibility of sequential circuits as an alternative intermediate representation for program analysis and enables head-to-head algorithmic comparison between hardware and software verification.

Keywords: Software verification · Hardware verification · C programs · Sequential circuits · Btor2 · Aiger · Tool combination · Portfolio

1 Introduction

Software verification is challenging. Numerous intermediate representations have been proposed to capture diverse software features and facilitate the development of program verifiers. Among various encodings of a state-transition system, *sequential circuits*, consisting of memory elements to represent states and combinational logic to capture state transitions, are commonly used in the hardware-verification domain, and abundant techniques have been invented for *hardware model checking*. Using sequential circuits as its intermediate representation, our tool CPV aims to answer the following question: *Are sequential circuits feasible as an alternative foundation to build software verifiers?* While previous studies on translating and cross-applying verification techniques for hardware and software exist [1, 2, 3, 4], to our knowledge, no participants in SV-COMP had used sequential circuits as their intermediate representations. This competition report outlines the software

* Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 365–370, 2024.
https://doi.org/10.1007/978-3-031-57256-2_22

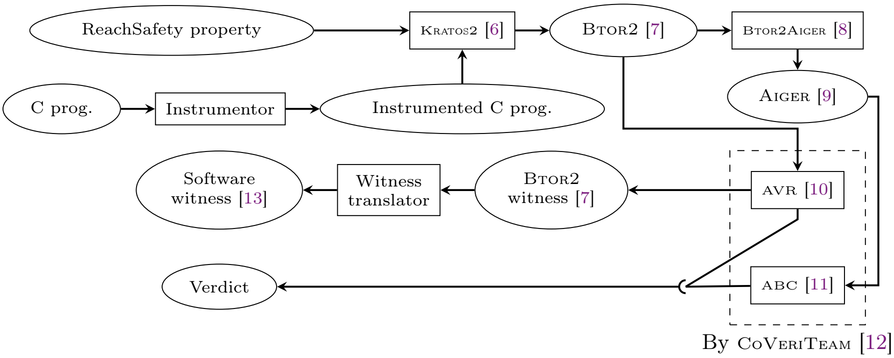


Fig. 1: Software architecture of CPV

architecture and verification approach of CPV and discusses its results against other mature program analyzers in SV-COMP 2024 [5].

2 Software Architecture

The software architecture of CPV is depicted in Fig. 1. Its verification workflow is divided into two stages: (1) In the frontend (the upper half of Fig. 1), an input C program with a reachability-safety property is first instrumented to allow for witness translation (details in Sect. 3) and then translated into a word-level Btor2 [7] circuit by Kratos2 [6]. The Btor2 language [7] is used in the Hardware Model Checking Competitions [14, 15], and many powerful hardware model checkers support this format. A bit-level Aiger [9] circuit is also generated by the tool Btor2Aiger [8]; (2) In the backend (the lower half of Fig. 1), CPV invokes hardware model checkers AVR [10] and ABC [11] to verify the translated circuits. Btor2 verification witnesses produced for the circuits are translated to software witnesses in the GraphML format [13] for the original program. CPV configures and executes the backend model checkers (either solely or as portfolios) via CoVeriTeam [12], a library for cooperative verification [16]. Thanks to the versatility of CoVeriTeam, it is convenient to choose the verification algorithms used by AVR and ABC, and the pool of the backend verifiers in CPV can be expanded with little effort.

3 Verification Approach

The approach of CPV is to translate a program into a circuit and applies hardware model checking to the translated verification task. To generate software-verification witnesses, CPV instruments an input program before translating it to a circuit, such that the information contained in a witness for the translated circuit can be mapped back to the original program.

Program-to-Circuit Translation. CPV utilizes Kratos2 [6] as its frontend to translate a verification task of a C program into a word-level sequential circuit in the Btor2 format [7]. Kratos2 applies *large-block encoding* [17] and introduces a symbolic program counter to fold the summarized program into a state-transition system. Executing a maximal loop-free block of the program is a one-step transition in the system. A call to an external function that models nondeterministic input values to the program, e.g., the functions `__VERIFIER_nondet_X()` in SV-COMP, is represented as an external input to the state-transition system. We configure Kratos2 to export the system as a sequential circuit in the Btor2 format because Btor2 is the prevailing format for hardware model checking. In order to leverage bit-level hardware model checkers, CPV additionally invokes Btor2Aiger [8] to translate the word-level Btor2 circuit into the Aiger format [9]. Currently, CPV supports the property of reachability safety. Violation to the reachability-safety property of the input program is captured by a circuit output asserting the equivalence between the symbolic program counter and the error location.

Hardware Model Checking. CPV employs AVR [10] and ABC [11], two state-of-the-art hardware model checkers for word-level Btor2 and bit-level Aiger circuits, respectively, to analyze the translated circuits. A hardware model checker decides whether the translated circuit has a computation trace to assert its circuit output, which indicates the error location in the original program is reachable. In this case, the verification verdict is `false`, and the original program is unsafe. If there is no trace to assert the circuit output, the verdict is `true`, and the original program is safe.

To achieve synergy, we combine the strengths of various hardware-verification algorithms, including property-directed reachability (PDR) [18, 19], interpolation-based model checking (IMC) [20], k-induction (KI) [21], and bounded model checking (BMC) [22]. For the tasks that can be translated into Aiger circuits,[1] a sequential portfolio of AVR-KI, AVR-PDR, ABC-IMC, ABC-PDR, and AVR-BMC is applied. A pre-determined time limit is imposed on each component in the portfolio by CoVeriTeam. AVR is executed first in the portfolio because it can produce a Btor2 witness [7] for the translated circuit if a property violation was found, whereas ABC does not export witnesses in a standardized format. CPV can then translate a Btor2 witness back to a software violation witness. Currently, CPV outputs a dummy violation witness if a bug is reported by ABC. Since both the Btor2 and Aiger languages do not define a format for correctness witnesses, CPV also outputs a dummy correctness witness in this case. For the remaining tasks that cannot be translated into Aiger circuits, CPV uses a sequential portfolio of AVR's KI, PDR, and BMC.

Program Instrumentation for Witness Translation. To map the information in a Btor2 witness back to the original program, CPV instruments the input program prior to the program-to-circuit translation. A Btor2 violation witness encodes a computation trace that asserts the output of the translated circuit. The trace consists of a sequence of values given to the circuit's external inputs, each corresponding to a call to a function `__VERIFIER_nondet_X()` in the program.

[1] The Btor2-to-Aiger translation may fail if a Btor2 circuit uses data sorts or operations unsupported by Aiger, such as arrays or non-constant register initialization.

Table 1: Summary of CPV's correct results in SV-COMP 2024

ReachSafety		#solved	#tasks solved by respective approach			
verdict	#tasks		AVR-KI	AVR-PDR	AVR-BMC	ABC-IMC
`true`	8 323	3 860	3 405	323	0	132
`false`	2 899	1 092	867	172	2	51

To assume these values at the control-flow locations where they are relevant for triggering the property violation, CPV's instrumentor assigns a fresh counter to each of these calls. A counter is incremented after each call, so its value can be inferred from the Btor2 witness. An input value is relevant if accompanied by a change in its counter. The witness translator of CPV traverses the Btor2 witness, extracts the relevant input values by tracking the changes in the counters, and exports the software violation witness in the GraphML format [13].

4 Results in SV-COMP 2024

CPV participated in the category *ReachSafety* of SV-COMP 2024 [5]. As a first-year participant, it surprisingly outperformed several mature software verifiers in terms of the number of correctly solved tasks. CPV is especially effective in the subcategory *ReachSafety-Hardware* and *ReachSafety-ECA*, solving the second and third most tasks among all participants, respectively. Its impressive results manifest the feasibility of using sequential circuits as an alternative intermediate representation to construct program verifiers.

The overall results of CPV is summarized in Table 1. Among the 11 222 verification tasks in the category *ReachSafety*, 8 439 were successfully translated to Btor2 circuits by Kratos2, and 7 773 could be further translated to Aiger circuits by Btor2Aiger. In total, CPV produced 4 952 correct and confirmed results. The k-induction implementation in AVR contributed the most correctly solved and confirmed tasks, followed by PDR of AVR and IMC of ABC.[2]

We will improve CPV in the following directions: First, we will generate non-trivial software correctness witnesses through extracting and translating the fixed points computed by hardware model checkers. We aim to enhance the witness-confirmation rate of CPV, currently about 90 %, to the level of other mature participants (more than 95 %). Second, we will investigate the 27 false alarms in the subcategory *ReachSafety-Hardness*.

5 Setup and Configuration

We submitted CPV at version 0.4 [23] to SV-COMP 2024 [5]. A Linux-based operating system is required to execute the tool, as the used library CoVeriTeam [12] relies on Linux-specific features, such as control groups, name spaces, and overlay file systems. Additional Python package requirement and the instructions to set up the execution environment can be found in the `README` file of the submitted tool archive.

[2] The observations are specific to the order of algorithms in CPV's sequential portfolios.

Data-Availability Statement. CPV is an open-source project, developed and maintained by the Software and Computational Systems Lab at LMU Munich. Its source code and executables are archived on Zenodo [23], and the project is maintained on GitLab at https://gitlab.com/sosy-lab/software/cpv.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and the LMU Postdoc Support Fund.

References

1. Mukherjee, R., Tautschnig, M., Kroening, D.: v2c: A Verilog to C translator. In: Proc. TACAS. pp. 580–586. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_38
2. Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In: Proc. TACAS. pp. 1–21. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12
3. Noureddine, M.A., Zaraket, F.A.: Model checking software with first order logic specifications using AIG solvers. IEEE Trans. Softw. Eng. **42**(8), 741–763 (2016). https://doi.org/10.1109/TSE.2016.2520468
4. Long, J.: Reasoning about High-Level Constructs in Hardware/Software Formal Verification. Ph.D. thesis, University of California, Berkeley (2017). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-150.html
5. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
6. Griggio, A., Jonáš, M.: Kratos2: An SMT-based model checker for imperative programs. In: Proc. CAV. pp. 423–436. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_20
7. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
8. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Source-code repository of Btor2, BtorMC, and Boolector 3.0. https://github.com/Boolector/btor2tools, accessed: 2023-01-29
9. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Institute for Formal Models and Verification, Johannes Kepler University (2007). https://doi.org/10.35011/fmvtr.2007-1
10. Goel, A., Sakallah, K.: AVR: Abstractly verifying reachability. In: Proc. TACAS. pp. 413–422. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23
11. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proc. CAV. pp. 24–40. LNCS 6174, Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
12. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
13. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
14. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Proc. FMCAD. p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233

15. Biere, A., Froleyks, N., Preiner, M.: 11th Hardware Model Checking Competition (HWMCC 2020). http://fmv.jku.at/hwmcc20/, accessed: 2023-01-29
16. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
17. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
18. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI. pp. 70–87. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
19. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proc. FMCAD. pp. 125–134. FMCAD Inc. (2011). https://dl.acm.org/doi/10.5555/2157654.2157675
20. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
21. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, pp. 127–144. LNCS 1954, Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
22. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
23. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier. Zenodo (2023). https://doi.org/10.5281/zenodo.10203472, version 0.4

EmergenTheta: Verification Beyond Abstraction Refinement (Competition Contribution)

Levente Bajczi(✉)⋆, Dániel Szekeres, Milán Mondok, Zsófia Ádám,
Márk Somorjai, Csanád Telbisz, Mihály Dobos-Kovács, and
Vince Molnár

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
`bajczi@mit.bme.hu`

Abstract. Theta is a model checking framework conventionally based on abstraction refinement techniques. While abstraction is useful for a large number of verification problems, the over-reliance on the technique led to Theta being unable to meaningfully adapt. Identifying this problem in previous years of SV-COMP has led us to create EmergenTheta, a sandbox for the new approaches we want Theta to support. By differentiating between mature and emerging techniques, we can experiment more freely without hurting the reliability of the overall framework. In this paper we detail the development route to EmergenTheta, and its first debut on SV-COMP'24 in the ReachSafety category.

Funding. This research was partially funded by the ÚNKP-23-{2,3}-I New National Excellence Program; Project no. 2019-1.3.1-KK-2019-00004 (implemented with the support provided from the NRDI Fund of Hungary under the 2019-1.3.1-KK funding scheme); and the Doctoral Excellence Fellowship Programme (funded by the NRDI Fund of Hungary and the BME University).

1 Software Architecture

Theta is a modular and configurable verification framework in the sense that multiple frontend subprojects are served by a vastly configurable, CEGAR-based backend ([10,6]). Frontends include *Petri-nets*, *AIGER* models, *timed automata*, and *C programs* among others (hence the modularity), and the CEGAR backend provides fine-grained access to its internal settings such as refinement and search strategy, abstract domains, and solver selection (hence the configurability). It is, however, not conventionally capable of using non-CEGAR based analyses. This behavior is engrained in the implementation in multiple ways, such as counterexamples and safety proofs requiring a partial or full abstract reachability

⋆ Jury member representing EmergenTheta at SV-COMP 2024.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 371–375, 2024.
https://doi.org/10.1007/978-3-031-57256-2_23

graph, and the interface of the backend containing references to *precision* [6]. Our main contribution as part of SV-COMP'24 is the removal of such dependencies on abstraction-specific classes. This enables the rapid prototyping and development of diverse verification algorithms such as this year's BMC, IMC, and k-induction algorithms [5,7,9], building the low-level core of Theta including the representation and manipulation of expressions and interfacing with several SMT-solvers.

To facilitate the implementation of these algorithms, we introduced a new `MonolithicTransitionFunction` interface to Theta, which returns a single nondeterministic action representing the whole transition system (i.e., it represents the structural information as additional variables and related guards). This is a counterpart to the previously existing TransitionFunction interface, which directly relies on the structural information for the enabledness of actions. This interface has been implemented for most of the formalisms supported by Theta.

Besides the changes detailed above, EmergenTheta still relies on Theta's Antlr-based C frontend and integrated support for SMT-solvers, as well as its existing counterexample-to-witness projection [1].

2 Verification Approach

In *bounded model checking* (BMC) [5], the transition system and the safety property are encoded as SMT [3] formulas. In each iteration of the algorithm, a path constraint is created from the formulas characterizing all execution paths of a given length k that start in an initial state and end in an error state. The satisfiability of the path constraint is checked using an SMT solver [8]. If a satisfying assignment is found, it is returned as a counterexample, else the bound k is increased until the available resources allow.

BMC is incomplete as it can only prove the absence of counterexamples up to a finite depth. *K-induction* [9] and *interpolation-based model checking* (IMC) [7] address this by adding checks that attempt to prove that the property holds for unbounded depth based on the unsatisfiability of the BMC query. K-induction does so by trying to prove the k-inductivity of the property with k being the current BMC length, while IMC derives Craig interpolants to compute an overapproximation of the set of reachable states.

Based on preliminary testing, we used a simple sequential portfolio (without algorithm selection) that executed an IMC-only verification phase first (for at most 90 seconds), then fell back to a combined BMC and k-induction-based verification phase for the rest of the time limit. EmergenTheta did not employ any of the CEGAR-based analysis methods already present it Theta, as we wanted to evaluate the newly implemented ones separately.

	All		False		True	
Tool	EmergenTheta	Theta	EmergenTheta	Theta	EmergenTheta	Theta
Category						
Arrays	13 (7)	13 (7)	0 (0)	5 (5)	13 (7)	8 (2)
BitVectors	16 (1)	23 (8)	7 (0)	10 (3)	9 (1)	13 (5)
Combinations	1 (0)	138 (137)	1 (0)	121 (120)	0 (0)	17 (17)
ControlFlow	7 (4)	9 (6)	1 (0)	2 (1)	6 (4)	7 (5)
ECA	1 (1)	307 (307)	0 (0)	133 (133)	1 (1)	174 (174)
Floats	25 (6)	54 (35)	2 (0)	23 (21)	23 (6)	31 (14)
Hardness	378 (269)	116 (7)	0 (0)	0 (0)	378 (269)	116 (7)
Hardware	134 (15)	194 (75)	60 (10)	89 (39)	74 (5)	105 (36)
Heap	2 (0)	2 (0)	0 (0)	0 (0)	2 (0)	2 (0)
Loops	232 (117)	161 (46)	34 (11)	40 (17)	198 (106)	121 (29)
Sequentialized	1 (0)	47 (46)	1 (0)	34 (33)	0 (0)	13 (13)
XCSP	2 (0)	45 (43)	2 (0)	44 (42)	0 (0)	1 (1)
Overall	812 (420)	1153 (761)	108 (21)	530 (443)	704 (399)	623 (318)

Table 1: Comparison of Theta and EmergenTheta for each subcategory

3 Discussion of Strengths and Weaknesses of the Approach

As our secondary goal (besides adapting Theta's architecture to a more flexible one) was to find out how the new algorithms implemented in EmergenTheta performed, we mainly compare and contrast the results of EmergenTheta (which used only the newly implemented algorithms) and Theta (which used only CEGAR). In the future, we aim to integrate the new algorithms into our mainline Theta tool, for which this evaluation is invaluable.

Table 1 compares the number of tasks correctly solved by Theta and EmergenTheta for each subcategory in ReachSafety (using official results) [4], distinguishing between true and false outputs. The numbers in parentheses show the number of correctly solved tasks that the other tool was unable to solve in time.

Looking at the overall results, we can see that Theta and EmergenTheta are suitable for different tasks: although Theta solved more tasks, EmergenTheta solved 420 tasks *that* Theta *could not solve*, which is 36% of the 1153 tasks solved by Theta. With an ideal portfolio, incorporating these algorithms could significantly increase the number of tasks solved by Theta.

Theta was much better at finding counterexamples (108 vs 530 false outputs), while EmergenTheta was slightly better at proving correctness (704 vs 623 true outputs). This goes against our intuition, as abstraction refinement is more tailored to proving correctness. This phenomenon warrants further investigation; our current hypothesis is that performing enough refinements to eliminate all spurious counterexamples had too large an overhead. More than half of the true results for each tool were for tasks that the other one could not solve, highlighting their complementary nature.

EmergenTheta was significantly better in the Loops and the Hardness categories, while it was worse in Combinations, ECA, Sequentialized and XCSP. As for Combinations and Sequentialized, this could be attributed to Theta being generally better at finding counterexamples, as false tasks are overrepresented in these categories; but for ECA and XCSP, tasks of both types are represented nearly equally.

These relatively positive results were achieved in spite of a misconfiguration: although our preliminary measurements had shown that CVC5 and MathSAT performed best with K-IND and IMC respectively, we accidentally enrolled EmergenTheta with its default solver Z3. We consider this a failure in the design of the portfolio engine of Theta, which allowed us to submit a faulty configuration without this being evident in the logs (that no runs were using solvers other than Z3). We will prioritize improving on this aspect of Theta for next year.

4 Tool Setup and Configuration

EmergenTheta remains vastly configurable, and successfully choosing a performant configuration for a verification task at hand can be complicated. If using the competition archive [2] for software verification, we recommend using the preassembled portfolio: `theta-start.sh <input> --backend IMC_THEN_KIND`. To minimize the output verbosity and produce a witness in the working directory, the flags `--loglevel RESULT` and `--witness-only` can be added to the arguments. We also used these options at SV-COMP 2024.

5 Software Project and Contributors

EmergenTheta is integrated into the Theta verification framework maintained by the Critical Systems Research Group[1] of the Budapest University of Technology and Economics. The project is available open-source on GitHub[2] under an Apache 2.0 license. The version (`5.0.0`) used in the competition is available at [2].

References

1. Ádám, Z., Bajczi, L., Dobos-Kovács, M., Hajdu, Á., Molnár, V.: Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) TACAS 2021. Lecture Notes in Computer Science, vol. 13244, pp. 474–478. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_34

[1] https://ftsrg.mit.bme.hu/en/

[2] https://github.com/ftsrg/theta/releases/tag/svcomp24

2. Bajczi, L., Szekeres, D., Mondok, M., Molnár, V.: EmergenTheta - SV-COMP'24 Verifier Archive (Nov 2023). https://doi.org/10.5281/zenodo.10198872
3. Barrett, C., Tinelli, C.: Satisfiability Modulo Theories. https://doi.org/10.1007/978-3-319-10575-8_11
4. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS (1999). https://doi.org/10.1007/3-540-49059-0_14
6. 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
7. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification (2003). https://doi.org/10.1007/978-3-540-45069-6_1
8. 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
9. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Formal Methods in Computer-Aided Design (2000). https://doi.org/10.1007/3-540-40922-X_8
10. Tóth', T.: Abstraction Refinement-Based Verification of Timed Automata. Ph.D. thesis, Budapest University of Technology and Economics (2021)

ESBMC v7.4: Harnessing the Power of Intervals

(Competition Contribution)

Rafael Sá Menezes[1,2], Mohannad Aldughaim[1,6], Bruno Farias[1], Xianzhiyu Li[1],
Edoardo Manino[1], Fedor Shmarov[1,5], Kunjian Song[1], Franz Brauße[1](✉)[⋆],
Mikhail R. Gadelha[3], Norbert Tihanyi[4], Konstantin Korovin[1],
and Lucas C. Cordeiro[1,2]

[1] The University of Manchester, Manchester, UK
franz.brausse@manchester.ac.uk
[2] Federal University of Amazonas, Manaus, Brazil
[3] Igalia, A Coruña, A Coruña, Spain
[4] Eötvös Loránd University, Budapest, Hungary
[5] Newcastle University, Newcastle upon Tyne, UK
[6] King Saud University, Riyadh, Saudi Arabia

Abstract. ESBMC implements many state-of-the-art techniques that combine abstract interpretation and model checking. Here, we report on new and improved features that allow us to obtain verification results for previously unsupported programs and properties. ESBMC now employs a new static interval analysis of expressions in programs to increase verification performance. This includes interval-based reasoning over booleans and integers, and forward-backward contractors. Other relevant improvements concern the verification of concurrent programs, as well as several operational models, internal ones, and also those of libraries such as pthread and the C mathematics library. An extended memory safety analysis now allows tracking of memory leaks that are considered still reachable.

1 Software Architecture

ESBMC [4,6] is a mature, permissively licensed open-source context-bounded model checker for the verification of single- and multi-threaded C programs for various code safety violations (e.g., buffer overflows, dangling pointers, arithmetic overflows) and user-defined assertions. It has been successfully participating in the SV-COMP competitions for many years due to our continuous work towards improving its performance. ESBMC transforms a given C program using a Clang-based [11] front-end into an intermediate representation in the GOTO language [3], which is symbolically executed to produce verification formulae passed to one or more SMT solvers. In addition, ESBMC implements state-of-the-art incremental BMC and k-induction proof-rule algorithms based on SMT and Constraint Programming (CP) solvers.

[⋆] Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 376–380, 2024.
https://doi.org/10.1007/978-3-031-57256-2_24

2 Verification Approach

Interval analysis In this year, ESBMC interval analysis was improved using Abstract Interpretation techniques [5]. We used the integer domain (with infinities) as the abstract domain for SV-COMP. The domain consists of, for each statement in the program, keeping the box interval (i.e., a *minimum* and *maximum*) for all variables. ESBMC also supports interval arithmetic and widening strategies (through extra- and interpolation). Once computed, the intervals are used for optimizations (i.e., dead code elimination and constant folding) and invariant instrumentation.

Regarding the new code instrumentation, the main use of intervals is to generate invariants which the k-induction strategy benefits from most. This is done by adding assumptions restricting the value of variables. In addition, the set of variables used for these assumptions has been reduced to those occurring in conditional statements and guards only. Lastly, we expanded the types of instrumented statements: assertions, conditionals, and function calls.

Contractors ESBMC v7.4 employs another method to refine intervals based on contractors. Contractors [10,14] are commonly used in the context of Constraint Satisfaction Problems (CSPs), that is, when variables, their (real-valued) domains, and constraints over those variables are fixed. A contractor is an operation on n-dimensional boxes (product of intervals) respecting the given constraints, i.e., it refines the domains such that no solutions to the CSP are lost. A particularly efficient one for CSPs containing a single constraint is the Forward-backward contractor [7,8,15]. It operates in two stages: forward evaluation and backward propagation [14,1]. In scenarios with multiple constraints, the forward-backward contractor is applied to each constraint independently.

ESBMC utilizes the forward-backward contractor implemented in the Ibex library [2] to refine the results of the interval analysis mentioned above. That is, conditions of statements such as “if” and loops in the program are relaxed to conditions over reals, where possible, and then the contractor is applied to this relaxed condition. The result is a refined set of intervals for the variables involved. These refined intervals are then restricted to the original variable domains, which – in case of, e.g., integers – results in a further reduction of the size of intervals. The intervals contracted in this way generally enhance the results of the interval analysis employed by ESBMC and benefit its k-induction strategy.

Memory leaks This year, ESBMC employs a refined check for the *valid-memtrack* property. This property is loosely described as only allowing those dynamically allocated objects to survive that are still reachable at the end of the program's execution by following a path of pointers stored in objects eventually referenced by global variables. A property violation witness has to contain proof of *unreachability* of a dynamic allocation starting from any global variable.

The new algorithm leverages the existing one tracking the lifetime of allocations for the *valid-memcleanup* property, but it specifically excludes still-reachable objects from the check. This condition is encoded into an SMT formula using the paths deterministically described by expressions of type struct, union, pointer, or array with constant size. Each possible successor along the path is obtained through the value-set, and the validity is encoded through guards which have to hold at the end of execution.

C mathematical library ESBMC v7.4 offers extended support for the `math.h` library. Accurate modeling of its semantics is crucial for reasoning on the behavior of complex floating-point software. For example, most neural network code relies on 32-bit floats and may invoke the `math.h` library to compute the result of activation functions, positional encodings, and vector normalisations [12].

The IEEE 754 standard [9] mandates bit-precise semantics for a small subset of the `math.h` library only. This subset includes addition, multiplication, division, `sqrt`, `fma`, and other support functions such as `remquo`. In contrast, the behavior of most transcendental functions (e.g., `sin`, `exp`, `log`) is platform-specific. Still, the standard recommends implementing the correct rounding whenever possible.

As a tradeoff between precision and verification speed, ESBMC now features a two-pronged design. For the most commonly-used `float` functions, we borrow the MUSL plain-C implementation of numerical algorithms [13]. For the corresponding `double` functions, we employ less complex algorithms with approximate behavior.

Data races Data races occur when multiple threads concurrently access the same memory location, and at least one of these accesses involves a write operation. ESBMC's algorithm for checking data races extends the static code instrumentation CBMC [3] uses. The idea is to add a flag A', initially true, to each variable A involved in an assignment. Directly after the assignment to A, A' is reset to false. To identify races, we assert that the value of A' is false when A is accessed. Subsequently, we outline the challenges encountered by ESBMC and the improvements we have implemented.

As this method introduces additional instructions into the program, the potentially larger number of thread interleavings is counteracted by inserting atomic blocks appropriately – subject to ensuring accuracy, the atomic block encompasses the assertion on A', original assignment to A, and setting A', in sequence. Data races are now also checked on access of arrays with non-constant indices. The most challenging aspect of data race detection is the dereference of pointers, as the pointer would have to be instrumented but is not statically known through the value-set analysis. Thus, the new implementation is hybrid, addressing cases unsuitable for static analysis during symbolic execution, thereby enabling ESBMC to detect more types of data races.

3 Strengths and Weaknesses

The interval analysis improved and provided better invariants for ESBMC. The new optimizations help ESBMC to solve new benchmarks in categories with multiple path conditions (i.e., ECA). The main weakness of the method is that our Abstract Interpreter only has partial support for widening, and it is not context-aware (i.e., function parameters and global variables cannot be tracked globally). This results in a slowdown for categories with loops with thousands of statements (e.g., Hardware).

While contractors are highly regarded for their ability to provide assured limits on solutions, their cautious approach may lead to overly broad results and less precise conclusions. Therefore, a more rigorous evaluation of contractors is essential to assess their advantages and limitations effectively.

The new algorithm for the *valid-memtrack* sub-property allowed ESBMC to identify 70/153 violations correctly with no incorrect verdicts (last year: 0/134). There is a theoretical weakness in the current implementation concerning dynamic allocations only reachable through pointers stored in arrays of statically unknown size. It could result in incorrect-false verdicts, but it has not been observed in test cases, yet.

Without operational models of the `math.h` library, ESBMC would assign nondeterministic results, which may cause incorrect counterexamples to be returned. This behavior is especially evident for older versions of ESBMC on neural network code [12], as it usually contains many mathematical operations. ESBMC v7.4 fixes this semantic issue by providing explicit operational models for many common functions in `math.h`, thus yielding no incorrect results on the benchmarks in [12], and achieving second place in the ReachSafety-Floats sub-category.

From the competition results, the data race detection of ESBMC v7.4 is promising. Compared to the previous version, the new algorithm supports more types of expressions and reduces the verification time. The relatively high number of 2.2% incorrect-true verdicts is mostly due to still missing support for detecting data races during dereferences of pointers to compound types.

We will address the weaknesses identified in this competition in the future.

4 Tool Setup and Configuration

To setup and run ESBMC, follow the instructions in the `README.md` file. ESBMC can also be run via the Python wrapper `esbmc-wrapper.py` for simplified usage in the competition. An example command line is:

```
esbmc-wrapper.py -s kinduction -a 64 -p unreach-call.prp example.c
```

5 Software Project

The ESBMC development is funded by ARM, EPSRC EP/T026995/1, EPSRC EP/V000497/1, Ethereum Foundation, EU H2020 ELEGANT 957286, UKRI Soteria, Intel, and Motorola Mobility (through Agreement N° 4/2021). It is publicly available at `http://esbmc.org` under the terms of the Apache License 2.0 and static release builds of ESBMC are provided at `https://github.com/esbmc/esbmc`. The version that participated in SV-COMP 2024 is available at `https://doi.org/10.5281/zenodo.10198805`.

References

1. M. Aldughaim, K. M. Alshmrany, M. R. Gadelha, R. de Freitas, and L. C. Cordeiro. FuSeBMC_IA: Interval analysis and methods for test case generation. In L. Lambers and S. Uchitel, editors, *Fundamental Approaches to Software Engineering*, pages 324–329, Cham, 2023. Springer Nature Switzerland.
2. G. Chabert and ibex team. ibex-lib, 2023. `https://github.com/ibex-team/ibex-lib` [Accessed: 19 December 2023].
3. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
4. L. C. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.
5. P. Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.
6. M. Y. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering ASE*, pages 888–891. ACM, 2018.
7. L. Granvilliers. Revising hull and box consistency. *Logic Programming*, pages 230–244, 1999.
8. E. Hansen and G. W. Walster. *Global optimization using interval analysis: revised and expanded*, volume 264. CRC Press, 2003.
9. IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.
10. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. Applied Interval Analysis. In *Springer London*, 2001.
11. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International symposium on code generation and optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.
12. E. Manino, R. S. Menezes, F. Shmarov, and L. C. Cordeiro. NeuroCodeBench: a plain C neural network benchmark for software verification, 2023.
13. musl community. musl libc, 2023. `https://musl.libc.org/` [Accessed: 15 December 2023].
14. M. Mustafa, A. Stancu, N. Delanoue, and E. Codres. Guaranteed SLAM—An interval approach. *Robotics and Autonomous Systems*, 100:160–170, 2018.
15. A. Neumaier. *Interval methods for systems of equations*, volume 37. Cambridge University Press, 1990.

Goblint: Abstract Interpretation for Memory Safety and Termination
(Competition Contribution)

Simmo Saan[1](✉)*, Julian Erhard[2,3], Michael Schwarz[2],
Stanimir Bozhilov[2], Karoliine Holter[1], Sarah Tilscher[2,3],
Vesal Vojdani[1], and Helmut Seidl[2]

[1] University of Tartu, Tartu, Estonia
{simmo.saan,karoliine.holter,vesal.vojdani}@ut.ee
[2] Technische Universität München, Garching, Germany
{julian.erhard,m.schwarz,stanimir.bozhilov,
sarah.tilscher,helmut.seidl}@tum.de
[3] Ludwig-Maximilians-Universität München, Munich, Germany

Abstract. Goblint is an abstract interpreter of C programs, focusing on the analysis of multi-threaded code. It is equipped with a variety of abstract domains, as well as analyses which allow it to reason about an array of program properties in a highly configurable manner. Goblint has been extended with support for the detection of memory safety bugs and non-termination.

1 Verification Approach

Goblint is an abstract-interpretation–based static analyzer of C code, with an emphasis on the sound analysis of multi-threaded programs [14, 15]. It uses side-effecting constraint systems [2] to combine context-sensitive analysis of local states with flow-insensitive analysis of data possibly shared between threads. Goblint is equipped with a range of different analyses that, in turn, build on multiple abstract domains for expressing candidate program invariants.

1.1 Memory Safety

Techniques for detecting memory-related bugs have been extensively studied [6, 9, 10, 20]. While Goblint did not target such bugs in the past, new analyses for the *sound* analysis of memory safety have been added for SV-COMP 2024. The analyzer already tracks abstract address sets for pointer variables. A single abstract address consists of a variable and an abstract offset. The analyzer distinguishes between regular program variables and allocated memory blocks, which are identified by their respective allocation sites together with the allocating thread and possibly an allocation counter.

* Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 381–386, 2024.
https://doi.org/10.1007/978-3-031-57256-2_25

The new analyses are concerned with the detection of the following memory-safety bugs: *invalid memory deallocations*, *invalid pointer dereferences*, as well as *memory leaks*. Beyond *null-pointer dereferences*, two further kinds of invalid dereferences are now considered: *memory out-of-bounds* accesses and *use-after-free* (UAF) bugs. Memory out-of-bounds accesses can be uncovered by obtaining the size, as well as the offset from the base address of the memory being accessed. To determine whether an access via some offset may be out of bounds, the analysis relies on an expressive combination of integer domains including intervals.

Invalid dereferences due to use-after-frees can be detected in the single- and multi-threaded case. For the single-threaded setting, the analysis uses the allocation-site abstractions in order to keep track of potentially already deallocated memory, and warns on accesses to such memory. Regarding the multi-threaded case, it additionally leverages Goblint's side-effecting functionality by maintaining a global invariant that, for each piece of deallocated memory, collects the set of all threads that may free it. Goblint tracks abstract thread IDs which allow reasoning about which threads may run in parallel [17]. The may-happen-in-parallel (MHP) information from the abstract thread ID domain (and a dedicated analysis of thread joins) is used to infer whether an access to a piece of memory may happen in parallel with (or after) the deallocation of the same piece of memory by another thread. In addition, invalid frees due to possibly occurring double frees are flagged by this analysis as well.

Potential memory leaks can be detected thanks to a dedicated analysis. To this end, all allocated memory blocks are tracked path- and context-sensitively. Furthermore, the allocation counter is relied on to potentially exclude memory leaks for a particular allocation site. Calls to deallocating functions, such as `free`, have the effect of removing pieces of tracked (and now deallocated memory) from the state, whenever the analysis determines that the passed pointer *must* point to an abstract block of memory which describes a single concrete memory location. At all exit points of the program, it is then checked whether the set of possibly still allocated memory is empty. In case any such set is non-empty, a memory leak is reported. In the multi-threaded case, the analysis checks the following stronger property and warns whenever that property may be violated:

1. all threads have terminated at the end point of `main`, and
2. `exit` and similar functions, causing early termination, are not called, and
3. at its return, each thread has freed all the memory it allocated.

This property allows for a thread-modular analysis, where sets of allocated and freed memory are maintained in a flow- and context-sensitive manner.

We remark that the analysis for memory leaks tracks which heap-allocated memory *may not* be freed yet, while the analysis to detect UAF issues tracks which memory *may* potentially already be freed. One direction of improvement would be to consider tracking relational pointer information along the lines of Seidl et al. [18] and, additionally, consider relational information about the lengths of arrays and memory blocks. This may be useful in the case of variable length arrays and dynamically allocated memory for which the size is not statically known.

1.2 Termination

A termination analysis has been added, largely leveraging existing features of the framework. This highlights the versatility of the framework. To account for non-termination due to loops, a counter variable is inserted into each loop and incremented in every loop iteration. A relational polyhedra analysis based on APRON [8] is then used to determine whether the counter variable is bounded. To detect potential non-termination due to recursion, the notion of a call graph is enhanced by considering functions together with their respective abstract calling contexts and taking dynamic calls via pointers into account. This graph is *a posteriori* extracted out of the analysis result and then checked for cycles in a post-processing phase. In case no cycles (including self-loops) exist in the abstract call graph, there can be no cycles in the concrete call graph.

The currently implemented termination analysis is just a first step in the realization of related techniques. Future work may, e.g., be the tuning of the abstract contexts for this use-case, or the incorporation of more involved techniques for termination analysis by abstract interpretation [4, 5]. Extending the presented approaches to the non-termination of concurrent programs while remaining as thread-modular as possible seems particularly challenging.

2 Software Architecture

GOBLINT is implemented in ∼54,000 lines of OCAML and uses an updated fork of CIL [12] as its parser frontend for the C language. It depends on APRON [8] for relational analyses. No other major libraries or external tools are required.

The modular architecture of GOBLINT [1] allows a combination of analyses to be selected and automatically configured at runtime [15]. Analyses are defined through their abstract domains and transfer functions, which can communicate with other analyses using predefined queries and events. The combined analyses together with the control-flow graphs of the functions yield a side-effecting constraint system [2], which is solved using a local generic solver [19]. The solution is post-processed to determine the verdict and construct a witness.

3 Strengths and Weaknesses

GOBLINT once again demonstrated its soundness in this year's competition, i.e., it did not produce any false negatives. The only other tools that did not produce any false negatives are AISE [21] (competing only in *ReachSafety-Loops*), BRICK (competing in three sub-categories of *ReachSafety*), and MOPSA [11] (competing in all categories except *ConcurrencySafety* and *Termination*). GOBLINT is thus the only *sound* tool in SV-COMP 2024 to support *all* properties, and the only sound tool represented in the overall ranking. Among the tools participating in the overall ranking, GOBLINT, despite targeting only proofs – which are traditionally considered to be more time-consuming than finding counter-examples – leads the pack in terms of points achieved in $\leq 9\,\mathrm{s}$. This is most pronounced when considering runtimes ≤ 1 s. This highlights the efficiency of Goblint. Beyond these observations, we briefly discuss the newly added analyses here. Support for soundly detecting memory safety bugs greatly broadens the applicability of the analyzer, evidencing the flexibility of the underlying framework. Of particular note is the support for verifying the memory-safety of multi-threaded programs in a thread-modular way, yielding the second-best score in *ConcurrencySafety-MemSafety*, after Deagle [7]. Turning to termination analysis, the added analysis demonstrates that a considerable chunk of the SV-COMP benchmarks in this category can be handled by using our extended dynamic call graph to deal with recursion and ghost counters together with numerical relational domains to deal with loops. Finally, Goblint now comes with dedicated support for analyzing programs using `setjmp`/`longjmp` and flagging their misuse [16]. We have contributed programs using this language feature to the benchmark suite.

A general weakness of Goblint currently is that, while it supports expensive but expressive relational domains such as polyhedra, it lacks a heuristic when to activate them, and thus only uses them for termination analysis. Activating these domains based on some program properties, or attempting analysis with such expensive domains after an analysis without them was inconclusive, may help to improve the precision of the analyzer without compromising its efficiency.

4 Tool Setup and Configuration

Goblint version `svcomp24-0-gc2e9465a7` participated in SV-COMP 2024 [3, 13]. It is available in both binary (Ubuntu 22.04) and source code form at our GitHub repository.[4] Instructions for building from source can be found in the `README`. Both the tool-info module and the benchmark definition for SV-COMP are named `goblint`. They correspond to running the tool as follows:

```
./goblint --conf conf/svcomp24.json \
          --set ana.specification property.prp input.c
```

Goblint participated in *all* the categories, while opting-out from *FalsificationOverall*.

5 Software Project and Contributors

Goblint development takes place on GitHub, while related publications are listed on its website.[5] It is an MIT-licensed project initiated by Technische Universität München and the University of Tartu.

Acknowledgments. This work was supported by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 ConVeY 2. We would like to thank everyone who has contributed to Goblint over the years, especially the students who contributed the termination analysis, namely: Thomas Lagemann, Johanna Franziska Schinabeck, Alexander Schlenga, and Isidor Zweckstetter.

[4] https://github.com/goblint/analyzer/releases/tag/svcomp24

[5] https://github.com/goblint/analyzer and https://goblint.in.tum.de

Data Availability Statement. All data of SV-COMP 2024 are archived as described in the competition report [3] and available on the competition website. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of Goblint as used in the competition is archived on Zenodo [13].

Bibliography

[1] Apinis, K.: Frameworks for analyzing multi-threaded C. Ph.D. thesis, Technische Universität München (2014)

[2] Apinis, K., Seidl, H., Vojdani, V.: Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In: APLAS '12, pp. 157–172, Springer (2012), doi: 10.1007/978-3-642-35182-2_12

[3] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: TACAS '24, Springer (2024)

[4] Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: POPL '12, pp. 245–258, ACM (2012), doi: 10.1145/2103656.2103687

[5] Dimovski, A.S.: Lifted termination analysis by abstract interpretation and its applications. In: GPCE '21, pp. 96–109, ACM (2021), doi: 10.1145/3486609.3487202

[6] Gui, B., Song, W., Xiong, H., Huang, J.: Automated use-after-free detection and exploit mitigation: How far have we gone? IEEE Trans. Software Eng. **48**(11), 4569–4589 (2022), doi: 10.1109/TSE.2021.3121994

[7] He, F., Sun, Z., Fan, H.: Deagle: An SMT-based verifier for multi-threaded programs. In: TACAS '22, vol. 2, pp. 424–428, Springer (2022), doi: 10.1007/978-3-030-99527-0_25

[8] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV '09, pp. 661–667, Springer (2009), doi: 10.1007/978-3-642-02658-4_52

[9] Jones, J., Wasson, J., Brown, S., Poulsen, S., Aldous, P., Mercer, E.: Memory safety in C by abstract interpretation. SIGSOFT Softw. Eng. Notes **43**(4), 56 (2019), doi: 10.1145/3282517.3282530

[10] Loginov, A., Yahav, E., Chandra, S., Fink, S., Rinetzky, N., Nanda, M.: Verifying dereference safety via expanding-scope analysis. In: ISSTA '08, pp. 213–224, ACM (2008), doi: 10.1145/1390630.1390657

[11] Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa-C: Improved verification for C programs, simple validation of correctness witnesses. In: TACAS '24, Springer (2024)

[12] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC '02, pp. 213–228, Springer (2002), doi: 10.1007/3-540-45937-5_16

[13] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint at SV-COMP 2024 (Nov 2023), doi: 10.5281/zenodo.10202867, tool artifact

[14] Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting

constraints. In: TACAS ’21, pp. 438–442, Springer (2021), doi: 10.1007/978-3-030-72013-1_28

[15] Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: Goblint: Autotuning thread-modular abstract interpretation. In: TACAS ’23, vol. 2, pp. 547–552, Springer (2023), doi: 10.1007/978-3-031-30820-8_34

[16] Schwarz, M., Erhard, J., Vojdani, V., Saan, S., Seidl, H.: When long jumps fall short: Control-flow tracking and misuse detection for non-local jumps in C. In: SOAP ’23, pp. 20–26, ACM (2023), doi: 10.1145/3589250.3596140

[17] Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered relational thread-modular abstract interpretation with local traces. In: ESOP ’23, pp. 28–58, Springer (2023), doi: 10.1007/978-3-031-30044-8_2

[18] Seidl, H., Erhard, J., Schwarz, M., Tilscher, S.: 2-pointer logic. In: Javier Esparza’s 60th Birthday, pp. 254–264, Springer (2024)

[19] Seidl, H., Vogler, R.: Three improvements to the top-down solver. Math. Struct. Comput. Sci. **31**(9), 1090–1134 (2021), doi: 10.1017/S0960129521000499

[20] Sui, Y., Ye, D., Xue, J.: Static memory leak detection using full-sparse value-flow analysis. In: ISSTA ’12, pp. 254–264, ACM (2012), doi: 10.1145/2338965.2336784

[21] Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution. In: TACAS ’24, Springer (2024)

Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)

Raphaël Monat[1] (✉) ⋆, Marco Milanese[2], Francesco Parolini[2], Jérôme Boillot[3], Abdelraouf Ouadjaout[4], and Antoine Miné[2]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
raphael.monat@inria.fr
[2] LIP6, Sorbonne Université, F-75005 Paris, France
[3] École Normale Supérieure, Université PSL, F-75005 Paris, France
[4] Grenoble, France

Abstract. We present advances we brought to Mopsa for SV-Comp 2024. We significantly improved the precision of our verifier in the presence of dynamic memory allocation, library calls such as `memset`, `goto`-based loops, and integer abstractions. We introduced a witness validator for correctness witnesses. Thanks to these improvements, Mopsa won SV-Comp's *SoftwareSystems* category by a large margin, scoring 2.5 times more points than the silver medalist, Bubaak-SpLit.

Keywords: Static Analysis · Abstract Interpretation · Competition on Software Verification · SV-Comp.

1 Verification Approach: the Mopsa platform

Mopsa is an open-source static analysis platform relying on abstract interpretation [6]. The implementation of Mopsa aims at exploring new perspectives for the design of static analyzers. Journault et al. [8] describe the core Mopsa principles, and Monat [12, Chapter 3] provides an in-depth introduction to Mopsa's design. The C analysis which we rely on for this competition is based on the work of Ouadjaout and Miné [16]; it proceeds by induction on the syntax, is fully context- and flow-sensitive, and committed to be sound. This is the second time Mopsa participates in SV-Comp [15]. We have brought precision improvements, described below; they have proved decisive for the *SoftwareSystems* category.

Dynamic memory allocation precision improvements. Mopsa relies on the recency abstraction [1] to handle dynamic allocation. For each allocation site, this abstraction keeps the last allocated block separated from the others, the latter being summarized into a single, weak memory block. Allocation sites are customizable [14], they are usually based on a program location. However, this summarization can be detrimental to precision. We implemented an alternative abstraction that keeps memory blocks separated during loop unrolling. This

⋆ Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 387–392, 2024.
https://doi.org/10.1007/978-3-031-57256-2_26

enhancement, combined with targeted loop unrolling helped us verify more tasks, including 246 from the *uthash* categories. Specifically, we proved correct half the tasks of *uthash-NoOverflow* category, which are out of reach of the other verifiers.

Integer abstractions. Mopsa only supported convex representations of integer sets, such as intervals. As such, it was impossible to precisely represent cases where $x \in [-10, 10]$ and $x \neq 0$. We have resolved this issue by adding an excluded set domain, which tracks a set of values a given variable cannot take. We have also implemented the symbolic rewriting domain of Boillot and Feret [4], which simplifies arithmetic expressions with overflows into simpler ones. This new implementation has been written in 1,200 lines of OCaml code.

Improved precision for goto-based loops. Since the analyzer iterates on the syntax of the program, `goto` statements require the usage of flows tokens [12] and a special fixpoint iteration scheme. We added support for a decreasing iteration pass, which allows to recover some precision after the generalization performed by the widening operator. In addition, we added a syntactical loop rewriting pass which turns few special goto patterns into equivalent while loops which are analyzed more precisely.

Precise stub initialization. Ouadjaout and Miné [16] implemented a stub language and its interpretation for the C standard library in Mopsa. Contiguous region initialization through functions such as `memset` were not handled precisely by our implementation of the cells domain [11], mainly to be scalable. We improved the domain to handle region initialization up to a given bound, and `NULL` pointer synthesis from a contiguous block of 0 bytes.

Other improvements. Some SV-Comp programs have specific symbolic argument initialization performed by client code, with variable parameters on the maximal size of all symbolic arguments. We have thus extended Mopsa to handle a wide range of parameters for symbolic argument initialization, matching those found in SV-Comp programs. We also rely on the flambda optimizer for OCaml, which brings more than a 15% performance improvements.

2 Software Architecture: the SV-Comp driver

By default, the C analysis of Mopsa detects all the runtime errors that may happen in the analyzed program, while SV-Comp tasks focus on a specific property at a time. We thus rely on an SV-Comp specific driver. It takes as input the task description (program, property, data model). It runs increasingly precise C analyses defined in Mopsa until the property of interest is proved or the most precise analysis is reached (or the resources are exhausted). Each analysis result is postprocessed by the driver to check if the property is proved.

An analysis configuration defines the set of domains used, and their parameters allowing modifications of the precision-efficiency ratio. A breakdown of the results is shown in Fig. 1. This year, we use five configurations. Conf. 1 relies on intervals and cells [11]. Conf. 2 additionally enables the string length domain [9], the excluded powerset domain, and congruences. It performs decreasing iterations for `goto` statements, unrolls the first 10 iterations of loops, enables the

Max. Conf.	Tasks proved correct		Tasks yielding timeout	
1	6995		368	
2	7775	(+780)	717	(+349)
3	8197	(+422)	2954	(+2237)
4	8257	(+60)	3527	(+573)
5	8400	(+143)	9532	(+6005)

Fig. 1. Max. Conf. i represents the sequence of increasingly precise analyses from Conf. 1 up to Conf. i. Max. Conf. 2 is able to prove 780 tasks correct in addition to the 6995 proved by conf. 1, although 717 tasks reach the resource limits when analyzed by Conf. 1 and 2 (349 more than by Conf. 1 alone). There are 25885 tasks in total, and 17851 correctness tasks. Mopsa can only prove program correctness for now (68% of the tasks); it yields "unknown" when unable to prove a program correct.

enhanced memory allocation abstraction, and the more precise evaluation of stubs. Conf. 3 adds a polyhedra abstract domain, relying on a static packing to scale [7]. This includes tracking numerical relations between string lengths and scalar variables. A pointer sentinel domain is added to symbolically track the position of the first `NULL` value of a pointer array. Decreasing iterations are also enabled `for`/`while` loops, and the first 15 iterations of loops are unrolled. Conf. 4 adds the symbolic rewriting domain of Boillot and Feret [4]. Loop unrolling is extended to 60 iterations. Conf. 5 performs a fully relational analysis of the analyzed program without packing.

Witness Validation. We extended our driver to support the witness validation phase of SV-Comp: we inject loop invariants of a witness, encoded as assertions into the original program. We then check that this patched program is correct. This approach is similar to Metaval's [3], but we used the new YAML format. The work of Saan et al. [22] is more involved: it leverages the witness to guide their analysis and yields precision improvements, compared to their bare analysis.

3 Strengths and Weaknesses

Mopsa participated in the following categories, targeting C programs: *ReachSafety, MemSafety, NoOverflows* and *SoftwareSystems*. It did not compete in the termination category and cannot precisely analyze concurrency-related verification tasks. The highlight of this year's participation is Mopsa's gold medal in the *SoftwareSystems* track, focusing on verifying real software systems. Mopsa scored 2.5 times more points than the second tool, Bubaak-SpLit [5]. Figure 2 breaks down the results of Mopsa in the subcategories of the *SoftwareSystems* track, highlighting our progress, and the best results obtained by this year's verifiers. An overview of results can be found in the competition report [2].

Strengths. Mopsa is quite scalable: our cheapest configuration is able to analyze a given program within the allocated resource budget in 98.6% of the cases. In addition, Mopsa is the only verifier of 2023 and 2024 able to gain points in the DDLL category, corresponding to large instances of instrumented Linux drivers.

Category	Prop.	|tasks|	Mopsa'23	Mopsa'24	Best score (2024)	
AWS	R	197	32	36	137	Symbiotic
coreutils	M	140	0	0	0	_
coreutils	N	30	0	4	4	Mopsa
BusyBox	N	54	4	8	8	Mopsa
DDL	R	2442	3174	3476	3476	Mopsa
DDLL	R	8	10	14	14	Mopsa
DDL	M	141	0	8	71	Bubaak-SpLit
other	R	22	0	10	10	Mopsa
other	M	34	0	12	12	Mopsa
uthash	R	138	0	192	228	Bubaak*, Symbiotic
uthash	M	138	0	96	204	Bubaak*, Symbiotic
uthash	N	114	0	204	204	Mopsa

Fig. 2. Mopsa's improvements for subcategories of the *SoftwareSystems* track. Property is either ***R**eachSafety*, ***M**emSafety* or ***N**oOverflow*. The last three columns show the score of Mopsa submitted last year, this year, and the best score reached by a verifier.

Mopsa is committed to being sound. Thanks to this, we have been able to fix 20 mislabeled verdicts this year, mainly in the DDL category (*DeviceDriversLinux*).

Weaknesses. Mopsa can only prove programs correct for now, and is currently unable to provide counterexamples otherwise. We plan to leverage the recent work of Milanese and Miné [10] to address this issue. Our SV-Comp driver currently tries a fixed sequence of increasingly precise configurations. We plan to reuse information between the different analyses of the sequence, and automatically adapt the options of Mopsa to the analyzed program (similar to Goblint's autotuning [21]). Our analysis is not competitive enough in the tracks besides *SoftwareSystems*: we plan to add new array abstractions as well as a partitioning mechanism. We also noted that Mopsa is imprecise on `longjmp`, following the addition of recent benchmarks from Schwarz et al. [23] to SV-Comp.

Methodology. We finish this section by explaining how we worked to improve Mopsa this year. We focused on the most important subcategories of *SoftwareSystems*. We encountered a few runtime errors in our analysis: we used automated testcase reduction [18] to pinpoint these issues and fix them. We investigated several timeouts in the *DeviceDriversLinux-Large* (DDLL) category, by using standard profiling tools (such as `perf`), but also by profiling which parts of a given program took long to analyze through custom plugins. The rest of the work consisted in performing manual inspection of some tasks to see how we could improve precision. We started by choosing tasks solved by competing tools relying on similar approaches, starting from Goblint [20, 21, 19].

4 Software Project and Contributors

Mopsa is available on Gitlab [17], and released under an GNU LGPL v3 license. Mopsa was originally developed at LIP6, Sorbonne Université following an ERC Consolidator Grant award to Antoine Miné. Mopsa is now additionally developed in other places, including Inria, ENS Airbus, and Nomadic Labs. The people who improved Mopsa for SV-Comp 2024 are the authors of this paper.

Data-Availability Statement. The exact version of Mopsa and the driver that participated in SV-Comp 2024 are available as a Zenodo archive [13].

Bibliography

[1] Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: SAS, Lecture Notes in Computer Science, vol. 4134, pp. 221–239, Springer (2006)

[2] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS, LNCS , Springer (2024)

[3] Beyer, D., Spiessl, M.: Metaval: Witness validation via verification. In: CAV (2), Lecture Notes in Computer Science, vol. 12225, pp. 165–177, Springer (2020)

[4] Boillot, J., Feret, J.: Symbolic transformation of expressions in modular arithmetic. In: SAS, Lecture Notes in Computer Science, vol. 14284, pp. 84–113, Springer (2023)

[5] Chalupa, M., Richter, C.: Bubaak-SpLit: Split what you cannot verify (competition contribution). In: Proc. TACAS, LNCS , Springer (2024)

[6] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)

[7] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the Astrée static analyzer. In: ASIAN, pp. 272–300 (2006)

[8] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, pp. 1–18 (2019)

[9] Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in C programs. In: SAS, pp. 243–262 (2018)

[10] Milanese, M., Miné, A.: Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In: VMCAI, Springer (2024)

[11] Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES (2006)

[12] Monat, R.: Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries. Ph.D. thesis, Sorbonne Université, France (2021)

[13] Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa at sv-comp 2024 (Nov 2023), `https://doi.org/10.5281/zenodo.10198570`

[14] Monat, R., Ouadjaout, A., Miné, A.: Value and allocation sensitivity in static python analyses. In: SOAP@PLDI, pp. 8–13, ACM (2020)

[15] Monat, R., Ouadjaout, A., Miné, A.: Mopsa-c: Modular domains and relational abstract interpretation for C programs (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 565–570, Springer (2023)

[16] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, pp. 223–247 (2020)

[17] Ouadjaout, A., Monat, R., Miné, A., Journault, M.: Mopsa (2022), URL `https://gitlab.com/mopsa/mopsa-analyzer`

[18] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: PLDI, pp. 335–346, ACM (2012)

[19] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination (competition contribution). In: Proc. TACAS, LNCS , Springer (2024)

[20] 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: TACAS (2021)

[21] Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: Goblint: Autotuning thread-modular abstract interpretation (competition contribution). In: Proc. TACAS (2), LNCS , Springer (2023)

[22] Saan, S., Schwarz, M., Erhard, J., Seidl, H., Tilscher, S., Vojdani, V.: Correctness witness validation by abstract interpretation. In: VCMAI, LNCS , Springer (2024)

[23] Schwarz, M., Erhard, J., Vojdani, V., Saan, S., Seidl, H.: When long jumps fall short: Control-flow tracking and misuse detection for non-local jumps in C. In: SOAP@PLDI, pp. 20–26, ACM (2023)

PROTON: PRObes for Termination Or Not (Competition Contribution)

Ravindra Metta[1,2](✉)*, Hrishikesh Karmarkar[1], Kumar Madhukar[3], R. Venkatesh[1], and Supratik Chakraborty[4]

[1] TCS Research, Tata Consultancy Services, Pune, India
{r.metta,hrishi.karmarkar,r.venky}@tcs.com
[2] School of CIT, Technical University of Munich, Munich, Germany
[3] Dept. of Computer Science and Engineering, IIT Delhi, New Delhi, India
madhukar@cse.iitd.ac.in
[4] Dept. of Computer Science and Engineering, IIT Bombay, Bombay, India
supratik@cse.iitb.ac.in

Abstract. PROTON is a tool to check whether a given C program has a non-terminating behaviour or not. It is built around the C Bounded Model Checker (CBMC). CBMC cannot prove non-termination directly, as all non-terminating runs are unbounded. PROTON annotates the loops in a given program with assertions that check for a recurrent program state. Violation of such an assertion shows the existence of a recurrent state and thereby proves non-termination. PROTON also transforms the violating trace returned by CBMC into a non-termination witness for the program.

1 Introduction

Given a program P for which we want to check termination under all inputs, a checker should either provide a witness for non-termination of P, or give a *correct* verdict that P always terminates. For termination checking, PROTON reuses the high confidence, but unsound, technique used in VeriFuzz 1.4 [9]. For proving non-termination, PROTON implements a novel sound technique that attempts to discover recurrent states inside loops. A recurrent state (RS) is a program state at the head of a loop such that (1) RS entails the loop guard; (2) RS is reachable from an initial state in some valid program execution and (3) RS is reachable from itself after the loop body is executed. This notion of an RS is a strengthening of the recurrent set definition proposed in [5].
Consider the example program P in Listing 1.1, adapted from the SV-COMP benchmark `WhileSingle.c`. This program does not terminate for any nondet value ≤ 3. For example, if nondet value on Line 1 is 3, then the if-condition on Line 3 gets evaluated to false and hence the value of i remains unchanged, causing the loop to run infinitely. PROTON works in three main phases, as described below.

[*] Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 393–398, 2024.
https://doi.org/10.1007/978-3-031-57256-2_27

Listing 1.1. Program P

```
i = __VERIFIER_nondet_int();
while (i < 10) {
   if (i != 3) {
      i = i+1;
   }
}
```

Listing 1.2. Program P'

```
1 i = __VERIFIER_nondet_int();
2 bool pStored0 = false;
3 while (i < 10) {
4 bool flag = __VERIFIER_nondet_bool();
5 static int oi; if(pStored0)
6 {__CPROVER_assert(!(oi==i),"RSF");}
7 if(flag){oi=i;pStored0=true;}
8 { if (i != 3) { i = i+1; }}}
```

Phase 1 - Program Instrumentation: PROTON instruments each loop in P with a __CPROVER_assert to check for a recurrent state. This is illustrated in Listing 1.2. PROTON first parses P using various Clang/LLVM APIs and collects the set of all program variables visible in the scope of each loop L_k in P. Following this, PROTON instruments each L_k as follows:-

- A boolean variable *pStoredk* is introduced just before the loop-guard of L_k and initialized to *false* (Line 2 of Listing 1.2).
- Another boolean variable *flag* is added inside the loop, immediately past the guard condition, which is nondeterministically initialized (Line 3).
- For each variable i, visible in the scope of L_k, a corresponding static variable *oi*; i.e. i prefixed with o is added, which tracks the "old" value of i (Line 5).
- An assertion that the "old" state of P never repeats in any later iteration of L_k (lines 5 and 6) is added.
- If *flag* is *true*, then the program state is stored as shown on line 7, and pStoredk is set to *true*.
- Lastly, PROTON emits the loop body as is, but enclosed in braces (Line 8).

The above instrumentation ensures that the assertion gets checked (due to the if-condition on Line 5), in every iteration after the one in which the state is stored, as pStoredk is set to *true* after this if-statement. So, in the very first iteration in which the program state is stored, the assertion is not invoked. This encoding allows a bounded model checker like CBMC [3,4] to check if the program state stored during a non-deterministically chosen iteration of L_k recurs during any subsequent iteration of L_k, subject to the loop iteration bound used for checking.

Phase 2 - Bounded Model Checking for recurrent states: After instrumenting P, PROTON iteratively invokes CBMC for different unwind bounds until a pre-configured max unwind bound (empirically chosen to be 1000, for SV-COMP 2024) for a pre-configured time limit (set to 2 minutes for SV-COMP 2024). If the recurrent state assertion ever gets violated, it proves the presence of a recurrent state and hence non-termination. When this happens, CBMC generates a corresponding counterexample trace. During Phase 1 described above, PROTON does additional instrumentation (not shown in Listing 1.2 for want of space) to help generate a corresponding non-termination witness in the graphml format. If the recurrent state assertion does not get violated until the max bound or if it times out, then PROTON moves to Phase 3, described below.

Phase 3 - Value-Bounded Termination Check In this phase, entered only if PROTON could not find a non-termination witness in Phase 2, PROTON invokes the termination check of VeriFuzz 1.4 [9], which is reimplemented in PROTON, for a pre-configured time limit (set to 2 minutes for SV-COMP 2024).

2 Software Architecture

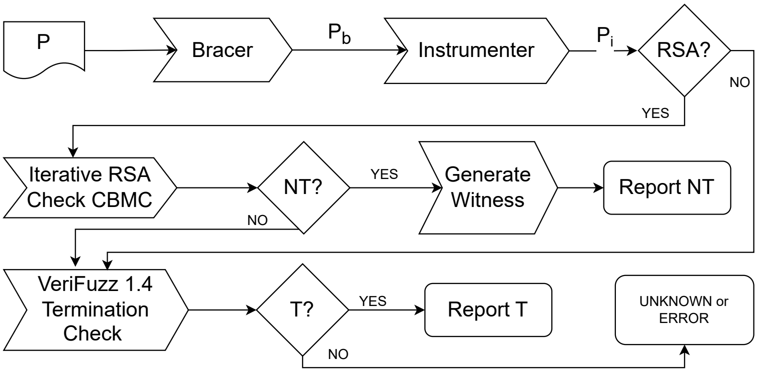


Fig. 1. PROTON architecture

Currently PROTON checks only termination and non-termination of programs. Figure 1 shows the tool flow of PROTON. Given an input program P, PROTON first invokes *Bracer*, which simply adds curly braces around all loop bodies in P to produce P_b. PROTON then invokes *Instrumenter* on P_b, which instruments P_b, as described in Phase 1, to produce P_i. Sometimes, due to internal errors, the Instrumenter may not be able to instrument the program. So, PROTON then checks if P_i has at least one Recurrent State Assertion (RSA). If so, it performs the non-termination check as described in Phase 2 above and generates a corresponding witness if it detects non-termination.

If P_i does not contain any RSA, or if this non-termination check is unsuccessful, PROTON then invokes confidence based termination check on P, mentioned in Phase 3 above. If this termination check concludes that P terminates, PROTON reports P to be terminating. Else, PROTON reports either UNKNOWN (when both checks failed) or ERROR (if there is any internal error).

PROTON is built using CBMC v5.95.0 [3] with Z3 4.12.2 [10] and Glucose Syrup [1] as the backend SMT and SAT solvers respectively. The Bracer and Instrumenter were implemented in C++ using the clang-14 and llvm-14 libraries. The tool flow is implemented in a bash shell script.

3 Strengths and Weaknesses

Here we present our analysis of strengths and weaknesses of PROTON's non-termination check, as that is the main novelty of PROTON.

Strengths: Of the 818 Non-termination tasks in SV-COMP 2024 [2], PROTON correctly solved 627, out of which 501 witnesses could be successfully validated. There are 18 tasks, all from *systemc* directory, such as `token_ring.10.cil-1.c` and `transmitter.08.cil.c`, for which PROTON was the only tool in the competition that could identify them as non-terminating. These programs have several function calls and while loops, with around 1000 lines of code. However, none of the corresponding witnesses generated by PROTON could be validated. Further, the total time taken by PROTON for the 818 tasks is 37000 seconds, which is well below other top tools such as ULTIMATE Automizer [6] (correct solved: 548, confirmed: 537, time 100000 seconds) and 2LS [8] (correctly solved: 685, confirmed: 484, time: 52000 seconds). This shows that PROTON's approach of checking for recurrent sets at shallow loop unwinding depths is both effective and efficient.

Weaknesses: As mentioned above in *Phase 2 - Bounded Model Checking for recurrent states*, PROTON checks for a recurrent state only up to an unwind to 1000 in SV-COMP 2024. Therefore, it cannot handle cases where recurrent-states occur beyond this unwind bound, such as in `cohencu1-both-nt.c`, where the first recurrent state occurs after 2^{32} iterations. Another technical limitation of our approach is the inability to handle arrays, as it requires instrumenting each array element, which does not scale for large arrays. So, we currently ignore loops that modify arrays, and hence could not solve cases such as `Arrays02-EquivalentConstantIndices.c`. Also, since our *instrumenter* does not handle recursion currently, PROTON could not identify benchmarks like `RecursiveNonterminating-1.c` as non-terminating. Lastly, due to a bug in the *instrumenter*, one pointer was nottracked by our insutrmenter, leading to PROTON incorrectly reporting the program as non-terminating.

4 Tool Configuration and Setup

PROTON comes with an MIT license, and is available at [7,11]. To install and run the tool, follow the instructions in the file named README.txt. The benchexec tool-info module is PROTON.py and the benchmark definition file is PROTON.xml. A sample run command is: `PROTON --graphml-witness witness.graphml --propertyfile termination.prp --64 example.c`.
PROTON opted to participate only in the Termination category in SV-COMP 2024.

5 Software Project and Contributors

PROTON is developed and maintained by the authors at IIT Delhi, TCS Research, and IIT Bombay. We thank everyone who has contributed to the development of PROTON, Clang and LLVM Infrastructure, CBMC, Glucose Syrup, and Z3.

6 Data-Availability Statement

PROTON is publicly available at https://github.com/kumarmadhukar/term. The SV-COMP 2024 competition version of PROTON is available at Zenodo: https://doi.org/10.5281/zenodo.10185252. For any queries, please contact the authors.

References

1. Audemard, G., Simon, L.: On the glucose SAT solver. Int. J. Artif. Intell. Tools pp. 1840001:1–1840001:25 (2018). https://doi.org/10.1142/S0218213018400018
2. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
3. C Bounded Model Checker. https://github.com/diffblue/cbmc
4. Clarke E., Kroening D., L.F.: A tool for checking ansi-c programs. In: TACAS. pp. 168–176 (2004). https://doi.org/10.1007/978-3-540-24730-2_15
5. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL. pp. 147–158. ACM (2008)
6. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schussele, F., Podelski, A.: Ultimate Automizer 2024 (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
7. Karmarkar, H., Madhukar, K., Metta, R.: Proton sv-comp 2024 competition version (Nov 2023). https://doi.org/10.5281/zenodo.10185252, https://doi.org/10.5281/zenodo.10185252
8. Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and loop unwinding (competition contribution). In: Proc. TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_31
9. Metta, R., Yeduru, P., Karmarkar, H., Medicherla, R.K.: Verifuzz 1.4: Checking for (non-)termination (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 594–599. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_42
10. Moura, L.M.d., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
11. Proton github. https://github.com/kumarmadhukar/term (2023), accessed: 22-Dec-2023

SWAT: Modular Dynamic Symbolic Execution for Java Applications using Dynamic Instrumentation (Competition Contribution)

Nils Loose(✉)[⋆], Felix Mächtle, Florian Sieck, and Thomas Eisenbarth

Institute for IT Security, University of Lübeck, Lübeck, Germany
{n.loose,f.maechtle,florian.sieck,thomas.eisenbarth}@uni-luebeck.de

Abstract. SWAT is a novel dynamic symbolic execution engine for Java applications utilizing dynamic instrumentation. SWAT's unique modular design facilitates flexible communication between its symbolic explorer and executor using HTTP endpoints, thus enhancing adaptability to diverse application scenarios. The symbolic executor's ability to attach to Java applications enables efficient constraint generation and path exploration. SWAT employs JavaSMT for constraint generation and ASM for bytecode instrumentation, ensuring robust performance. SWAT's efficacy is evaluated in the Java Track of SV-COMP 2024, achieving fourth place.

Keywords: Dynamic Symbolic Execution · Java · Dynamic Instrumentation

1 Verification Approach

The symbolic execution of a System-under-Test (SuT) is a well-known verification technique where the state space is systematically explored by using constraint modeling to compute new valid inputs for the SuT. Dynamic Symbolic Execution (DSE), in particular, has shown recent successes with JDart [15] winning the Java track of SV-COMP 2022 [4] as the first DSE tool and GDart [16] achieving second place in 2023 [5]. Generally, DSE utilizes a symbolic executor to evaluate a SuT by observing the concrete execution for a given assignment of the symbolic variables. Constraints are recorded during execution, reflecting all operations involving symbolic variables. In particular, each branching point that depends on a symbolic variable is modeled as a path constraint. After the execution terminates, the symbolic explorer can select a previously unexplored branch. Given the recorded constraints, an SMT solver is used to determine whether a model for the symbolic variables under the given constraints exists. If so, a concrete instantiation for each value can be obtained to drive execution to previously unexplored regions of the state space. By repeating this process, the state space of the SuT can be systematically explored.

JDart, the winning candidate from 2022, relies on Java Pathfinder (JPF) [9] and its implementation of the Java Virtual Machine (JVM) for symbolic

[⋆] Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 399–405, 2024.
https://doi.org/10.1007/978-3-031-57256-2_28

execution [15]. While the JPF-JVM offers robust analysis tools, it limits JDart's applicability and causes a significant overhead. Coastal [8], on the other hand, relies on a standard JVM. The symbolic execution is realized using dynamic instrumentation. While Coastal provides a loosely coupled design between the symbolic execution engine and the symbolic explorer, both components are still located in the same Java program used as the driver to start and execute the SuT symbolically. GDart extends the notion of modularity introduced by Coastal with a fully decoupled explorer and executor that communicate using a custom protocol [16]. SWAT offers a fully modular design comparable to GDart while relying on HTTP endpoints for communication between the symbolic explorer and executor. In addition, GDart relies on the GraalVM [20] for driving symbolic execution while SWAT attaches to the SuT, thus enabling symbolic execution inside native JVM implementations.

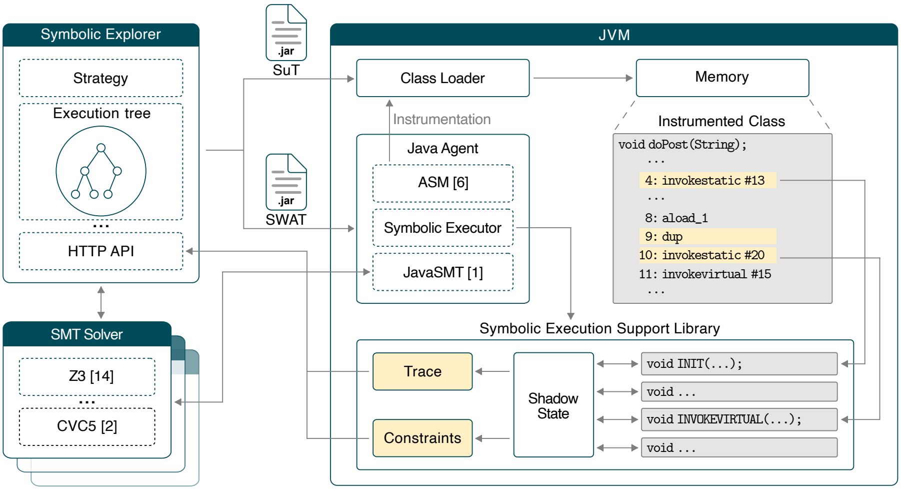
2 System Architecture

SWAT's decoupled design allows for a persistent symbolic explorer that receives relevant information from instances of the symbolic executor. The executor observes the SuT by attaching to the JVM and adding symbolic capabilities using dynamic instrumentation. An overview of the design and interaction between the different components is shown in Figure 1 and described in more detail below.

Symbolic executor The executor attaches to the JVM running the SuT via the Java agent interface and dynamically instruments each class at load time with additional (non-interfering) instructions that dynamically build and manage a symbolic shadow state responsible for maintaining the symbolic constraints. This leads to a symbolic executor that does not actively drive symbolic execution and instead records relevant information during normal execution. SWAT utilizes the ASM framework [6] for bytecode manipulation via the `Java.lang.instrument` API [17]. Historically, this part builds on CATG [19] as a basis for dynamic symbolic execution. Significant parts of CATG are reworked, and the language support is lifted to Java 17, including most of its features. The symbolic shadow state and the symbolic constraint handling are extended and wholly rewritten to utilize the API offered by JavaSMT [1] as an abstraction layer between constraint generation and the solver. The symbolic scope and variables, as well as the entry and exit points for symbolic tracking, are fully configurable, allowing for broad applicability of the system. The instrumentation logic is also modularized, allowing us to easily extend SWAT to various use cases, such as the SV-COMP.

When the execution of the SuT reaches a symbolic entry point, the symbolic executor records control-flow information as well as the constraints, and after the exit point has been reached, both the trace and the corresponding constraints are sent to the symbolic explorer using HTTP requests. Constraints are serialized using the SMT-LIB v2 [3] format.

Symbolic explorer The explorer, written in Python using the FastAPI [18] web framework, receives the language agnostic trace and constraint information. These are stored in a binary execution tree. The tree can be searched using a



```
void doPost(String);
  ...
  4: invokestatic #13
  ...
  8: aload_1
  9: dup
 10: invokestatic #20
 11: invokevirtual #15
  ...
```

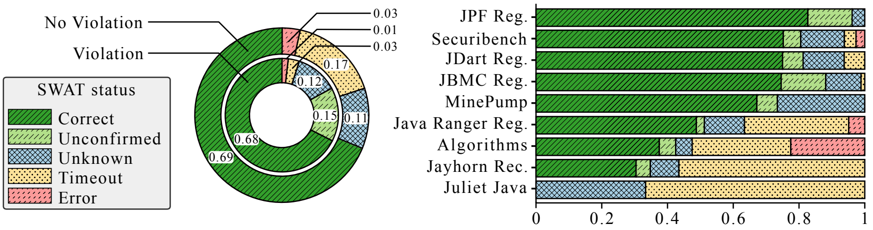
```
void INIT(...);
void ...
void INVOKEVIRTUAL(...);
void ...
```

Fig. 1. Schematic overview of SWAT's modular architecture.

configurable and modularized strategy to select unexplored branches. To obtain new inputs, the constraints are sent to Z3 [14]. The inputs can either be made available to external drivers, such as fuzzers, using an endpoint or, in the case of SV-COMP, are directly used to initiate a new concrete execution. This structure makes SWAT widely applicable and even enables straightforward testing of web services, for example, where each controller is configured as the entry and exit point and user-controlled values are tracked symbolically. This allows the same JVM to keep running in between symbolic runs and even allows for multiple (non-interfering) executions in parallel.

3 Evaluation

In the first participation on the Java category of SV-COMP 2024, SWAT reached fourth place with 566 out of 828 total points while MLB [7], the winning candidate, scored 676 points. Overall, SWAT correctly classified 68% of test cases. Figure 2a visualizes the result distribution for test cases containing violations and those without. The number of correctly classified cases is similar for both groups. However, due to issues during witness generation, several correctly identified violations did not produce correct witnesses. Hence, without considering the witnesses, the number of identified violations rises significantly from 68% to 83%. Generally, DSE frameworks are expected to identify violations (one concrete path) better than proving their absence (full state-space exploration). This is also reflected in the distribution of timeouts, with a five times increase between violation and safe test cases. Roughly 10% of test cases are labeled as unknown by SWAT. This case comprises several possibilities: Out-of-scope invocations



(a) SWAT results for safe and violation test cases

(b) SWAT results for each subcategory

Fig. 2. SWAT results divided based on the ground truth of each test case (a) and results for each subcategory of the Java category (b).

without a symbolic model, inability to determine satisfiability or unsupported behavior such as uncaught exceptions.

Further dividing the results based on the different subgroups (see Figure 2b) highlights differences in the status distributions. SWAT generally performs well for regression test categories, as these usually test specific functionalities, resulting in small programs that do not lead to a state space explosion. With the increasing complexity of test suites, the number of timeouts is expected to rise. The Jayhorn recursive test cases cause many timeouts as SWAT currently does not support advanced recursion handling. Lastly, SWAT is holistically unable to solve the test cases provided by the Juliet test suite due to the extensive use of socket connections, which require explicit mocking.

While the results demonstrate the impact of state space explosion on the performance of DSE engines, generally, the results highlight the potential of SWAT, especially when considering the overhead incurred by starting a new JVM instance for each run of the test case. In SWAT's current form, this causes instrumentation at each iteration whereas test cases that can be re-initiated without restarting the JVM would result in significantly faster executions.

4 Software Project

SWAT is developed by the Institute for IT Security at the University of Lübeck and published on GitHub [12] under the BSD 2-Clause. Installation instructions, documentation, and examples can be found on our GitHub Page [11]. Global configuration options chosen for the participation include the exclusive usage of the Z3 [14] solver, a breadth-first search strategy, and an SV-COMP specific driver modules inside the symbolic explorer and executor.

Data-Availability Statement The version of SWAT used for the SV-COMP 2024 Java category is available at Zenodo [13] and on GitHub [10].

5 Acknowledgments

This work has been supported by the Bundesministerium für Bildung und Forschung (BMBF) through the PeT-HMR project.

References

1. Baier, D., Beyer, D., Friedberger, K.: Javasmt 3: Interacting with SMT solvers in java. 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. 195–208. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_9, https://doi.org/10.1007/978-3-030-81688-9_9
2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
3. Barrett, C., Stump, A., Tinelli, C., et al.: The smt-lib standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)
4. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 375–402. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20, https://doi.org/10.1007/978-3-030-99527-0_20
5. Beyer, D.: Competition on software verification and witness validation: Sv-comp 2023. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 495–522. Springer Nature Switzerland, Cham (2023)
6. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems **30**(19) (2002)
7. Bu, L., Liang, Y., Xie, Z., Qian, H., Hu, Y., Yu, Y., Chen, X., Li, X.: Machine learning steered symbolic execution framework for complex software code. Formal Aspects Comput. **33**(3), 301–323 (2021). https://doi.org/10.1007/S00165-021-00538-3, https://doi.org/10.1007/s00165-021-00538-3
8. Geldenhuys, J., Visser, W.: Coastal. https://github.com/DeepseaPlatform/coastal, accessed 12/2023
9. Havelund, K., Pressburger, T.: Model checking JAVA programs using JAVA pathfinder. Int. J. Softw. Tools Technol. Transf. **2**(4), 366–381 (2000). https://doi.org/10.1007/S100090050043, https://doi.org/10.1007/s100090050043
10. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT Competition Version. https://github.com/SWAT-project/SWAT/tree/SV-COMP-Submission-2024, accessed 12/2023
11. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT Documentation. https://swat-project.github.io/docs/, accessed 12/2023

12. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT Repository. `https://github.com/swat-project/swat`, accessed 12/2023
13. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: Swat (2023). https://doi.org/10.5281/zenodo.10418643, `https://doi.org/10.5281/zenodo.10418643`
14. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, `https://doi.org/10.1007/978-3-540-78800-3_24`
15. Mues, M., Howar, F.: Jdart: Dynamic symbolic execution for java bytecode (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 398–402. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_28, `https://doi.org/10.1007/978-3-030-45237-7_28`
16. Mues, M., Howar, F.: Gdart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 435–439. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_27, `https://doi.org/10.1007/978-3-030-99527-0_27`
17. Oracle: Java Instrumentation. `https://docs.oracle.com/en/java/javase/17/docs/api/java.instrument/java/lang/instrument/package-summary.html`, accessed 12/2023
18. Ramírez, S.: FastAPI, `https://github.com/tiangolo/fastapi`, accessed 12/2023
19. Tanno, H., Zhang, X., Hoshino, T., Sen, K.: Tesma and CATG: Automated test generation tools for models of enterprise applications. In: Bertolino, A., Canfora, G., Elbaum, S.G. (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. pp. 717–720. IEEE Computer Society (2015). https://doi.org/10.1109/ICSE.2015.231, `https://doi.org/10.1109/ICSE.2015.231`
20. 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: Hosking, A.L., Eugster, P.T., Hirschfeld, R. (eds.) ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013. pp. 187–204. ACM (2013). https://doi.org/10.1145/2509578.2509581, `https://doi.org/10.1145/2509578.2509581`

Symbiotic 10: Lazy Memory Initialization and Compact Symbolic Execution*
(Competition Contribution)

Martin Jonáš[1](✉), Kristián Kumor[1], Jakub Novák[1], Jindřich Sedláček[1], Marek Trtík[1], Lukáš Zaoral[2], Paulína Ayaziová[1], and Jan Strejček[1]

[1] Masaryk University, Brno, Czech Republic
martin.jonas@mail.muni.cz
[2] Red Hat, Brno, Czech Republic

Abstract. SYMBIOTIC 10 brings four substantial improvements. First, we extended our clone of KLEE called JETKLEE with *lazy memory initialization*. With this extension, JETKLEE can symbolically execute a function without knowing its context. In SV-COMP, we use it to handle `extern` variables. Second, we have implemented the technique called *compact symbolic execution* to SLOWBEAST. Third, we have implemented a non-trivial *may-happen-in-parallel* analysis, which improves slicing of parallel programs. Finally, we have implemented support for violation witnesses in the new *witness format 2.0*.

1 Verification Approach

Just like previous versions, SYMBIOTIC 10 relies on a combination of static analysis, code instrumentation, and several flavors of *symbolic execution* (SE) [8]. It employs two symbolic executors: SLOWBEAST and our fork of KLEE [2] called JETKLEE. SLOWBEAST implements standard (forward) SE, backward SE with loop folding [5], and compact SE [13]. JETKLEE implements standard SE.

The rest of the section describes the precise workflow for various types of properties and discusses the differences between SYMBIOTIC 10 and SYMBIOTIC 9.1, which is the version that competed in SV-COMP 2023.

Verification of the Property `unreach-call` For this property, SYMBIOTIC 10 performs slicing of the given program to remove the parts that have no influence on reaching the target function, and executes sequential portfolio of the following engines. Each of the engines is executed for the given number of seconds. The execution can be shorter if the engine decides or fails to decide, e.g., due to an unsupported feature of the input program like threads or symbolic floats.

1. Forward symbolic execution by JETKLEE for 333 seconds. JETKLEE is efficient industrial-strength symbolic executor and most of the solved benchmarks are solved by JETKLEE.

[*] This work has been supported by the Czech Science Foundation grant GA23-06506S.
[✉] Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 406–411, 2024.
https://doi.org/10.1007/978-3-031-57256-2_29

2. Compact symbolic execution (CSE) [13] by Slowbeast for 60 seconds. In most cases, CSE either finishes quickly or brings no benefit compared to standard forward symbolic execution.
3. Backward symbolic execution with loop folding (BSELF) [5] by Slowbeast without time limit.
4. If BSELF fails, we perform forward symbolic execution by Slowbeast without time limit. The reason for this is that Slowbeast has better support for floating point arithmetic and threads than JetKlee.

If an error is found by any of the engines, it is replayed on the unsliced code. If the replay succeeds, we generate a violation witness. If the program is decided safe by BSELF, we generate a correctness witness containing the generated invariants. The other engines do not support invariant generation, therefore if the program is decided safe by any other of the engines, we generate a trivial correctness witness.

Verification of Other Properties For other properties, Symbiotic 10 uses the same workflow as Symbiotic 9 [4]. In a nutshell, we identify program instructions that can violate the property, instrument the program with code that dynamically checks the property violation before each of the identified instructions, slice the program, and run either JetKlee or Slowbeast.

Compact Symbolic Execution We extended Slowbeast with *compact symbolic execution* (CSE) [13]. CSE analyzes each looping path of the execution and tries to summarize it by a quantified formula that describes the effect of κ iterations of that cyclic path, where κ is a free variable. For example, if we apply compact symbolic execution to the loop

```
while (i < n) { if (A[i] = 0) { break; }; i += 2; },
```

the path condition will be augmented by the quantified formula

$$\kappa \geq 0 \quad \wedge \quad \forall \tau. \, (0 \leq \tau < \kappa \rightarrow (i + 2\tau < n \wedge A[i + 2\tau] \neq 0)) \, .$$

This allows symbolic execution to fully explore some programs with unbounded loops and find deep counterexamples. However, it works only for looping paths of specific form and requires potentially expensive quantified SMT reasoning.

Lazy Memory Initialization We extended JetKlee with *lazy memory initialization*, which constructs symbolic memory objects lazily during the first access to that object, not during its initialization. This allows isolated symbolic execution of functions without knowing their arguments and calling context. As all programs in SV-COMP start with the `main` function and there is no need to analyze an isolated function, we use this feature in the competition only to support externally defined variables. Note that this cannot be achieved by merely making the externally defined variable symbolic, as it can be a pointer to external memory, which needs to be properly initialized. For this reason, externally defined variables were not supported by the previous version of Symbiotic.

Table 1. The comparison of Symbiotic 9.1 and Symbiotic 10 on the intersection of benchmarks from SV-COMP 2023 and SV-COMP 2024. The table is computed from the official results of SV-COMP 2023 and SV-COMP 2024.

Property	Benchmarks	Both solved	Only 10 solved	Only 9.1 solved
`no-data-race`	783	0	0	0
`no-overflow`	7502	442	4102	1
`termination`	1809	1220	10	31
`unreach-call`	9537	3577	116	225
`valid-memcleanup`	61	35	0	0
`valid-memsafety`	4113	416	1427	34

May-Happen-in-Parallel Analysis We improved slicing of parallel programs by employing a static *may-happen-in-parallel* analysis [11], which overapproximates the set of pairs of program locations that can happen in parallel in different threads. Previously, Symbiotic assumed that all possible pairs of instructions can happen in parallel, which reduced effectivity of slicing. The implementation currently does not consider thread synchronization. For more details, see the bachelor's thesis about the implementation [12]. In the future, we want to use this analysis also for proving some `no-data-race` properties.

Other Changes All external dependencies of Symbiotic 10 have been updated to newer versions and all parts of Symbiotic 10 have been ported to llvm 14. Notably, this concerns JetKlee, into which we merged most of the upstream changes from the base Klee (more than 300 commits).

We extended JetKlee with support for generating yaml-based violation witnesses in witness format 2.0[3]. Slowbeast still supports only the older witness format 1.0 based on GraphML.

We also fixed incorrect overflow checking of 64-bit integers and incorrect modeling of `fscanf` for the purposes of static analysis and instrumentation. Due to these problems, Symbiotic 9.1 did not support any of `*- uliet` benchmarks, which are now fully supported.

Unlike the previous versions of Symbiotic, Symbiotic 10 does not employ Predator [6] as a static analyzer. This is due to technical difficulties during porting our version of Predator to llvm 14. This is a temporary solution and we plan include Predator in the future versions of Symbiotic.

2 Strengths and Weaknesses

Standard forward symbolic execution suffers from path explosion and is unable to fully analyze programs with unbounded loops. Backward symbolic execution with loop folding and compact symbolic execution can finish analysis even for

[3] https://gitlab.com/sosy-lab/benchmarking/sv-witnesses

some programs with unbounded loops, yet they still suffer from path explosion and will time out on programs with a large number of branching paths.

The results of SV-COMP 2024 show that the combination of static analysis, instrumentation, program slicing, and several variants of symbolic execution are efficient in practice, in particular for bug hunting. The static analyses are often able to prove that some parts of the code are correct or do not influence the property. These parts of the code then can be removed by slicing. This partly mitigates the scalability problem caused by path explosion.

Results of Symbiotic 10 in SV-COMP 2024 Symbiotic 10 participated in all categories of SV-COMP 2024 for C programs. It won silver medals in categories *MemSafety* and *FalsificationOverall* [1]. Symbiotic 10 produced 19 wrong answers; most of these are caused by imprecise modeling of the system functions `setlocale` and `getopt_long`. They are not fundamental problems of the approach and will be fixed.

Table 1 compares the results of Symbiotic 9.1 in SV-COMP 2023 and Symbiotic 10 in SV-COMP 2024 on the benchmarks that were used in both years. Symbiotic 10 was able to correctly solve 5655 benchmarks that were not solved by Symbiotic 9.1. From these, 5366 benchmarks (3990 `no-overflow` + 1376 `valid-memsafety`) are from subcategories `*- uliet`, which the previous version of Symbiotic did not support. Unfortunately, 147 of the previously decided benchmarks from `ConcurrencySafety-main` with property `unreach-call` were not decided by Symbiotic 10 due to a bug in our version of Slowbeast. Additionally, 31 of previously decided benchmarks (16 in `Memsafety-Heap` and 15 in `Memsafety-LinkedLists`) were not decided by Symbiotic 10 due to exclusion of Predator. If Predator had not been excluded or the wrong results had been fixed, Symbiotic 10 would have won the *MemSafety* category.

3 Software Architecture, Usage, and Contributors

All components of Symbiotic 10 use llvm 14 [9] for the intermediate representation. To obtain the llvm bitcode from the verified C program, Symbiotic relies on clang. Slicer and instrumentation module are written in C++ and rely on the library DG [3]. JetKlee is implemented in C++ and Slowbeast [14] is written in Python. Both symbolic executors use Z3 [10] as the smt solver. Control scripts are written in Python. All the components and external dependencies have permissive open-source licenses.

Binary form of Symbiotic 10 is available Zenodo [7], source code is available from https://github.com/staticafi/symbiotic under the tag `svcomp24`. You can run Symbiotic with

```
bin/symbiotic --sv-comp --prp <prpfile> [--32] <source>.
```

For details, see the file `README.md` in the mentioned repository.

Symbiotic 10 has been developed at the Faculty of Informatics of Masaryk University by the authors of this paper under the supervision of Jan Strejček.

References

1. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
2. 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
3. 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
4. Chalupa, M., Mihalkovič, V., Řechtáčková, A., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding - (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) TACAS 2022. Lecture Notes in Computer Science, vol. 13244, pp. 462–467. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32, https://doi.org/10.1007/978-3-030-99527-0_32
5. 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, https://doi.org/10.1007/978-3-030-88806-0_3
6. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer (2011), https://doi.org/10.1007/978-3-642-36742-7_49
7. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Shandilya, S., Trtík, M., Zaoral, L., Strejček, J.: Symbiotic 10: Submission to SV-COMP 2024 (Nov 2023). https://doi.org/10.5281/zenodo.10202594
8. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976)
9. 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
10. 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
11. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An Efficient Algorithm for Computing *MHP* Information for Concurrent Java Programs. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC / SIGSOFT FSE 1999. Lecture Notes in Computer Science, vol. 1687, pp. 338–354. Springer (1999). https://doi.org/10.1007/3-540-48166-4_21, https://doi.org/10.1007/3-540-48166-4_21
12. Sedláček, J.: May-Happen-in-Parallel Analysis for Slicing of Parallel Programs. Bachelor's thesis, Masaryk University (2024), https://is.muni.cz/th/he6cd/
13. Slaby, J., Strejček, J., Trtík, M.: Compact symbolic execution. In: Hung, D.V., Ogawa, M. (eds.) ATVA 2013. Lecture Notes in Computer Science, vol. 8172, pp. 193–207. Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_15, https://doi.org/10.1007/978-3-319-02444-8_15
14. Slowbeast repository. https://gitlab.com/mchalupa/slowbeast (2021)

Theta: Abstraction Based Techniques for Verifying Concurrency (Competition Contribution)

Levente Bajczi(✉) ⋆, Csanád Telbisz, Márk Somorjai, Zsófia Ádám, Mihály Dobos-Kovács, Dániel Szekeres, Milán Mondok, and Vince Molnár

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
bajczi@mit.bme.hu

Abstract. Theta is a model checking framework, with a strong emphasis on effectively handling concurrency in software using abstraction refinement algorithms. In SV-COMP 2024, we use 1) an abstraction-aware partial order reduction; 2) a dynamic statement reduction technique; and 3) enhanced support for call stacks to handle recursive programs. We integrate these techniques in an improved architecture with inherent support for portfolio-based verification using dynamic algorithm selection, with a diverse selection of supported SMT solvers as well. In this paper we detail the advances of Theta regarding concurrent and recursive software support.

Funding. This research was partially funded by the ÚNKP-23-{2,3}-I New National Excellence Program; Project no. 2019-1.3.1-KK-2019-00004 (implemented with the support provided from the NRDI Fund of Hungary under the 2019-1.3.1-KK funding scheme); and the Doctoral Excellence Fellowship Programme (funded by the NRDI Fund of Hungary and the BME University).

1 Verification Approach

Theta [15,8] first competed at SV-COMP as a standalone tool in 2022, with initial support for some multi-threaded tasks using a crude version of a partial order reduction (POR) algorithm [2], and no practical support for recursion.

This year, we implemented a novel *abstraction-based partial order reduction* algorithm [13] that enables Theta to solve significantly more tasks compared to previous SV-COMPs, especially in the ReachSafety category. Our algorithm considers two program statements independent even if they use the same shared variable when the current abstraction has no information about this variable. For example, the statements $y = x$ and $x = 1$ are classically considered dependent

⋆ Jury member representing Theta at SV-COMP 2024.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 412–417, 2024.
https://doi.org/10.1007/978-3-031-57256-2_30

due to x. However, if the current abstraction has no information about x (e.g., we only track the predicates $y > 0$ and $z = y$), we consider these statements independent as they are commutative in the abstract state space. We extend a static source-set based POR algorithm [1] with our abstraction-based technique.

A novel statement reduction algorithm has also been developed for the verification of concurrent programs [14]. Our algorithm is similar to program slicing and cone-of-influence techniques in the sense that it detects and removes statements that do not affect the verified property [5,9]. However, our approach analyzes the current local states of concurrent threads and data-flow between threads to dynamically detect irrelevant statements that do not affect the verified property in the current thread interleaving. The evaluation of such statements is skipped which considerably reduces the time cost of successor state calculation during state space exploration. Our technique is especially useful for concurrent tasks where the reducing capability of existing slicing and cone-of-influence techniques is limited due to the many possible interleavings of threads: our algorithm can skip (sub-)statements in certain contexts even if these statements cannot be removed generally (that is, statements that may be important in other thread interleavings). Our algorithm is different from dynamic program slicing [9] since those techniques do not consider the current interleaving of threads for slicing.

THETA has been extended with enhanced interprocedural analysis [12]. Previously, all procedures have been inlined at all of their calls before verification. Procedure support was implemented last year, which handles procedures dynamically during verification, using a stack to keep track of calling locations. This year, procedure support is further improved by applying abstraction to location stacks. If an abstract state overapproximates another with the bottom of their stacks abstracted away, then all abstract paths going out from the covered state are present at the covering state until the current procedure returns. Therefore, the top location of the covered state is popped and exploration continues from the outer procedure, eliding unnecessary exploration [12].

The main advantage of handling procedures dynamically is that it allows THETA to verify recursive programs, which was not possible with inlining. Applying abstraction to stacks also enables the verification of some infinitely recursive programs. Additionally, it reduces the size of the abstract state-space and improves THETA's verification performance with predicate abstraction.

2 Software Architecture

Since last year, we opted to keep our initial portfolio-based approach [2], but used a separate process for each configuration, which can easily be killed using signals, as opposed to the thread-based approach of THETA at SV-COMP'22. Furthermore, we created a generic interface that allows easy co-development of portfolios without having to recompile THETA. The architecture of THETA can be seen in Figure 1: THETA parses and transforms the input program into an eXtended CFA, then, based on the configuration in the portfolio, spawns one or more worker THETA processes that perform the verification. The portfolio en-

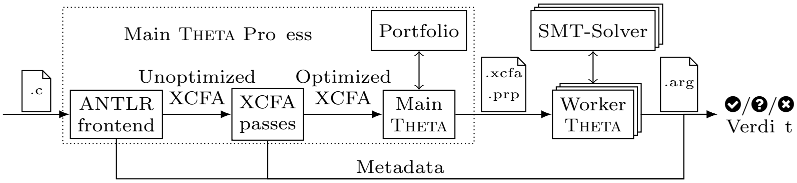


Fig. 1: The architecture of Theta for software verification

gine has been re-written this year to better support pre-compiled configurations written in kotlin instead of kotlin scripts, due to uncovering the dire performance implications of using the script execution engine, which often takes multiple tens of seconds to initialize and start. Dynamic algorithm selection is used to select a suitable configuration for each input task, with several ways of recovering, should the first algorithm take too long or encounter an exception.

Theta uses Z3 [10] versions 4.12.2 and 4.5.0 (the latter is integrated natively via the Java API, while the former is used via SMT-LIB), MathSAT [7] version 5.6.10, CVC5 [4] version 1.0.8 and Princess [11] version 2023-06-19 as SMT solvers under the hood. Compared to previous years, Theta utilizes the new interpolation API of Z3 to support interpolation-dependent refinement strategies with the new solver (removed previously in 4.8.0).

Theta has seen several major updates in its C-frontend for the new tasks introduced to the benchmark repository since SV-COMP'23. The most notable improvements were made around its Antlr-based grammar for lexing and parsing C files, and some further tweaks in the transformation step from the AST to CFA to avoid some wrong verdicts that plagued Theta in earlier SV-COMPs.

3 Strengths and Weaknesses of the Approach

In ReachSafety, Theta achieved a score of 2119 [6]. Although Theta still has known limitations regarding some C elements (e.g., structs), recent technical improvements of the frontend resulted in Theta not giving any wrong results in any categories, except for 3 wrong results in ConcurrencySafety-NoOverflows. Furthermore, Theta achieved a score of 2354 in ConcurrencySafety. To show the negative influence of frontend limitations, we recalculated the score for the participating tools on those ConcurrencySafety tasks that did not end in a frontend failure for Theta. In this alternative scoring Theta would move from the 7th to the 3rd place, highlighting the serious need for further frontend development.

It is worth looking at Theta's performance in the reachability category over the years. As seen in Figure 2, Theta has dipped in performance for last year's installment of SV-COMP (the figure shows only those tasks that have been the same for the last 3 years) from that of SV-COMP'22 [2]. This year we managed to bring the performance back to even outperform Theta'22, especially in the ConcurrencySafety, Sequentialized and Combinations subcategory. However, we did lose a significant number of tasks in some other subcategories, such as Loops.

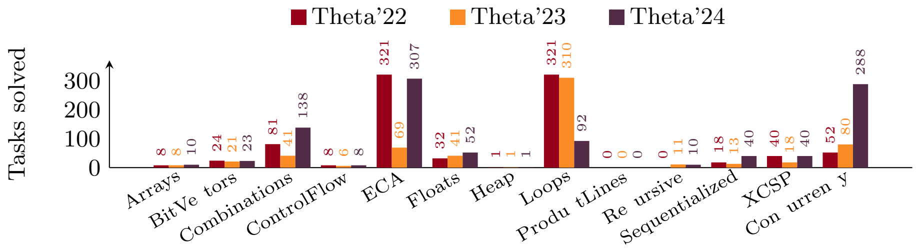


Fig. 2: Overview of successful tasks for THETA per year on common tasks

This can either be a result of a suboptimal portfolio for such tasks, or the result of some tweaks we had to make in order to achieve this year's outstanding 0 incorrect tasks, a feat performed only by 3 other tools. We plan to prioritize the analysis of these cases for future development. We also plan to support categories such as ProductLines and Heap, where we have almost no successful results. This entails supporting structs, function pointers, and heap manipulation.

The novel algorithms implemented in THETA especially helped recursive and multithreaded programs. THETA gained support for recursive programs by implementing the aforementioned stack-based approach, and support for reachability queries in multithreaded programs grew more than 3.5-fold since last year, as seen in Figure 2. In particular, our internal evaluation shows that the size of the state space reduced by the abstraction-based partial order reduction algorithm is 15% smaller on average compared to the case when we use traditional partial order reduction. Our dynamic statement reduction technique can eliminate 22% of statements reducing the time of successor state calculation by up to 60% and the overall verification time by 15% on average depending on the configuration.

4 Tool Setup and Configuration

THETA is vastly configurable [8], and successfully choosing a performance configuration for a verification task at hand can be complicated. For software verification, we recommend using the portfolio (`complex`) in the competition archive [3]: `./theta-start.sh <input> --portfolio COMPLEX`. To minimize the output verbosity and produce a witness, `--loglevel RESULT` and `--witness-only` can be added to the arguments. We also used these options at SV-COMP 2024.

5 Software Project and Data Availability

THETA is a verification framework maintained by the Critical Systems Research Group of the Budapest University of Technology and Economics. The project is available open-source on GitHub[1] under an Apache 2.0 license. The version (`5.0.0`) used in the competition is available at [3].

[1] https://github.com/ftsrg/theta/releases/tag/svcomp24

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Comparing source sets and persistent sets for partial order reduction. Lecture Notes in Computer Science, vol. 10460, pp. 516–536. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_26
2. Ádám, Z., Bajczi, L., Dobos-Kovács, M., Hajdu, Á., Molnár, V.: Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) TACAS 2021. Lecture Notes in Computer Science, vol. 13244, pp. 474–478. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_34
3. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Molnár, V.: Theta - SV-COMP'24 Verifier Archive (Nov 2023). https://doi.org/10.5281/zenodo.10202679
4. Barbosa, H., et al.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. pp. 415–442. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
5. Berezin, S., Campos, S.V.A., Clarke, E.M.: Compositional Reasoning in Model Checking. Lecture Notes in Computer Science, vol. 1536, pp. 81–102. Springer (1997). https://doi.org/10.1007/3-540-49213-5_4
6. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
7. 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
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. Harman, M., Hierons, R.M.: An overview of program slicing. Softw. Focus **2**(3), 85–92 (2001). https://doi.org/10.1002/swf.41
10. 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
11. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. LNCS, vol. 5330, pp. 274–289. Springer (2008). https://doi.org/10.1007/978-3-540-89439-1_20
12. Somorjai, M.: Abstraction-Based Interprocedural Software Verification. Students' scientific association (tdk) submission, Budapest University of Technology and Economics (2023), https://tdk.bme.hu/VIK/DownloadPaper/Absztrakcioalapu-interproceduralis
13. Telbisz, C.: Partial Order Reduction for Abstraction-Based Verification of Concurrent Software in the Theta Framework. Bachelor's thesis, Budapest University of Technology and Economics (2022), https://tdk.bme.hu/VIK/DownloadPaper/Reszleges-rendezes-redukcio-tobbszalu
14. Telbisz, C.: Abstract Data-Flow-Based Statement Reduction for Model Checking Concurrent Software. Students' scientific association (tdk) submission, Budapest University of Technology and Economics (2023), https://tdk.bme.hu/VIK/DownloadPaper/Absztrakt-adatfolyamalapu-utasitasredukcio
15. 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 Automizer and the Abstraction of Bitwise Operations

(Competition Contribution)

Frank Schüssele, Manuel Bentele, Daniel Dietsch, Matthias Heizmann*, Xinyu Jiang, Dominik Klumpp, and Andreas Podelski

University of Freiburg, Freiburg im Breisgau, Germany
schuessf@informatik.uni-freiburg.de

Abstract. The verification of Ultimate Automizer works on an SMT-LIB-based model of a C program. If we choose an SMT-LIB theory of (mathematical) integers, the translation is not precise, because we overapproximate bitwise operations. In this paper we present a translation for bitwise operations that improves the precision of this overapproximation.

1 Verification Approach

Ultimate Automizer (in the following abbreviated as UAutomizer) is a software verifier that implements the trace abstraction approach [6,9]. In trace abstraction, a verification problem is considered as a formal language and decomposed via automata-theoretic methods into smaller verification problems. While verifying a C program, UAutomizer applies trace abstraction to a model of the program that consists of a control-flow graph (CFG) and SMT-LIB formulas that express how the program's data is modified while moving along an edge of the CFG. We obtain this model by first translating the C program into a Boogie [10] program and afterwards translating the Boogie program into the CFG and SMT-LIB formulas. We have two variants of these translations, we call them the *integer-based translation* and the *bitvector-based translation*. The integer-based translation results in a Boogie program over mathematical integers that is later translated to SMT-LIB formulas from the integer theory. The bitvector-based translation results in a Boogie program over bitvectors that is later translated to SMT-LIB formulas from the bitvector theory. The integer-based translation uses modulo operations to make sure that the result of arithmetic operation is in the correct range. It also overapproximates the result of bitwise operations and is hence not very precise. If the trace abstraction-based verification algorithm returns a counterexample that contains an overapproximated operation, UAutomizer does not return the counterexample but *unknown* instead. The bitvector-based translation returns a result that is precise but whose verification is costly. In order to mitigate the shortcomings of both translations, UAutomizer first runs the verification on the integer-based model. If the result is *unknown*, the tool is run again on the bitvector-based model.

* Jury Member: Matthias Heizmann

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 418–423, 2024.
https://doi.org/10.1007/978-3-031-57256-2_31

2 Abstraction of Bitwise Operations

In the past our integer translation overapproximated the bitwise operators, i.e. `&`, `|`, `^`, `~`, `<<`, `>>` returned some non-deterministic value. In this paper we show how to translate bitwise operators more precisely. Our translation is a generalization of the work of Liu et al. [11]. First we describe the translation of the operators `&`, `|`, `^`. The remaining operators will be explained at the end of this section. For the operators `&`, `|`, `^` we distinguish three different cases:

- If both operands are literals, we replace the operation by its result.
- If one operand is a literal with a specific bit-pattern, we rewrite the expression directly.
- Otherwise we overapproximate with additional constraints for the return value.

Rewrite rules. If one of the operands is a literal, we try to replace the bitwise operation by an arithmetic operation based on the bit-pattern of the literal. These rewrite rules are shown in Table 1 (omitting symmetric cases). The first two cases are simple. In the first row every bit is zero (i.e. the operand is 0). Zero is the absorbing element for `&` and the neutral element for `|` and `^`. In the second row every bit is one (i.e. the operand is -1 for signed integers or the maximum value for unsigned integers). This is the neutral element for `&` and the absorbing element for `|`. The last two cases are motivated by typical bitmasks and are a generalization of the first two cases. In a C program, bitmasks are used to set bits to zero or to one. For example the expression `x & 255` can be used to replace every bit of `x` by zero except for the last 8 bits. The third row is motivated by Liu et al. [11]. They rewrote `x & 1` (i.e. only the last bit is one) to `x % 2`, whereas we generalize this case for any pattern that only ends with ones. With the rule on the third row the expression `x & 255` is rewritten to `x % 256`. In the last row only the starting bits are one. This case works analogously to the third row, it is rewritten using a combination of modulo and other arithmetic operators. We implemented these rules in our translation from C to Boogie. Boogie has mathematical integer semantics, so the evaluation of the expressions in the table can never lead to an overflow. The rules for the operators `|` and `^` are based on the equalities $a \mid b = a + b - (a \,\&\, b)$ and $a\,\hat{}\,b = a + b - 2 \cdot (a \,\&\, b)$.

Constrained Overapproximation. If none of the operands are literals with a bit-pattern from above, we translate the bitwise operations to calls to functions as implemented in Fig. 1 in Boogie as follows: `x & y` is translated to `and(x, y)`, `x | y` is translated to `or(x, y)` and `x ^ y` is translated to `xor(x, y)`. We omitted

Table 1: Rewrite rules based on the bit-pattern of `c`

`bits(c)`	`x & c`	`x | c`	`x ^ c`
`0...0`	`c`	`x`	`x`
`1...1`	`x`	`c`	`c - x`
`0...01...1`	`x%(c+1)`	`x+c - x%(c+1)`	`x+c - 2*(x%(c+1))`
`1...10...0`	`x - x%(c+1)`	`c + x%(c+1)`	`c-x - 2*(x%(c+1))`

```
procedure and(a: int, b: int) {
  if (a == 0 || b == 0) return 0;
  if (a == b) return a;

  var r: int;
  assume (a>=0 || b<0) ==> r<=a;
  assume (a<0 || b>=0) ==> r<=b;
  assume (a>=0 || b>=0) ==> r>=0;
  assume (a<0 || b<0) ==> r>a+b;
  return r;
}
```

```
procedure xor(a: int, b: int) {
  if (a == 0) return b;
  if (b == 0) return a;
  if (a == b) return 0;

  var r: int;
  assume (a>=0 <==> b>=0) ==> r>0;
  assume !(a>=0 <==> b>=0) ==> r<0;
  assume (a>=0 || b>=0) ==> r<=a+b;
  return r;
}
```

Fig. 1: Procedures to overapproximate the operators `&` and `^`

the definition for the function `or(a, b)` here, because a possible implementation could simply use the relation between `&` and `|` to return `a + b - and(a, b)`. The first lines of `and` and `xor` cover the cases that are handled precisely, i.e. where one of the operands is zero or both are equal. For all other cases return a non-deterministic value to overapproximate the behavior of the bitwise operators. We constrain this value via the assumptions that often provide lower and upper bounds. For example, if `a` and `b` are both non-negative, `and(a, b)` returns also a non-negative value that is also smaller or equal to both `a` and `b`. Similarly `xor(a, b)` returns a positive value that is smaller or equal to the sum `a + b` in that case.

Negation and Shifts. We rewrite the negation ∼`x` to the equivalent expression `-1 - x`. We rewrite shift operators if the second operand is a literal. The left shift `x << y` is rewritten to `x * c` and the right shift `x >> y` is rewritten to `x / c`, where `c` is the literal that is obtained by evaluating `pow(2, y)`. The rewritten expression `x * c` has an overflow if and only if the original expression `x << y` has an overflow.

3 Strengths and Weaknesses

UAutomizer won the overall category and the category NoOverflows in SV-COMP 2024 [2]. UAutomizer reported 10 incorrect results, which were due to incorrect modelling of C features.

We evaluated the abstraction of bitwise operations on selected benchmarks from SV-COMP 2024. The evaluation was performed on a AMD Ryzen Threadripper 3970X using 2 cores at 3.7 GHz with a time limit of 900 s and a memory limit of 8 GB. In Table 2 you can see the results of the evaluation on the category ReachSafety. We choose this category, because it contains a wide range of benchmarks, including several that make use of bitwise operators. There we compared three settings: the bitvector-based translation, the old integer-based translation where every bitwise operation is allowed to return any value and the integer-based translation with the optimizations described in Section 2. The results show that the new integer-based translation can verify 25 more benchmarks than the old integer-based translation (from various folders, e.g. hardness-nfm22

Table 2: Comparison on ReachSafety

	Bitvector			Integer (optimized)			Integer (old)		
		time	mem		time	mem		time	mem
	#	(h)	(GB)	#	(h)	(GB)	#	(h)	(GB)
total (10 205)	1 958	65	1 862	2 076	37	2 600	2 051	36	2 550
safe (7 557)	1 183	41	1 030	1 350	22	1 510	1 324	21	1 440
unsafe (2 648)	775	24	832	726	15	1 090	727	15	1 110

Table 3: Comparison on Termination-BitVectors

	Integer (optimized)			Integer (old)		
		time	mem		time	mem
	#	(s)	(GB)	#	(s)	(GB)
total (37)	31	410	12.1	12	122	4.2
safe (23)	23	325	9.2	7	73	2.5
unsafe (14)	8	85	2.9	5	49	1.7

and hardware-verification) and 118 more than the bitvector-based translation. The bitvector-based translation is precise in contrast to the integer-based translation. Overall this precision does not pay off, as the result of the bitvector-based translation is often too costly to verify. However, the precision can also be helpful, as the bitvector-based translation can find 48 (resp. 49) more bugs than the integer-based translations.

We also evaluated our approach on the subcategory Termination-BitVectors, where most of the benchmarks contain bitwise operations. For termination we do not support bitvectors, therefore we compared only our approach with the old integer-based translation. The results in Table 3 show that the our optimized approach is sufficient to prove the (non-)termination of 31 of the total 37 tasks, whereas the trivial overapproximation is only sufficient for 12.

4 Architecture, Setup, Configuration, and Project

UAutomizer is part of Ultimate [15,16], a program analysis framework written in Java and licensed under LGPLv3. UAutomizer is an automaton-based model checker using a CEGAR-loop approach [8]. The submitted version 0.2.4-0e0057cc requires Java 11 and Python 3.6. Its Linux version, binaries of the required SMT solvers Z3 [12,13], CVC4 [1,14], MathSAT [4,7], and a Python wrapper script were submitted as a `.zip` archive. UAutomizer 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 witness is written to the files `witness.graphml` and `witness.yml`. The benchmarking tool BenchExec [3] supports UAutomizer through the tool-info module `ultimateautomizer.py`. UAutomizer participates in all categories, as declared in its benchmark definition file `uautomizer.xml`.

Data Availability. The competition contribution for UAutomizer is available as an archive on Zenodo [5].

References

1. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
2. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
3. 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
4. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
5. Dietsch, D., Bentele, M., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Automizer SV-COMP 2024 Competition Contribution (Nov 2023). https://doi.org/10.5281/zenodo.10203545
6. Dietsch, D., Heizmann, M., Klumpp, D., Naouar, M., Podelski, A., Schätzle, C.: Verification of concurrent programs using Petri net unfoldings. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12597, pp. 174–195. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_9
7. Fondazione Bruno Kessler, D.: MathSAT, https://mathsat.fbk.eu, (retrieved 2024-02-12)
8. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018. Lecture Notes in Computer Science, vol. 10806, pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
9. 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
10. Leino, K.R.M.: This is Boogie 2 (June 2008), https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
11. Liu, Y.C., Pang, C., Dietsch, D., Koskinen, E., Le, T., Portokalidis, G., Xu, J.: Proving LTL properties of bitvector programs and decompiled binaries. In: Oh, H. (ed.) Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13008, pp. 285–304. Springer (2021). https://doi.org/10.1007/978-3-030-89051-3_16

12. Microsoft Corporation: Z3, https://github.com/Z3Prover/z3, (retrieved 2024-02-12)
13. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Stanford University, U.: CVC4, https://cvc4.github.io, (retrieved 2024-02-12)
15. University of Freiburg: Ultimate source code repository, https://github.com/ultimate-pa/ultimate, (retrieved 2024-02-12)
16. University of Freiburg: Ultimate website, https://ultimate-pa.org, (retrieved 2024-02-12)

Author Index

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 425–428, 2024.
https://doi.org/10.1007/978-3-031-57256-2