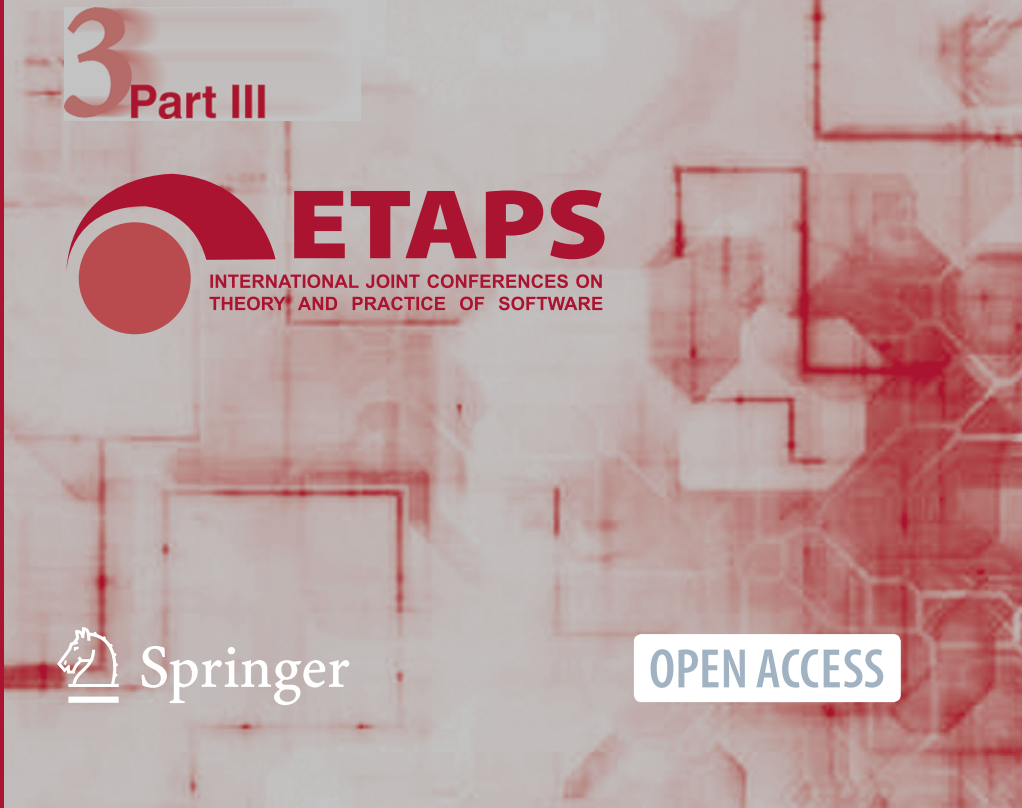
Arie Gurfinkel
Marijn Heule (Eds.)

LNCS 15698 ARCoSS

Tools and Algorithms for the Construction and Analysis of Systems

31st International Conference, TACAS 2025
Held as Part of the International Joint Conferences
on Theory and Practice of Software, ETAPS 2025
Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part III



Lecture Notes in Computer Science **15698**

Founding Editors

Gerhard Goos
Juris Hartmanis

Editorial Board Members

Elisa Bertino, USA
Wen Gao, China
Bernhard Steffen, Germany
Moti Yung, USA

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome 'La Sapienza', Italy*
Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *TU Munich, Germany*
Benjamin C. Pierce, *University of Pennsylvania, USA*
Bernhard Steffen, *University of Dortmund, Germany*
Deng Xiaotie, *Peking University, Beijing, China*
Jeannette M. Wing, *Microsoft Research, Redmond, WA, USA*

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

Arie Gurfinkel · Marijn Heule
Editors

Tools and Algorithms for the Construction and Analysis of Systems

31st International Conference, TACAS 2025
Held as Part of the International Joint Conferences
on Theory and Practice of Software, ETAPS 2025
Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part III

Springer

Editors
Arie Gurfinkel
University of Waterloo
Waterloo, ON, Canada

Marijn Heule
Carnegie Mellon University
Pittsburgh, PA, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-90659-6 ISBN 978-3-031-90660-2 (eBook)
https://doi.org/10.1007/978-3-031-90660-2

© The Editor(s) (if applicable) and The Author(s) 2025. This book is an open access publication.

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

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

If disposing of this product, please recycle the paper.

ETAPS Foreword

Welcome to the 28th ETAPS! ETAPS 2025 took place in Hamilton, Canada. It is the first time ETAPS was held outside of Europe.

ETAPS 2025 was the 28th instance of the International Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of four conferences: ESOP, FASE, FoSSaCS, and TACAS. Each conference has its own Program Committee (PC) and its own Steering Committee (SC). The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations of programming languages, analysis tools, and formal approaches to software engineering. Organizing these conferences in a coherent, highly synchronized conference 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 2025 received 329 submissions in total, 106 of which were accepted, yielding an overall acceptance rate of 32.2%. 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 2025 featured the unifying invited speakers Ina Schaefer (Karlsruhe Institute of Technology, Germany) and Matthew B. Dwyer (University of Virginia, USA), and the invited speakers Amal Ahmed (Northeastern University, USA) for ESOP and José Meseguer (University of Illinois Urbana-Champaign, USA) for FASE. Invited tutorials were provided by Suguman Bansal (Georgia Institute of Techology, USA) on reinforcement learning from logical specifications and Arun Ross (Michigan State University, USA) on biometrics.

ETAPS 2025 was organized by McMaster University. The Faculty of Engineering at McMaster University has a reputation for innovative programs, cutting-edge research, leading faculty, and aspiring students. It has earned a strong reputation as a center for academic excellence and innovation. The Faculty has approximately 180 faculty members, along with close to 4,500 undergraduate and 1,000 graduate students. The local organization team consisted of Claudio Menghi and Mark Lawford (general chairs), Melissa Alzaeim (event organizer), Alan Wassyng and Angelo Gargantini (workshop chairs), Sébastien Mosser and Matt Luckcuck (publicity chairs), Patrizio Pelliccione (sponsor chair), Silvia Bonfanti and Andrea Bombarda (web chairs), Jacques Carette and Christos Tsigkanos (local proceedings chair), Lena Liberale and Martin von Mohrenschildt (finance chairs), Damiano Torre and Lina Marsso (registration chairs), and Vera Pantelic and Denise Geiskkovitch (student volunteer chairs).

ETAPS 2025 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-Essen), Arnd Hartmanns (Twente), Caterina Urban (Inria), Jan Křetínský (Munich), Elizabeth Polgreen (Edinburgh), and Lenore Zuck (Chicago).

Other members of the steering committee are: Elvira Albert (Madrid), Maurice ter Beek (Pisa), Nathalie Bertrand (Rennes), Dirk Beyer (Munich), Artur Boronat (Leicester), Luís Caires (Lisboa), Ferruccio Damiani (Torino), Gordon Fraser (Passau), Arie Gurfinkel (Waterloo), Reiner Hähnle (Darmstadt), Reiko Heckel (Leicester), Marijn Heule (Pittsburgh), Sebastian Junges (Nijmegen), Joost-Pieter Katoen (Aachen and Twente), Guy Katz (Jerusalem), Delia Kesner (Paris), Fabrice Kordon (Paris), Robbert Krebbers (Nijmegen), Kim Guldstrand Larsen (Aalborg), Mark Lawford (Hamilton), Claudio Menghi (Hamilton and Bergamo), Stefan Milius (Erlangen-Nürnberg), Andrzej Murawski (Oxford), Corina Păsăreanu (Ames), Laure Petrucci (Paris), Peter Y.A. Ryan (Luxembourg), Don Sannella (Edinburgh), Viktor Vafeiadis (Kaiserslautern), and Anton Wijs (Eindhoven).

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 2025 was also generously supported by Tourism Hamilton and the Tutte Institute for Mathematics and Computing. I hope you all enjoyed ETAPS 2025.

Finally, a big thanks to Claudio, Mark and Melissa and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

May 2025

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

This three-volume proceedings contains the papers presented at the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2025). TACAS 2025 was part of the 28th International Joint Conferences on Theory and Practice of Software (ETAPS 2025), which was held in Hamilton, Ontario, Canada.

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 2025 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 2025:

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 were restricted to 6 pages, excluding the bibliography and appendices.

TACAS 2025 received 148 submissions, consisting of 103 regular research papers, 6 case study papers, 29 regular tool papers, and 10 tool demonstration papers. Each submission was assigned for review to at least three Program Committee (PC) members, who made use of sub-reviewers. Regular research papers were reviewed as double-blind, whereas case study, regular tool, and tool-demonstration papers were reviewed using a single-blind reviewing process.

As in previous years, authors had the option to submit an artifact alongside their paper. For TACAS 2025, artifact submission was mandatory for regular tool and tool demonstration papers, and optional for regular research and case study papers. Artifacts could include tools, models, proofs, or any other data necessary to validate the paper's results. The Artifact Evaluation Committee (AEC), which was composed of 89 members of the community, was responsible for reviewing these submissions, assessing their

documentation, usability, and, most importantly, whether the results presented in the corresponding paper could be successfully reproduced. Most evaluations were conducted using a standardized virtual machine or a Docker image to ensure consistency, except in cases where specific hardware or software requirements applied. Artifact evaluation at TACAS 2025 was carried out in two rounds. The first round, which took place alongside the work of the Program Committee (PC), involved the mandatory evaluation of regular tool and tool demo papers. The decisions of the AEC were shared with the PC and factored into their discussions. The second round focused on the voluntary evaluation of regular research and case study papers and was conducted after paper acceptance notifications were issued. In both rounds, each artifact received three reviews, and authors had the opportunity to anonymously communicate with the AEC to resolve technical issues. In total, 65 artifacts (39 were mandatory as they were associated with tool papers, and 26 were voluntary) were evaluated for their availability, functionality, and re-usability. Papers with successfully assessed artifacts were awarded one or more badges which are placed on the first page of the corresponding paper, certifying the relevant claims and properties of the tool.

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 46 papers, out of which there were 28 regular research papers, 11 regular tool papers, 3 case study papers, and 4 tool demonstration papers. This corresponds to an overall acceptance rate of 31%.

TACAS 2025 also hosted SV-COMP 2025, the 14th International Competition on Software Verification. This event evaluated 62 tools for automatic verification of C and Java programs and 18 tools for witness validation, where 35 verification and 13 validation tools were registered and actively supported by development teams. The TACAS 2025 proceedings contain a competition report by the SV-COMP chairs and 13 short papers selected by the competition jury. One short paper deals with reproduction aspects of the competition and the remaining short papers describe 12 out of the tools participating with active team support. The 13 short papers were reviewed by a separate program committee (jury); each was assessed by at least four jury members. Two sessions in the TACAS 2025 program were reserved for SV-COMP: (1) a presentation session with a report by the competition chairs and summaries by the development 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 2025 successful. We thank the authors for submitting their papers to TACAS 2025. 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 2025 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 2025.

May 2025

Arie Gurfinkel
Marijn Heule
PC Chairs

Daniela Kaufmann
Mark Santolucito
AEC Chairs

Dirk Beyer
Jan Strejček
SV-COMP Chairs

Organization

Program Committee Chairs

Arie Gurfinkel	University of Waterloo, Canada
Marijn Heule	Carnegie Mellon University, USA

Program Committee

Erika Abraham	RWTH Aachen University, Germany
Elvira Albert	Universidad Complutense de Madrid, Spain
Cyrille Valentin Artho	KTH Royal Institute of Technology, Sweden
Haniel Barbosa	Universidade Federal de Minas Gerais, Brazil
Clark Barrett	Stanford University, USA
Dirk Beyer	LMU Munich, Germany
Armin Biere	University of Freiburg, Germany
Nikolaj Bjørner	Microsoft, USA
Roderick Bloem	Graz University of Technology, Austria
Rose Bohrer	Worcester Polytechnic Institute, USA
Borzoo Bonakdarpour	Michigan State University, USA
Mingshuai Chen	Zhejiang University, China
Mila Dalla Preda	University of Verona, Italy
Rayna Dimitrova	CISPA Helmholtz Center for Information Security, Germany
Grigory Fedyukovich	Florida State University, USA
Hadar Frenkel	Bar Ilan University, Israel
Alberto Griggio	Fondazione Bruno Kessler, Italy
Ichiro Hasuo	National Institute of Informatics, Japan
Holger Hermanns	Saarland University, Germany
Alan J. Hu	University of British Columbia, Canada
Sebastian Junges	Radboud University, Netherlands
Temesghen Kahsai	Amazon, USA
Joost-Pieter Katoen	RWTH Aachen University, Germany
Guy Katz	Hebrew University of Jerusalem, Israel
Umang Mathur	National University of Singapore, Singapore
Anastasia Mavridou	KBR/NASA Ames Research Center, USA
Kuldeep S. Meel	University of Toronto, Canada
Magnus O. Myreen	Chalmers University of Technology, Sweden

Nina Narodytska	VMware Research by Broadcom, USA
Jorge A. Navas	Certora, USA
Laura Nenzi	University of Trieste, Italy
Aina Niemetz	Stanford University, USA
André Platzer	Karlsruhe Institute of Technology, Germany
Elizabeth Polgreen	University of Edinburgh, UK
Kristin Yvonne Rozier	Iowa State University, USA
Christian Schilling	Aalborg University, Denmark
Anne-Kathrin Schmuck	Max Planck Institute for Software Systems, Germany
Natasha Sharygina	USI, Switzerland
Sharon Shoham	Tel Aviv University, Israel
Xujie Si	University of Toronto, Canada
Mihaela Sighireanu	Université Paris-Saclay, CNRS, ENS Paris-Saclay, France
Tanel Tammet	Tallinn University of Technology, Estonia
Yong Kiam Tan	Institute for Infocomm Research (I^2R), A*STAR and Nanyang Technological University, Singapore
Silvia Lizeth Tapia Tarifa	University of Oslo, Norway
Cesare Tinelli	University of Iowa, USA
Hazem Torfah	Chalmers University of Technology, Sweden
Yakir Vizel	Technion, Israel
Tomas Vojnar	Masaryk University and Brno University of Technology, Czech Republic
Jingbo Wang	University of Southern California, USA
Georg Weissenbacher	TU Wien, Austria
Anton Wijs	Eindhoven University of Technology, Netherlands
Sarah Winkler	Free University of Bozen-Bolzano, Italy
Haoze Wu	Amherst College, USA

Artifact Evaluation Committee Chairs

Daniela Kaufmann	TU Wien, Austria
Mark Santolucito	Barnard College at Columbia University, USA

Artifact Evaluation Committee

Mohammad Afzal	TCS Research Pune and IIT Bombay India, India
Mohammad M. Ahmadpanah	Chalmers University of Technology, Sweden

Hichem Rami Ait El Hara	OCamlPro/Université Paris-Saclay, France
Ashwani Anand	Max Planck Institute for Software Systems, Kaiserslautern, Germany
Bruno Andreotti	Universidade Federal de Minas Gerais, Brazil
Åsmund Aqissiaq Arild Kløvstad	University of Oslo, Norway
Paulína Ayaziová	Masaryk University, Czech Republic
Anna Becchi	Fondazione Bruno Kessler, Italy
Raven Beutner	CISPA Helmholtz Center for Information Security, Germany
Alberto Bombardelli	Fondazione Bruno Kessler, Italy
Konstantin Britikov	USI, Switzerland
Marco Campion	Inria & École Normale Supérieure | Université PSL, France
Filip Cano	Graz University of Technology, Austria
Falke Carlsen	Aalborg Universityi, Denmark
Abha Chaudhary	Binghamton University (SUNY), USA
Lei Chen	HKUST (GZ), China
Siyu Chen	Purdue University, USA
Weiyi Chen	Purdue University, USA
Robin Coutelier	TU Wien, Austria
Vlad Constantin Craciun	BitDefender, UAIC, Romania
Leyi Cui	Columbia University, USA
Charles de Haro	École Normale Supérieure | Université PSL, France
Rafael Dewes	CISPA Helmholtz Center for Information Security, Germany
Oyendrila Dobe	Amazon Web Services, USA
Aoyang Fang	Chinese University of Hong Kong, Shenzhen, China
Mathias Fleury	University of Freiburg, Germany
Rui Ge	University of British Columbia, Canada
Pritam Gharat	Microsoft Research, USA
Pablo Gordillo	Universidad Complutense de Madrid, Spain
Jan Heemstra	Eindhoven University of Technology, Netherlands
Philippe Heim	CISPA Helmholtz Center for Information Security, Germany
Maximilian Heisinger	JKU Linz, Austria
Alejandro Hernández-Cerezo	Complutense University of Madrid, Spain
Matthias Hetzenberger	TU Wien, Austria
Nikolaus Holzer	Columbia University, USA
Petra Hozzová	Czech Technical University, Czech Republic
Guangyu Hu	Hong Kong University of Science and Technology (HKUST), China

Miguel Isabel	Universidad Politécnica de Madrid, Spain
Omri Isac	Hebrew University of Jerusalem, Israel
Tobias John	University of Oslo, Norway
Basel Khouri	Technion, Israel
Elad Kinsbruner	Technion, Israel
Paul Kobialka	University of Oslo, Norway
Tomáš Kolárik	USI, Switzerland
Wietze Koops	Lund University, Sweden and University of Copenhagen, Denmark
Martin Kristjansen	Aalborg University, Denmark
Marco Lewis	Inria, France
Changyue Li	Chinese University of Hong Kong, China
Xuyang Li	Purdue University, USA
Jing Liu	University of California, Irvine, USA
Yonghui Liu	Monash University, Australia
Nils Lommen	RWTH Aachen University, Germany
Filip Macák	Brno University of Technology, Czech Republic
Benedikt Maderbacher	Graz University of Technology, Austria
Kaushik Mallik	IMDEA Software Institute, Spain
Baoluo Meng	GE Aerospace Research, USA
Srinidhi Nagendra	Max Planck Institute for Software Systems, Germany
Tobias Nießen	TU Wien, Austria
Andy Oertel	Lund University and University of Copenhagen, Denmark
Elizaveta Pertseva	Stanford, USA
Anja Petković Komel	TU Wien, Austria
Mark Peyrer	JKU Linz, Austria
Jyoti Prakash	University of Passau, Germany
Juliane Päßler	University of Oslo, Norway
Arshia Rafieioskouei	Michigan State University, USA
Idan Refaeli	Hebrew University of Jerusalem, Israel
Simon Robillard	Université de Montpellier, France
Clara Rodriguez Nuñez	Complutense University of Madrid, Spain
Raven Rothkopf	Barnard College, USA
Neea Rusch	Augusta University, USA
Kartik Sabharwal	University of Iowa, USA
Tobias Seufert	University of Freiburg, Germany
Abhishek Singh	Tel Aviv University, Israel
Mallku Soldevila	Universidade Federal de Minas Gerais, Brazil
Reza Soltani	University of Twente, Netherlands
Yusen Su	University of Waterloo, Canada

Geoff Sutcliffe	University of Miami, USA
Joseph Tafese	University of Waterloo, Canada
Gefei Tan	Northwestern University, USA
Yun Chen Tsai	National Institute of Informatics, Japan
Divyesh Unadkat	Synopsys Inc., India
Christoph Wernhard	University of Potsdam, Germany
Aosen Xiong	University of Waterloo, Canada
Beyazit Yalcinkaya	University of California, Berkeley, USA
Jiong Yang	Georgia Institute of Technology, USA
Bohan Zhang	Binghamton University, USA
Yi Zhou	Carnegie Mellon University, USA

SV-COMP Organization

Chairs

Dirk Beyer	LMU Munich, Germany
Jan Strejček	Masaryk University, Czechia

Benchmark Quality Assurance

Zsófia Ádám	BUTE, Budapest, Hungary
Raphaël Monat	Inria and University of Lille, France
Simmo Saan	University of Tartu, Estonia
Frank Schüssele	University of Freiburg, Germany

Benchmark Categories

Thomas Lemberger	LMU Munich, Germany

Infrastructure

Philipp Wendler	LMU Munich, Germany (BenchExec)
Po-Chun Chien	LMU Munich, Germany (BenchCloud)
Marek Jankola	LMU Munich, Germany (BenchCloud)
Henrik Wachowitz	LMU Munich, Germany (FM-Weck)
Matthias Kettl	LMU Munich, Germany (Competition Scripts)
Marian Lingsch-Rosenfeld	LMU Munich, Germany (WitnessLint)

Reproducibility

Levente Bajczi	BUTE, Budapest, Hungary

SV-COMP Program Committee and Jury

Dirk Beyer (Co-chair)	LMU Munich, Germany
Jan Strejček (Co-chair)	Masaryk University, Czechia
Zsófia Ádám	BUTE, Budapest, Hungary
Paulína Ayaziová	Masaryk University, Czechia
Levente Bajczi	BUTE, Budapest, Hungary
Manuel Bentele	University of Freiburg, Germany
Martin Blicha	University of Lugano, Switzerland
Lei Bu	Nanjing University, China
Marek Chalupa	ISTA, Austria
Zhenbang Chen	National University of Defense Technology, China
Po-Chun Chien	LMU Munich, Germany
Tomáš Dacík	Brno University of Technology, Czechia
Priyanka Darke	Tata Consulting Services, India
Daniel Dietsch	University of Freiburg, Germany
Marcel Ebbinghaus	University of Freiburg, Germany
Gidon Ernst	LMU Munich, Germany
Fei He	Tsinghua University, China
Matthias Heizmann	University of Freiburg, Germany
Falk Howar	TU Dortmund, Germany
Soha Hussein	Ain Shams University, Egypt
Martin Jonáš	Masaryk University, Czechia
Dominik Klumpp	University of Freiburg, Germany
Marian Lingsch-Rosenfeld	LMU Munich, Germany
Nils Lommen	RWTH Aachen, Germany
Nils Loose	University of Luebeck, Germany
Viktor Malík	Brno University of Technology, Czechia
Ravindra Metta	Tata Consulting Services, India
Raphaël Monat	Inria and University of Lille, France
Hernán Ponce de León	Huawei Dresden Research Center, Germany
Matthew Richards	University of New South Wales, Australia
Simmo Saan	University of Tartu, Estonia
Peter Schrammel	Diffblue Ltd., UK
Frank Schüssele	University of Freiburg, Germany
Vesal Vojdani	University of Tartu, Estonia

Henrik Wachowitz	LMU Munich, Germany
Tong Wu	University of Manchester, UK

TACAS Steering Committee

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

Additional Reviewers

Abou El Wafa, Noah
Anand, Ashwani
Ang, Zhendong
Antal, László
Arceri, Vincenzo
Assolini, Nicola
Baier, Daniel
Banerjee, Subarno
Bassan, Shahaf
Berthon, Raphaël
Blicha, Martin
Bork, Alexander
Bovy, Eline
Brand, Sebastiaan
Britikov, Konstantin
Butte, Julia
Campion, Marco
Cano, Filip
Chen, Xin
Dacík, Tomáš
Dalleiger, Sebastian
Dengler, Gabriel
Dewes, Rafael
Elad, Neta
Elboher, Yizhak
Eshghie, Mojtaba
Fazekas, Katalin
Fesefeldt, Ira
Fleury, Mathias
Frankel, Guy
Frenkel, Eden
Frohn, Florian
Froleyks, Nils
Gamboa Guzman, Laura P.
Garagiola, Nazareno
Gehnen, Christina
Geng, Chuqin
Gerlach, Carolina
Gozzi, Riccardo
Gstrein, Bernhard
Hamza, Ameer
Hansen, Jonas
Haring, Johannes
Heemstra, Jan
Heim, Philippe
Holík, Lukáš
Hsu, Tzu-Han
Iosif, Radu
Isac, Omri
Jankola, Marek
Johannsen, Chris
Kabra, Aditi
Katis, Andreas
Kettl, Matthias

Klinkenberg, Lutz
Kløvstad, Åsmund Aqissiaq Arild
Kobialka, Paul
Koenighofer, Bettina
Kolárik, Tomáš
Kovács, József
Krogmeier, Paul
Köhl, Maximilian Alexander
Labbaf, Faezeh
Larraz, Daniel
Laurent, Jonathan
Lemberger, Thomas
Lengal, Ondrej
Li, Yixuan
Li, Zhiyang
Lingsch-Rosenfeld, Marian
Lorch, Robert
Lotan, Raz
Luque Cerpa, Alejandro
Maderbacher, Benedikt
Malík, Viktor
Miao, Mingkai
Nalbach, Jasper
Nayak, Satya Prakash
Negrini, Luca
Noll, Thomas
Novozhilov, Sergei
Otoni, Rodrigo
Panja, Promit
Patault, Paul
Perez-Lopez, Áron Ricardo
Pertseva, Elizaveta
Pferscher, Andrea
Pinto, Alessandro
Pollitt, Florian
Pranger, Stefan
Preiner, Mathias
Promies, Valentin
Päßler, Juliane
Pîrlea, George
Qian, Long
Quatmann, Tim
Rafieioskouei, Arshia
Raha, Ritam
Refaeli, Idan
Reynolds, Andrew
Riley, Daniel
Rogalewicz, Adam
Rosentrater, Alec
Saglam, Irmak
Sangnier, Arnaud
Saveri, Gaia
Schlatte, Rudolf
Schlichtkrull, Anders
Schmidt, Andreas
Schreiber, Dominik
Schurr, Hans-Jörg
Shankar, Raghav
Sierra, Sebastian
Silvetti, Simone
Skurka, Antonina
Sogokon, Andrew
Soldevila, Mallku
Sun, Yutao
Tan, Grace
Tappler, Martin
Trtík, Marek
Tschaikowski, Max
Tsiskaridze, Nestan
Vaandrager, Frits
van den Haak, Lars B.
van der Vegt, Marck
Varanasi, Sarat Chandra
Wachowitz, Henrik
Wang, Zhongyi
Wang, Zili
Wendler, Philipp
Winkler, Tobias
Wu, Min
Yang, Jiong
Yang, Mingqi
Ye, Leiqi
Yu, Hengzhi
Zavalia, Lucas
Zhang, David
Zhao, Huan
Zhou, Xiaqing
Zimmer, Dominic
Zimmermann, Martin

Contents – Part III

Verification 2

Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM 3
Nikolaus Huber, Naomi Spargo, Nicolas Osborne, Samuel Hym, and Jan Midtgaard

Weakly Acyclic Diagrams: A Data Structure for Infinite-State Symbolic Verification 23
Michael Blondin, Michaël Cadilhac, Xin-Yi Cui, Philipp Czerner, Javier Esparza, and Jakob Schulz

Pushing the Limit: Verified Performance-Optimal Causally-Consistent Database Transactions 43
Shabnam Ghasemirad, Christoph Sprenger, Si Liu, Luca Multazzu, and David Basin

Certifiably Robust Policies for Uncertain Parametric Environments 63
Yannik Schnitzer, Alessandro Abate, and David Parker

Quantum and GPU

AutoQ 2.0: From Verification of Quantum Circuits to Verification of Quantum Programs 87
Yu-Fang Chen, Kai-Min Chung, Min-Hsiu Hsieh, Wei-Jia Huang, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai

Parallel Equivalence Checking of Stabilizer Quantum Circuits on GPUs 109
Muhammad Osama, Dimitrios Thanos, and Alfons Laarman

SliQSim: A Quantum Circuit Simulator and Solver for Probability and Statistics Queries 129
Tian-Fu Chen and Jie-Hong R. Jiang

GPUexplore $^{\text{PROB}}$: Markov Chain State Space Construction and Verification with GPUs 139
Jan Heemstra and Anton Wijs

14th Competition on Software Verification (SV-COMP 2025)

Improvements in Software Verification and Witness Validation: SV-COMP 2025 151
Dirk Beyer and Jan Strejček

SV-COMP'25 Reproduction Report (Competition Contribution) 187
Levente Bajczi, Zsófia Ádám, and Zoltán Micskei

CPAchecker 4.0 as Witness Validator: (Competition Contribution) 192
Dirk Beyer and Marian Lingsch-Rosenfeld

AISE v2.0: Combining Loop Transformations: (Competition Contribution) 199
Yao Lin, Zhenbang Chen, and Ji Wang

AProVE (KoAT + LoAT): (Competition Contribution) 205
Nils Lommen and Jürgen Giesl

BUBAAK: Dynamic Cooperative Verification: (Competition Contribution) 212
Marek Chalupa and Cedric Richter

EmergenTheta: Variations on Symbolic Transition Systems (Competition Contribution) 217
Milán Mondok, Levente Bajczi, Dániel Szekeres, and Vince Molnár

ESBMC v7.7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction: (Competition Contribution) 223
Tong Wu, Xianzhiyu Li, Edoardo Manino, Rafael Sá Menezes, Mikhail R. Gadelha, Shale Xiong, Norbert Tihanyi, Pavlos Petoumenos, and Lucas C. Cordeiro

Mopsa-C with Trace Partitioning and Autosuggestions (Competition Contribution) 229
Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné

Nacpa: Native Checking with Parallel-Portfolio Analyses: (Competition Contribution) 236
Thomas Lemberger and Henrik Wachowitz

PROTON 2.1: Synthesizing Ranking Functions via fine-tuned locally Hosted LLM (Competition Contribution) 242
Diganta Mukhopadhyay, Ravindra Metta, Hrishikesh Karmarkar, and Kumar Madhukar

RACERF: Data Race Detection with Frama-C (Competition Contribution) 248
Tomáš Dacík and Tomáš Vojnar

SVF-SVC: Software Verification Using SVF (Competition Contribution) 254
Cameron McGowan, Matthew Richards, and Yulei Sui

THETA: Various Approaches for Concurrent Program Verification (Competition Contribution) 260
Csanád Telbisz, Levente Bajczi, Dániel Szekeres, and András Vörös

Author Index 267

Verification 2

Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM

Nikolaus Huber[1,2(✉)], Naomi Spargo[1,3], Nicolas Osborne[1], Samuel Hym[1], and Jan Midtgaard[1]

[1] Tarides, Paris, France
{nicolas.osborne,samuel,jan}@tarides.com
[2] Uppsala University, Uppsala, Sweden
nikolaus.huber@it.uu.se
[3] Galois Inc, Arlington, VA, USA
nspargo@galois.com
https://tarides.com

Abstract This paper introduces the QCheck-STM plugin for Ortac, a framework for dynamic verification of OCaml code. Ortac/QCheck-STM consumes OCaml module signatures annotated with behavioural specification contracts expressed in the Gospel language, extracts a functional model of a mutable data structure from it, and generates code for automated runtime assertion checking. We report on the implementation of the tool, the structure of the generated code, and on errors found in established OCaml libraries.

Keywords: OCaml · verification · property-based testing · functional models · runtime assertion checking · random testing

1 Introduction

OCaml is an industrial strength, multi-paradigm programming language. While fundamentally functional at its core, OCaml includes many imperative features, such as references, mutable arrays, I/O, and exceptions. These pose unique challenges when trying to test and verify programs written in it.

While OCaml has been used as a platform for the implementation of various code analysis and verification tools, e.g., the interactive theorem prover Coq [44] and the C code analysis framework Frama-C [15], there is a lack of general purpose tools for the verification of OCaml programs themselves. In order to remedy this void, the Gospel project [9] equips OCaml with its own behavioural specification language.

The Gospel language is tool-agnostic, it only offers a way of expressing formal contracts, which can be leveraged by separate tools in order to perform analysis and verification tasks. Different such tools have been developed, including Cameleer [39], a deductive verification tool, and gospel2cfml[4], a translator of

[4] https://github.com/ocaml-gospel/gospel2cfml

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 3–22, 2025.
https://doi.org/10.1007/978-3-031-90660-2_1

annotated OCaml module signatures into separation logic terms embedded in Coq. In this paper, we focus on another tool consuming Gospel annotations called Ortac. Ortac provides a framework for automated runtime assertion checking, and is therefore a member of the family of dynamic verification tools. Ortac offers a modular architecture, where analysis and verification tasks are implemented as *plugins*. This paper highlights the QCheck-STM plugin, which focuses on black-box, model-based state-machine testing in the style of QuickCheck [3,26].

Given its multi-paradigm nature, OCaml is naturally suited to a number of different verification strategies. While it is possible to test purely functional code with Ortac/QCheck-STM, its strength lies in the verification of specifications relating to mutable data structures. Therefore, this paper will put emphasis on such programs. It is customary to refer to such a data structure as the *System Under Test* (SUT), and the functions provided to work with it as its *Application Programming Interface* (API).

This paper provides an overview of how Ortac/QCheck-STM is implemented, and how it may be used as a dynamic verification tool. To do so, we demonstrate the translation of specifications for a simple array library. The OCaml interface for the library is introduced in Section 2, the Gospel contracts for it in Section 3. After a short overview of Ortac and its plugin structure in Section 4 we showcase the generated code for the array example in Section 5. In Section 6 we evaluate the approach and share examples of bugs found in existing OCaml libraries. Finally, we discuss related work in Section 7, before we remark on future work and conclude in Section 8.

Ortac is an open-source project and its source code is available from the following URL:

https://github.com/ocaml-gospel/ortac

2 Running Example: Array

For illustration, we will test a library providing mutable arrays, which is an excerpt from OCaml's standard library `Array` module:

```
type 'a t

val make : int -> 'a -> 'a t
val length : 'a t -> int
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> unit
```

For those unfamiliar with the syntax of OCaml, the code above defines a `type 'a t` representing arrays of a parametric type `'a`. In addition, this lists the type signatures of four OCaml functions. The first function `make` creates a fresh array from a given size and initialisation element. Function `length` accepts an array parameter and returns the size of it. Finally, `get` and `set` both take an array parameter and return and modify the element of an array at a given index, respectively.

3 Gospel by Example

In order to test the array library, the type and function signatures need to be annotated with Gospel specifications. To start, the type must be annotated with a model. In the tradition of other specification languages [6,29], annotations are added as special comments:

```
type 'a t
(* @ model size : integer
     mutable model contents : 'a sequence *)
```

An array can conceptually be thought of as a fixed capacity container. This logical model can be directly translated to Gospel by annotating `type 'a t` with two *model fields*, one for the immutable ***size*** and one for the mutable ***contents*** of the array. The types ***integer*** and ***'a sequence*** are part of the Gospel standard library and describe arbitrary precision integers and lists of values of type `'a`, respectively.

The `make` function creates new array instances given a size and an initial element:

```
val make : int -> 'a -> 'a t
(* @ t = make size a
     checks size >= 0
     ensures t.size = size
     ensures t.contents = Sequence.init size (fun j -> a) *)
```

The ***checks*** clause introduces a *pre-condition* that must hold at function entry. The two ***ensures*** clauses express that the resulting array has the expected size and that all entries are initialised to the given element ***a***. In addition to a ***checks*** clause, Gospel also offers a ***requires*** clause. Unlike a ***requires*** clause, with the above ***checks*** clause the behaviour of `make` is well-defined in case the pre-state does not meet the condition, as it means that the function raises an `Invalid_argument` exception in that case. The function ***Sequence.init*** is again part of the Gospel standard library.

The `set` function changes the value at a given position in the array:

```
val set : 'a t -> int -> 'a -> unit
(* @ set t i a
     checks 0 <= i < t.size
     modifies t.contents
     ensures t.contents = Sequence.set (old t.contents) i a *)
```

Again, the function ***Sequence.set*** is part of the Gospel standard library. The specification of `set` checks if the given index ***i*** is within the array bounds and

modifies the contents of the argument array, which is indicated by the ***modifies*** clause. For each model field marked as modified, the user needs to provide a corresponding ***ensures*** clause specifying how to construct the modified model. Note that it is implicitly assumed that in case the ***check*** fails, the argument SUT remains unchanged. It is possible to define custom exceptions and give equations for the model state after such an exception has been raised. For further information about other Gospel features, the interested reader is referred to the documentation[5].

For brevity, we will not explain all function contracts here, as both `length` and `get` follow analogously. The full specification of the example array library is shown below:

```
type 'a t
(* @ model size : integer
     mutable model contents : 'a sequence *)

val make : int -> 'a -> 'a t
(* @ t = make size a
     checks size >= 0
     ensures t.size = size
     ensures t.contents = Sequence.init size (fun j -> a) *)

val length : 'a t -> int
(* @ i = length t
     ensures i = t.size *)

val get : 'a t -> int -> 'a
(* @ a = get t i
     checks 0 <= i < t.size
     ensures a = t.contents[i] *)

val set : 'a t -> int -> 'a -> unit
(* @ set t i a
     checks 0 <= i < t.size
     modifies t.contents
     ensures t.contents = Sequence.set (old t.contents) i a *)
```

4 Ortac

Gospel itself does not perform any kind of verification. It is the job of other tools to take the provided specifications and perform further analysis.

The Ortac [18] tool provides functions for converting the given annotations into OCaml code. Ortac is extensible through *plugins*, which can make use of

[5] https://ocaml-gospel.github.io/gospel/

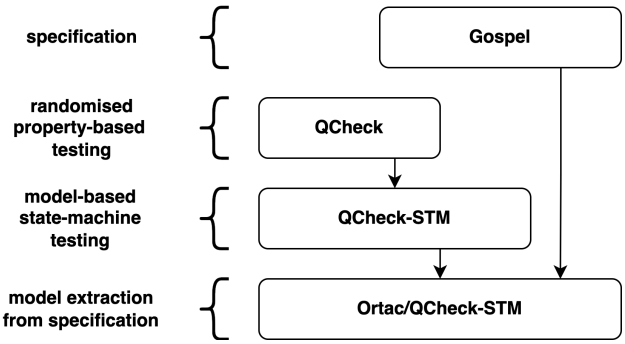


Figure 1. Inner architecture of Ortac/QCheck-STM.

these functions to check specifications. Currently, three different plugins are implemented: Wrapper, Monolith, and QCheck-STM. The original Ortac prototype was developed as part of Clément Pascutto's PhD work [38].

Ortac/Wrapper generates a wrapper module from an annotated module signature, which instruments each function with assertions on the argument and result values according to the given specifications. Ortac/Monolith [35] generates code to interface with Monolith [40], a fuzzing tool for OCaml. However, both the Wrapper and Monolith plugins currently do not support the definition of models. Therefore, they are of limited use when testing mutable data structures. The new QCheck-STM plugin lifts this limitation, by translating Gospel specifications into tests using the QCheck-STM framework. The basic idea behind Ortac/QCheck-STM is to extract a purely functional model of the SUT from the provided Gospel specifications and compare the behaviour of both while running random call sequences of the associated API.

5 Implementation

In this section, we describe the overall architecture of Ortac/QCheck-STM. We then describe how to generate code for testing a single SUT, and finally, how to generalise this approach to support multiple SUTs.

5.1 Ortac Underneath the Hood

In order to understand the working of Ortac/QCheck-STM, it is insightful to look at its constituent parts, as illustrated in Figure 1. At its core, it uses the QCheck library[6], which offers utilities for randomised property-based testing. If a user provides a random generator for a type `'a` and a property as a function `'a -> bool`, QCheck can then run a sequence of tests by randomly generating test inputs of the given type and checking that the property holds for each of

[6] https://github.com/c-cube/qcheck

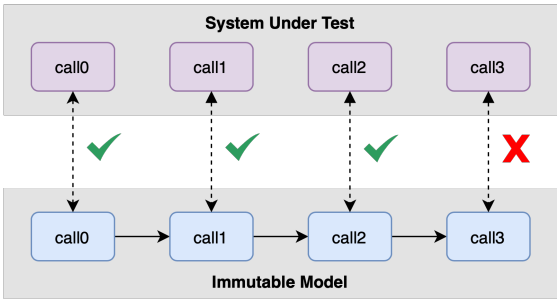


Figure 2. Comparing the behaviour of the SUT and model on randomly generated API call sequences.

them. In case of a violation, QCheck automatically shrinks the given test input to show a minimised counterexample to the user.

QCheck-STM [32] builds upon this functionality by providing a framework for testing a mutable data structure (the SUT) against an immutable reference model, as illustrated in Figure 2. It generates random API call sequences, and then checks the property that the behaviour of the SUT and its functional model coincide for each such test input. In case of a violation of that property, QCheck-STM also reports minimised call sequence traces.

To test a particular data structure and its API with QCheck-STM, the user has to write the model manually. Ortac/QCheck-STM offers the ability to generate the functional model automatically from the Gospel specification. It uses the *model* annotations in order to generate the functional model, and the *modifies* and *ensures* clauses to update the model on each function call.

5.2 Generated code

To illustrate the working of Ortac/QCheck-STM, we will show parts of the generated code from the array specifications introduced in Section 2. First, however, Ortac/QCheck-STM needs another input besides the annotated interface file, which is the configuration of the SUT:

```
type sut = char Array.t
let init_sut = Array.make 16 'a'
```

A minimal configuration needs to provide the `type sut` and a value of that type named `init_sut`. As the name suggests, `type sut` defines the particular type we would like to test, which needs to be fully instantiated. E.g., here we instantiate the polymorphic array type to `char`. The `init_sut` value specifies the initial SUT value from which to start each test. If the Gospel annotated interface is in a file *array.mli* and the configuration in *config.ml*, Ortac/QCheck-STM can be invoked as follows:

```
    ortac qcheck-stm array.mli config.ml
```

The generated code consists of multiple functor specifications and error reporting information. We will simplify the shown code examples to aid conciseness. As a starting point, the SUT and model are defined:

Generated

```
type sut = char Array.t
let init_sut () = Array.make 16 'a'
type state = {
    size : integer;
    contents : char sequence;
}
let init_state = {
    size = integer_of_int 16;
    contents = Sequence.init (integer_of_int 16) (fun _ -> 'a');
}
```

The astute reader will have realised that the definitions of `type sut` and `init_sut` are taken from the provided configuration module (`init_sut` is turned into a function here, as the generated test executable will run multiple instances of random API sequences, for which fresh initial SUT values need to be created). The `type state` defines the two model fields from the specification of `type 'a t` shown in Section 2, where the type variable `'a` has been instantiated to `char` according to the configuration. The initial state value has been synthesised from the given specification of the `make` function by comparing its signature to the `init_sut` function from the configuration module. It does not need to be turned into a function, since it is immutable. The functions `integer_of_int` and `Sequence.init` are provided by the Gospel standard library.

Next, the type of available function calls (i.e., commands) is defined:

Generated

```
type cmd =
  | Length
  | Get of int
  | Set of int * char
```

Each constructor carries argument values according to the formal arguments of the respective signature. Notice, that the SUT argument is missing, as it is not randomly generated, but rather kept and updated by the testing runtime. Furthermore, the `make` function is not present, as it returns a new SUT. This will be amended in Section 5.3.

In order to perform randomised property-based testing with QCheck-STM, a QCheck generator for the `cmd` type is defined:

Generated

```
let arb_cmd state = QCheck.make show_cmd Gen.(oneof [
  pure Length;
  pure (fun i -> Get i) <*> int;
  pure (fun i a -> Set (i, a)) <*> int <*> char;
])
```

This definition needs some explanation. The function `arb_cmd` takes as the only input the current state (i.e., the functional model). This is currently unused, but might be used in the future to define smarter random command generators (see Section 8). `QCheck.make` creates new instances of random generators for a given type by taking both a function that can print values (here defined by `show_cmd` which is left out for brevity) and a generator which can create random values of that type. The `Gen` module provides basic generators and combinators to define new ones. `Gen.oneof` randomly selects from a list of generators. `Gen.pure` always returns its argument value. For simple cases like `Length`, this is enough, however, some command constructors carry fields for their argument values, which need to be provided by random generators as well. For example, the `Get` constructor needs an integer for the index it shall fetch from the array. The infix operator `val ( <*> ) : ('a -> 'b) Gen.t -> 'a Gen.t -> 'b Gen.t` can be used to turn a function generator into a generator of its return type by providing a generator of its argument type. This is done by using the provided generators of base types such as `int` and `char`.

Next, we generate a function that can run a command on a given SUT:

Generated

```
let run cmd sut = match cmd with
  | Length -> Res (int, length sut)
  | Get i -> Res ((result char exn), protect (get sut i))
  | Set (i, a) -> Res ((result unit exn), protect (set sut i a))
```

The function `run` matches on the current command and calls the respective array function. The result constructor `Res` is provided by QCheck-STM and carries as fields the returned value and a pretty-printer for its respective type (e.g., `int` and `result char exn`). The function `protect` turns functions raising exceptions into functions returning values of type `result` (in OCaml all exceptions are part of the extensible variant type `exn`).

As the SUT value is mutable, its internal state will change in-place during the execution of `run`. The functional model of the SUT has to be updated separately. Therefore, a function `next_state` is defined:

Generated

```
let next_state cmd state = match cmd with
  | Length -> state
  | Get _ -> state
  | Set (i, a) ->
```

```
      if (0 <= i) && (i < state.size) then
        {
          size = state.size;
          contents = Sequence.set state.contents i a;
        }
      else state
```

Functions `length` and `get` do not mutate the array, and therefore they return the argument state unchanged. When setting a value at a particular index, the next state depends on if the index is within the array bounds. If so, the new state has the same size as the old one, and the contents are the same besides at the given index (`Sequence.set` is again part of the Gospel standard library). If the check fails, the underlying array stays unchanged. The individual cases within `next_state` are extracted from the *ensures* clauses of the respective function contract in Section 3. This is why each field that is marked as modified needs to provide a corresponding post-condition describing the model of the post-state.

Finally, a post-condition function is generated:

Generated

```
let ortac_postcond cmd state res =
  let new_state = next_state cmd state in
  match (cmd, res) with
    | Length, Res (_, i) ->
          if i = new_state.size then None
          else (* error report *)
    | Get i, Res (_, a) ->
        if (0 <= i) && (i < new_state.size) then
          (match a with
            | Ok a -> if a = Sequence.get new_state i then None
                else (* error report *)
            | _ -> (* error report *))
        else
          (match a with
            | Error (Invalid_argument _) -> None
            | _ -> (* error report *))
    (* further cases *)
```

The function `ortac_postcond` takes the current command, the current state, and the result after calling `run` as input, and returns an optional error report. As all post-conditions refer to the state after executing a given command, it first defines the new state by calling `next_state` (Gospel offers the *old* operator to refer to the pre-state, which we utilise in the specification of the `set` function). We only show the post-conditions for the `length` and `get` functions, the others follow analogously.

In the case of `length` we expect the returned integer to coincide with the `size` field of `new_state`. For `get` the returned value depends on if the given index was

within bounds. If it was, the returned character should be the same as the one taken from the functional model, and otherwise we expect an `Invalid_argument` exception.

We have left out the actual error reporting mechanism for brevity. The careful reader may have realised that there are in fact two different categories of post-conditions. In the case of `make` and `set`, the *ensures* clauses define the model after executing the respective function, which is used in `next_state`. For `length` and `get` they state a property of the return value, which can be checked in `ortac_postcond`.

5.3 Functions returning and consuming multiple SUTs

Thus far, all testable functions only took one SUT argument as input, and did not return a SUT value as an output (recall that the `make` function was temporarily omitted). Let us extend the available array API with another function `append` from the OCaml standard library's `Array` module. The function `append` takes two arrays, and returns a fresh array with the contents of both arguments appended. The specification is straightforward, when utilising the sequence concatenation operator `( ++ )` from the Gospel standard library:

```
val append : 'a t -> 'a t -> 'a t
(*@ t = append a b
    ensures t.size = a.size + b.size
    ensures t.contents = a.contents ++ b.contents *)
```

In order to allow testing functions that take multiple arguments of the SUT type, or likewise return a value of that type, the generated code is adapted:

Generated

```
type sut = char Array.t Stack.t
let init_sut () =
    let s = Stack.create () in
    for _ = 0 to max_sut - 1 do
        Stack.push (Array.make 16 'a') s
    done;
    s
type element = {
    size : integer;
    contents : char sequence;
}
type state = element list
let init_state = List.init max_sut (fun _ -> {
    size = integer_of_int 16;
    contents = Sequence.init (integer_of_int 16) (fun _ -> 'a');
})
```

The `type sut` is not a single SUT any more, it represents a (mutable) stack of SUT values (`Stack` is part of the OCaml standard library). Correspondingly, `type state` must describe a functional model of the SUT stack, as is done here through a list of model elements.

The variable `max_sut` is defined as the maximum number of SUT arguments ever needed by any single function of the API under test. For our running array example, `max_sut` is 2, as `append` expects two arguments of type `t`. This number can easily be determined during code generation. By starting with a SUT stack already filled with the maximum number of initial elements ever needed by any API call, each available function can immediately be used.

The generator for arbitrary commands now includes the full API:

Generated

```
let arb_cmd state = QCheck.make show_cmd Gen.(oneof [
  pure (fun s a -> Make (s, a)) <*> small_signed_int <*> char;
  pure Length;
  pure (fun i -> Get i) <*> int;
  pure (fun i a -> Set (i, a)) <*> int <*> char;
  pure Append
])
```

Notice, that the size argument `s` to `Make` is not provided by the `int` generator. Ortac/QCheck-STM uses a simple heuristic of classifying functions that produce a SUT but take no SUT argument as *initialisation functions*, i.e., functions that create new SUT instances. For these functions, any `int` generator is automatically changed to `small_signed_int` in order to keep the runtime of the generated test executable low.

For brevity, we will not show all the adapted code examples from the previous section again, but only describe the overall behaviour. When a function requires multiple SUT arguments, the required amount of SUTs is popped from the stack, the function is run, and the SUTs pushed back onto the stack in reverse order (this allows to capture changes to the argument SUTs in the post-condition). If a function returns a SUT, this value is pushed on the stack as well (so that the post-condition has access to it). This is illustrated in Figure 3 for the `append` function.

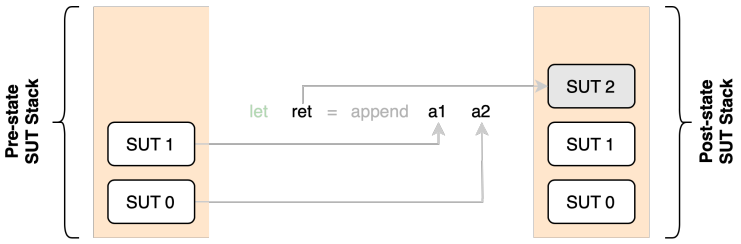


Figure 3. Function call with arguments and return value on the SUT stack.

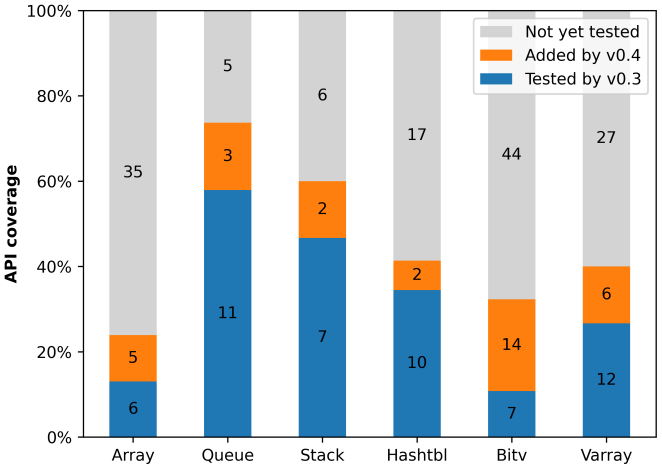


Figure 4. Number of API functions covered by the generated code.

6 Evaluation

We have used Ortac/QCheck-STM to test 6 OCaml modules as summarised in Figure 4, including the `Array`, `Stack`, `Queue`, and `Hashtbl` modules from the OCaml standard library. We have found 5 crashing bugs, and 1 function from the standard library needing a documentation fix (the source code for these tests, including the respective Gospel contracts, is available online [25]). Resolving the reported issues later revealed 2 additional unexpected exceptions in one of the tested modules. We will elaborate further on these findings later in this section.

Figure 4 displays the number of API functions covered by the generated testing code. With Ortac/QCheck-STM version 0.3 without the extension described in Section 5.3, this covers 11% (7/65) to 58% (11/19) of the module APIs. Ortac/QCheck-STM version 0.4 adds support for testing functions that take multiple SUT arguments and return new SUT values, as described in Section 5.3. Doing so increases the module API coverage further to 24% (11/46) to 74% (14/19). Interestingly, 7 out of 8 of the reported errors in this paper are all due to functions that were not testable with version 0.3. Ortac automatically skips a function for which no suitable Gospel annotation is provided. The biggest remaining contributor to untested functions is higher-order functions, such as `map` and `iter`. We expand on lifting this restriction in Section 8.

While no bugs were found within the standard library modules, one curiosity was discovered: The function `Hashtbl.create` is documented in the following way[7]:

[7] https://ocaml.org/manual/5.2/api/Hashtbl.html

```
val create : ?random:bool -> int -> ('a, 'b) t
(** [Hashtbl.create n] creates a new, empty hash table, with
   initial size [n]. For best results, [n] should be on the
   order of the expected number of elements that will be in
   the table. The table grows as needed, so [n] is just an
   initial guess. ... *)
```

A natural Gospel specification would therefore be to model the hash table type as an association list (similar to dictionaries in other languages). The specification of the `create` function could then look similar to the following:

```
type ('a, 'b) t
(*@ mutable model contents : ('a * 'b) list *)

val create : ?random:bool -> int -> ('a, 'b) t
(*@ h = create ?random size
    checks size >= 0
    ensures h.contents = [] *)
```

Running the test generated by Ortac/QCheck-STM reveals the following:

```
Gospel specification violation in function create

  File "hashtbl.mli", line 7, characters 11-20:
    size >= 0

when executing the following sequence of operations:

  [@@@ocaml.warning "-8"]
  open Hashtbl
  let protect f = try Ok (f ()) with e -> Error e
  let sut0 = create ~random:false 16
  let r = protect (fun () -> create true (-8))
  assert (match r with
            | Error (Invalid_argument _) -> true
            | _ -> false)
  (* returned Ok (<sut>) *)
```

It turns out, that the initial guess can be negative, in which case it has the same effect as providing zero. After reporting this, the documentation for the `create` function has been revised in OCaml 5.3[8].

[8] https://github.com/ocaml/ocaml/pull/13535

Outside of the standard library, 2 libraries from the official OCaml package repository have been tested as well, which revealed errors in both of them:

The `Bitv` library[9] is a mature, 25-year-old OCaml library implementing mutable bit-vectors of arbitrary but fixed length, with an API very similar to arrays. In three of its functions (`fill`, `sub`, and `blit`) Ortac/QCheck-STM tests discovered that their index-bound checking could lead to an integer overflow, resulting in a segmentation fault on at least one tested platform (the usage of unsafe language features or external code can lead to diverging behaviour on different operating systems or processor architectures). The issue has been reported and fixed[10] consequently. Furthermore, Ortac/QCheck-STM found two cases of unexpected exceptions being raised when trying to rotate a zero-length vector. The issue has been reported[11] and fixed[12] as well.

The `Varray` library[13] implements extensible arrays. It uses an intricate data structure [22] along with some unsafe OCaml tricks in order to obtain good amortised performance, of which many can lead to crashing programs if used incorrectly. Such a crashing scenario was found when starting from an empty array and then adding and removing an element from different ends of the array[14]. Anecdotally, tests of the `Varray` library have been part of the Ortac code-base almost from the initial release for internal testing. These discovered an initial bug early on[15]. The latest error, however, was only recently discovered when Ortac/QCheck-STM was extended to cover functions returning SUT values as described in Section 5.3.

Given the automated nature of the code generated by Ortac/QCheck-STM, it is particularly suited to be included in a *Continuous-Integration* (CI) pipeline, as is demonstrated by the CI used in Ortac's GitHub repository[16]. So far, all observed test runs have shown runtimes in the range of hundreds of milliseconds, even with the extension described in Section 5.3. Despite these tests still not reaching full API coverage, we believe this highlights the tool's usability in CI.

7 Related Work

Fundamental ideas within modern verification can be traced back to Hoare. This is the case for invariants and pre- and post-conditions as found in Hoare logic triples [23] as well as proving an implementation correct with respect to a model [24]. Meyer later put the concepts into use in the *design-by-contract* methodology of the Eiffel programming language [31]. The `require(s)` and `ensure(s)` keywords of modern specification languages such as Gospel thus have roots in Eiffel.

[9] https://github.com/backtracking/bitv
[10] https://github.com/backtracking/bitv/pull/32
[11] https://github.com/backtracking/bitv/issues/33
[12] https://github.com/backtracking/bitv/commit/f30e7a8
[13] https://github.com/art-w/varray
[14] https://github.com/art-w/varray/issues/2
[15] https://discuss.ocaml.org/t/ann-varray-0-2/13492
[16] https://github.com/ocaml-gospel/ortac

The Gospel specification language for OCaml follows a line of specification languages such as the Java Modeling Language (JML) for Java programs [29], the ANSI/ISO C Specification Language (ACSL) for C programs [6], and Spec# for C# programs [5].

Software engineering tools to validate program specifications can roughly be divided into two categories: One group of tools works by dynamic *runtime assertion checking*, whereas another group of tools performs static verification. The JML-consuming ESC/Java tool [21] targets both of these categories. The ACSL-consuming Frama-C tool [15] targets the latter. The Gospel-consuming Ortac tool targets the former, but other Gospel tools (under development) [39] target the latter.

Whereas the above specification languages and verification tools have been developed for existing programming languages a posteriori, a newer class of programming languages have verification fundamentally built in from the beginning. This is the case for Lean [33], Why3 [19], F* [43], and Dafny [30], among others.

Another approach to formally verified software development is *correct-by-construction* techniques. Filliâtre et al. [17] describe a development process that builds upon both Gospel and Why3 as part of the VOCAL [10] project. The user provides an OCaml module signature with Gospel annotations, which is translated to a WhyML specification. The module is then implemented in WhyML, proven correct with respect to the translated specification by Why3, and automatically translated back to OCaml code.

The term *property-based testing* was introduced by Fink and Bishop [20] originally. It became popular with QuickCheck, an embedded domain-specific language for the functional programming language Haskell [12], and has since been ported to numerous other languages, including OCaml [46]. QuickCheck introduced modular combinators for building up generators of complex test inputs, how each input is tested on properties in the form of Boolean-valued functions, and test input shrinking when finding a counterexample. Whereas the original QuickCheck formulation targeted purely functional code, in follow-up work Claessen and Hughes presented extensions to target monadic, effectful Haskell code, including the idea of model-based testing [13]. While QuickCheck was primarily conceived as a testing tool, the method has since been introduced in various interactive theorem provers in order to quickly provide counterexamples during the proof development process. Examples of this can be found in Isabelle [7,8], Coq [37], and Agda [16].

With roots in *model-based testing* from outside the functional programming language community [45], the Gast framework for the Clean programming language offered *state machines* to specify the intended behaviour of stateful reactive systems [27]. The design of the state-machine framework for the commercial Erlang QuickCheck [3,26] has since influenced framework ports for other languages, e.g., for Scala [34] and QCheck-STM for OCaml as used in this work [32]. The state-machine approach furthermore extends to testing stateful code for race conditions under concurrent usage [14].

In an impressive feat, Arts et al. [4] have developed state-machine models of the AUTOSAR specification to test automotive software. The range of defects found in doing so underlines the usefulness of the approach. Other successful QuickCheck applications include testing of telecommunication software [3], data structures [2], election software [28], computational geometry algorithms [41], compilers [36], and run-time systems [32].

8 Conclusion and Future Work

In this paper, we have presented Ortac/QCheck-STM, a tool that consumes behavioural contracts expressed in the Gospel specification language, and generates code to automatically test a given OCaml module against a functional reference model derived from these contracts. Despite being a relatively young tool with the first version released in October 2023, Ortac/QCheck-STM has already proven useful in finding bugs in established OCaml libraries, as well as pointing out inconsistencies in documentation. We expect to find more errors with it as we continue annotating more libraries with Gospel contracts.

While this paper focuses on Ortac/QCheck-STM, previous work [42] has investigated verification of OCaml code by leveraging both static and dynamic verification tools for Gospel, including Ortac. In that work, the authors remark on various limitations of Ortac, of which many have been lifted (including, for example, the verification of functions taking multiple SUT arguments, or returning SUT values). However, Ortac/QCheck-STM still has various limitations, which we would like to lift in future work:

Given its nature as a dynamic verification tool, Ortac is inherently restricted to the executable fragment of the Gospel specification language. At times, this results in contracts that are not as natural as their purely logical counterpart. By extending the accepted syntactic forms, contracts could be written in ways that would make them more amenable to other forms of verification (and their respective tools) as well.

By design, the Gospel specification language implicitly requires that mutable arguments do not alias, i.e., that they occupy separate memory locations in the OCaml heap [9]. Therefore, Ortac/QCheck-STM does not attempt to generate calls with aliased SUTs. Once Gospel is enhanced to express aliasing properties, we would like to extend Ortac/QCheck-STM accordingly to exercise such specifications.

The majority of functions in Figure 4 currently not covered by the generated code comprise idiomatic higher-order functions such as `map`, `fold`, and `iter`. Gospel currently does not have a way of specifying effectful function arguments, whereas it is possible to specify the behaviour of functions accepting pure function arguments [9]. As a first step, Ortac should be able to lift a restriction for the latter, e.g., using Claessen's approach to function generation [11]. Secondly, once Gospel has set on a way for specifying effectful function arguments, we hope to extend Ortac to cover such API calls as well.

Currently, preconditions introduced by *requires* clauses are not considered during the generation of arbitrary command lists on which to test the SUT and the model. During the execution of each test case, if a given pre-state does not fulfil the stated pre-condition for a particular command, it is simply skipped. The random generator could be extended to take *requires* clauses into account while generating random sequences of commands, which would increase the efficiency of the tool.

QCheck-STM was originally developed to test the new multicore runtime arriving with OCaml 5 [32]. It can therefore also produce parallel sequences of random API calls, and test if the observed behaviour is sequentially consistent by reconciling each run with a sequential execution of a given model. By extending Ortac to use the parallel test generator, it would be possible to also test concurrent data-structures (as for example done by Artho et al. [1]).

Acknowledgments. This work was funded by ANR grant ANR-22-CE48-0013 and Tarides.

References

1. Artho, C., Gros, Q., Rousset, G., Banzai, K., Ma, L., Kitamura, T., Hagiya, M., Tanabe, Y., Yamamoto, M.: Model-Based API Testing of Apache ZooKeeper. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 288–298 (2017). https://doi.org/10.1109/ICST.2017.33
2. Arts, T., Castro, L.M., Hughes, J.: Testing Erlang data types with Quviq QuickCheck. In: Proceedings of the 7th ACM SIGPLAN Workshop on Erlang, ERLANG'08. pp. 1–8 (2008). https://doi.org/10.1145/1411273.1411275
3. Arts, T., Hughes, J., Johansson, J., Wiger, U.T.: Testing telecoms software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang. pp. 2–10 (2006). https://doi.org/10.1145/1159789.1159792
4. Arts, T., Hughes, J., Norell, U., Svensson, H.: Testing AUTOSAR software with QuickCheck. In: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops. pp. 1–4 (2015). https://doi.org/10.1109/ICSTW.2015.7107466
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. pp. 49–69 (2005). https://doi.org/10.1007/978-3-540-30569-9_3
6. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/download/acsl.pdf
7. Berghofer, S., Nipkow, T.: Random Testing in Isabelle/HOL. In: SEFM. vol. 4, pp. 230–239 (2004). https://doi.org/10.1109/SEFM.2004.10049
8. Bulwahn, L.: The New Quickcheck for Isabelle. In: Certified Programs and Proofs. pp. 92–108 (2012). https://doi.org/10.1007/978-3-642-35308-6_10
9. Charguéraud, A., Filliâtre, J.C., Lourenço, C., Pereira, M.: GOSPEL - Providing OCaml with a Formal Specification Language. In: FM 2019 - 23rd International Symposium on Formal Methods (2019). https://doi.org/10.1007/978-3-030-30942-8_29
10. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL – A Verified OCAml Library (Sep 2017), https://inria.hal.science/hal-01561094, ML Family Workshop 2017

11. Claessen, K.: Shrinking and showing functions: *(functional pearl)*. In: Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012. pp. 73–80 (2012). https://doi.org/10.1145/2364506.2364516
12. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00. pp. 268–279 (2000). https://doi.org/10.1145/351240.351266
13. Claessen, K., Hughes, J.: Testing monadic code with QuickCheck. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002. pp. 65–77 (2002). https://doi.org/10.1145/581690.581696
14. Claessen, K., Palka, M.H., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.T.: Finding race conditions in Erlang with QuickCheck and PULSE. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009. pp. 149–160 (2009). https://doi.org/10.1145/1596550.1596574
15. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Software Engineering and Formal Methods. pp. 233–247 (2012). https://doi.org/10.1007/978-3-642-33826-7_16
16. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining Testing and Proving in Dependent Type Theory. In: Basin, D., Wolff, B. (eds.) Theorem Proving in Higher Order Logics. pp. 188–203 (2003). https://doi.org/10.1007/10930755_12
17. Filliâtre, J.C., Gondelman, L., Lourenço, C., Paskevich, A., Pereira, M., Melo de Sousa, S., Walch, A.: A Toolchain to Produce Verified OCaml Libraries (Jan 2020), https://hal.science/hal-01783851
18. Filliâtre, J.C., Pascutto, C.: Ortac: Runtime Assertion Checking for OCaml (Tool Paper). In: Runtime Verification: 21st International Conference, RV 2021, Proceedings. pp. 244–253 (2021). https://doi.org/10.1007/978-3-030-88494-9_13
19. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128 (2013). https://doi.org/10.1007/978-3-642-37036-6_8
20. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. SIGSOFT Softw. Eng. Notes **22**(4), 74–80 (1997). https://doi.org/10.1145/263244.263267
21. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. pp. 234–245. PLDI'02 (2002). https://doi.org/10.1145/512529.512558
22. Goodrich, M.T., Kloss, J.G.: Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences. In: Algorithms and Data Structures. pp. 205–216 (1999). https://doi.org/10.1007/3-540-48447-7_21
23. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
24. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**, 271–281 (1972). https://doi.org/10.1007/BF00289507
25. Huber, N., Osborne, N., Hym, S., Midtgaard, J.: Dynamic Verification of OCaml Software with Gospel and Ortac/QCheck-STM - Software Artifact (Oct 2024). https://doi.org/10.5281/zenodo.13988146
26. Hughes, J.: Software testing with QuickCheck. In: Central European Functional Programming School - Third Summer School, CEFP 2009, Revised Selected Lec-

tures. Lecture Notes in Computer Science, vol. 6299, pp. 183–223 (2009). https://doi.org/10.1007/978-3-642-17685-2_6
27. Koopman, P.W.M., Plasmeijer, R.: Testing reactive systems with GAST. In: Revised Selected Papers from the Fourth Symposium on Trends in Functional Programming, TFP 2003. vol. 4, pp. 111–129 (2003)
28. Koopman, P.W.M., Plasmeijer, R.: Testing with functional reference implementations. In: Trends in Functional Programming - 11th International Symposium, TFP 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6546, pp. 134–149 (2010). https://doi.org/10.1007/978-3-642-22941-1_9
29. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (May 2006). https://doi.org/10.1145/1127878.1127884
30. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370 (2010). https://doi.org/10.1007/978-3-642-17511-4_20
31. Mandrioli, D., Meyer, B., et al.: Advances in object oriented software engineering. Prentice Hall (1992)
32. Midtgaard, J., Nicole, O., Osborne, N.: Multicoretests — Parallel testing libraries for OCaml 5.0. In: OCaml Users and Developers Workshop (2022), https://github.com/ocaml-multicore/multicoretests
33. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Automated Deduction - CADE-25. pp. 378–388 (2015). https://doi.org/10.1007/978-3-319-21401-6_26
34. Nilsson, R.: ScalaCheck: The Definitive Guide. Artima (2014)
35. Osborne, N., Pascutto, C.: Leveraging Formal Specifications to Generate Fuzzing Suites. In: OCaml Users and Developers Workshop (2021), https://inria.hal.science/hal-03328646
36. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 91–97. AST'11 (2011). https://doi.org/10.1145/1982595.1982615
37. Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational Property-Based Testing. In: ITP 2015 - 6th conference on Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 9236 (2015). https://doi.org/10.1007/978-3-319-22102-1_22
38. Pascutto, C.: Runtime verification of OCaml programs. Theses, Université Paris-Saclay (Oct 2023), https://theses.hal.science/tel-04696708
39. Pereira, M., Ravara, A.: Cameleer: A deductive verification tool for OCaml. In: Computer Aided Verification. pp. 677–689 (2021). https://doi.org/10.1007/978-3-030-81688-9_31
40. Pottier, F.: Strong Automated Testing of OCaml Libraries. In: JFLA 2021 - 32es Journées Francophones des Langages Applicatifs (Feb 2021), https://inria.hal.science/hal-03049511
41. Sergey, I.: Experience report: growing and shrinking polygons for random testing of computational geometry algorithms. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016. pp. 193–199 (2016). https://doi.org/10.1145/2951913.2951927
42. Soares, T.L., Chirica, I., Pereira, M.: Static and dynamic verification of OCaml programs: The Gospel ecosystem (extended version) (2024), https://arxiv.org/abs/2407.17289

43. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016. pp. 256–270 (2016). https://doi.org/10.1145/2837614.2837655
44. The Coq development team: The Coq proof assistant (Jun 2024). https://doi.org/10.5281/zenodo.11551307
45. Tretmans, J.: Testing concurrent systems: A formal approach. In: CONCUR '99: Concurrency Theory, 10th International Conference, Proceedings. Lecture Notes in Computer Science, vol. 1664, pp. 46–65 (1999). https://doi.org/10.1007/3-540-48320-9_6
46. Wikipedia: QuickCheck Wikipedia page (Accessed: 2024-10-06), https://en.wikipedia.org/wiki/QuickCheck

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

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

Weakly acyclic diagrams: A data structure for infinite-state symbolic verification*

Michael Blondin[1(✉)], Michaël Cadilhac[2], Xin-Yi Cui[3], Philipp Czerner[3], Javier Esparza[3], and Jakob Schulz[3]

[1] Université de Sherbrooke, Sherbrooke, Canada
michael.blondin@usherbrooke.ca
[2] DePaul University, Chicago, IL, USA
[3] Technical University of Munich, Munich, Germany

Abstract. Ordered binary decision diagrams (OBDDs) are a fundamental data structure for the manipulation of Boolean functions, with strong applications to finite-state symbolic model checking. OBDDs allow for efficient algorithms using top-down dynamic programming. From an automata-theoretic perspective, OBDDs essentially are minimal deterministic finite automata recognizing languages whose words have a fixed length (the arity of the Boolean function). We introduce weakly acyclic diagrams (WADs), a generalization of OBDDs that maintains their algorithmic advantages, but can also represent infinite languages. We develop the theory of WADs and show that they can be used for symbolic model checking of various models of infinite-state systems.

Keywords: binary decision diagrams · weakly acyclic languages · partially ordered automata · well-structured transition systems.

1 Introduction

Ordered binary decision diagrams (OBDDs) are a fundamental data structure for the manipulation of Boolean functions [8,13], widely used to support symbolic model checking of finite-state systems [14,16]. In this approach to model checking, sets of configurations of a system are encoded as Boolean functions of fixed arity, say n, and the transition relation is encoded as a Boolean function of arity $2n$. After fixing a total order on the Boolean variables, the OBDD representation of a Boolean function is unique up to isomorphism, and one can implement different model checking algorithms on top of an OBDD library.

The fundamental operations on sets and relations used by symbolic model-checking algorithms are—besides union, intersection, and complement of sets and relations—the *pre* and *post* operations that return the immediate predecessors or successors of a given set of configurations with respect to the transition relation. All of them are efficiently implemented on OBDDs using top-down dynamic programming: a call to an operation for functions of arity n invokes one or more

* M. Blondin was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 23–42, 2025.
https://doi.org/10.1007/978-3-031-90660-2_2

calls to the same or other operations for functions of arity $n-1$; intermediate results are stored using memoization [9,14,16].

OBDDs can be presented in automata-theoretic terms. A Boolean function $f(x_1,\ldots,x_n)$ with order $x_1 < \cdots < x_n$ is representable by the language $L_f := \{b_1 \cdots b_n \in \{0,1\}^n : f(b_1,\ldots,b_n) = \mathit{true}\}$. The OBDD for f with this variable order is very close to the minimal DFA (MDFA) for L_f.[4] From this point of view, OBDDs are just MDFAs recognizing *fixed-length languages*—languages whose words have all the same length—and operations on OBDDs are just operations on fixed-length languages canonically represented by their unique MDFAs.

The automata-theoretic view of OBDDs immediately suggests a generalization from fixed-length languages to arbitrary regular languages: canonically represent a regular language by its unique MDFA, and manipulate MDFAs using automata-theoretic operations. (This is for example the approach of the tool MONA [28].) However, in this generalization crucial advantages of OBDD algorithms get lost. Consider the implementation of the *post* operation that, given a MDFA $\mathcal{A}$ recognizing a set of configurations and the MDFA $\mathcal{T}$ recognizing the transition relation, computes the MDFA for the set of immediate successors of $L(\mathcal{A})$ with respect to $L(\mathcal{T})$ (see *e.g.* [20, Chap. 5]). The algorithm proceeds in three steps. First, it constructs an NFA $\mathcal{A}'$ recognizing the set of immediate successors; then, it applies the powerset construction to determinize $\mathcal{A}'$ into a DFA $\mathcal{A}''$; finally, it minimizes $\mathcal{A}''$ to yield the final MDFA $\mathcal{A}'''$. The crucial point is that the intermediate DFA $\mathcal{A}''$ can be *exponentially larger* than $\mathcal{A}'''$, and so, in the worst case, *post* uses exponential space in the size of $\mathcal{A}'''$, the final output. This is not the case for the OBDD implementation for fixed-length languages, which constructs $\mathcal{A}'''$ *directly* from $\mathcal{A}$ and $\mathcal{T}$, using only space $\mathcal{O}(|\mathcal{A}'''|)$.

This observation raises the question of whether OBDDs can be generalized beyond fixed-length languages *while maintaining the advantages of OBDD algorithms*. We answer it in the affirmative. We introduce *weakly acyclic diagrams (WADs)*, a generalization of OBDDs for the representation of *weakly acyclic languages*. A language is weakly acyclic if it is recognized by a weakly acyclic DFA, and a DFA is weakly acyclic if every simple cycle of its state-transition diagram is a self-loop. So, for example, the language a^*ba^* is weakly acyclic. Note that, contrary to fixed-length languages, weakly acyclic languages can be infinite.

We develop the theory of WADs and in particular show that small modifications to the top-down dynamic programming algorithms for OBDDs generalize them to WADs. We also present a first application of WADs to symbolic infinite-state model checking. A fundamental technique in this area is the backwards reachability algorithm for well-structured transition systems [23,3]. The algorithm computes the set of all predecessors of a given upward-closed set of configurations (with respect to the partial order making the system well-structured). For that, it iteratively computes the immediate predecessors of the current set of configurations, until a fixpoint is reached. Well-structuredness ensures that all intermediate sets remain upward-closed, and termination. We show that for

[4] See *e.g.* [20, Chap. 6], which even introduces a slight generalization of DFAs such that the OBDD *is* the unique minimal deterministic generalized automaton for L_f.

many well-structured transition systems, including lossy channel systems [4,6], Petri nets [31], and broadcast protocols [21], upward-closed sets of configurations can be easily encoded as weakly acyclic languages, and so the backwards reachability algorithm can be implemented with WADs as data structure.

We implemented our algorithms in WADL, a prototype library for weakly acyclic diagrams. We conduct experiments on over 200 inputs for the backwards reachability algorithm, including instances of lossy channel systems, Petri nets, and broadcast protocols. We compare with some established tools using dedicated data structures and show that our generic approach is competitive.

Related work. Weakly acyclic automata have been studied extensively. However, past work has focused on algebraic, logical, and computational complexity questions, not on their algorithmics as a data structure.

In [15], Brzozowski and Fich showed that weakly acyclic automata, called *partially ordered automata* there, capture the so-called $\mathcal{R}$-trivial languages. They credit Eilenberg for earlier work on $\mathcal{R}$-trivial monoids [19]. Weakly acyclic automata have also been called *extensive automata*, *e.g.* in [32].

Weak acyclicity has also been defined in terms of *nondeterministic* automata. As we shall see, one such definition yields a model as expressive as the deterministic one, while another one is more expressive. These models have sometimes been called *restricted partially ordered NFAs (rpoNFAs)* and *partially ordered NFAs (poNFAs)*. Schwentick, Thérien and Vollmer have shown that poNFAs characterize level 3/2 of the Straubing-Thérien hierarchy [34]. Bouajjani *et al.* characterized the class of languages accepted by poNFAs as languages closed under permutation rewriting, *i.e.*, languages described by so-called *alphabetic pattern constraints* [12]. As mentioned, *e.g.*, in the introduction of [30], further equivalences are known in first-order logic.

More recently, Krötzsch, Masopust and Thomazo have studied the computational complexity of basic questions for several types of partially ordered automata [29,30], and Ryzhikov and Wolf have studied the complexity of problems for upper triangular Boolean matrices [33]; these matrices arise naturally from the transformation monoid of partially ordered automata.

In the context of symbolic verification, there is work on *dedicated* data structures for the representation of configuration sets for *specific* systems. In particular, Delzanno *et al.* introduced *covering sharing trees* to represent upward-closed sets of Petri net markings [18], which were later extended to *interval sharing trees* by Ganty *et al.* [24]. Boigelot and Godefroid studied *queue-content decision diagrams* to analyze channel systems [11].

Structure of the paper. In Section 2, we introduce and study weakly acyclic languages. In Sections 3 and 4, we introduce weakly acyclic diagrams and explain how to implement many operations. In Section 5, we explain how our framework can be used for infinite-state symbolic verification, and we then report on experimental results in Section 6. We conclude in Section 7. Some proofs and experimental results are deferred to the appendix of the full version [10].

2 Weakly acyclic languages

We assume familiarity with basic automata theory. Let us recall some notions. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton (DFA). We write $\mathrm{L}(\mathcal{A})$ to denote the language accepted by $\mathcal{A}$. The language accepted by a state $q \in Q$, denoted $\mathrm{L}(q)$, is the language accepted by the DFA $(Q, \Sigma, \delta, q, F)$. We write Σ^+ to denote the set of all nonempty words over alphabet Σ.

For every word $w \in \Sigma^*$, we define $\alpha(w)$ as the set of letters that occur within w, *e.g.* $\alpha(baacac) = \{a, b, c\}$. The *residual* of a language $L \in \Sigma^*$ with respect to a word $u \in \Sigma^*$ is $L^u := \{v \in \Sigma^* : uv \in L\}$. Recall that a language is regular iff it has finitely many residuals. Moreover, every regular language has a unique minimal DFA. The states of a minimal DFA $\mathcal{A}$ accept distinct residuals of $\mathrm{L}(\mathcal{A})$, and each residual of $\mathrm{L}(\mathcal{A})$ is accepted by a state of $\mathcal{A}$. For example, let $L \subseteq \{a, b\}^*$ be the language described by $(ab^* + b + \varepsilon)ab^*$. The minimal DFA for L is depicted in Figure 1. Its residuals are L^ε, L^a, L^b, L^{aa} and L^{aaa}.

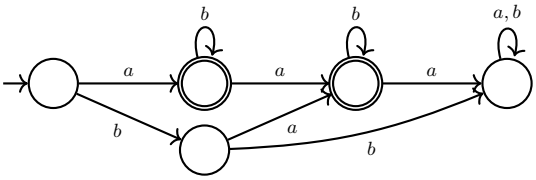


Fig. 1. Example of a minimal DFA (whose language is weakly acyclic).

We say that a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is *weakly acyclic* if for every $q \in Q$, $w \in \Sigma^+$ and $a \in \alpha(w)$, it is the case that $\delta(q, w) = q$ implies $\delta(q, a) = q$. For example, Figure 1 depicts a weakly acyclic DFA. Equivalent definitions of weak acyclicity are as follows:

- the binary relation $\preceq$ over Q given by "$q \preceq q'$ if $\delta(q, w) = q'$ for some word w" is a partial order;
- each strongly connected component of the underlying directed graph of $\mathcal{A}$ contains a single state; and
- the underlying directed graph of $\mathcal{A}$ does not have any cycle beyond self-loops.

Analogously, a nondeterministic finite automaton (NFA) $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is *weakly acyclic* if $q \in \delta(q, w)$ implies $\delta(q, a) = \{q\}$ for all $a \in \alpha(w)$. In other words, the underlying directed graph does not contain any simple cycle beyond self-loops, and nondeterminism with a letter a can only occur from states without self-loops labeled by a.

We make some observations whose proofs appear in the appendix of the long version of this work. Let us observe that the powerset construction (determinization), and DFA minimization, both preserve weak acyclicity.

Proposition 1. *Applying the powerset construction to a weakly acyclic NFA yields a weakly acyclic DFA.*

Proposition 2. *Let $\mathcal{A}$ be a weakly acyclic DFA. The minimal DFA that accepts* $\mathrm{L}(\mathcal{A})$ *is also weakly acyclic.*

We say that a language is *weakly acyclic* if it is accepted by a weakly acyclic automaton $\mathcal{A}$. By the above propositions, this means that $\mathcal{A}$ can interchangeably be an NFA, a DFA, or a minimal DFA. The lemma below yields a characterization of weakly acyclic languages in terms of residuals rather than automata.

Lemma 1. *A regular language $L \subseteq \Sigma^*$ is weakly acyclic iff for all $u \in \Sigma^*$ and $v \in \Sigma^+$ it is the case that $L^u = L^{uv}$ implies $L^u = L^{uw}$ for every $w \in \alpha(v)^*$.*

It can also be shown that weakly acyclic languages are precisely the languages described by such *weakly acyclic expressions*: $r ::= \emptyset \mid \Gamma^* \mid \Lambda^* a r \mid r + r$ where $\Gamma, \Lambda \subseteq \Sigma$ and $a \in \Sigma \setminus \Lambda$ (see the appendix of the full version). Note that these expressions are essentially "$\mathcal{R}$-expressions", which are known to be equivalent to weakly acyclic languages via algebraic automata theory. Moreover, [29, Thm. 7], shows that weaky acyclic DFAs and NFAs are equivalent, and the proof is very similar our conversion from weakly acyclic NFAs to expressions given in the appendix of the full version.

Complementing a weakly acyclic DFA preserves weak acyclicity since no cycle is created. Moreover, weakly acyclic regular expressions allow for union. Thus:

Proposition 3. *Weakly acyclic languages are closed under complementation, union and intersection.*

3 A data structure for weakly acyclic languages

Note that if a language L is weakly acyclic, then so is L^a for all $a \in \Sigma$. From this simple observation, one can imagine an infinite deterministic automaton where each state is a weakly acyclic language L, and each a-transition leads to L^a. Let us define this "master automaton" formally.

Definition 1. *The* master automaton *over alphabet Σ is the tuple $\mathcal{M} = (Q_\mathcal{M}, \Sigma, \delta_\mathcal{M}, F_\mathcal{M})$, where*

- *$Q_\mathcal{M}$ is is the set of all weakly acyclic languages over Σ;*
- *$\delta_\mathcal{M} \colon Q_\mathcal{M} \times \Sigma \to Q_\mathcal{M}$ is given by $\delta_\mathcal{M}(L, a) := L^a$ for all $L \in Q_\mathcal{M}$ and $a \in \Sigma$;*
- *$L \in F_\mathcal{M}$ iff $\varepsilon \in L$.*

Given weakly acyclic languages K and L, let $K \preceq L$ denote that $K = L^w$ for some word w. An immediate consequence of the definition of weakly acyclic DFAs is that $\preceq$ is a partial order. The minimal elements of $\preceq$ satisfy the equation $L = L^a$ for every letter a. The equation has exactly two solutions: $L = \emptyset$ and $L = \Sigma^*$. Moreover, we can easily show by structural induction that $\preceq$ has no infinite descending chains. This allows to reason about a weakly acyclic language recursively through its residuals until reaching $\emptyset$ or Σ^*.

Proposition 4. *Let L be a weakly acyclic language. The language accepted from the state L of the master automaton is L.*

By Proposition 4, we can look at the master automaton as a structure containing DFAs recognizing all the weakly acyclic languages. To make this precise, each weakly acyclic language L determines a DFA $\mathcal{A}_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$ as follows: Q_L is the set of states of the master automaton reachable from the state L, q_{0L} is the state L, δ_L is the projection of $\delta_{\mathcal{M}}$ onto Q_L, and $F_L \coloneqq F_{\mathcal{M}} \cap Q_L$. It is easy to show that $\mathcal{A}_L$ is the *minimal* DFA recognizing L.

Proposition 5. *For every weakly acyclic language L, automaton $\mathcal{A}_L$ is the minimal DFA recognizing L.*

Proposition 5 allows us to define a data structure for representing finite sets of weakly acyclic languages. Loosely speaking, the structure representing the languages $\mathcal{L} = \{L_1, \ldots, L_n\}$ is the fragment of the master automaton containing the states recognizing $L_1, \ldots, L_n$ and their descendants. It is a DFA with multiple initial states which we call the *weakly acyclic diagram for* $\mathcal{L}$. Formally:

Definition 2. *Let $\mathcal{L} = \{L_1, \ldots, L_n\}$ be weakly acyclic languages over Σ. The* weakly acyclic diagram *$\mathcal{A}_{\mathcal{L}}$ is the tuple $(Q_{\mathcal{L}}, \Sigma, \delta_{\mathcal{L}}, Q_{0\mathcal{L}}, F_{\mathcal{L}})$ where $Q_{\mathcal{L}}$ is the subset of states of the master automaton reachable from at least one of $L_1, \ldots, L_n$; $Q_{0\mathcal{L}} \coloneqq \{L_1, \ldots, L_n\}$; $\delta_{\mathcal{L}}$ is the projection of δ_M onto $Q_{\mathcal{L}}$; and $F_{\mathcal{L}} = F_{\mathcal{M}} \cap Q_{\mathcal{L}}$.*

In order to manipulate weakly acyclic diagrams, we represent them as tables of nodes. Let us fix $\Sigma = \{a_1, a_2, \ldots, a_m\}$. A *node* is a triple $q\colon (s, b)$ where

- q is the node *identifier*;
- $s = (q_1, q_2, \ldots, q_m)$ is the *successor tuple* of the node, where each q_i is either a node identifier or the special symbol SELF; and
- $b \in \{0, 1\}$ is the *acceptance flag* that indicates whether q is accepting or not.

We write q^{a_i} to denote q_i. We write $\mathrm{L}(s, b)$ for the language defined recursively as follows, where $i \in [1..m]$ and $w \in \Sigma^*$:

$$\begin{aligned} \varepsilon \in \mathrm{L}(q) &\iff b = 1, \\ a_i w \in \mathrm{L}(q) &\iff ((q^{a_i} = \textsc{self} \wedge w \in \mathrm{L}(q)) \vee (q^{a_i} \neq \textsc{self} \wedge w \in \mathrm{L}(q^{a_i}))) . \end{aligned}$$

For every node $q\colon (s, b)$, we write $\mathrm{L}(q)$ to denote $\mathrm{L}(s, b)$.

We denote the identifiers of the nodes for languages $\emptyset$ and Σ^* by $q_\emptyset$ and q_{Σ^*}, respectively, *i.e.*, $q_\emptyset\colon ((\textsc{self}, \ldots, \textsc{self}), 0)$ and $q_{\Sigma^*}\colon ((\textsc{self}, \ldots, \textsc{self}), 1)$.

Example 1. Let K and L be the languages over $\Sigma = \{a, b, c\}$ described by the regular expressions $a^*(\varepsilon + b^+ a\Sigma^*)$ and $ab^*a\Sigma^* + (b + c)\Sigma^*$. These languages are represented by the table depicted on the left of Figure 2, which corresponds to the diagram on the right. It is readily seen that $\mathrm{L}(q_3) = K$ and $\mathrm{L}(q_4) = L$. □

Note that there is a risk that two distinct nodes represent the same language. For example, suppose $\Sigma = \{a, b\}$ and that we create a node $p\colon ((\textsc{self}, q_{\Sigma^*}), 1)$. We obtain the following table, where $\mathrm{L}(p) = \mathrm{L}(q_{\Sigma^*}) = \{a, b\}^*$:

identifier	succ. tuple			flag
	a	b	c	
q_4	q_2	q_{Σ^*}	q_{Σ^*}	0
q_3	SELF	q_2	$q_\emptyset$	1
q_2	q_{Σ^*}	SELF	$q_\emptyset$	0
q_{Σ^*}	SELF	SELF	SELF	1
$q_\emptyset$	SELF	SELF	SELF	0

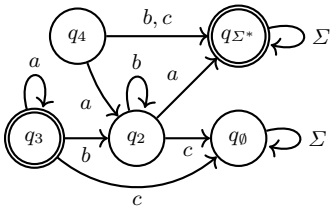


Fig. 2. Example of the representation of weakly acyclic languages.

identifier	succ. tuple		flag
	a	b	
p	SELF	q_{Σ^*}	1
q_{Σ^*}	SELF	SELF	1

We will avoid this: we will maintain a table of nodes such that the language of distinct nodes is distinct. The procedure `make` of Algorithm 1 serves this purpose. Given a successor tuple s and an acceptance flag b, it first checks whether there is already an entry for language $\mathrm{L}(s, b)$, and if not it creates one with a new identifier. The identifiers created by `make` are generated in increasing order. We assume that the table initially contains $q_\emptyset$ and q_{Σ^*} as the first two identifiers.

Algorithm 1: Algorithm to maintain nodes.

Input: $s = (q_1, \ldots, q_m)$ and $b \in \{0, 1\}$ where $q_i = \text{SELF}$ or an existing identifier
Result: unique identifier q with language $\mathrm{L}(s, b)$ and $q \notin s$

```
1  make(s, b):
2      if the table contains q: (s, b) then return q
3      else
4          q' ← max{r ∈ s : r ≠ SELF}
5          s' ← s[q'/SELF]
6          if the table contains q: (s', b) then return q
7          else
8              q ← next fresh identifier
9              add q: (s, b) to the table
10             return q
```

Let us explain how `make` checks whether a node must be created. Given a tuple $s = (q_1, q_2, \ldots, q_m)$, we write $s[r/r']$ to denote the tuple obtained from s by replacing each occurrence $q_i = r$ by r'. As already explained, when trying to add (s, b) to the table, there might already be an entry $q\colon (s', b')$ with $\mathrm{L}(s', b') = \mathrm{L}(s, b)$. It can be shown that this happens iff $b' = b$ and there exists $q' \in s$ such that $s' = s[q'/\text{SELF}]$ (see the proof of Proposition 6 below in the full version [10]). In fact, it can be shown that such a q' must be the maximal identifier of s; hence,

there is no need to check all elements of s. For example, in the aforementioned problematic case, we have $s = (\text{SELF}, q_{\Sigma^*})$ and $s[q_{\Sigma^*}/\text{SELF}] = (\text{SELF}, \text{SELF})$ which is already in the table, so no node is created.

The amortized time of `make` belongs to $\mathcal{O}(|\Sigma|)$. Further, it works as intended:

Proposition 6. *A table obtained from successive calls to Algorithm 1 does not contain nodes $q \neq q'$ such that $\mathrm{L}(q) = \mathrm{L}(q')$.*

4 Operations on weakly acyclic languages

4.1 Unary and binary operations

Algorithms 2 and 3 describe recursive procedures that respectively complement a weakly acyclic language, and intersect two weakly acyclic languages. They are *very similar* to the classical algorithms for negation and conjunction in OBDDs (see [20, Chap. 6, Algorithms 27 and 26]); they essentially only differ in the usage of SELF. For this reason, their correcteness and time complexity are established with the same arguments. Let us expand for readers less familiar with OBDDs.

Procedure `comp` terminates because on each recursive call, q^a is a smaller identifier than q. Procedure `inter` terminates because on each recursive call, either $p^a \neq \text{SELF}$ and p^a is a smaller identifier than p, or likewise for q^a and q.

Algorithm 2: Complement.

Input: identifier q
Result: id. $r : \mathrm{L}(r) = \overline{\mathrm{L}(q)}$

```
1  comp(q):
2      if q = q_∅ then
3          return q_{Σ*}
4      else if q = q_{Σ*} then
5          return q_∅
6      else
7          for a ∈ Σ do
8              if q^a = SELF then
9                  s_a ← SELF
10             else
11                 s_a ← comp(q^a)
12         return make(s, ¬flag(q))
```

Algorithm 3: Intersection.

Input: identifiers p and q
Result: id. $r : \mathrm{L}(r) = \mathrm{L}(p) \cap \mathrm{L}(q)$

```
1  inter(p, q):
2      if p = q_∅ or q = q_∅ then
3          return q_∅
4      else if p = q_{Σ*} then
5          return q
6      else if q = q_{Σ*} then
7          return p
8      else
9          for a ∈ Σ do
10             p' ← p^a if p^a ≠ SELF else p
11             q' ← q^a if q^a ≠ SELF else q
12             if p' = p and q' = q then
13                 s_a ← SELF
14             else
15                 s_a ← inter(p', q')
16         return make(s, flag(p) ∧ flag(q))
```

The two procedures may have an exponential complexity since the recursion may recompute the same values repeatedly. However, this is easily avoided with memoization. This will apply to all our algorithms, *i.e.*, each recursive procedure

$\mathtt{t}(x)$ can first check in a dictionary whether the output value for x has already been computed, and only proceed to compute it if not. To avoid cluttering the presentation, we will not explicitly describe such memoization in our pseudocode.

The following proposition is established with standard arguments (see the appendix of the full version).

Proposition 7. *Algorithm 2 and Algorithm 3 are correct. Moreover, with memoization, they respectively work in time $\mathcal{O}(|\Sigma| n_q)$ and $\mathcal{O}(|\Sigma| n_p n_q)$, where n_x is the number of nodes reachable from node x.*

It is easy to adapt Algorithm 3 to other binary operations such as union: It suffices to change the base cases and the way the acceptance flag is set.

Note that testing emptiness, universality and language equality can be done in *constant time*. Indeed, as the table keeps a unique identifier for each language, we have $\mathrm{L}(q) = \emptyset$ iff $q = q_\emptyset$, $\mathrm{L}(q) = \Sigma^*$ iff $q = q_{\Sigma^*}$, and $\mathrm{L}(p) = \mathrm{L}(q)$ iff $p = q$.

4.2 Fixed-length relations: a general approach

For a relation $R \subseteq \Sigma^* \times \Sigma^*$ and a language $L \subseteq \Sigma^*$, we define $\mathrm{Post}_R(L) \coloneqq \{v \in \Sigma^* : u \in L, (u, v) \in R\}$ and $\mathrm{Pre}_R(L) \coloneqq \{u \in \Sigma^* : (u, v) \in R, v \in L\}$. A relation is said to be *fixed-length* if $(u, v) \in R$ implies $|u| = |v|$.

We consider fixed-length regular relations, *i.e.*, those that can be represented by automata over alphabet $\Sigma \times \Sigma$, which we call transducers. Given a fixed-length regular relation R and a weakly acyclic language L, $\mathrm{Post}_R(L)$ and $\mathrm{Pre}_R(L)$ may not be weakly acyclic. For example, consider the relation R and the language L depicted by the transducer and automata of Figure 3. We have $\mathrm{Post}_R(L) = (a+b)^*b$ which is not weakly acyclic (as its minimal DFA has a nontrivial cycle).

Fig. 3. *Left*: A transducer $\mathcal{T}$ that converts each occurrence of letter c into letter b. *Right*: A weakly acyclic DFA $\mathcal{A}$ accepting language $(a + b)^*c$.

Yet, under the guarantee that the resulting language is weakly acyclic, we can compute $\mathrm{Post}_R(L)$ and $\mathrm{Pre}_R(L)$. The key observation is that if a DFA accepting a weakly acyclic language has a cycle, then this cycle can be "contracted". Thus, given a transducer $\mathcal{T}$ and a node q, we apply the powerset contruction on the pairing of T and q, and we contract cycles on the fly. Informally, cycle detection is achieved by testing whether a successor is already on the call stack. Algorithm 4 implements this idea for $\mathrm{Pre}_R(L)$. In the pseudocode, symbols S and S' denote sets of pairs of states.

Algorithm 4: General algorithm for the Pre operation

Input: transducer $\mathcal{T} = (P, \Sigma \times \Sigma, \delta, p_0, F)$, and identifier q such that $\mathrm{Pre}_{\mathrm{L}(T)}(\mathrm{L}(q))$ is weakly acyclic
Result: identifier r such that $\mathrm{L}(r) = \mathrm{Pre}_{\mathrm{L}(\mathcal{T})}(\mathrm{L}(q))$

```
1  pre(T, q):
2      succ(q, b):
3          if q^b ≠ SELF then return q^b
4          else return q
5      pre-aux(S):
6          mark S
7          for a ∈ Σ do
8              S' ← ⋃_{b∈Σ} {(p', q') : (p, q) ∈ S, p' ∈ δ(p, (a, b)), q' = succ(q, b)}
9              if S' is marked then s_a ← SELF    // Contract detected cycle
10             else                 s_a ← pre-aux(S')
11         unmark S
12         return make(s, ∃(p, q) ∈ S : p ∈ F ∧ flag(q))
13     return pre-aux({(p_0, q)})
```

Let us analyze Algorithm 4. We write $\mathrm{Pre}_p(q)$ to denote $\mathrm{Pre}_{\mathrm{L}(p)}(\mathrm{L}(q))$. Let $\mathrm{L}(S) \coloneqq \bigcup_{(p,q)\in S} \mathrm{Pre}_p(q)$. Given S and $a \in \Sigma$, we write $S \to^a S'$ where

$$S' \coloneqq \bigcup_{b\in\Sigma} \{(p', q') : (p, q) \in S, p' \in \delta(p, (a, b)), q' = \texttt{succ}(q, b)\}.$$

In words, "$S \to^a S'$" denotes that `pre-aux(S)` constructs the set S' on Line 8. By extension, we write $S \to^\varepsilon S'$ if $S = S'$, and we write $S \to^{a_1 a_2 \cdots a_n} S'$ if $S = S_0 \to^{a_1} S_1 \to^{a_2} S_2 \cdots \to^{a_n} S_n = S'$ for some $S_0, S_1, \ldots, S_n$.

Proposition 8. *Let $w \in \Sigma^*$. If $S \to^w S'$, then $\mathrm{L}(S)^w = \mathrm{L}(S')$.*

Proposition 9. *If $S_0 \to^* S' \to^* S \to^a S'$ for some $a \in \Sigma$, and $\mathrm{L}(S_0)$ is weakly acyclic, then $\mathrm{L}(S) = \mathrm{L}(S')$.*

Proposition 10. *Algorithm 4 is correct and terminates.*

Proof. Termination follows easily by the fact that only finitely many subsets can be generated (see the appendix of the full version). Let us prove correctness.

Let $S_0 \coloneqq \{(p_0, q)\}$ be the input and let `pre-aux(S)` be a recursive call made by `pre(S_0)`. Recall that $\mathrm{L}(S) = \bigcup_{(p,q)\in S} \mathrm{Pre}_p(q)$. So, we must show that $\mathrm{L}(\texttt{pre-aux}(S)) = \mathrm{L}(S)$. To do so, we must prove that, for each $a \in \Sigma$, Algorithm 4 assigns to s_a the state accepting $\mathrm{L}(S)^a$. Moreover, we must show that the acceptance flag is set correctly. We show the latter first, and then the former.

We have $\varepsilon \in \bigcup_{(p,q)\in S} \mathrm{Pre}_p(q)$ iff there exists $(p, q) \in S$ such that $(\varepsilon, \varepsilon) \in \mathrm{L}(p)$ and $\varepsilon \in \mathrm{L}(q)$ iff there exists $(p, q) \in S$ such that $p \in F$ and $\mathrm{flag}(q) = \mathit{true}$. Thus, the flag is set correctly on Line 12.

Let $a \in \Sigma$. Let S' be the set computed on Line 8 for letter a. By definition, we have $S \to^a S'$. We make a case distinction on whether S' is marked or not.

Case 1: S' is marked. Since S' is marked, we have $S_0 \to^* S' \to^* S$. Therefore, $S_0 \to^* S' \to^* S \to^a S'$. By Proposition 9, we have $\mathrm{L}(S) = \mathrm{L}(S')$. Moreover, by Proposition 8, we have $\mathrm{L}(S)^a = \mathrm{L}(S')$. Thus, $\mathrm{L}(S)^a = \mathrm{L}(S') = \mathrm{L}(S)$. As S' is marked, Algorithm 4 executes Line 9, which correctly assigns "$s_a \leftarrow$ SELF".

Case 2: S' is unmarked. By Proposition 8, $\mathrm{L}(S)^a = \mathrm{L}(S')$. As S' is unmarked, Algorithm 4 executes Line 10, which correctly assigns "$s_a \leftarrow$ `pre-aux(`S'`)`". □

Note that Algorithm 4 has exponential worse-case time complexity since this is already the case for the particular case of OBDDs (see [20, Chap. 6.5, pp. 148–149]). Indeed, given an arbitrary NFA A, one can find a (deterministic) transducer and a DFA whose composition yields A, and it is well known that determinizing an NFA blows up its size exponentially in the worst case.

4.3 Fixed-length relations: a more efficient approach

As just mentioned, pairing a transducer and an automaton can result in a non-deterministic automaton. For this reason, Algorithm 4 determinizes on the fly by constructing subsets of states, and minimizes on the fly by contracting cycles.

We describe a setting in which determinism is guaranteed, and hence where cycle contraction is avoided. This allows for a polynomial-time time procedure.

Observe that a weakly acyclic language K over $\Sigma \times \Sigma$ can be represented with our data structure. As shown in Figure 3, even if K and L are weakly acyclic, it is not necessarily the case that $\mathrm{Post}_K(L)$ and $\mathrm{Pre}_K(L)$ are weakly acyclic. Yet, we provide a sufficient condition under which weak acyclicity is guaranteed.

Let K and L be weakly acyclic languages respectively over $\Sigma \times \Sigma$ and Σ. We say that K and L are *pre-compatible* if for every $a \in \Sigma$ the following holds:

- there exists at most one $b \in \Sigma$ such that $K^{(a,b)} \neq \emptyset$ and $L^b \neq \emptyset$; and
- if $K^{(a,b)} \neq \emptyset$ and $L^b \neq \emptyset$, then $K^{(a,b)}$ and L^b are pre-compatible.

Algorithm 5 describes a procedure that computes a node accepting $\mathrm{Pre}_K(L)$. Similarly to Algorithms 2 and 3, a simple analysis yields the following proposition (see the appendix of the full version).

Proposition 11. *Algorithm 5 is correct. Moreover, with memoization, it works in time $\mathcal{O}(|\Sigma|^2 n_p n_q)$, where n_x is the number of nodes reachable from node x.*

5 Weakly acyclic (regular) model checking

In this section, we present interesting examples of systems that can be modeled with WADs. In our setting, a system has a set of configurations equipped with a partial order $\preceq$ and a monotone labeled transition relation, *i.e.*, if $C \to^t D$ and

Algorithm 5: Algorithm for the Pre operation under pre-compatibility

Input: identifier p over $\Sigma \times \Sigma$, identifier q over Σ such that $\mathrm{L}(p)$ and $\mathrm{L}(q)$ are pre-compatible

Result: identifier r such that $\mathrm{L}(r) = \mathrm{Pre}_{\mathrm{L}(p)}(\mathrm{L}(q))$

```
1  pre(p, q):
2      for a ∈ Σ do
3          if ¬(∃b ∈ Σ : p^(a,b) ≠ p_∅ ∧ q^b ≠ q_∅) then  s_a ← q_∅
4          else
5              let b be the unique letter such that p^(a,b) ≠ p_∅ and q^b ≠ q_∅
6              p' ← p^(a,b)  if p^(a,b) ≠ SELF  else p
7              q' ← q^b      if q^b ≠ SELF      else q
8              if p' = p ∧ q' = q then  s_a ← SELF
9              else                    s_a ← pre(p', q')
10     return make(s, flag(p) ∧ flag(q))
```

$C' \succeq C$, then $C' \to^t D'$ for some $D' \succeq C'$. The *safety verification problem* asks, given configurations C and C', whether $C \to^* C''$ for some $C'' \succeq C'$.

For every D, let $\mathrm{Pre}(D) \coloneqq \{C : C \to^* D\}$. We extend this notation to sets, *i.e.*, $\mathrm{Pre}(X) \coloneqq \bigcup_{C \in X} \mathrm{Pre}(C)$. Let $\mathrm{Pre}^*(X) \coloneqq \bigcup_{i \geq 0} \mathrm{Pre}^i(X)$. A set of configurations X is *upward-closed* if $\uparrow X = X$, where $\uparrow X \coloneqq \{C' : C' \succeq C \text{ and } C \in X\}$.

Safety verification amounts to testing whether $C \in \mathrm{Pre}^*(\uparrow C')$. If the system is well structured, namely if $\preceq$ is a well-quasi-order, then it is well-known that $\mathrm{Pre}^*(\uparrow C') = \mathrm{Pre}^i(\uparrow C')$ for some $i \geq 0$. Thus, verification can be done by the *backward algorithm*: compute $\mathrm{Pre}^0(\uparrow C') \cup \mathrm{Pre}^1(\uparrow C') \cup \cdots$ until reaching a fixed point. So, for well-structured transition systems, the approach is as follows:

- Choose a suitable alphabet Σ;
- For each configuration D, define an encoding $\mathrm{enc}(\uparrow D) \subseteq \Sigma^*$ of $\uparrow D$ which is a weakly acyclic language;
- For each t, define a transducer $\mathcal{T}_t$ for the language $\{(\mathrm{enc}(D), \mathrm{enc}(D')) : D \to^t D'\} \subseteq (\Sigma \times \Sigma)^*$;
- Create a weakly acyclic diagram for $\mathrm{L}(q_0) = \mathrm{enc}(\uparrow C')$ and compute $\mathrm{L}(q_{i+1}) = \mathrm{Pre}_{\mathcal{T}_t}(\mathrm{L}(q_i))$ iteratively until $q_{i+1} = q_i$.

Since $\preceq$ is a well-quasi-order, any upward-closed set X of configurations has a finite basis, *i.e.*, there exist configurations $C_1, \ldots, C_n$ such that $X = \bigcup_{i \in [1..n]} \uparrow C_i$. Thus, as weakly acyclic languages are closed under union, $\mathrm{enc}(X)$ must be weakly acyclic. Moreover, since $\mathrm{Pre}(X)$ is upward-closed by monotonicity, $\mathrm{Pre}_{\mathcal{T}_t}(\mathrm{enc}(X))$ is necessarily weakly acyclic.

5.1 Lossy channel systems

A *channel system* is a finite directed graph whose nodes P are called *states*, and whose arcs, called *transitions*, are labeled by $\mathtt{read}_i(a)$, $\mathtt{write}_i(a)$ or $\mathtt{nop}$ with

$a \in \Gamma$ and $i \in [1..k]$. There are m processes starting in some states, and evolving by reading from the left and writing to the right of channels, and by moving to other states. A *lossy channel system (LCS)* is a channel system where any letter may be lost at any moment from any channel.

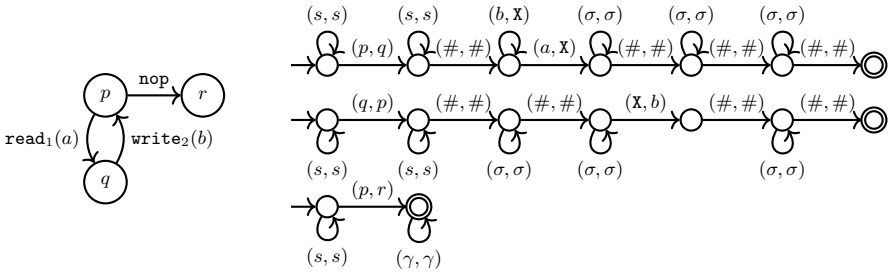


Fig. 4. *Left*: Example of a lossy channel system with $\Gamma = \{a, b\}$. *Right*: Transducers encoding respectively the transitions $\texttt{read}_1(a)$, $\texttt{write}_2(b)$ and $\texttt{nop}$ (under the semi-lossy semantics), where $s \in \{p, q, r\}$, $\sigma \in \{a, b\}$ and γ stands for any letter.

For example, consider the channel system depicted on the left of Figure 4. Consider its configuration $C \coloneqq ([p, q], ab, \varepsilon, b)$ with two processes in states p and q, and with three channels currently containing ab, ε and b. We have, *e.g.*,

$$C \xrightarrow{\texttt{read}_1(a)} ([q, q], b, \varepsilon, b) \xrightarrow{\texttt{write}_2(b)} ([q, p], b, b, b) \xrightarrow{\texttt{nop}} ([q, r], b, b, b).$$

We encode configurations as words over $\Sigma \coloneqq P \cup \Gamma \cup \{\texttt{\#}\}$, *e.g.* the encoding of C is $pq\texttt{\#}ab\texttt{\#\#}a\texttt{\#}$. Moreover, we encode operations as transducers with an extra letter $\texttt{X}$. Let us explain this through an example.

The transducers for the three transitions of our example are depicted on the right of Figure 4. As a preprocessing step for operation $\texttt{write}_i(\sigma)$, we pad channels of a configuration on the right with $\texttt{X}^*$. This way, the first occurrence of $\texttt{X}$ can be replaced by the letter to insert. This padding can then be removed in a postprocessing step. Furthermore, we model the "lossiness" of the channels through $\texttt{read}_i(\sigma)$: when reading an a, all letters in front of the first a may be lost. Transducers for $\texttt{write}_i(\sigma)$ and $\texttt{nop}$ are nonlossy. For safety verification, the standard "pure lossy" semantics is equivalent to our "semi-lossy" semantics.

Let us formalize it. Given configurations $C = (X, w_1, \ldots, w_k)$ and $C' = (X', w'_1, \ldots, w'_k)$, we write $C \preceq C'$ if $X = X'$ and $w_i \sqsubseteq w'_i$ for all $i \in [1..k]$, where $\sqsubseteq$ denotes the subword order. For example, $([p, q], ab, \varepsilon, b) \preceq ([p, q], aab, a, baa)$. By Higman's lemma, the subword order $\sqsubseteq$ is a well-quasi-order. Moreover, equality over a finite set is a well-quasi-order. As the order $\preceq$ over configurations is a product of these orders, it is also a well-quasi-order. Moreover:

Proposition 12. *We have $C \xrightarrow{*}_{\text{pure-lossy}} C''$ for some $C'' \succeq C'$ iff $C \xrightarrow{*}_{\text{semi-lossy}} C''$ for some $C'' \succeq C'$. Furthermore, $\rightarrow_{\text{semi-lossy}}$ is monotone.*

Note that $\mathrm{enc}(\uparrow C)$ is weakly acyclic. Indeed, for $C = ([p_1, \ldots, p_m], w_1, \ldots, w_k)$, the language $\mathrm{enc}(\uparrow C)$ can be expressed by a weakly acyclic expression:

$$p_1 \cdots p_m \ \# \prod_{i \in [1..k]} \left(\prod_{a \in w_i} (\Gamma \setminus \{a\})^* a \right) \Gamma^* \ \#.$$

5.2 Petri nets

A Petri net is a tuple (P, T, F) where P is a finite set of *places*, T is a finite set of *transitions* disjoint from P, and $F \colon (P \times T) \cup (T \cup P) \to \mathbb{N}$. A *marking* is an assignment $\boldsymbol{m} \colon P \to \mathbb{N}$ of tokens to places. A transition t is *enabled* in $\boldsymbol{m}$ if $\boldsymbol{m}(p) \geq F(p,t)$ for every place p. If t is enabled, then it can be *fired*, which leads to the marking $\boldsymbol{m}'$ obtained by removing $F(p,t)$ tokens and adding $F(t,p)$ tokens to each place p, *i.e.*, $\boldsymbol{m}'(p) = \boldsymbol{m}(p) - F(p,t) + F(t,p)$.

Consider, *e.g.*, the Petri net depicted on the left of Figure 5. From the marking $[p \mapsto 3, q \mapsto 1]$, written as $\boldsymbol{m} \coloneqq (3,1)$, we have, *e.g.*, $\boldsymbol{m} \to^t (2,3) \to^t (1,5)$.

We encode markings over $\Sigma \coloneqq \{\bullet, \#\}$, *e.g.* the encoding of $\boldsymbol{m}$ is $\bullet\bullet\bullet\#\bullet\#$. Moreover, we encode operations as transducers with an extra letter `X`. Let us explain this through an example. The transducer for the transition of our example is depicted on the right of Figure 5. As a preprocessing step, we pad places of a configuration on the right with `X`*. This symbol can be used to remove and add tokens. The padding is then removed in a postprocessing step.

We write $\boldsymbol{m} \leq \boldsymbol{m}'$ if $\boldsymbol{m}(p) \leq \boldsymbol{m}'(p)$ for all p. By Dickson's lemma, $\leq$ is a well-quasi-order. Note that $\mathrm{enc}(\uparrow \boldsymbol{m})$ is weakly acyclic. Indeed, for $\boldsymbol{m} = (n_1, \ldots, n_k)$, it is the language of this weakly acyclic expression: $\bullet^{n_1}\bullet^*\#\cdots\bullet^{n_k}\bullet^*\#$.

5.3 Broadcast protocols

A *broadcast protocol* is a finite directed graph whose nodes P are called *states*, and where each arc (*transition*) is labeled by a, b`!`, b`?`, c`!!` or c`??` where $a \in \Gamma$, $b \in \Gamma'$ and $c \in \Gamma''$ (three disjoint alphabets). There are m processes starting in some states, and evolving by local, rendez-vous or broadcast communication:

- A local update a only changes the state of a single process;
- A rendez-vous occurs when two processes respectively take a b`!`-transition and a b`?`-transition;

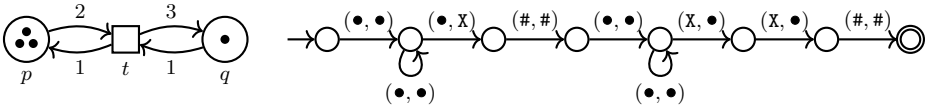


Fig. 5. *Left*: Example of a Petri net where $P = \{p, q\}$, $T = \{t\}$, $F(p,t) = 2$, $F(t,p) = 1 = F(q,t)$ and $F(t,q) = 3$. *Right*: Transducer encoding transition t.

– A broadcast occurs when a process takes a $c!!$-transition; when this happens, any other process that can take a $c??$-transition takes it.

For example, consider the broadcast protocol depicted on the left of Figure 6. Let $C := [p, p, p, p]$ be the configuration with four processes in state p. We have, *e.g.*, $C \to^b [q, r, p, p] \to^b [q, r, q, r] \to^c [p, q, p, r] \to^c [p, p, p, q]$. We encode configurations as words over $\Sigma := P$, *e.g.* $[p, r, p, q]$ becomes $prpq$. The transducers for our example are depicted on the middle and right of Figure 6.

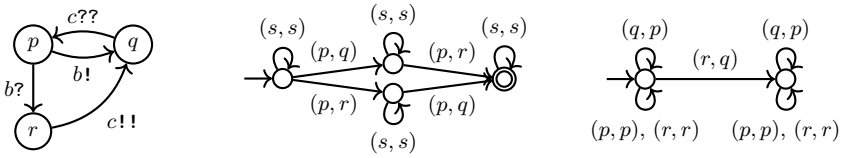


Fig. 6. *Left*: Example of a broadcast protocol with $P = \{p, q, r\}$, $\Gamma = \emptyset$, $\Gamma' = \{b\}$ and $\Gamma'' = \{c\}$. *Middle and right*: Transducers encoding the transitions, where $s \in \{p, q, r\}$.

6 Experimental results

We developed a prototype C++ library for weakly acyclic diagrams, which we refer to as WADL. It implements all operations described in Section 4, and backwards reachability for lossy channel systems, Petri nets and broadcast protocols, as in Section 5. To benchmark the performance, we use four different sets:

– *Lossy channel systems.* We collected 13 instances used by the tools BML [25] and McScM [26,27]. They arise from a mutual exclusion algorithm and from protocols (communication, business, etc.) The instance `ring2` comes from a parameterized family. Thus, we generalized it to `ring3`, `ring4`, etc.
– *Petri nets.* We collected 173 instances used in many papers on Petri net safety verification (*e.g.* see [22]). They model protocols (mutual exclusion, communication, etc.); concurrent C and Erlang programs; provenance analysis of messages of a medical messaging system and a bug-tracking system.
– *Broadcast protocols.* We collected 38 broadcast protocol instances from the benchmark set of DODO, a tool for regular model checking that uses the encoding of Section 5.3 for broadcast protocols [17,35]. The instances correspond to safety properties of several cache-coherence protocols.
– *Regular model checking problems.* Backwards reachability can also be applied to systems which are not well structured, as long as all the intermediate sets of configurations remain weakly acyclic. The algorithm may not terminate, but if it does then it yields the correct answer. We collected 35 further instances from DODO modeling parameterized mutual exclusion algorithms, variants of the dining philosophers, algorithms for termination detection and leader election, and token passing algorithms.

We carried experiments for the four above categories with the goal of testing the potential and versatility of WADs. We used a machine with an 8-Core 11$^{\text{th}}$ Gen Intel® Core™ i7-1165G7 CPU @ 2.80GHz running on Kubuntu 24.04 with 31.1GiB of memory. The time was determined with the sum of the `user` and `sys` times given by Linux tool `time`. We used a timeout of 10 minutes per instance, except for Petri nets where we used a timeout of 20 minutes[5].

Lossy channel systems. We made a few optimizations compared to Section 5. First, states are represented in binary to avoid using $|P|$ letters. Second, we tweaked the transducers so that they yield the pre-compatibility of Section 4.3. The transducers of Figure 4 do not necessarily satisfy it as the initial states use letters (x, y) and (x', y') with $x = x'$. This can be fixed, *e.g.* make the initial state of the top transducer loop on (q, q) and (r, r), but not (p, p). This slightly changes the semantics: only the first process in p can execute. Yet, as processes in a common state are indistinguishable, only the state count matters and so the change does not alter safety verification. Moreover, in our benchmarks, processes evolve in disjoint subgraphs and hence can never even be in a common state.

We compared with BML [25] which runs the backward algorithm with pruning (modes: CSRE, MOF, SI). Pruning can be disabled to yield the basic algorithm. We also compared to McScM [26,27] which (in its default mode) uses an adaptive extrapolated backwards computation. The results are depicted on the left of Figure 7. WADL solves the most instances and does so generally faster, although the competition is close. We further tested the same tools on `ring3`, `ring4`, etc. Our tool terminates on large n (e.g. `ring99` in 2m38s and `ring187` is the largest solved), but the other tools did not scale, as depicted in Table 1.

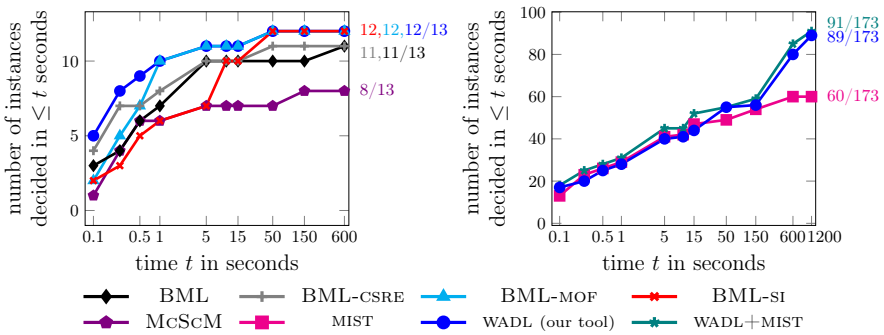


Fig. 7. Cumulative number of instances decided over time (semi-log scale) for lossy channel systems excluding {`ring3`, `ring4`, ...} (left), and Petri nets (right). WADL+MIST refers to the best outcome on instances solved by at least one of the two tools.

[5] During the reviewing process, a reviewer requested a longer timeout for this case.

	BML	BML-CSRE	BML-MOF	BML-SI	MCSCM	WADL
ring2	0.63	1.02	0.89	0.37	TO	0.04
ring3	TO	TO	TO	9.13	416.48	0.10
ring4	TO	TO	TO	307.98	SO	0.19
ring5	TO	TO	TO	TO	TO	0.33

Table 1. Time in seconds to solve ring$n$ (TO = timeout, SO = stack overflow).

Petri nets. We compared with MIST [18,24], an established efficient tool that, in particular, offers an implementation of the backward algorithm without pruning, as in our case, but using interval sharing trees as the data structure. The results are depicted on the right of Figure 7. MIST is competitive, but we solved more instances in total. There are 31 instances that we solved that MIST did not; 2 instances for the other way around; and 58 instances solved by both tools.

Broadcast protocols. We solved 32/38 instances, with 25 solved within 100ms. We do not compare with DODO as its goal is not to check the property, but rather to compute small invariants explaining why it holds. So, although we would look good in comparison, it would be meaningless (the runtimes of DODO reported in [35] are at least ten times higher than ours in every instance).

Regular model checking. While there is *a priori* no reason why the instances should be checkable with WADs, we solved 22/35 instances (all within 10ms). We further timed out on 6 instances, and could not verify 7 instances. For the latter, a cycle was contracted in Line 9 of Algorithm 4, and hence we cannot be certain that the result is weakly acyclic.

The fact we solved many instances suggests that weakly acyclic languages may be common in regular model checking. In fact, popular examples used to introduce the technique, like the token-passing protocol, FIFO channel, and stack examples of the surveys [5,1,7,2], or the alternating bit protocol of [4] are modeled with weakly acyclic languages.

7 Conclusion

We introduced weakly acyclic diagrams as a general data structure that extends binary decision diagrams to (possibly) infinite languages, while maintaining their algorithmic advantages. Many instances in regular model checking fall within the class of weakly acyclic languages. Moreover, weakly acyclic diagrams allow for the manipulation and verification of various well-structured transition systems such as lossy channel systems, Petri nets and broadcast protocols.

As future work, it would be interesting to see whether the good algorithmic properties can be retained under a slight extension of nondeterminism, *e.g.* for languages like $(a + b)^*b$ which are simple but not weakly acyclic per se.

References

1. Abdulla, P.A.: Regular model checking. International Journal on Software Tools for Technology Transfer **14**(2), 109–118 (2012). https://doi.org/10.1007/S10009-011-0216-8
2. Abdulla, P.A.: Regular model checking: Evolution and perspectives. In: Model Checking, Synthesis, and Learning – Essays Dedicated to Bengt Jonsson on The Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13030, pp. 78–96. Springer (2021). https://doi.org/10.1007/978-3-030-91384-7_5
3. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: Proc. 11th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 313–321. IEEE Computer Society (1996). https://doi.org/10.1109/LICS.1996.561359
4. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. In: Proc. 8th Annual Symposium on Logic in Computer Science (LICS). pp. 160–170 (1993). https://doi.org/10.1109/LICS.1993.287591
5. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Proc. 15th International Conference on Concurrency Theory (CONCUR). vol. 3170, pp. 35–48. Springer (2004). https://doi.org/10.1007/978-3-540-28644-8_3
6. Abdulla, P.A., Kindahl, M.: Decidability of simulation and bisimulation between lossy channel systems and finite state systems (extended abstract). In: Proc. 6th International Conference on Concurrency Theory (CONCUR). vol. 962, pp. 333–347. Springer (1995). https://doi.org/10.1007/3-540-60218-6_25
7. Abdulla, P.A., Sistla, A.P., Talupur, M.: Model checking parameterized systems. In: Handbook of Model Checking, pp. 685–725. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_21
8. Akers, S.: Binary decision diagrams. IEEE Transactions on Computers **27**(6), 509–516 (1978), https://doi.org/10.1109/TC.1978.1675141
9. Andersen, H.R.: An introduction to binary decision diagrams (1998)
10. Blondin, M., Cadilhac, M., Cui, X., Czerner, P., Esparza, J., Schulz, J.: Weakly acyclic diagrams: A data structure for infinite-state symbolic verification. CoRR **abs/2411.17250** (2024). https://doi.org/10.48550/arXiv.2411.17250
11. Boigelot, B., Godefroid, P.: Symbolic verification of communication protocols with infinite state spaces using QDDs. Formal Methods in System Design (FMSD) **14**(3), 237–255 (1999). https://doi.org/10.1023/A:1008719024240
12. Bouajjani, A., Muscholl, A., Touili, T.: Permutation rewriting and algorithmic verification. Information and Computation **205**(2), 199–224 (2007). https://doi.org/10.1016/J.IC.2005.11.007
13. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
14. Bryant, R.E.: Binary decision diagrams. In: Handbook of Model Checking, pp. 191–217. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_7
15. Brzozowski, J.A., Fich, F.E.: Languages of $\mathcal{R}$-trivial monoids. Journal of Computer and System Sciences **20**(1), 32–49 (1980), https://doi.org/10.1016/0022-0000(80)90003-3
16. Chaki, S., Gurfinkel, A.: BDD-based symbolic model checking. In: Handbook of Model Checking, pp. 219–245. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_8

17. Czerner, P., Esparza, J., Krasotin, V., Welzel-Mohr, C.: Computing inductive invariants of regular abstraction frameworks. In: Proc. 35th International Conference on Concurrency Theory (CONCUR). LIPIcs, vol. 311, pp. 19:1–19:18 (2024). https://doi.org/10.4230/LIPICS.CONCUR.2024.19, The tool DODO and the instances are available at https://zenodo.org/records/8354894.
18. Delzanno, G., Raskin, J., Begin, L.V.: Covering sharing trees: a compact data structure for parameterized verification. International Journal on Software Tools for Technology Transfer (STTT) **5**(2-3), 268–297 (2004). https://doi.org/10.1007/S10009-003-0110-0
19. Eilenberg, S.: Automata, Languages, and Machines, vol. B. Academic Press (1976)
20. Esparza, J., Blondin, M.: Automata theory: An algorithmic approach. MIT Press (2023)
21. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Proc. 14th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 352–359. IEEE Computer Society (1999). https://doi.org/10.1109/LICS.1999.782630
22. Esparza, J., Ledesma-Garza, R., Majumdar, R., Meyer, P.J., Niksic, F.: An SMT-based approach to coverability analysis. In: Proc. 26th International Conference on Computer Aided Verification (CAV). pp. 603–619. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_40
23. Finkel, A., Schnoebelen, Ph.: Well-structured transition systems everywhere! Theoretical Computer Science **256**(1-2), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X
24. Ganty, P., Meuter, C., Delzanno, G., Kalyon, G., Raskin, J.F., Van Begin, L.: Symbolic data structure for sets of k-uples. Tech. Rep. 570, Université Libre de Bruxelles, Belgium (2007)
25. Geffroy, T., Leroux, J., Sutre, G.: Backward coverability with pruning for lossy channel systems. In: Proc. 24th ACM SIGSOFT International Symposium on Model Checking of Software (SPIN). pp. 132–141. ACM (2017). https://doi.org/10.1145/3092282.3092292, The tool BML and the lossy channel system instances are available at https://dept-info.labri.fr/~tgeffroy/lcs/.
26. Heußner, A., Gall, T.L., Sutre, G.: Extrapolation-based path invariants for abstraction refinement of FIFO systems. In: Proc. 16th International on Model Checking Software (SPIN). pp. 107–124. Springer (2009). https://doi.org/10.1007/978-3-642-02652-2_11, The tool McScM is available at https://svn.labri.fr/repos/acs/www/redmine/projects/mcscm/wiki.html
27. Heußner, A., Gall, T.L., Sutre, G.: McScM: A general framework for the verification of communicating machines. In: Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 478–484. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_34
28. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, University of Aarhus (January 2001), notes Series NS-01-1. Available from http://www.brics.dk/mona/
29. Krötzsch, M., Masopust, T., Thomazo, M.: Complexity of universality and related problems for partially ordered NFAs. Information and Computation **255**, 177–192 (2017). https://doi.org/10.1016/j.ic.2017.06.004
30. Masopust, T., Krötzsch, M.: Partially ordered automata and piecewise testability. Logical Methods in Computer Science **17**(2) (2021). https://doi.org/10.23638/LMCS-17(2:14)2021
31. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989). https://doi.org/10.1109/5.24143

32. Éric Pin, J.: Varieties of formal languages. North Oxford, London and Plenum (1986)
33. Ryzhikov, A., Wolf, P.: Monoids of upper triangular matrices over the Boolean semiring. In: Proc. 49$^{\text{th}}$ International Symposium on Mathematical Foundations of Computer Science (MFCS). LIPIcs, vol. 306, pp. 81:1–81:18 (2024). https://doi.org/10.4230/LIPICS.MFCS.2024.81
34. Schwentick, T., Thérien, D., Vollmer, H.: Partially-ordered two-way automata: A new characterization of DA. In: Proc. 5$^{\text{th}}$ International Conference on Developments in Language Theory (DLT). vol. 2295, pp. 239–250. Springer (2001). https://doi.org/10.1007/3-540-46011-X_20
35. Welzel-Mohr, C.: Inductive Statements for Regular Transition Systems. Ph.D. thesis, Technical University of Munich, Germany (2024), https://mediatum.ub.tum.de/1721365

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

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

Pushing the Limit: Verified Performance-Optimal Causally-Consistent Database Transactions

Shabnam Ghasemirad(✉), Christoph Sprenger, Si Liu, Luca Multazzu, and David Basin

ETH Zurich, Zurich, Switzerland
shabnam.ghasemirad@inf.ethz.ch

Abstract. Modern web services crucially rely on high-performance distributed databases, where concurrent transactions are isolated from each other using concurrency control protocols. Relaxed isolation levels, which permit more complex concurrent behaviors than strong levels like serializability, are used in practice for higher performance and availability. In this paper, we present Eiger-PORT+, a concurrency control protocol that achieves a strong form of causal consistency, called TCCv (Transactional Causal Consistency with convergence). We show that Eiger-PORT+ also provides performance-optimal read transactions in the presence of transactional writes, thus refuting an open conjecture that this is impossible for TCCv. We also deductively verify that Eiger-PORT+ satisfies this isolation level by refining an abstract model of transactions. This yields the first deductive verification of a complex concurrency control protocol. Furthermore, we conduct a performance evaluation showing Eiger-PORT+'s superior performance over the state-of-the-art.

1 Introduction

Modern web services are built on top of high-performance database systems operating in partitioned, geo-distributed environments. These systems provide distributed transactions that group the users' read and write requests. To balance data consistency and system performance, databases provide a spectrum of *isolation levels* (I in ACID: Atomicity, Consistency, Isolation, and Durability [46]), defining the degree of separation between concurrent transactions. Isolation is enforced by *concurrency control protocols* (also called *transaction protocols*).

Many applications, such as social networks, opt for weak isolation levels to avoid the performance overhead of stronger levels like serializability [43]. These weaker guarantees allow distributed transactions to remain functional even during network partitions, while still providing useful properties. Notably, *transactional causal consistency* (TCC) represents a successful integration of ideas from the distributed computing and database communities. It extends causal consistency [2,44]—the strongest consistency level achievable in an always-available system [4]—by incorporating transactional guarantees. The past decade has seen sustained efforts in designing databases supporting performant causally-consistent distributed transactions [35,3,11,39,38,47], along with their growing adoption in industry [41,14,40]. Nearly all of these systems provide a stronger

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 43–62, 2025.
https://doi.org/10.1007/978-3-031-90660-2_3

variant of TCC, known as TCCv [3,35], that includes *data convergence* requiring views across different clients to eventually converge to the same state.

In this paper, we present a case study on developing and verifying a performance-optimal, causally-consistent, database transaction protocol. Our protocol, Eiger-PORT+, provides TCCv, for which we give a formal proof. Our work faced two challenges. First, it is not a priori clear that such an isolation guarantee is achievable for a performance-optimal protocol. In fact, Lu et al. [38] conjectured that for distributed performance-optimal read-only transactions (PORTs) in the presence of transactional writes, TCC (without convergence) is the strongest achievable isolation level. They presented the Eiger-PORT protocol, which provides this guarantee. In this paper, we constructively refute their conjecture by designing our novel protocol, Eiger-PORT+, which achieves the stronger TCCv guarantee.

Second, transaction protocols are notoriously hard to get right, as witnessed by numerous design-level isolation errors in production databases [24,20,28,21,22], and even in protocols that have undergone pen-and-pencil proofs [42] and model-checking analysis [37]. We thus aim for a full deductive verification of Eiger-PORT+, covering *all* possible behaviors. To our knowledge, the deductive verification of transaction protocols for weak isolation levels, which exhibit complex concurrent behaviors, has not been attempted so far. Previous efforts in this area have focused on simple textbook protocols like two-phase locking achieving serializability or employed model checking, which requires bounding the number of processes and transactions. We address this challenge using our Isabelle/HOL framework [15] built around Xiong et al.'s abstract transaction model [50]. We formalize Eiger-PORT+ and show that it satisfies TCCv using reduction [26] in combination with a refinement of the TCCv instance of the abstract transaction model.

Furthermore, we implement and deploy Eiger-PORT+, along with Eiger-PORT and its precursor Eiger, and conduct a comprehensive performance comparison of these three protocols. Our evaluation demonstrates Eiger-PORT+'s superior performance in terms of system throughput and latency across various scenarios, e.g., with a growing number of clients and servers.

The complete formal development accompanying this paper, including all definitions and proofs, as well as a protocol implementation are available at [17].

Contributions Overall, we see our contributions as three-fold:

- **Conjecture refutal.** We formally refute the conjecture that TCC is the strongest achievable isolation level for PORTs in the presence of transactional writes by designing a protocol, Eiger-PORT+, that provably achieves TCCv.
- **Proof of correctness.** We model Eiger-PORT+ in Isabelle/HOL and verify its correctness by showing that its behavior conforms to the TCCv instance of the abstract transaction model [50]. This represents the first complete formal verification of a complex distributed database transaction protocol.
- **Superior performance.** We deploy Eiger-PORT+, along with two state-of-the-art causally-consistent transaction protocols, in a cluster and evaluate their performance. Our experimental results demonstrate Eiger-PORT+'s superior performance, with both lower latency and higher throughput.

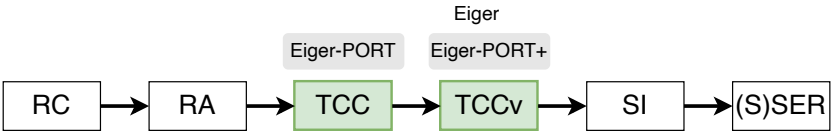


Fig. 1: A spectrum of isolation levels. $A \rightarrow B$ means A is weaker than B. RC: read committed [7]; RA: read atomicity [6]; TCC: transactional causal consistency [38], provided by Eiger-PORT [38]; TCCv: TCC with convergence [3,35], offered by Eiger and our Eiger-PORT+; SI: snapshot isolation [7]; (S)SER: (strict) serializability [43]. Protocols supporting PORTs are highlighted in gray.

2 Background

2.1 Distributed Database Transactions

In a distributed database, vast amounts of data are split up and stored across multiple servers, also called partitions. User requests are submitted as database transactions, initiated by front-end clients. Each client executes the transactions in its own session, where a transaction comprises a sequence of read and/or write operations on data items (or keys) distributed across partitions.

Isolation levels Distributed databases offer various isolation levels, differing on how they balance data consistency and system performance. Figure 1 shows a spectrum of practically relevant isolation levels, ranging from weaker ones like Read Committed, through various forms of transactional causality, to stronger guarantees such as Serializability. We briefly explain Read Atomicity and two variants of transactional causality, which are the focus of this work.

Read Atomicity (RA). This is also known as *atomic visibility*, requiring that all or none of a transaction's updates are observed by other transactions. It prohibits *fractured reads* anomalies, such as Carol only observing one direction of a new (bi-directional) friendship between Alice and Bob in a social network.

Transactional Causal Consistency (TCC). In addition to RA, this level requires that two causally related transactions appear to all client sessions in the same causal order [2,44]. It prevents *causality violations*, such as Carol observing Bob's response to Alice's message without seeing the message itself.

TCC with Convergence (TCCv). With TCC, different clients may observe causally unrelated transactions in different orders. TCCv's *convergence* property prevents this by requiring these clients' views to converge to the same state [34,3]. For example, this prevents confusion created by Alice and Bob independently posting "Let's meet at my place" in a road trip planner. In practice, most causally-consistent databases provide convergence.

Performance-optimal read-only transactions NOCS [38] is the state-of-the-art impossibility result that captures conflicts between distributed transactions'

performance and their isolation guarantees. NOCS proves that read-only transactions cannot complete with **N**on-blocking communication in **O**ne round and **C**onstant-size metadata, while achieving **S**trict serializability (SSER). At best, three of the four NOCS properties can be satisfied. In particular, protocols satisfying the NOC properties (under isolation levels weaker than SSER) are said to provide *performance-optimal* read-only transactions (PORTs). The NOCS authors [38] introduce the Eiger-PORT protocol providing TCC and PORTs. They also state the following conjecture, which has remained unresolved for four years.

Conjecture. TCC is the strongest isolation level achievable for PORTs in the presence of transactional writes.

Recent studies show that write-heavy workloads involving transactional writes are more prevalent than previously assumed and are expected to become increasingly prominent [52]. This, along with the practical significance of TCCv, motivates our work on refuting the above conjecture and pushing the boundary.

2.2 Transition Systems and Refinement

We use *labeled transition systems* (LTSs) to model database protocols and the abstract transaction model. An LTS $\mathcal{E} = (S, I, \{\xrightarrow{e} \mid e \in E\})$ consists of a set of states S, a non-empty set of initial states $I \subseteq S$, and transition relations $\xrightarrow{e} \subseteq S \times S$, one for each event $e \in E$. We assume an idling event $\mathsf{skip} \in E$ with $s \xrightarrow{\mathsf{skip}} s$. We often define the relations $\xrightarrow{e}$ using *guard* predicates G_e and *update* functions U_e by $s \xrightarrow{e} s'$ if and only if $G_e(s) \wedge s' = U_e(s)$. A state s is *reachable* if a sequence of transitions leads from an initial state to s. We denote the set of reachable states of $\mathcal{E}$ by $\mathsf{reach}(\mathcal{E})$. A set of states J is an *invariant* if $\mathsf{reach}(\mathcal{E}) \subseteq J$.

Refinement relates two LTSs $\mathcal{E}_i = (S_i, I_i, \{\xrightarrow{e}_i \mid e \in E_i\})$, for $i \in \{1, 2\}$. Given *refinement mappings* $r\colon S_2 \to S_1$ and $\pi\colon E_2 \to E_1$ between the LTSs' states and events, we say $\mathcal{E}_2$ refines $\mathcal{E}_1$, written $\mathcal{E}_2 \preccurlyeq_{r,\pi} \mathcal{E}_1$, if (i) $r(s) \in I_1$ for all $s \in I_2$ and (ii) $r(s) \xrightarrow{\pi(e)}_1 r(s')$ whenever $s \xrightarrow{e}_2 s'$. Using guards and updates, (ii) reduces to two proof obligations: assuming $G^2_e(s)$ prove (a) $G^1_{\pi(e)}(r(s))$ (*guard strengthening*) and (b) $r(U^2_e(s)) = U^1_{\pi(e)}(r(s))$ (*update correspondence*). Refinement guarantees the inclusion of sets of reachable states (modulo r), i.e., $r(\mathsf{reach}(\mathcal{E}_2)) \subseteq \mathsf{reach}(\mathcal{E}_1)$, where r is applied to each element of $\mathsf{reach}(\mathcal{E}_2)$. Refinement proofs often require invariants to strengthen the refinement mapping.

2.3 An Abstract Transaction Model

Xiong et al. [50] introduced a centralized operational model for atomic transactions operating on distributed multi-versioned key-value stores (KVSs), where the database stores data as key-value pairs and each key may be mapped to multiple versions for increased data availability. This model can be instantiated to different isolation guarantees, including RA, TCCv, and SSER (cf. Figure 1). They prove

the equivalence of these model instances to their declarative counterparts based on abstract executions. The model can be used to prove the correctness of both concurrency control protocols and client programs. We have formalized this framework in Isabelle/HOL and extended it with an extensive library of lemmas supporting protocol correctness proofs [15].

Xiong et al.'s model is formulated as an LTS, called the *abstract transaction model*, which abstracts the protocols' distributed collection of KVSs (each representing a shard and/or replica) into a single (centralized) multi-versioned KVS $\mathcal{K} : \texttt{key} \to \texttt{version list}$ that maps each key to a list of *versions*. Each version $\mathcal{K}(k,i)$ of a key k at the list index i records (i) the value v stored, (ii) the writer transaction t that has produced this version, and (iii) the reader set T, i.e., the set of transactions that have read this version. The pairs (t,t') for any $t' \in T$ are called *write-read dependencies*. The relation $\mathsf{WR}_{\mathcal{K}}$ contains all such pairs. The fact that, in a real, distributed system, each client cl has a different partial *client view* of $\mathcal{K}$ is modeled by explicitly representing these views in the model's configurations as mappings $\mathcal{U}(cl) : \texttt{key} \to \texttt{nat set}$. This describes, for each key, the set of versions (denoted by list indices) visible to the client. Clients are assumed to process transactions sequentially. The *session order* relation SO captures the order of their transactions.

The model assumes the *snapshot property*, ensuring that each transaction reads and writes at most one version of each key. Hence, transactions can be represented by a *fingerprint* $\mathcal{F} : \texttt{key} \times \{\texttt{R},\texttt{W}\} \rightharpoonup \texttt{value}$, which maps each key and operation (read or write) to at most one value. It also assumes that views are *atomic*, i.e., clients observe either all or none of a transaction's effects. These properties together establish *atomic visibility*, also called Read Atomicity (RA) (cf. Section 2.1), as the model's baseline isolation guarantee.

The model has two events (plus skip): *commit*, which atomically executes an entire transaction, and *view extension*, which monotonically extends a client's view of the KVS. The commit event's executability depends on the isolation guarantee. Here, we focus on the model's TCCv instantiation, called $\mathcal{I}_{\mathsf{TCCv}}$.

Commit The commit event's transition relation for TCCv is defined by:

$$\frac{\begin{array}{c}\mathcal{U}(cl) \sqsubseteq u \quad \mathsf{canCommit}_{\mathsf{TCCv}}(\mathcal{K},u,\mathcal{F}) \quad u \sqsubseteq u' \quad \mathsf{RYW}(\mathcal{K},\mathcal{K}',u') \\ \mathsf{LWW}(\mathcal{K},u,\mathcal{F}) \quad \mathsf{wf}(\mathcal{K},u) \quad \mathsf{wf}(\mathcal{K}',u') \\ t^{cl}_{sn} \in \mathsf{nextTxids}(\mathcal{K},cl) \quad \mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K},t^{cl}_{sn},u,\mathcal{F})\end{array}}{(\mathcal{K},\mathcal{U}) \xrightarrow{\mathsf{commit}(cl,sn,u,\mathcal{F})}_{\mathsf{TCCv}} (\mathcal{K}',\mathcal{U}[cl \mapsto u'])}$$

The transition in the conclusion updates the configuration $(\mathcal{K},\mathcal{U})$ to the new configuration $(\mathcal{K}',\mathcal{U}[cl \mapsto u'])$, where $\mathcal{K}'$ is the updated KVS and $\mathcal{U}[cl \mapsto u']$ updates the client cl's view to u'. Both $\mathcal{K}'$ and u' are determined by the rule's premises, which act as the event's guards with the following meanings:

- $\mathcal{U}(cl) \sqsubseteq u$: This condition allows one to extend the client cl's current view to a (point-wise) larger one before committing.

- $\mathsf{canCommit}_{\mathsf{TCCv}}(\mathcal{K}, u, \mathcal{F})$: This is the central commit condition, which ensures that it is safe to commit a transaction at the TCCv isolation level. It requires that the set of visible transactions $\mathsf{visTx}(\mathcal{K}, u)$ (i.e., the writers of the versions that the view u points to) is *closed* under the relation $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$, i.e.,

$$((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})^{-1})^{+}(\mathsf{visTx}(\mathcal{K}, u)) \subseteq \mathsf{visTx}(\mathcal{K}, u) \cup \mathsf{rdonly}(\mathcal{K}).^{1} \quad (1)$$

  In other words, following the *causal dependency* relation $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$ backwards from visible transactions, we only see visible or read-only transactions.
- $u \sqsubseteq u'$: This condition captures the *monotonic reads* session guarantee, i.e., the view u' extends the view u.
- $\mathsf{RYW}(\mathcal{K}, \mathcal{K}', u')$: This condition expresses the *read-your-writes* (RYW) session guarantee, stating that each client sees all versions previously written by itself.
- $\mathsf{LWW}(\mathcal{K}, u, \mathcal{F})$: This captures the *last-write-wins* conflict resolution policy, whereby a client reads each key's latest version in its view.
- $\mathsf{wf}(\mathcal{K}, u)$ and $\mathsf{wf}(\mathcal{K}, u')$: This requires that the views u and u' are *wellformed*, i.e., they are atomic and contain indices that point to existing versions.
- $t^{cl}_{sn} \in \mathsf{nextTxids}(\mathcal{K}, cl)$: Transaction identifiers $t^{cl}_{sn} \in \mathit{TxID}$ are indexed by the issuing client cl and a (monotonically increasing) sequence number sn. This condition obtains a fresh transaction ID t^{cl}_{sn}, where the sequence number sn is larger than any of the client cl's sequence numbers used in $\mathcal{K}$.
- $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, t^{cl}_{sn}, u, \mathcal{F})$: The KVS $\mathcal{K}'$ is obtained from $\mathcal{K}$ by adding the operations described by the fingerprint $\mathcal{F}$: the writes append a new version with the writer ID t^{cl}_{sn} to the respective key's version list, and the reads add t^{cl}_{sn} to the respective versions' reader sets.

View extension The view extension event for TCCv is defined by the rule:

$$\frac{\mathcal{U}(cl) \sqsubseteq u \qquad \mathsf{wf}(\mathcal{K}, u)}{(\mathcal{K}, \mathcal{U}) \xrightarrow{\mathsf{xview}(cl,u)}_{\mathsf{TCCv}} (\mathcal{K}, \mathcal{U}[cl \mapsto u])}$$

and extends a client cl's view from $\mathcal{U}(cl)$ to a wellformed view u. It abstractly models that additional versions of certain keys become visible to the client.

3 Eiger-PORT+: An Overview

Causally-consistent transactions have attracted the attention of both academia and industry in recent years. Eiger [35] is among the first distributed databases providing TCCv. Eiger-PORT [38] improves Eiger's overall performance by optimizing its read-only transactions while sacrificing data convergence, thus allowing diverging client views of concurrent conflicting writes (to the same keys) as in TCC. We show that this sacrifice is unnecessary and design Eiger-PORT+ based on Eiger-PORT. Eiger-PORT+ provides TCCv with both read-only and

[1] This condition differs from the one presented in [50], but is equivalent.

write-only transactions.[2] The key idea of achieving convergent client views is to share with clients the *total order* of versions on a server, which has already been established by uniquely assigned timestamps across versions. In contrast, Eiger-PORT constructs individual, possibly different, orders per client. Eiger-PORT+'s read-only transactions satisfy the NOC properties and are therefore performance-optimal (PORT). This is achieved in the same way as for Eiger-PORT. In particular, both protocols' read operations use only a fixed number of timestamps as metadata. We now give a high-level description of both Eiger-PORT and Eiger-PORT+, starting with their commonalities and then highlighting their differences.

3.1 Timestamps

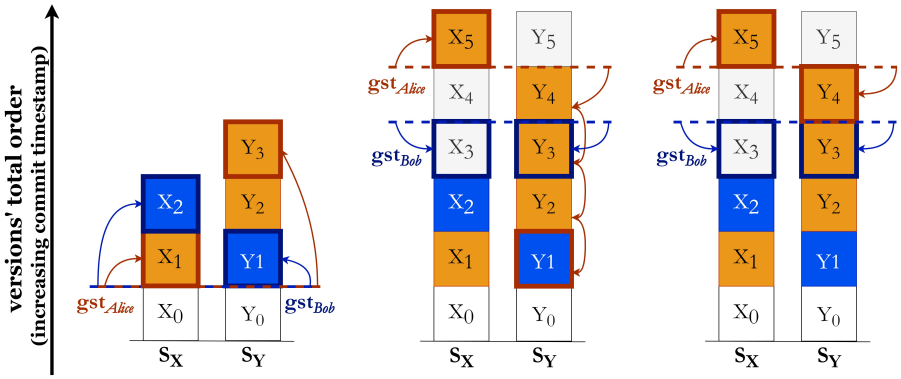
Both distributed transaction protocols leverage *timestamp-based* concurrency control. The timestamps are based on Lamport clocks [25], which clients and servers maintain and update with each local or communication event. Whenever a transaction commits a new version, the current clock reading is paired with the transaction's client ID to generate a globally unique commit timestamp. The lexicographic order of these pairs induces a total order on commit timestamps.

Each server maintains a *local safe time* `lst` that corresponds to the minimum of the uncommitted transactions' timestamps or, if there is none, the maximum committed timestamp for that server. Each client maintains a variable `lst_map`, which maps server IDs to their latest known `lst` value, and a *global safe time* `gst`. The latter is updated to the minimum timestamp in `lst_map` when a read-only transaction starts and acts as the *stable frontier* for that client: all transactions with earlier timestamps are guaranteed to be committed on all servers. Each read sent to a server includes `gst` as a read timestamp, which is used to safely read a committed version with a timestamp lower than the `gst`, or the client's latest own write (to achieve RYW), if its timestamp is higher than the `gst`.

3.2 Read and Write Transactions

Write transactions are similar in both protocols and follow a variant of the two-phase commit (2PC) protocol that always commits [35]. In the *prepare* phase, each timestamped write is sent to the corresponding partition, which adds the write to its local data store as a pending version. In the *commit* phase, each partition sets the version as committed, along with its commit timestamp. However, the two protocols differ in how they handle read transactions in the absence of an own write newer than `gst`. While Eiger-PORT+ always reads the *latest* version below `gst` in this case, Eiger-PORT searches for the latest version below `gst` that either has no write conflicts or is written by a different client. This backward scan is presumably done to maintain read atomicity (RA). However, we show that this scan is unnecessary for RA, can harm performance, and cause client view divergence. Consequently, in addition to providing convergence, Eiger-PORT+ improves performance by eliminating this scan's overhead.

[2] Eiger-PORT+'s pseudocode, together with its description, is given in [16, Appx. A].



(a) Initial (b) Eiger-PORT (c) Eiger-PORT+

Fig. 2: Alice and Bob reading (shown by arrows) from servers storing X and Y, illustrating convergence in Eiger-PORT+ and lack thereof in Eiger-PORT. Each square represents a version, and its color determines the version's writer, where orange, blue, and gray correspond to Alice, Bob, and other clients respectively.

Example 1. Figure 2 illustrates the difference between Eiger-PORT and Eiger-PORT+ in reading versions. In Figure 2a, Alice (orange) and Bob (blue) have written some versions on servers (partitions storing keys) X and Y, and their `gst`s are 0. Alice and Bob each perform a transaction reading from servers X and Y. In both protocols, Alice reads $\{X_1, Y_3\}$ and Bob $\{X_2, Y_1\}$ according to RYW.

As new versions are added to the servers, the `gst`s advance to higher timestamp values. Assume that versions Y_1 to Y_4 are conflicting writes, i.e., the transactions writing Y_2 to Y_4 had already started when Y_1 was committed. Alice and Bob then again read keys X and Y. Bob behaves the same in both protocols (cf. Figures 2b and 2c): He reads the last committed versions below its `gst`, $\{X_3, Y_3\}$, as he has no writes above its `gst` and the read versions are written by other clients. However, Alice's behavior differs in the two protocols. Given Alice's new `gst`, in Eiger-PORT she reads $\{X_5, Y_1\}$ (cf. Figure 2b), while in Eiger-PORT+ she reads $\{X_5, Y_4\}$ (cf. Figure 2c). As X_5 is her newest own write on X, this is the same for both protocols. But Alice has no newer own writes to Y and the latest committed version, Y_4, is Alice's write and has write conflicts. Thus, Eiger-PORT performs a scan to find the latest version written by a *different* client below Alice's `gst`, i.e., Y_1, while Eiger-PORT+ reads Y_4, irrespective of its writer or write conflicts.

Hence, in Eiger-PORT, Alice reads Y_3 before Y_1, while Bob reads Y_1 before Y_3, which results in diverging client views. This behaviour is allowed by TCC where convergence is not required. In contrast, in Eiger-PORT+, Bob reads Y_1 before Y_3 and Alice reads Y_3 before Y_4, both agreeing with the same convergent order, i.e., the versions' total order established on the servers ($Y_1 < Y_3 < Y_4$).

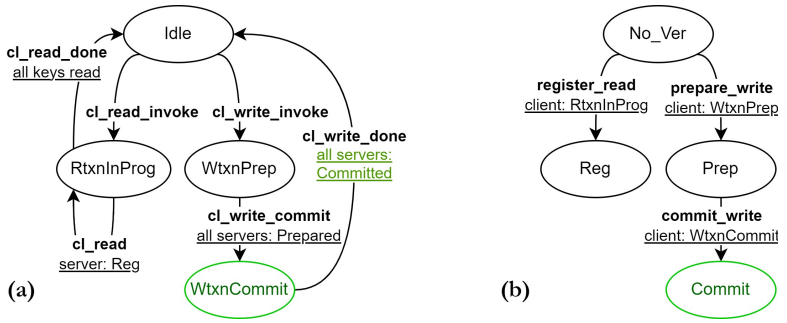


Fig. 3: Eiger-PORT+: state diagrams of (a) a client's `cl_state` and (b) a server's `svr_state` for a given transaction.

4 Formal Modeling and Verification

We formally model Eiger-PORT+ (Section 4.1), and use our proof technique (Section 4.2) to verify its TCCv isolation guarantee (Sections 4.3 and 4.4).

4.1 Formalizing Eiger-PORT+

To model the protocol, we consider a distributed KVS with one transaction coordinator per client and several servers that handle the clients' transactions. For simplicity, we integrate the coordinator into the client and assume that each server manages one key. We also assume clients execute transactions sequentially.

We formalize Eiger-PORT+ as an LTS. Its states are the protocol's global configurations, consisting of the clients' and the servers' local configurations. As depicted in Figure 3, these local configurations include control states indicating a protocol execution's progress. Each event changes either one client's or one server's configuration and advances its respective control state, ensuring that the clients and servers are independent components with interleaved events. We allow these components to directly access each other's local configurations to exchange information. This is a standard abstraction in protocol modeling, which can later be refined into explicit message-passing communication.

We next define our LTS model's configurations and describe the sequences of events associated with read and write transactions in more detail. We slightly deviate from the Isabelle syntax to stay closer to standard mathematical notation.

Configurations We model the client, server, and global configurations as records in Isabelle/HOL (Figure 4). The global configuration contains the client (`cls`) and server (`svrs`) local configurations, whose types are parameterized by the type `'v` of values, and three history variables, which we discuss in Section 4.3.

Besides the global safe time, `gst`, and the `lst_map`, already discussed above, the client configuration consists of its state, `cl_state`, a transaction sequence number, `cl_sn`, and the client's (Lamport) clock, `cl_clock`. The state is described by the type `txn_state`, which has four constructors for idle (`Idle`), read

```
(* transaction state *)
datatype 'v txn_state =
 Idle |
 RtxnInProg ts (key set) (key ⇀ 'v) |
 WtxnPrep (key ⇀ 'v) |
 WtxnCommit ts (key ⇀ 'v)

(* version state *)
datatype 'v ver_state =
 No_Ver |
 Reg |
 Prep ts ts 'v |
 Commit ts ts ts 'v (txid ⇀ ts × ts)
```

```
(* client configuration *)
record 'v cl_conf =
 cl_state : 'v txn_state
 cl_sn : sqn
 cl_clock : ts
 gst : ts
 lst_map : key → ts

(* server configuration *)
record 'v svr_conf =
 svr_state : txid → 'v ver_state
 svr_clock : ts
 lst : ts
```

```
(* global configuration *)
record 'v global_conf =
 cls : cl_id → 'v cl_conf
 svrs : key → 'v svr_conf
 rtxn_rts : txid ⇀ ts        (* three history variables *)
 wtxn_cts : txid ⇀ ts
 cts_order : key → txid list
```

Fig. 4: Eiger-PORT+: the client, server, and global configurations.

transaction in progress (`RtxnInProg`), write transaction prepare (`WtxnPrep`), and write transaction commit (`WtxnCommit`). The latter three states include a key-value map describing the values (to be) read or written.

A server's configuration consists of a function mapping each transaction ID to a version state (`ver_state`), the server's (Lamport) clock, and its local safe time (`lst`). A version state may either be idle (`No_Ver`), registered read (`Reg`) for a read transaction, or prepared (`Prep`) or committed (`Commit`) for a write transaction. The latter two states include timestamps in their parameters and the commit state includes a readermap to record information about the transactions reading this version (similar to the abstract model's reader sets).

Notation. To improve readability, we sometimes omit the projections `cls` and `svrs`, writing, e.g., `(gst s cl)` for `(gst (cls s cl))`.

Read-only transactions proceed as follows (cf. Figure 3). The `cl_read_invoke` event of a client starts a read-only transaction and transitions from `Idle` to `(RtxnInProg clk keys ∅)` state, where `clk` is the client's current clock reading, `keys` is the (finite and non-empty) set of keys to be read, and ∅ is the empty key-value mapping, where the subsequently read values will be recorded. As mentioned, this event also updates the client's global safe time, `gst`, to the minimum of the servers' local safe times stored in `lst_map`. As a result, more up-to-date versions of certain keys may become visible to the client.

Once a client has invoked a read, the involved servers (in `keys`) follow with a `register_read` event, where they transition from `No_Ver` to `Reg` state and access the client's `gst` to determine the latest own write newer than the `gst` (if any), or the latest transaction with a commit timestamp `cts` $\leq$ `gst`. This transaction is

```
1  definition cl_write_commit cl kv_map cts sn u clk s s' ⟷
2    —— guards:
3    cl_state s cl = WtxnPrep kv_map ∧
4    (∀k ∈ dom kv_map. is_prepared (svr_state s k (Tn sn cl)) ∧
5    cts = Max {get_prepared_ts (svr_state s k (Tn sn cl)) |
6               k ∈ dom kv_map} ∧
7    sn = cl_sn s cl ∧
8    u = views_of s cl ∧
9    clk = cts + 1 ∧
10   —— updates (unmentioned variables remain unchanged):
11   cl_state s' cl = WtxnCommit cts kv_map ∧
12   cl_clock s' cl = clk ∧
13   wtxn_cts s' = (wtxn_cts s)(Tn sn cl ↦ cts) ∧
14   cts_order s' = extend_cts_order s (Tn sn cl) cts kv_map
```

Fig. 5: Eiger-PORT+'s client commit event.

recorded in the read version's readermap along with the current `lst` and updated server clock. This information is then accessed in the client's subsequent `cl_read` event, reading the version's value and updating its own clock and `lst_map`. When the client has read all requested values, i.e., `dom kv_map = keys` holds for its state `(RtxnProg clk keys kv_map)`, the event `cl_read_done` brings it back to `Idle`.

Write-only transactions are initiated by the `cl_write_invoke` event, in which the client transitions from `Idle` to the state `(WtxnPrep kv_map)`. The key-value map `kv_map` describes the keys and associated values to be written and corresponds to the transaction's (write-only) fingerprint (Section 2.3). Once all servers have followed into their prepared state, the client can execute the `cl_write_commit` event, which is defined in Figure 5.

This event has eight parameters: the client ID `cl`, the key-value map `kv_map`, the commit timestamp `cts`, which is the maximum of the involved server's prepared timestamps for the current transaction `(Tn sn cl)` (lines 5-6), the client's current sequence number `sn` (line 7), the abstract view `u` (line 8, see Section 4.3), the updated Lamport clock `clk` (line 9), and the global states `s` and `s'` before and after the event. As `cts > cl_clock` always holds here, there is no need to take their maximum to determine `clk`. The guards at lines 3 and 4 require that the client is in the prepared state and that all involved servers have followed into their own prepared state. The client's state is updated to `(WtxnCommit cts kv_map)` (line 11) and the clock is updated (line 12). We will discuss the history variable updates at lines 13-14 in Section 4.3.

After the client's commit event, the involved servers commit the transaction on their side using `commit_write` events. When all servers have committed, the client executes the `cl_write_done` event to return to its idle state.

4.2 Proof Technique

We now discuss our protocol verification technique based on refinement, and a technique for commuting independent events required to complement refinement.

The main goal of our verification is to prove the following result, stating that all of Eiger-PORT+'s reachable states are allowed by the abstract model $\mathcal{I}_{\mathsf{TCCv}}$:

$$r_{\mathtt{EPP}}\ (\mathsf{reach}\ \mathtt{EPP}) \subseteq \mathsf{reach}\ \mathcal{I}_{\mathsf{TCCv}}. \tag{2}$$

Recall from Section 2.2, that this would follow from a proof of $\mathtt{EPP} \preccurlyeq_{r_{\mathtt{EPP}},\pi_{\mathtt{EPP}}} \mathcal{I}_{\mathsf{TCCv}}$, for suitable refinement mappings $r_{\mathtt{EPP}}$ and $\pi_{\mathtt{EPP}}$ on protocol states and events. However, such a direct proof would fail for the following reason.

Eiger-PORT+ uses timestamps to define an order on versions and to identify "safe-to-read" versions. Hence, to ensure that clients always read the latest version in their view, the refinement mapping must reconstruct the version lists of the abstract KVS in the order of their commit timestamps. Otherwise, the proof of the abstract guard LWW (Section 2.3) will fail. However, the *execution order* of commits and the *order of the associated commit timestamps* may not coincide. We call such commits *inverted commits*. Having inverted commits in an execution may require *inserting* a key's new version to its version list rather than *appending* it. Since the abstract model only ever appends new versions at the end of the version lists, the refinement proof alone would fail for executions with inverted commits.

To address this problem, we introduce an extra proof step to reorder the inverted commits before the refinement. To this end, we define a modified protocol model $\widehat{\mathtt{EPP}}$ that restricts transaction commits to those that do not introduce any inverted commits. We decompose the proof of (2) into the following two steps:

$$\mathsf{reach}\ \mathtt{EPP} = \mathsf{reach}\ \widehat{\mathtt{EPP}}, \tag{3}$$

$$r_{\mathtt{EPP}}\ (\mathsf{reach}\ \widehat{\mathtt{EPP}}) \subseteq \mathsf{reach}\ \mathcal{I}_{\mathsf{TCCv}}. \tag{4}$$

We prove (4) by the refinement $\widehat{\mathtt{EPP}} \preccurlyeq_{r_{\mathtt{EPP}},\pi_{\mathtt{EPP}}} \mathcal{I}_{\mathsf{TCCv}}$, which works for the restricted model. To prove (3), we use a proof technique based on Lipton's reduction method [26] to successively reorder inverted commits in executions by commuting causally independent events, while preserving the executions' final state.

4.3 Refinement Mapping

For the refinement proof, we need to find the refinement mappings $\pi_{\mathtt{EPP}}$ and $r_{\mathtt{EPP}}$. To define $\pi_{\mathtt{EPP}}$, we must identify protocol events that refine the abstract commit event and the abstract view extension event. The protocol events `cl_read_done` and `cl_write_commit` refine the abstract commit event since reads and writes are guaranteed to commit after these events are executed. The event `cl_read_invoke` refines the abstract view extension event, as this event updates a client's `gst` and thus extends its view. All other events refine skip.

To define $r_{\mathtt{EPP}}$, we must reconstruct an abstract configuration from Eiger-PORT+'s protocol configuration s, i.e.,

$$r_{\mathtt{EPP}}\ s = (\mathtt{kvs\_of}\ s,\ \mathtt{views\_of}\ s),$$

where the function ($\mathtt{kvs\_of}\ s$) reconstructs the abstract KVS and ($\mathtt{views\_of}\ s$) reconstructs the abstract client views. To help define these components, we add

history variables to the global configuration. We now describe these history variables and then the functions `kvs_of` and `views_of`.

History variables The global configuration includes three history variables. The variables `rtxn_rts` and `wtxn_cts` serve as shortcuts to respectively map transaction IDs directly to the read timestamp (`gst`) and commit timestamps of the corresponding read-only and write-only transactions. These variables get updated in the corresponding commit events. The variable `cts_order` maps each key to a list of client-committed write-only transactions, *ordered by their commit timestamps*. The client commit event extends `cts_order` by *inserting* the committed transaction's ID at the position corresponding to its commit timestamp into the transaction ID list of each key written by the transaction (line 13 in Figure 5). This variable is used to facilitate the reconstruction of the abstract KVS.

Abstract KVS The function `kvs_of` reconstructs the abstract KVS from the `cts_order` history variable by mapping each key's list of client-committed transactions to an abstract version list. For a given key `k` and transaction `t` in the list, we extract each version's value and readerset from server `k`'s (prepared or committed) state for `t`. Since the abstract model always reads the latest versions in a client's view, as expressed by the guard LWW of the abstract commit event (see Section 2.3), the `cts_order` must be sorted by commit timestamp.

Abstract views We define the function `views_of`, reconstructing the abstract views from Eiger-PORT+'s model configurations in two steps. We first construct a function `get_view`, where `(get_view s cl k)` denotes the set of client-committed transactions `t`, whose commit timestamp is less than or equal to the client's `gst`, i.e., `(wtxn_cts s t)` $\leq$ `(gst s cl)` or that are the client `cl`'s own transactions (for RYW). Second, we use `cts_order` to map the transactions IDs in the range of `get_view` to their positions in the `cts_order`, which correspond to indices into the abstract version lists.

4.4 Correctness: Eiger-PORT+ satisfies TCCv

We can now state our main result of this section.

Theorem 1 (Correctness of Eiger-PORT+). *The Eiger-PORT+ model* EPP *satisfies* TCCv*, i.e.,* r_{EPP} (reach EPP) $\subseteq$ reach $\mathcal{I}_{\mathsf{TCCv}}$.

As described in Section 4.2, we combine refinement and reduction in this proof. We devote the remainder of this subsection to sketching both parts of our proof, stated as Lemmas 1 and 2 below, followed by describing the invariants used in these proofs.

Restricted model and reduction proof We define the restricted model $\widehat{\mathsf{EPP}}$ by adding a guard to the event `cl_write_commit`, which requires that the unique commit timestamp `(cts, cl)` of the client's transaction is greater than any commit timestamp of a transaction in the `cts_order`. This ensures that the client commit only appends, but does not insert, the new transaction into the `cts_order`. For this model, we prove the following lemma.

Lemma 1. reach $\texttt{EPP}$ = reach $\widehat{\texttt{EPP}}$.

Proof (sketch). By construction of $\widehat{\texttt{EPP}}$, the inclusion "$\supseteq$" is easily shown by a refinement. For the inclusion "$\subseteq$", consider any execution e of $\texttt{EPP}$ ending in some state s. We prove by reduction that we can reorder all inverted commits in e of $\texttt{EPP}$, while preserving its final state s. We first show that the relevant events of two transactions with inverted commit timestamps are pairwise causally independent. We then prove that adjacent causally independent events in e can be commuted. Using a measure function on executions, we show that this process terminates in an execution $\widehat{e}$ without inverted commits ending in state s, thus an execution of $\widehat{\texttt{EPP}}$. Hence, any state reachable in $\texttt{EPP}$ is also reachable in $\widehat{\texttt{EPP}}$. □

Refinement proof Next, we establish a refinement between the restricted model $\widehat{\texttt{EPP}}$ and the abstract model $\mathcal{I}_{\mathsf{TCCv}}$ instantiated to TCCv, using the refinement mapping defined in Section 4.3.

Lemma 2. $r_{\texttt{EPP}}$ (reach $\widehat{\texttt{EPP}}$) $\subseteq$ reach $\mathcal{I}_{\mathsf{TCCv}}$.

Proof (sketch). We show guard strengthening and update correspondence for every event of $\widehat{\texttt{EPP}}$. This is easy for most events, which refine skip. The interesting cases are the read invoke event, which refines the abstract view extension event, and the client commit and read done events, refining the abstract commit event.

We focus here on the client commit event. The update correspondence proof relies on the absence of inverted commits in $\widehat{\texttt{EPP}}$ and thus client commits appending versions to KVS version lists. For guard strengthening, we must show that all guards of the abstract commit event (cf. Section 2.3) are implied by the concrete guards. We discuss the most interesting ones. View atomicity (part of view wellformedness) holds by construction, since all versions of a transaction have the same `cts` and the abstracted view includes all transactions with a `cts` below the client's `gst` (and also its own writes). Similarly, we prove that LWW holds, i.e., a client reads the latest version in its view. To show that $\mathsf{canCommit}$ holds, we prove an invariant stating that the clients' views remain closed under $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$. This proof in turn requires several invariants about timestamps. □

Invariants and lemmas Our proofs rely on numerous invariants and lemmas. We present the most important ones categorized as follows.

- **Freshness of transaction IDs**: The clients' current transaction ID is fresh, i.e., does not occur in the KVS until the commit.
- **Past and future transactions**: stating that the respective client and servers are in particular start states (e.g., `Idle`) or end states (e.g., `Commit`).
- **Views**: These invariants include view wellformedness, view closedness (for $\mathsf{canCommit}_{\mathsf{TCCv}}$), and session guarantees (monotonic reads and RYW).
- **Timestamps**: This category includes lemmas showing the monotonic increase of timestamps and the following invariant for any client `cl` and server `k`:

$$\texttt{gst s cl} < \texttt{lst\_map s cl k} < \texttt{lst s k} < \texttt{svr\_clock s k}.$$

This invariant states that the following timestamps are in a strictly increasing order: client `cl`'s global safe time, client `cl`'s entry for server `k` in its map of local safe times, server `k`'s local safe time, and `k`'s local clock value.

– **Client commit order**: The `cts_order` history variable is sorted by commit timestamps and contains only client-committed (and distinct) transactions.

Note that the first three categories are generic and many of their invariants directly imply related guards needed in the abstract commit event's refinement. The last two categories can easily be adapted to other timestamp-based protocols.

5 Deployment and Evaluation

We have implemented and deployed our Eiger-PORT+ protocol on a cluster for a comprehensive performance evaluation [17]. Eiger-PORT+ pushes the limit of the state-of-the-art, with *superior performance* and a *stronger isolation guarantee*.

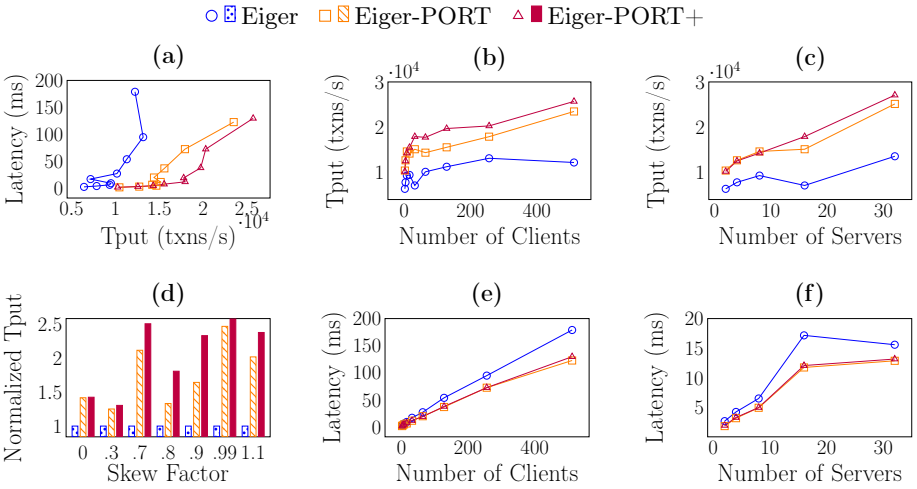


Fig. 6: Performance comparison among the Eiger-family protocols.

Deployment We implement Eiger-PORT+, along with Eiger and Eiger-PORT, in the same codebase, each consisting of around 12 kLoC in Java. We use Eiger-PORT's workload generator with default parameters of 32 threads per client, 1 million keys, 90% read proportion, and the Zipfian key-access distribution with a skew factor of 0.8. We deploy these three protocols on a CloudLab [13] cluster of machines, each with 2.4 GHz Quad-Core Xeon CPU and 12 GB RAM. By default, we use eight servers to partition the database and eight client machines to load the servers. We plot each data point using the average over five 60-second trials.

Evaluation Overall, Eiger-PORT+ is highly performant and superior to both competing protocols (Figure 6a). In particular, compared to the performance-optimal transaction protocol Eiger-PORT providing TCC, Eiger-PORT+ exhibits

higher throughput with a *stronger* isolation guarantee including convergence. Eiger-PORT+ also scales well with an increasing number of clients (Figure 6b) and servers (Figure 6c), with up to 1.8x (resp. 2.5x) throughput improvement over Eiger-PORT (resp. Eiger). In addition, despite varying skews, Eiger-PORT+'s throughput consistently surpasses that of its competitors (Figure 6d). This improvement becomes more pronounced with highly skewed workloads (or larger skew factors). This is because higher skewness results in increased concurrency, which would trigger additional server-side computation in Eiger-PORT and more rounds of communication in Eiger. Notably, despite its higher throughput and stronger isolation guarantee, Eiger-PORT+ demonstrates similar latency to Eiger-PORT (Figures 6e and 6f), which has been proven to be latency-optimal [36].

6 Related Work

We compare our work with other efforts on verifying transaction protocols, distinguishing them based on testing, model checking, and deductive verification.

Testing Previous work has devised various testers either for specific isolation levels, such as SI [20] and SER [49], or for a range of levels [19,22,24,28]. The underlying techniques are usually based on a characterization of anomalies, e.g., specified using dependency graphs [1] or axioms [8,9]. In contrast to our work, testing can only verify individual protocol executions and therefore can easily miss rarely occurring isolation bugs. On the other hand, while we are verifying a protocol model, testing can be done on the actual implementation.

Model checking The Maude model checker has been used to verify the RAMP and LORA protocols for RA [30,27], the Walter protocol for (parallel) SI [32], the ROLA protocol for Update Atomicity [31], and the MegaStore and P-Store protocols for SER [18,42]. Maude has also been used to verify various non-transactional consistency properties of the Cassandra key-value store [33]. Using the TLA+ model checker, the Azure CosmosDB has been verified against several non-transactional consistency properties [5] and TiDB against SI [45]. This model checker has also been used to verify some properties of concurrency control protocols other than isolation guarantees [23,53]. For model checking to be feasible, one must usually impose certain bounds (e.g., on the number of processes and transactions). In contrast, our work provides fully general verification results, which hold for arbitrary protocol executions.

Deductive verification Xiong et al. [50] also combine reduction and refinement of their abstract transaction model to prove that the COPS protocol [34] (with read-only transactions but single writes) satisfies TCCv and the Clock-SI protocol [12] satisfies SI. However, while being general, pen-and-paper proofs of such complex protocols are error-prone. Previous work on mechanized deductive verification, to the best of our knowledge, only covers either non-transactional consistency properties or serializability of textbook protocols. Chapar is a framework for verifying causal consistency of non-transactional KVSs. PVS and Event-B have been applied to verify (S)SER of the two-phase locking (2PL) protocol [10,51]. In

contrast, we have designed a new protocol, Eiger-PORT+, which is substantially more complex than 2PL, and we have verified that it satisfies TCCv.

7 Conclusion

We have designed Eiger-PORT+, a novel, causally-consistent, database transaction protocol, and formally verified its isolation guarantee of TCCv in Isabelle/HOL. In particular, TCCv was previously conjectured to be incompatible with transactional writes in the presence of performance-optimal read-only transactions. We have formally refuted this conjecture by our protocol design and its verification. Moreover, this case study represents the first complete formal verification of a complex distributed database transaction protocol. Our verification effort, excluding the verification framework, amounts to 10.3k lines of Isabelle/HOL code, composed of 0.7 kLoC for the model and 9.6 kLoC for the proof, which required 108 invariants. In addition, we have conducted a comprehensive evaluation, demonstrating Eiger-PORT+'s superior performance over two state-of-the-art protocols. We believe our protocol is an attractive choice for database applications opting for TCCv.

We see several avenues for future work. First, to facilitate formal protocol modeling and correctness proofs, we will develop an abstract distributed protocol model as an intermediate refinement step. This model will capture structure common to protocols and factor out recurring parts of correctness proofs. Second, we intend to support additional protocol features, for example, open-loop clients, which optimize transactional writes by immediately starting a new transaction, once they commit the previous one. An interesting case study in this context would be Eiger-NOC2 [29], a recent successor of Eiger-PORT+ that improves its performance by using open-loop clients and other features. However, this protocol has not yet been formally verified to provide TCCv. Third, we envision verifying our implementation and connecting it to our protocol verification results, possibly following the Igloo methodology [48].

Acknowledgements. We thank the anonymous reviewers for their valuable feedback. This research is supported by an ETH Zurich Career Seed Award and the Swiss National Science Foundation project 200021-231862 "Formal Verification of Isolation Guarantees in Database Systems".

References

1. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science (1999)
2. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Comput. **9**(1), 37–49 (1995)

3. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N.M., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: ICDCS 2016. pp. 405–414. IEEE Computer Society (2016)
4. Attiya, H., Ellen, F., Morrison, A.: Limitations of highly-available eventually-consistent data stores. In: PODC 2015. pp. 385–394. ACM (2015)
5. Azure: azure-cosmos-tla. https://github.com/Azure/azure-cosmos-tla (2022)
6. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with ramp transactions. ACM Transactions on Database Systems (TODS) **41**(3), 1–45 (2016)
7. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. ACM SIGMOD Record **24**(2), 1–10 (1995)
8. Biswas, R., Enea, C.: On the complexity of checking transactional consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 165:1–165:28 (2019)
9. Cerone, A., Gotsman, A.: Analysing snapshot isolation. J. ACM **65**(2), 11:1–11:41 (2018)
10. Chkliaev, D., Hooman, J., van der Stok, P.: Mechanical verification of transaction processing systems. In: ICFEM 2000. pp. 89–100. IEEE Computer Society (2000)
11. Didona, D., Guerraoui, R., Wang, J., Zwaenepoel, W.: Causal consistency and latency optimality: Friend or foe? Proc. VLDB Endow. **11**(11), 1618–1632 (2018)
12. Du, J., Elnikety, S., Zwaenepoel, W.: Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In: SRDS '13. pp. 173–184. IEEE Computer Society (2013)
13. Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., Mishra, P.: The design and operation of CloudLab. In: USENIX ATC'19. pp. 1–14 (Jul 2019)
14. ElectricSQL: https://electric-sql.com/ (2024)
15. Ghasemirad, S., Liu, S., Sprenger, C., Liu, S., Multazzua, L., Basin, D.: VerIso: Verifiable isolation guarantees for database transactions. Proc. VLDB Endow. **18** (2025), To appear.
16. Ghasemirad, S., Sprenger, C., Liu, S., Multazzu, L., Basin, D.: Pushing the limit: Verified performance-optimal causally-consistent database transactions. CoRR **abs/2411.07049** (2025). https://doi.org/10.48550/ARXIV.2411.07049
17. Ghasemirad, S., Sprenger, C., Liu, S., Multazzu, L., Basin, D.: Pushing the limit: Verified performance-optimal causally-consistent database transactions (artifacts) (Jan 2025). https://doi.org/10.5281/zenodo.14622073
18. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi. LNCS, vol. 8373, pp. 494–519. Springer (2014)
19. Gu, L., Liu, S., Xing, T., Wei, H., Chen, Y., Basin, D.: IsoVista: Black-box checking database isolation guarantees. Proc. VLDB Endow. **17**(12) (2024)
20. Huang, K., Liu, S., Chen, Z., Wei, H., Basin, D., Li, H., Pan, A.: Efficient black-box checking of snapshot isolation in databases. Proc. VLDB Endow. **16**(6), 1264–1276 (2023)
21. Jepsen: Jepsen Analyses (2024), https://jepsen.io/analyses
22. Jiang, Z.M., Liu, S., Rigger, M., Su, Z.: Detecting transactional bugs in database engines via graph-based oracle construction. In: OSDI'23. USENIX Association (2023)
23. Katsarakis, A., Ma, Y., Tan, Z., Bainbridge, A., Balkwill, M., Dragojevic, A., Grot, B., Radunovic, B., Zhang, Y.: Zeus: locality-aware distributed transactions. In: EuroSys '21. pp. 145–161. ACM (2021)

24. Kingsbury, K., Alvaro, P.: Elle: Inferring isolation anomalies from experimental observations. Proc. VLDB Endow. **14**(3), 268–280 (2020)
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (jul 1978)
26. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
27. Liu, S.: All in one: Design, verification, and implementation of SNOW-optimal read atomic transactions. ACM Trans. Softw. Eng. Methodol. **31**(3) (Mar 2022)
28. Liu, S., Gu, L., Wei, H., Basin, D.: Plume: Efficient and complete black-box checking of weak isolation levels. Proc. ACM Program. Lang. **8**(OOPSLA2) (2024)
29. Liu, S., Multazzu, L., Wei, H., Basin, D.: NOC-NOC: Towards Performance-optimal Distributed Transactions. Proc. ACM Manag. Data **2**(1) (mar 2024)
30. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016. ACM (2016)
31. Liu, S., Ölveczky, P.C., Wang, Q., Gupta, I., Meseguer, J.: Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. Formal Aspects Comput. **31**(5), 503–540 (2019)
32. Liu, S., Ölveczky, P.C., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store. In: WRLA '18. LNCS, vol. 11152, pp. 136–152. Springer (2018)
33. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: ICFEM '14. LNCS, vol. 8829, pp. 332–347. Springer (2014)
34. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: SOSP 2011. pp. 401–416. ACM (2011)
35. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: NSDI 2013. pp. 313–328. USENIX Association (2013)
36. Lu, H., Hodsdon, C., Ngo, K., Mu, S., Lloyd, W.: The SNOW theorem and latency-optimal read-only transactions. In: OSDI 2016. pp. 135–150. USENIX Association (2016)
37. Lu, H., Mu, S., Sen, S., Lloyd, W.: NCC: Natural concurrency control for strictly serializable datastores by avoiding the Timestamp-Inversion pitfall. In: OSDI '23. pp. 305–323. USENIX Association (2023)
38. Lu, H., Sen, S., Lloyd, W.: Performance-optimal read-only transactions. In: OSDI 2020. pp. 333–349. USENIX Association (2020)
39. Mehdi, S.A., Littley, C., Crooks, N., Alvisi, L., Bronson, N., Lloyd, W.: I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In: NSDI 2017. pp. 453–468. USENIX Association (2017)
40. Microsoft: Azure CosmosDB DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels (2024)
41. Neo4j: https://neo4j.com/ (2024)
42. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude. In: WADT '16. LNCS, vol. 10644, pp. 189–207. Springer (2016)
43. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM (JACM) **26**(4), 631–653 (1979)
44. Perrin, M., Mostefaoui, A., Jard, C.: Causal consistency: Beyond memory. SIGPLAN Not. **51**(8), 26:1–26:12 (Feb 2016)

45. PingCAP: tla-plus. https://github.com/pingcap/tla-plus (2022)
46. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, Seventh Edition. McGraw-Hill Book Company (2020), https://www.db-book.com/
47. Spirovska, K., Didona, D., Zwaenepoel, W.: Optimistic causal consistency for geo-replicated key-value stores. IEEE Trans. Parallel Distributed Syst. **32**(3), 527–542 (2021)
48. Sprenger, C., Klenze, T., Eilers, M., Wolf, F., Müller, P., Clochard, M., Basin, D.: Igloo: Soundly linking compositional refinement and separation logic for distributed systems verification. In: ACM Program. Lang. 4, OOPSLA, Article 152 (2020)
49. Tan, C., Zhao, C., Mu, S., Walfish, M.: Cobra: Making transactional key-value stores verifiably serializable. In: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020. pp. 63–80. USENIX Association (2020)
50. Xiong, S., Cerone, A., Raad, A., Gardner, P.: Data consistency in transactional storage systems: A centralised semantics. In: 34th European Conference on Object-Oriented Programming, ECOOP 2020. LIPIcs, vol. 166, pp. 21:1–21:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
51. Yadav, D., Butler, M.J.: Rigorous design of fault-tolerant transactions for replicated database systems using Event B. In: Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]. LNCS, vol. 4157, pp. 343–363. Springer (2006)
52. Yang, J., Yue, Y., Rashmi, K.V.: A large-scale analysis of hundreds of in-memory key-value cache clusters at Twitter. ACM Trans. Storage **17**(3), 17:1–17:35 (2021)
53. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: SOSP '15. p. 263–278. ACM (2015)

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

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

Certifiably Robust Policies for Uncertain Parametric Environments

Yannik Schnitzer(✉), Alessandro Abate, and David Parker

University of Oxford, Oxford, UK
{yannik.schnitzer,alessandro.abate,david.parker}@cs.ox.ac.uk

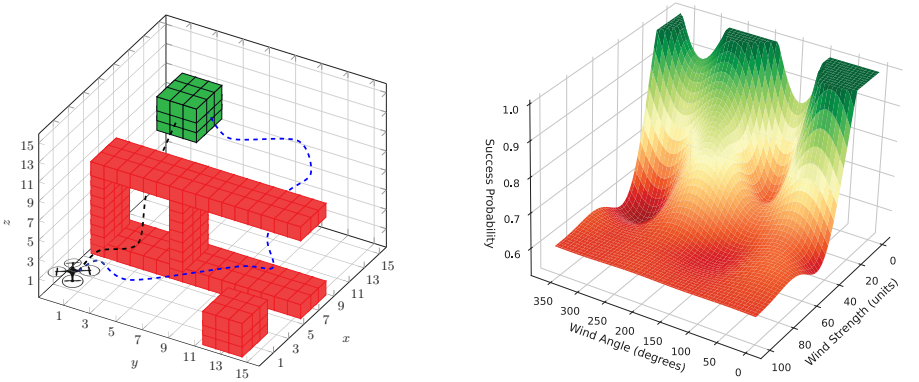
Abstract. We present a data-driven approach for producing policies that are provably robust across unknown stochastic environments. Existing approaches can learn models of a single environment as an interval Markov decision processes (IMDP) and produce a robust policy with a probably approximately correct (PAC) guarantee on its performance. However these are unable to reason about the impact of environmental parameters underlying the uncertainty. We propose a framework based on parametric Markov decision processes with unknown distributions over parameters. We learn and analyse IMDPs for a set of unknown sample environments induced by parameters. The key challenge is then to produce meaningful performance guarantees that combine the two layers of uncertainty: (1) multiple environments induced by parameters with an unknown distribution; (2) unknown induced environments which are approximated by IMDPs. We present a novel approach based on scenario optimisation that yields a single PAC guarantee quantifying the risk level for which a specified performance level can be assured in unseen environments, plus a means to trade-off risk and performance. We implement and evaluate our framework using multiple robust policy generation methods on a range of benchmarks. We show that our approach produces tight bounds on a policy's performance with high confidence.

1 Introduction

Ensuring the safety and robustness of autonomous systems in safety-critical tasks, such as unmanned aerial vehicles (UAVs), robotics or autonomous control, is paramount. A standard model for sequential decision making in these settings is a Markov decision process (MDP), which provides a stochastic model of the environment. However, real-world dynamics are complex, not fully known, and may evolve over time. Reasoning about *epistemic uncertainty*, which quantifies the lack of knowledge about the environment, can help construct *robust* policies that perform well across multiple possible stochastic environments.

Consider the UAV motion planning problem shown in Figure 1a, based on [6]. The goal is to navigate the drone safely to the target zone (green box) whilst avoiding obstacles (red regions). The drone's dynamics are influenced by weather conditions, such as wind strength or direction, potentially perturbing the drone from its intended route. These conditions can vary over time and may be difficult

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 63–83, 2025.
https://doi.org/10.1007/978-3-031-90660-2_4



(a) UAV motion planning environment with sample trajectories [6].

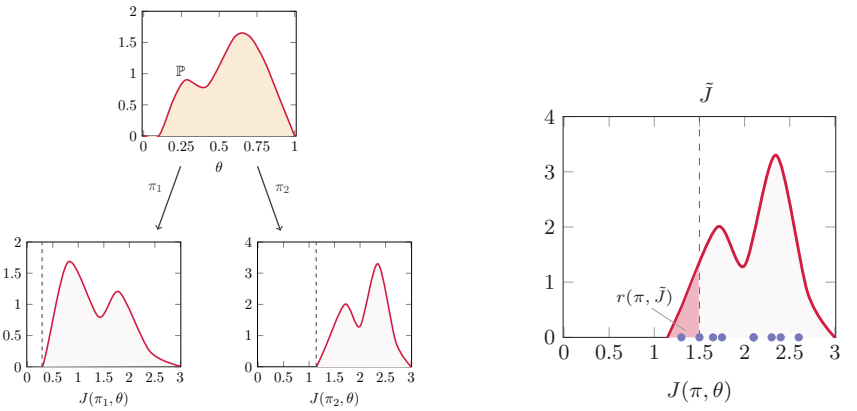
(b) Probability of task completion $J(\pi, \theta)$ for different parameter valuations θ.

Fig. 1: Example parametric environment with induced performance function.

to observe exactly. In low disturbance conditions (e.g., light wind), the drone can safely take the shorter route to the target (black dashed line). However, the drone should fly safely under all conditions, even if it flies overly cautiously in some. Therefore, a robust policy might take a detour through a less cluttered region of the environment (longer blue dashed line), ensuring a high probability of task completion even under more severe disturbances.

Epistemic uncertainty about the environment can be captured using *uncertain* MDPs, such as *interval MDPs* (IMDPs), which define a range of possible values for the probability of each transition between states of the model [21,53]. Under assumptions of independence, techniques such as robust dynamic programming [28,34] can then be used to efficiently generate *robust* policies for these IMDPs, i.e., policies that are optimal under *worst-case* assumptions about the true values of the transition probabilities. Furthermore, data-driven approaches, for example based on sampled trajectories through the environment, can be used to simultaneously learn both an IMDP and a robust policy for it [3,33,45,48], along with a probably approximately correct (PAC) guarantee on its performance.

In this paper, we present a general framework for synthesising provably robust policies in settings where environmental uncertainty is influenced by one or more *parameters*, e.g., wind strength/direction in the UAV example above. We model environments as *uncertain parametric MDPs* (upMDPs) [6], comprising a *parameter space* Θ of which each *parameter valuation* $\theta \in \Theta$ induces a standard MDP. Furthermore, an (unknown) distribution $\mathbb{P}$ over Θ represents the likelihood of each parameter valuation. Based on trajectories through multiple sampled instances of the environment, our goal is to produce *certifiably robust* policies. Instead of making worst-case assumptions across all possible parameter values, which can be overly conservative, we adopt a risk-based approach, providing a guaranteed level of performance for the policy with an associated *risk level* that



(a) Measure $\mathbb{P}$ and policy-induced densities over performance values $J(\pi_i, \theta)$.

(b) Performance guarantee $\tilde{J}$ and risk, estimated by sample performances (blue).

Fig. 2: For a fixed policy π, $J(\pi, \theta)$ is a random variable over performance values with measure $\mathbb{P}$ over valuations $\theta \in \Theta$ (left). We sample performances to bound the risk $r(\pi, \tilde{J})$, i.e., the probability for J to take a value less than $\tilde{J}$ (right).

quantifies the possibility of this level being violated. We also provide a tuning mechanism to adapt the trade-off between performance and risk.

To quantify the performance of a policy π in an MDP induced by parameter valuation $\theta \in \Theta$, we use an *evaluation function* $J(\pi, \theta)$. Typical examples include the probability to satisfy a specification expressed in temporal logics such as LTL [35] or PCTL [24] or an expected reward (see Section 2). For a fixed policy, J becomes a function in the valuations θ, as depicted in Figure 1b for the UAV example. When additionally considering the distribution $\mathbb{P}$ over parameter valuations, J becomes a random variable with respect to $\mathbb{P}$ describing the performance likelihood under policy π (see Figure 2a). Whereas the dashed vertical lines in Figure 2a indicate the worst-case performance of each policy, Figure 2b illustrates the risk measure $r(\pi, \tilde{J})$ that we use in this paper: the probability with which performance falls below a specified threshold $\tilde{J}$.

Deriving policies that are robust, i.e., which achieve high performance across either many or all possible environments, is a challenging problem. When Θ is finite, and assuming worst-case performance (i.e., ignoring $\mathbb{P}$), the model is referred to as a *multi-environment MDP*, for which finding an optimal robust memoryless policy is NP-hard, even for just two fully known environments [38]. For general upMDPs, recent work [39] finds robust policies but assumes that $\mathbb{P}$ can be sampled directly and that the resulting MDPs are fully known. In our setting, transition probabilities are unknown and are inferred from trajectories. This is comparable to the aforementioned work on PAC-learning of IMDPs [3,33,45,48], but these methods assume a single, fixed (but unknown) MDP.

In our work, there are *two layers* of uncertainty, resulting from (1) unknown parameter valuations inducing unknown MDPs, sampled from (2) the unknown

parameter distribution. In this setting, various learning-based methods known as *robust meta reinforcement learning* have been proposed [15,22,50]. Crucially, though, none of the existing learning algorithms are able to provide theoretical guarantees as to the performance of the generated policies in unseen environments; this is a core contribution of our framework.

An overview of our approach is illustrated in Figure 3. We assume access to multiple sampled environments $\mathcal{M}[\theta_i]$, each of which is an MDP induced by a parameter valuation θ_i from the unknown distribution $\mathbb{P}$. These are not fully known; instead we are able to access a set of sample trajectories from each one. In our UAV example, this equates to taking the drone outside on a new day and encountering a new set of environmental conditions (or a simulation of this).

Our framework divides the sample environments into two groups, a *training set* and a *verification* set. The training set is used to learn a robust policy. For this we build on existing IMDP-based policy learning methods and also consider robust meta reinforcement learning techniques. The verification set is used to derive the guarantees on the performance of the robust policy obtained from the training set. For this, we apply PAC IMDP learning to each unknown MDP in the verification set. Concretely, we use sample trajectories to infer IMDP overapproximations, which contain the true, unknown MDPs with a user-specified confidence. From those, we can derive lower bounds on the performance J of the learned policy in each of the environments, which hold with the specified confidence.

To tackle the higher layer of uncertainty and infer a bound for the policy's robust performance over the entire unknown distribution $\mathbb{P}$ underlying the sample MDPs, we develop a new approach based on *scenario optimisation* [11,12]. This takes samples of the performance J and a user-specified performance bound $\tilde{J}$ and provides a PAC guarantee on the probability of the performance on a new sample being less than $\tilde{J}$, i.e., the *risk*. However, in our setup we do not obtain samples of J directly, but derive lower bounds from the learned PAC IMDPs, which only hold up to a certain confidence. Our key theoretical contribution, presented in Section 3.3, is a generalisation of the scenario approach that can handle samples whose values are only known to lie in a confidence interval.

Our theoretical results combine the two layers of uncertainty: (1) the finite sampling of MDPs from the distribution $\mathbb{P}$, (2) the fact that sampled MDPs are unknown, so the performances of the learned policy are only inferable up to a certain confidence. The result is a single PAC guarantee on the policy's performance which holds with a high, user-specified confidence.

Furthermore, our framework allows tuning of the trade-off between performance guarantee and risk. By excluding the k worst-case sample environments, users can discard unlikely outliers, resulting in a higher performance guarantee at the cost of an increased risk bound, adjustable to the level the user considers admissible. We implement our framework as an extension of the PRISM model checker [31] and show that it can tightly quantify the performance and associated risk of learned policies on a range of benchmarks.

In summary, our contributions are: (1) a novel framework and techniques for producing certifiably robust policies in uncertain parametric MDPs for which

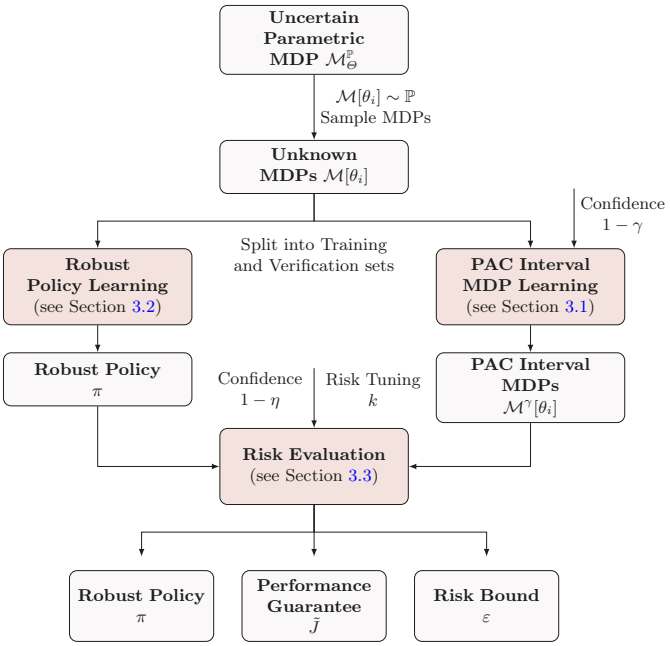


Fig. 3: Overview of our framework to derive performance and risk guarantees for policies learned on upMDPs. The setup includes two layers of uncertainty: we sample and analyse unknown environments from an unknown distribution.

both the parameters and transition probability functions are unknown; (2) new theoretical results which yield PAC guarantees on a policy's robust performance on unseen environments, where sample environments are unknown and can only be estimated from trajectories; (3) an implementation and evaluation of the framework on a range of benchmarks.

An extended version of this paper, with full proofs of all results, extended experiments and further technical details, can be found in [42].

1.1 Related Work

Epistemic uncertainty in MDPs has received broad attention across many areas, including formal methods, planning and reinforcement learning [5]. As mentioned above, there are various ways to model this using uncertain MDPs [21,53]. There are also techniques such as robust dynamic programming to synthesise robust policies for these models [28,34] and approaches to learn the models from trajectory data [3,33,45,48]. In this work, however, we investigate *parametric* uncertainty sets with unknown distributions over parameter valuations.

Uncertain parametric MDPs have emerged as a common model in meta reinforcement learning [15,19,20,22,23] and gained attention in formal methods [6,39]. On the one hand, meta reinforcement learning trains policies on multiple unknown environments sampled from an upMDP, using policy gradient methods [49], in

order to generalise to unseen environments. However, to our knowledge, none of these algorithms provide theoretical generalisation guarantees, either on their average [19,23] or their robust performance [15,22].

On the other hand, existing formal methods approaches to upMDPs do not offer the generality of meta RL setups. The work in [6] uses scenario methods and provides PAC guarantees, but for the existence of a policy that achieves a certain performance, not robust policy synthesis; they also require full knowledge of sampled parameter valuations and environments. In [39], concrete robust policies are synthesised with a PAC guarantee for performance on unseen environments, but this also relies on complete knowledge of all sampled valuations, reducing it to a special case of our approach. In our work, we address the very general problem of unknown sample environments; we target the scalability and generality of meta RL, while providing formal guarantees that are independent of the model size and previously unattainable for policy training methods like those in [39].

Also related is [16] which uses parametric MDPs in a Bayesian setting; parameter valuations are unknown but the model's transition functions are known and assumed to be defined by polynomial expressions. Other recent work in [14] combines, like us, the scenario approach and parametric MDPs, but for a different setting that assumes uniform distributions over parameter spaces. We also mention [7], which uses scenario optimisation with imprecise constraints to analyse continuous-time Markov chains with uncertain transition rates.

2 Preliminaries

We review the key formalisms used in our approach. Let $\Delta(S) = \{\mu \colon S \to [0,1] \mid \sum_s \mu(s) = 1\}$ denote the set of all probability distributions over a finite set S.

Definition 1 (Parametric MDP). *A* parametric Markov decision process *(pMDP) is a tuple $M_\Theta = (S, s_I, A, P_\Theta)$, where S and A are finite state and action spaces, $s_I \in \Delta(S)$ is the initial state distribution, and $P_\Theta \colon \Theta \times S \times A \to \Delta(S)$ is the parametric transition probability function over the parameter space Θ. Fixing a valuation $\theta \in \Theta$ induces a standard MDP $\mathcal{M}_\Theta[\theta]$, or $\mathcal{M}[\theta]$ for short, with transition kernel $P_\theta \colon S \times A \to \Delta(S)$ defined as $P_\theta(s,a,s') = P_\Theta(\theta,s,a,s')$.*

Parametric MDPs can be seen as an abstract model for a *set* of MDPs, i.e., they represent the instantiations induced by all possible valuations $\theta \in \Theta$. They are closely related to the model class of uncertain MDPs [34,53], in which each transition is associated with (potentially interdependent) sets of possible values.

Definition 2 (Uncertain Parametric MDP). *An* uncertain parametric Markov decision process *(upMDP) $\mathcal{M}^{\mathbb{P}}_\Theta = (\mathcal{M}_\Theta, \mathbb{P})$ is a pMDP $\mathcal{M}_\Theta$ with a (potentially unknown) probability measure $\mathbb{P}$ over the parameter space Θ.*

We assume that upMDPs are *graph preserving*, meaning that induced MDPs share a common topology: $\forall s, s' \in S, a \in A \colon (\forall \theta \in \mathrm{supp}(\mathbb{P}) \colon P_\theta(s,a,s') = 0) \vee (\forall \theta \in \mathrm{supp}(\mathbb{P}) \colon P_\theta(s,a,s') > 0)$. Although not strictly required, this assumption

can be crucial for efficiently solving learned IMDP approximations of the induced MDPs [33,53]. We describe in Section 3.4 how to lift this assumption by resorting to techniques which only approximate the performance and not the model.

Policies resolve action choices in MDPs and upMDPs. They are mappings $\pi\colon (S \times A)^* \times S \to \Delta(A)$, assigning a distribution over actions based on (finite) histories of states and actions. In this work, we focus on synthesising memoryless policies $\pi\colon S \to \Delta(A)$. While our theoretical results apply to arbitrary policy classes, learning and evaluating more expressive policies, such as those with finite memory, can be computationally expensive. We elaborate on this in Section 3.4.

Definition 3 (Evaluation Function). *For an upMDP $\mathcal{M}_\Theta^{\mathbb{P}} = (\mathcal{M}_\Theta, \mathbb{P})$, an* evaluation function $J\colon \Pi \times \Theta \to \mathbb{R}$ *maps a policy π and a parameter valuation θ to a performance value. We also denote by $J(\pi, \mathcal{M})$ the evaluation of policy π on an arbitrary MDP $\mathcal{M}$, such that $J(\pi, \theta) = J(\pi, \mathcal{M}[\theta])$.*

Typical evaluation functions include the reachability probability $\mathbf{Pr}^\pi_{\mathcal{M}}(\Diamond T)$ of eventually reaching a set of target states $T \subseteq S$, the reach-avoid probability $\mathbf{Pr}^\pi_{\mathcal{M}}(\neg C \,\mathcal{U}\, T)$ of reaching states in T while not entering a set of avoid states $C \subseteq S$, the expected reward $\mathbb{E}^\pi_{\mathcal{M}}(\Diamond T)$ accumulated before reaching a set T, given a reward structure, expected accumulated reward over finite or infinite time horizons, or more sophisticated properties expressed in temporal logics such as LTL [35] or PCTL [24]. Throughout this work's technical presentation, we assume that performance J is maximised. By changing the directions of optimisation and inequalities, our results also directly apply to the dual minimisation case.

For a fixed policy π and upMDP $\mathcal{M}_\Theta^{\mathbb{P}}$, the evaluation function J only depends on the valuations θ. Hence, J is a random variable with measure $\mathbb{P}$ (see Figure 2a). The *violation risk* is the probability that policy π achieves a value less than a stated performance guarantee $\tilde{J}$ on $\mathcal{M}_\Theta^{\mathbb{P}}$ (see Figure 2b).

Definition 4 (Violation Risk). *The* violation risk *of policy π for performance guarantee $\tilde{J} \in \mathbb{R}$, denoted by $r(\pi, \tilde{J})$, is defined as:*

$$r(\pi, \tilde{J}) = \mathbb{P}\left\{\theta \in \Theta\colon J(\pi, \theta) < \tilde{J}\right\}. \tag{1}$$

There is an inherent trade-off between violation risk and performance guarantee: a higher guarantee is associated with a higher risk, regardless of $\mathcal{M}_\Theta^{\mathbb{P}}$ and J.

The framework we present in this paper is based on learning and solving approximations of MDPs in the form of *interval MDPs* [21,53].

Definition 5 (Interval MDP). *An interval Markov decision process (IMDP) $\mathcal{M}^I = (S, s_I, A, P^I)$ is an MDP with a probabilistic interval transition function $P^I\colon S \times A \times S \to \mathbb{I}$, where $\mathbb{I} = \{[a, b] \mid 0 < a \leq b \leq 1\} \cup \{[0, 0]\}$ is the set of all graph-preserving intervals. We say that IMDP $\mathcal{M}^I$* includes *MDP $\mathcal{M}$, denoted by $\mathcal{M} \in \mathcal{M}^I$, if $\mathcal{M}$ and $\mathcal{M}^I$ share the same state and action spaces, and $P(s, a, s') \in P^I(s, a, s')$ for all $s, s' \in S, a \in A$.*

For IMDPs, we typically adopt a *robust* view of a policy's performance, i.e., the *worst-case* (minimum) value over any included MDP. We lift the evaluation function J to an IMDP $\mathcal{M}^I$ as follows:

$$J(\pi, \mathcal{M}^I) = \min_{\mathcal{M} \in \mathcal{M}^I} J(\pi, \mathcal{M}) \implies J(\pi, \mathcal{M}^I) \leq J(\pi, \mathcal{M}) \text{ for all } \mathcal{M} \in \mathcal{M}^I. \quad (2)$$

For key classes of properties used here (e.g., reachability probabilities, rewards), this value can be obtained via robust dynamic programming [28,34,53].

3 Learning Certifiably Robust Policies for upMDPs

This section presents our framework for computing performance and risk guarantees for learned policies in uncertain parametric MDPs. An overview of this framework was illustrated in Figure 3 and introduced in Section 1. We assume a fixed upMDP $\mathcal{M}^{\mathbb{P}}_{\Theta}$ and access to a sample set $\mathcal{D} = \{\mathcal{M}[\theta_i] \mid \theta_i \sim \mathbb{P}\}$ of unknown MDPs. This sample set is split into disjoint training and verification sets. The training set is used to compute a robust policy π, which is then evaluated on the verification set to derive a performance guarantee $\tilde{J}$ and bound the violation risk when the policy is deployed in an unseen environment sampled from $\mathbb{P}$. The overall goal is formally stated as follows.

Problem 1 *Given a upMDP $\mathcal{M}^{\mathbb{P}}_{\Theta}$ with unknown parameter distribution $\mathbb{P}$, an evaluation function J, and a confidence level $\eta > 0$, find a robust policy π, a (tight) performance guarantee $\tilde{J}$, and a (tight) risk bound $\varepsilon > 0$, such that:*

$$\Pr\left\{r(\pi, \tilde{J}) \leq \varepsilon\right\} \geq 1 - \eta.$$

There exist trivial solutions to Problem 1, such as selecting a very high risk bound or a very low performance guarantee. We aim to identify a tight solution that maximizes the performance guarantee with a precise, low risk bound.

The core part of our framework is the means to establish these performance and risk bounds for policies evaluated in unknown environments. From each unknown MDP in the verification set, we sample trajectories to learn an IMDP approximation that includes the MDP with high confidence. Solving these IMDPs yields probabilistic bounds on the policy's performance in each environment.

Our main theoretical results build upon scenario optimisation [11,12], the principal challenge being to incorporate the additional layer of uncertainty introduced by only being able to estimate the policy's performance in each unknown sampled environment. The result is a single PAC guarantee on the policy's performance in unseen environments, stated in Problem 1 above.

The process of establishing these guarantees is agnostic to the manner in which policies are produced. We consider two approaches, first taking a novel combination of existing methods for IMDPs and upMDPs [39,48], and secondly adopting a gradient-based technique from robust meta reinforcement learning [22].

The remainder of the section is structured as follows. Since PAC learning of IMDPs is used in multiple places, we discuss this first, in Section 3.1. Section 3.2 covers robust policy learning, Section 3.3 presents our main theoretical contributions and Section 3.4 describes several optimisations and extensions.

3.1 PAC IMDP Learning of Unknown MDPs

We follow established approaches for PAC learning of IMDP approximations introduced in [3,33,45,48]. Consider an unknown MDP $\mathcal{M}[\theta_i]$. We assume access to trajectories from $\mathcal{M}[\theta_i]$, consisting of sequences of triples (s, a, s') representing states, chosen actions and successor states. Leveraging the Markov property of MDPs, we treat each triple as an independent Bernoulli experiment to estimate the transition probability to state s' from s when choosing action a. We denote the number of times action a was chosen in state s across all sample trajectories as $\#(s, a)$, and the number of times this choice led to s' as $\#(s, a, s')$. The resulting point estimate for the transition probability is thus given by:

$$\tilde{P}(s, a, s') = \frac{\#(s, a, s')}{\#(s, a)}, \tag{3}$$

for $\#(s, a) > 0$. We construct an IMDP by transforming the point estimates into PAC confidence intervals [10]. Traditionally, this is achieved with Hoeffding's inequality [27,45]. Recent work has demonstrated that significantly tighter model approximations can be obtained by employing inequalities tailored to the binomial distribution, such as the Wilson score interval with continuity correction [33,54].

Let $1 - \gamma$ with $\gamma > 0$ be the desired confidence level for $\mathcal{M}[\theta_i]$ to be included in the IMDP, which we denote $\mathcal{M}^\gamma[\theta_i]$. This confidence is distributed over all n_u unknown transitions as $\gamma_P = \gamma / n_u$. Let $H = \#(s, a)$ and $\tilde{p} = \tilde{P}(s, a, s')$. For each unknown transition, the transition probability interval is given by:

$$P^\gamma(s, a, s') = [\max(\mu, \underline{p}_{wcc}), \min(\overline{p}_{wcc}, 1)], \tag{4}$$

with:

$$\underline{p}_{wcc} = \left(2H\tilde{p} + z^2 - z\sqrt{z^2 - \frac{1}{H} + 4H\tilde{p}(1-\tilde{p}) + 4\tilde{p} - 2} - 1\right) / \left(2(H + z^2)\right), \tag{5}$$

$$\overline{p}_{wcc} = \left(2H\tilde{p} + z^2 + z\sqrt{z^2 - \frac{1}{H} + 4H\tilde{p}(1-\tilde{p}) - 4\tilde{p} + 2} + 1\right) / \left(2(H + z^2)\right), \tag{6}$$

where z is the $1 - \frac{\gamma_P}{2}$ quantile of the standard normal distribution [33] and $\mu > 0$ is an arbitrarily small quantity to preserve the known graph structure. For unvisited state action pairs with $\#(s, a) = 0$, we set $P^\gamma(s, a, s') = [\mu, 1]$, for all s' in the known support. $P^\gamma(s, a, s')$ contains the true transition probability $P(s, a, s')$ with a confidence of at least $1 - \gamma_P$. By applying a union bound over the unknown transitions, we obtain the following overall guarantee:

Lemma 1 ([33]). *The true, unknown MDP $\mathcal{M}[\theta_i]$ is contained in its IMDP overapproximation $\mathcal{M}^\gamma[\theta_i]$ with probability at least $1 - \gamma$.* □

The confidence in the approximation of each environment is independent of the number or length of the trajectories analysed. However, more or longer trajectories generally lead to higher state-action counts, resulting in tighter intervals. In Section 4, we examine how the number of trajectories analysed influences the tightness of our performance guarantee and the associated risk.

3.2 Robust Policy Learning

We consider two distinct approaches to robust policy learning: *robust IMDP policy learning* and *robust meta reinforcement learning*. For the former, we propose a combination of techniques for robust policy synthesis for upMDPs with access to fully known sample environments [39] and IMDP learning for single unknown environments [48]. For the latter, we adopt a class of algorithms that optimise a policy's robust performance using policy gradient-methods [15,22].

Robust IMDP Policy Learning. Similarly to PAC IMDP learning in Section 3.1, we use sample trajectories to compute an IMDP overapproximation [48] for each unknown MDP in the training set. Then, like in [39], we combine models across the training set to perform policy synthesis. To obtain a policy that is robust across all samples, the learned IMDPs are combined by merging the transition intervals of each IMDP as $[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)]$. The resulting IMDP over-approximates all training MDPs, and the corresponding optimal deterministic policy considers the worst-case probability for each transition [53]. As the IMDPs for the training set are only used for policy synthesis and not for formal risk or performance analysis, we are not restricted to PAC IMDP learning. We can leverage a rich pool of interval learning algorithms, which, while lacking formal inclusion guarantees, provide empirically tighter intervals from fewer trajectories. A detailed overview and comparison of interval learning methods and their model approximation capabilities is conducted in [48]. We evaluate the best-performing approaches in our benchmarks in Section 4.

Robust Meta Reinforcement Learning. IMDP policies can be overly conservative, as they consider the worst-case scenario for each transition independently [39]. Additionally, IMDP learning produces memoryless deterministic policies. While sufficient for optimality in IMDPs, there exist upMDPs where an optimal robust policy requires randomization or memory [38].

Robust meta-reinforcement learning (RoML) extends classical reinforcement learning from a single MDP to upMDPs [15,22]. RoML employs policy gradient methods [49] to optimize a policy's performance across sampled training environments, estimating performance from sampled trajectories in each unknown environment. Unlike standard meta-RL, which maximises expected rewards across environments [9] and often yields strong average but poor worst-case performance, RoML prioritises robustness. It trains a policy using RL techniques to optimise the performance in the worst-case environment (via the max-min objective) [15] or in the α-quantile of worst-case environments (via a risk-aware CVaR objective) [22]. RoML differs from *robust RL* [18,52], which focuses on a single uncertain MDP, seeking a policy that is robust to its internal uncertainty. In contrast, RoML

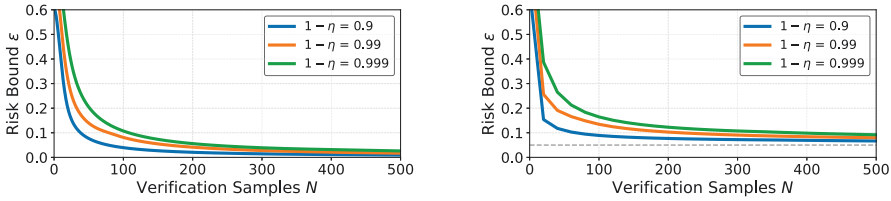


Fig. 4: Example risk bounds obtained from Theorem 1 (left) and Theorem 2 (right) for IMDP confidence $\gamma = 10^{-4}$. For Theorem 2, 5% of samples are discarded.

addresses multiple MDPs sampled from an unknown distribution, aiming for a policy that generalizes to the full distribution by achieving a strong robust performance across the sampled MDPs. This aligns well with our setup, and we refer the reader to [15,22] for further details on this approach.

3.3 Certifying Policy Performance and Risk in upMDPs

We now present our main theoretical contributions for quantifying the performance and violation risk of a policy π deployed in unseen environments of an upMDP. Specifically, we provide two results that derive PAC guarantees from lower bounds on π's performance, which we obtain by building and analysing PAC IMDPs (see Section 3.1) for the sampled environments that make up the verification set.

The first result extends the scenario approach [11,12] to incorporate additional uncertainty, as the lower bounds hold only with a certain confidence. This provides a PAC guarantee on the policy's performance, reasoning over the unknown distribution of true environments $\mathbb{P}$, based solely on estimations from sampled unknown environments attained by IMDP learning. The second result introduces flexibility in balancing the risk-performance trade-off. By extending a second result of the scenario approach, we allow the performance bound to be tuned by excluding worst-case outlier samples from the analysis, potentially leading to a higher performance guarantee at the cost of an increased risk bound.

Assume that the verification set comprises N sampled MDPs, from which we have learnt the PAC IMDPs $\{\mathcal{M}^\gamma[\theta_i]\}_{1\leq i\leq N}$. This establishes probabilistic lower bounds on the policy π's performance in the underlying unknown MDPs. From Equation (2), we have that $\mathcal{M}[\theta_i] \in \mathcal{M}^\gamma[\theta_i] \Rightarrow J(\pi, \mathcal{M}^\gamma[\theta_i]) \leq J(\pi, \mathcal{M}[\theta_i])$, where $J(\pi, \mathcal{M}^\gamma[\theta_i])$ can be obtained by standard solution methods for IMDPs, such as robust dynamic programming [28,34]. By Lemma 1, it follows that:

$$\mathbb{P}\{J(\pi, \mathcal{M}^\gamma[\theta_i]) \leq J(\pi, \mathcal{M}[\theta_i])\} \geq 1 - \gamma. \tag{7}$$

To bound the violation risk, we formulate the problem as a convex optimisation problem with randomised constraints—a *scenario program* [13]. We extend the generalisation theory of the *scenario approach* [11] to account for the uncertainty in the lower performance bounds (see Equation (7)). The detailed formulation and derivation of our generalisation can be found in [42].

Theorem 1. *Given N i.i.d. sample MDPs $\mathcal{M}[\theta_i]$ and IMDPs $\mathcal{M}^\gamma[\theta_i]$, such that $\mathbb{P}\{\mathcal{M}[\theta_i] \in \mathcal{M}^\gamma[\theta_i]\} \geq 1-\gamma$. For any policy π and confidence level $1-\eta$, with $\eta > 0$, it holds that:*

$$\mathbb{P}^N \left\{ r(\pi, \tilde{J}^\gamma) \leq \varepsilon(N, \gamma, \eta) \right\} \geq 1 - \eta, \tag{8}$$

where $\tilde{J}^\gamma = \min_i J(\pi, \mathcal{M}^\gamma[\theta_i])$, and $\varepsilon(N, \gamma, \eta)$ is the solution to the following, for any $K \leq N$:

$$\sum_{i=K}^{N} \binom{N}{i} (1-\gamma)^i \gamma^{N-i} - (1-\eta) = \sum_{i=0}^{N-K} \binom{N}{i} \varepsilon^i (1-\varepsilon)^{N-i}. \tag{9}$$

Proof sketch. The standard scenario approach in one dimension considers a set of i.i.d. performance samples $J_1, \ldots, J_N$ and provides a bound on the probability that the next sampled performance is lower than the minimum. In our case, we only have lower bounds on the actual performances, with $\mathbb{P}\{J_i^\gamma \leq J_i\} \geq 1-\gamma$. Assuming all lower bounds are valid with probability $(1-\gamma)^N$, we obtain an under-approximation of the true solution to the scenario program, and the union bound combines the confidences of the scenario approach and the validity of the lower bounds. However, this confidence becomes very small for large sample sizes N, even when $1-\gamma$ is close to 1. Conversely, with small sample sizes, the overall confidence remains low due to the weaker scenario confidence. To mitigate this, we require only K of the N lower bounds to be valid, which holds with high probability for small values of γ, even when K is close to N. As a result, we can exclude a small number $N-K$ of samples, assuming them to be violated under the scenario approach, thereby soundly over-approximating the risk. This only marginally increases the stated risk bound while significantly reducing the confidence overhead. Since we cannot specify which bounds are valid, we use the minimum over all lower performance bounds as an under-approximation of the solution to all scenario sub-programs with $N-K$ discarded constraints, providing a sound performance guarantee. The complete proof, including detailed bounds and derived inequalities, can be found in [42].

Theorem 1 bounds the risk for a policy to achieve a performance less than the minimum performance on any of the IMDPs. This bound only depends on values we can observe from the learned IMDP approximations. The theorem includes a tuning parameter $K \leq N$. The bound is valid for any value of K, and to obtain the tightest bound, we enumerate possible values and solve the resulting equation. For a fixed K, the left-hand side of Equation (9) is constant, and the right-hand side is the cumulative distribution function of a beta distribution with $K+1$ and $N-K$ degrees of freedom [13], which is easy to solve numerically for its unique solution in the interval $[0, 1]$ using bisection [6,40]. To the best of our knowledge, this is the first result to establish PAC guarantees on policy performance in unseen environments of upMDPs, in a setting where sample environments are unknown and can only be estimated from trajectories. Figure 4 illustrates the resulting risk bounds for example values. We assess the quality and tightness of our performance and risk bounds in the benchmarks presented in Section 4.

We extend Theorem 1 to allow tuning of the risk-performance trade-off by *discarding* samples [12]. Instead of bounding the risk for the policy to achieve a performance less than the minimum, we state a bound for the kth order statistic of the verification set. Users can choose k for a permissible risk level and a potentially higher performance guarantee, avoiding constraints from samples in the unlikely tail of the distribution.

Definition 6 (Order Statistic). *For a set of N samples $J_1, \ldots, J_N \in \mathbb{R}$ and $0 \leq k < N$, the kth order statistic $\tilde{J}_{(k)}$ is the kth smallest element when arranging all samples from smallest to largest.*

Theorem 2. *Given N i.i.d. sample MDPs $\mathcal{M}[\theta_i]$ and IMDPs $\mathcal{M}^\gamma[\theta_i]$, such that $\mathbb{P}\{\mathcal{M}[\theta_i] \in \mathcal{M}^\gamma[\theta_i]\} \geq 1-\gamma$, for any policy π, confidence level $1-\eta$ with $\eta > 0$, and number k of discarded samples, it holds that*

$$\mathbb{P}^N \left\{ r(\pi, \tilde{J}^\gamma_{(k)}) \leq \varepsilon_{(k)}(N, \gamma, \eta) \right\} \geq 1 - \eta, \tag{10}$$

where $\tilde{J}^\gamma_{(k)}$ is the kth order statistic of the performances $J(\pi, \mathcal{M}^\gamma[\theta_i])$, and $\varepsilon_{(k)}(N, \gamma, \eta)$ is the solution to the following, for any $K \leq N - k$:

$$\sum_{i=K}^{N-k} \binom{N-k}{i} (1-\gamma)^i \gamma^{N-k-i} - (1-\eta) = \sum_{i=0}^{N-K} \binom{N}{i} \varepsilon^i (1-\varepsilon)^{N-i}. \tag{11}$$

□

When $k = 0$ and no samples are discarded, Theorem 2 specialises to Theorem 1. The proof is an extension incorporating the additional uncertainty of the lower bounds obtained from the PAC IMDPs into the *sampling-and-discarding* theorem from scenario optimisation [12]. We detail the derivation of the bound in [42] and analyse its tightness in the experiments in Section 4.

3.4 Optimisations and Extensions

Finally in this section, we present some optimisations and extensions for our approach. First we show that, if there is additional knowledge as to the parametric structure of the upMDP, we can leverage this to obtain tighter approximations of the sample environments. Conversely, we describe how to seamlessly apply our framework and results to setups with *less* model knowledge, i.e., where not even the graph structure nor the (possibly infinite) state space is known. Furthermore, we outline how our results apply to more general setups where parameters influence not only transition probabilities, but also the evaluation functions, i.e., the specifications or *tasks* may vary across samples, aligning it with the setup commonly considered in meta reinforcement learning [15,22].

Model-based Optimisations. IMDP learning as described in Section 3.1 requires no knowledge of an MDP beyond its graph structure. However, additional information about the environment can yield tighter approximations with fewer samples. In cases where certain parameters, like temperature or air pressure, and their effect on some transition probabilities are known exactly, those transitions

can be treated as singleton intervals. This reduces the need for approximation and decreases the number of learned transitions n_u.

Additionally, we can apply *parameter tying* [36,37] to parameters appearing across different transitions. For instance, consider two transitions sharing the same parameterisation, $P_\Theta(s,a,s') = P_\Theta(t,b,t')$. We can combine the counts from both transitions since they represent the same Bernoulli experiment. Let $sim(s,a,s') = \{(t,b,t') \mid P_\Theta(s,a,s') = P_\Theta(t,b,t')\}$ denote the set of transitions with identical parametrisation. By plugging the combined counts, $\#^T(s,a,s') = \sum_{(t,b,t')\in sim(s,a,s')} \#(t,b,t')$ and $H^T(s,a,s') = \sum_{(t,b,t')\in sim(s,a,s')} \#(t,b)$ into Equations (3) and (4), we can obtain a tighter interval for both transitions.

Our experiments in Section 4 use model-based optimisations and the full evaluation in [42] compares the results with and without optimisations.

Statistical Model Checking. To approximate the performance of a learned policy in unknown sample environments, our framework is not limited to PAC IMDP learning. Various forms of statistical model checking (SMC) [1,26,32] can be applied, as long as they provide a lower bound J_i^γ on a policy's performance in a single environment induced by parameters θ_i, with a formally quantified confidence $\Pr\{J(\pi,\theta_i) \geq J_i^\gamma\} \geq 1-\gamma$. SMC techniques that require less information than PAC IMDP learning include those that rely on the minimum probability p_{min} potentially present in the model to infer the MDP's end-components with the desired confidence [3,17,33], or those that operate in a fully black-box setting with no model knowledge, approximating only the performance value. However, the latter techniques are typically restricted to finite-horizon properties [43,56].

Uncertain Specifications or Tasks. The meta reinforcement learning literature usually considers upMDPs where both transition probabilities and reward structure depend on parameters [19,22,23]. Our framework encompasses this problem class, and our results carry across directly. Parameterised rewards or specifications can be integrated into the evaluation function J, allowing parameters to affect both transitions and rewards. While the formal methods community has explored uncertain rewards and specifications to a lesser extent [4,41,44], we believe this is a promising direction for future work, particularly in extending PAC guarantees to uncertain specifications and objectives.

Non-memoryless Policies. In this work, we focus on synthesis of *memoryless* policies, which are sufficient for optimal performance under many common performance functions, such as reachability and reach-avoid specifications, in both single MDPs and IMDPs. However, for multiple environments or more complex specifications, such as general linear-time (LTL) properties, an optimal robust policy may require memory [34,38,55]. Our theoretical results in Theorems 1 and 2 apply to arbitrary policy classes, provided that policy performance can be evaluated on the learned IMDP approximations to obtain sample performance values. But solution of IMDPs in these cases presents challenges. For example, the standard automaton product constructions can be used to find finite-memory policies that optimise LTL specifications [55] but, in addition to increased compu-

Table 1: Salient characteristics of the evaluated benchmarks.

Benchmark	Evaluation J	Opt.	#Parameters	#States	#Transitions
UAV [6]	$\mathbf{Pr}(\neg C\,\mathcal{U}\,T)$	max	15	4096	86912
Aircraft Collision [30]	$\mathbf{Pr}(\neg C\,\mathcal{U}\,T)$	max	2	303	3468
Firewire [25]	$\mathbf{Pr}(\Diamond T)$	min	1	80980	112990
Semi-Auton. Vehicle [29]	$\mathbf{Pr}(\Diamond T)$	max	2	411	1503
Betting Game [8]	$\mathbb{E}(\Diamond T)$	max	1	480	2730
Chain [2]	$\mathbb{E}(\Diamond T)$	min	1	7	42

tational cost, the addition of memory means moving from a static to a dynamic uncertainty model [47,55], yielding overly conservative performance bounds.

4 Experimental Evaluation

We implemented our framework as an extension of the PRISM model checker [31], which provides trajectory generation and (robust) value iteration for MDPs and IMDPs[1]. We have evaluated our approach on a range of established benchmark environments used in similar work: an Aircraft Collision Avoidance [30], a Chain Problem [2], a Betting Game [8], a Semi-Autonomous Vehicle [29,46], the Firewire protocol [25], and the previously mentioned UAV [6]. Table 1 shows the salient features of the environments. We provide detailed descriptions of each benchmark, including the parameters and their distributions $\mathbb{P}$ in [42].

Setup. Since our approach is the first to provide policy performance guarantees under two layers of uncertainty, i.e., unknown sample environments from an unknown distribution, our experiments focus on assessing the quality of these guarantees. We study: (1) the tightness of the performance level $\tilde{J}$, assessing how closely our stated robust performance guarantee derived from the learned PAC IMDPs aligns with the actual robust performance on the true underlying sample MDPs that are hidden from the algorithm; (2) the quality of the risk bound ε derived from Theorems 1 and 2 with respect to the true violation risk $r(\pi, \tilde{J})$.

Our approach includes two sampling dimensions corresponding to the two layers of uncertainty: (1) the number of unknown MDPs induced by parameter valuations sampled from the distribution $\mathbb{P}$; (2) the number of sample trajectories generated in each unknown MDP. For the first dimension we consider a total of 600 sample MDPs, which we divide equally into training and verification sets. For the Firewire benchmark, we consider 150 verification samples. The second dimension is evaluated for up to 10^6 trajectories in each sampled environment.

For policy learning, we consider the two methods described in Section 3.2: robust IMDP policy learning and robust meta reinforcement learning with the max-min objective, implemented using the *Gymnasium* Python framework [51]. This illustrates the applicability of our approach to distinct state-of-the-art policy learning algorithms. We focus here on guarantees, rather than comparing policy learning methods, but we include statistics for both in [42] and refer the reader to, e.g., [15,22,48] for an in-depth comparison of methods.

[1] The implementation is available at: https://doi.org/10.5281/zenodo.14717176.

Table 2: Resulting performances, guarantees and risk bounds.

Benchmark	Performance J	Guarantee $\tilde{J}$	Risk Bound ε	Empirical Risk $r(\pi, \tilde{J})$	Risk Bound $\varepsilon_{(5)}$/Empirical Risk	Risk Bound $\varepsilon_{(10)}$/Empirical Risk	Runtime per 10^4 trajectories
UAV	0.7110	0.7100	0.027	0.003	0.052 / 0.023	0.075 / 0.057	1.51s
Aircraft Collision	0.5949	0.5900	0.027	0.004	0.052 / 0.017	0.075 / 0.046	0.35s
Firewire	0.1946	0.1967	0.055	0.004	0.103 / 0.039	0.146 / 0.081	14.9s
Semi-Auton. Vehicle	0.7854	0.7767	0.027	0.004	0.052 / 0.018	0.075 / 0.033	0.50s
Betting Game	30.78	30.65	0.027	0.005	0.052 / 0.016	0.075 / 0.026	1.12s
Chain	127.2	128.0	0.027	0.002	0.052 / 0.032	0.075 / 0.054	0.32s

For producing guarantees, we set the inclusion confidence level for the PAC IMDPs learned on the verification set to $\gamma = 10^{-4}$ and the overall confidence when applying Theorems 1 and 2 to $\eta = 10^{-2}$. Optimisations from Section 3.4 are applied (see [42] for results of their impact). All experiments were conducted on a 3.2 GHz Apple M1 Pro CPU with 10 cores and 16 GB of memory.

Results. Table 2 summarises the resulting performances and guarantees for the best-performing policy. All results are obtained after processing the full set of trajectories. We first give the *true robust performance* J, i.e., the policy's performance on the worst-case true MDP, which is hidden from the algorithm. We then report the key outputs of our approach: the *robust performance guarantee* $\tilde{J}$, representing the worst-case performance on the learned PAC IMDPs, and the *risk bound* ε for the performance guarantee $\tilde{J}$, obtained using Theorem 1. We also show an empirical estimate of the *true violation risk* $r(\pi, \tilde{J})$, computed over 1000 fresh sample MDPs. To evaluate the bounds derived via Theorem 2, we include the risk bounds $\varepsilon_{(k)}$ for discarding $k = 5$ and $k = 10$ worst-case samples, alongside estimates of the corresponding true violation risks.

For example, in the UAV case study, the table shows that the learned policy achieves at least $J = 0.711$ probability of reaching the goal without crashing into an obstacle on any sampled true MDP hidden from the algorithm. The learned IMDP approximations certify a minimum performance of $\tilde{J} = 0.71$, with the probability of performing worse on a fresh sample MDP bounded by $\varepsilon = 0.027$. On 1000 fresh samples, the policy actually performed worse in only 0.3% of cases.

Figure 5 shows the learning process and the derived performance guarantee for the Betting Game benchmark. We plot the true robust performance J (solid line), and the robust performance guarantee $\tilde{J}$ (dashed line) against the number of trajectories processed for each unknown MDP. We depict the progress for robust IMDP policy learning (purple) and robust meta reinforcement learning (yellow). The dotted red line corresponds to the *existential guarantee* [6], i.e., the minimum performance on any MDP from the verification set, when applying the individual optimal policies, which constitutes a natural upper bound on robust policy performance. Figure 6 illustrates the risk-tuning with performance guarantees obtained via Theorem 2. We depict the performances on the PAC IMDPs learned for the verification set (pink dots) and the performance guarantees $\tilde{J}_{(k)}$ when discarding the $k = 0, 5,$ and 10 worst-case samples (dashed lines), corresponding to the risk bounds $\varepsilon_{(k)}$ in Table 2. The full results for all policy learning techniques and benchmarks can be found in [42].

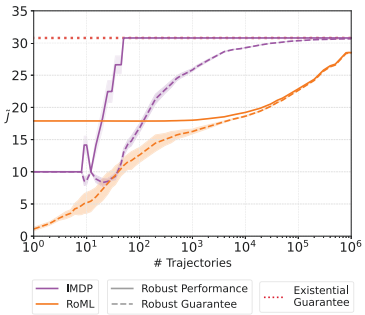


Fig. 5: True robust performances J and guarantees $\tilde{J}$ against number of processed trajectories (betting game).

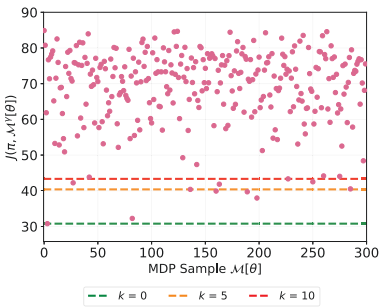


Fig. 6: Robust performance guarantees $\tilde{J}_{(k)}$, when discarding the $k = 0, 5$, and 10 worst-case samples (betting game).

Discussion. The results show that our framework generates tight bounds on the performance and risk of learned policies in upMDPs. Our approach effectively addresses and integrates the two layers of uncertainty: (1) an unknown environment distribution and (2) unknown sample environments. Our results yield high-quality risk bounds for the performance of policies in unseen environments. They enable tuning the risk-performance trade-off, and despite incorporating two layers of uncertainty, provide useful bounds with high user-specified confidence, constituting the first PAC guarantee for this general setup. While policy learning and solving PAC IMDPs scales with the model size and the number of sample MDPs, the computation of the risk bounds via Theorems 1 and 2 depends solely on the number of verification samples N and is independent of the model size. Regarding scalability, we briefly note that the range of model sizes we handle (see Table 1) includes the largest instances handled by comparable methods that perform PAC IMDP learning from trajectories [3,48,33].

5 Conclusion

We have presented a novel approach for producing certifiably robust policies for MDPs with epistemic uncertainty, where transition probabilities depend on parameters with unknown distributions. We have demonstrated that our approach yields tight bounds on a policy's performance in unseen environments from the same distribution. Future work includes extending certifiably robust policies to settings where the specification or task is also uncertain and parameter-dependent.

Acknowledgments. This work was part funded by the ARIA projects SAINT and Super Martingale Certificates, the UKRI AI Hub on Mathematical Foundations of AI and the ERC under the European Union's Horizon 2020 research and innovation programme (FUN2MODEL, grant agreement No. 834115).

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018)
2. Araya-López, M., Buffet, O., Thomas, V., Charpillet, F.: Active learning of MDP models. In: EWRL. Lecture Notes in Computer Science, vol. 7188, pp. 42–53. Springer (2011)
3. Ashok, P., Kretínský, J., Weininger, M.: PAC statistical model checking for Markov decision processes and stochastic games. In: CAV (1). Lecture Notes in Computer Science, vol. 11561, pp. 497–519. Springer (2019)
4. Bacci, G., Hansen, M., Larsen, K.G.: Model checking constrained Markov reward models with uncertainties. In: QEST. Lecture Notes in Computer Science, vol. 11785, pp. 37–51. Springer (2019)
5. Badings, T., Simão, T.D., Suilen, M., Jansen, N.: Decision-making under uncertainty: Beyond probabilities (2023)
6. Badings, T.S., Cubuktepe, M., Jansen, N., Junges, S., Katoen, J., Topcu, U.: Scenario-based verification of uncertain parametric MDPs. Int. J. Softw. Tools Technol. Transf. **24**(5), 803–819 (2022)
7. Badings, T.S., Jansen, N., Junges, S., Stoelinga, M., Volk, M.: Sampling-based verification of CTMCs with uncertain rates. In: CAV (2). Lecture Notes in Computer Science, vol. 13372, pp. 26–47. Springer (2022)
8. Bäuerle, N., Ott, J.: Markov decision processes with average-value-at-risk criteria. Math. Methods Oper. Res. **74**(3), 361–379 (2011)
9. Beck, J., Vuorio, R., Liu, E.Z., Xiong, Z., Zintgraf, L.M., Finn, C., Whiteson, S.: A survey of meta-reinforcement learning. CoRR **abs/2301.08028** (2023)
10. Boucheron, S., Lugosi, G., Massart, P.: Concentration Inequalities - A Nonasymptotic Theory of Independence. Oxford University Press (2013)
11. Campi, M.C., Garatti, S.: The exact feasibility of randomized solutions of uncertain convex programs. SIAM J. Optim. **19**(3), 1211–1230 (2008)
12. Campi, M.C., Garatti, S.: A sampling-and-discarding approach to chance-constrained optimization: Feasibility and optimality. J. Optim. Theory Appl. **148**(2), 257–280 (2011)
13. Campi, M.C., Garatti, S.: Introduction to the scenario approach. SIAM (2018)
14. Chi, Z., Liu, Y., Turrini, A., Zhang, L., Jansen, D.N.: A scenario approach for parametric Markov decision processes. In: Principles of Verification (2). Lecture Notes in Computer Science, vol. 15261, pp. 234–266. Springer (2024)
15. Collins, L., Mokhtari, A., Shakkottai, S.: Task-robust model-agnostic meta-learning. In: NeurIPS (2020)
16. Costen, C., Rigter, M., Lacerda, B., Hawes, N.: Planning with hidden parameter polynomial MDPs. Proceedings of the AAAI Conference on Artificial Intelligence **37**(10), 11963–11971 (Jun 2023)
17. Daca, P., Henzinger, T.A., Kretínský, J., Petrov, T.: Faster statistical model checking for unbounded temporal properties. In: TACAS. Lecture Notes in Computer Science, vol. 9636, pp. 112–129. Springer (2016)
18. Derman, E., Mankowitz, D., Mann, T., Mannor, S.: A Bayesian approach to robust reinforcement learning (2019)
19. Finn, C., Abbeel, P., Levine, S.: Model-agnostic meta-learning for fast adaptation of deep networks. In: ICML. Proceedings of Machine Learning Research, vol. 70, pp. 1126–1135. PMLR (2017)

20. Ghosh, D., Rahme, J., Kumar, A., Zhang, A., Adams, R.P., Levine, S.: Why generalization in RL is difficult: Epistemic POMDPs and implicit partial observability (2021)
21. Givan, R., Leach, S.M., Dean, T.L.: Bounded-parameter Markov decision processes. Artif. Intell. **122**(1-2), 71–109 (2000)
22. Greenberg, I., Mannor, S., Chechik, G., Meirom, E.A.: Train hard, fight easy: Robust meta reinforcement learning. In: NeurIPS (2023)
23. Gupta, A., Mendonca, R., Liu, Y., Abbeel, P., Levine, S.: Meta-reinforcement learning of structured exploration strategies. In: NeurIPS (2018)
24. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects Comput. **6**(5), 512–535 (1994)
25. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS (1). Lecture Notes in Computer Science, vol. 11427, pp. 344–350. Springer (2019)
26. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: VMCAI. Lecture Notes in Computer Science, vol. 2937, pp. 73–84. Springer (2004)
27. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Springer Series in Statistics (1994)
28. Iyengar, G.N.: Robust dynamic programming. Math. Oper. Res. **30**(2), 257–280 (2005)
29. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.: Safety-constrained reinforcement learning for MDPs. In: TACAS. Lecture Notes in Computer Science, vol. 9636, pp. 130–146. Springer (2016)
30. Kochenderfer, M.: Decision Making Under Uncertainty: Theory and Application (2015)
31. 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)
32. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: RV. Lecture Notes in Computer Science, vol. 6418, pp. 122–135. Springer (2010)
33. Meggendorfer, T., Weininger, M., Wienhöft, P.: What are the odds? improving the foundations of statistical model checking. CoRR **abs/2404.05424** (2024)
34. Nilim, A., Ghaoui, L.E.: Robust control of Markov decision processes with uncertain transition matrices. Oper. Res. **53**(5), 780–798 (2005)
35. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57. IEEE Computer Society (1977)
36. Polgreen, E., Wijesuriya, V.B., Haesaert, S., Abate, A.: Data-efficient Bayesian verification of parametric Markov chains. In: QEST. Lecture Notes in Computer Science, vol. 9826, pp. 35–51. Springer (2016)
37. Poupart, P., Vlassis, N., Hoey, J., Regan, K.: An analytic solution to discrete Bayesian reinforcement learning. In: ICML. ACM International Conference Proceeding Series, vol. 148, pp. 697–704. ACM (2006)
38. Raskin, J., Sankur, O.: Multiple-environment Markov decision processes. In: FSTTCS. LIPIcs, vol. 29, pp. 531–543. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2014)
39. Rickard, L., Abate, A., Margellos, K.: Learning robust policies for uncertain parametric Markov decision processes. In: Proc. L4DC'24. Proceedings of Machine Learning Research, vol. 242, pp. 876–889. PMLR (2024)
40. Ross, S.M.: Introduction to Probability Models. Academic Press (2014)

41. Scheftelowitsch, D., Buchholz, P., Hashemi, V., Hermanns, H.: Multi-objective approaches to Markov decision processes with uncertain transition parameters. In: VALUETOOLS. pp. 44–51. ACM (2017)
42. Schnitzer, Y., Abate, A., Parker, D.: Learning provably robust policies in uncertain parametric environments. CoRR **abs/2408.03093** (2024)
43. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: CAV. Lecture Notes in Computer Science, vol. 3114, pp. 202–215. Springer (2004)
44. Steimle, L.N., Kaufman, D.L., Denton, B.T.: Multi-model Markov decision processes. IISE Trans. **53**(10), 1124–1139 (2021)
45. Strehl, A.L., Littman, M.L.: A theoretical analysis of model-based interval estimation. In: ICML. ACM International Conference Proceeding Series, vol. 119, pp. 856–863. ACM (2005)
46. Stückler, J., Schwarz, M., Schadler, M., Topalidou-Kyniazopoulou, A., Behnke, S.: Nimbro explorer: Semiautonomous exploration and mobile manipulation in rough terrain. Journal of Field Robotics (2015)
47. Suilen, M., Badings, T.S., Bovy, E.M., Parker, D., Jansen, N.: Robust markov decision processes: A place where AI and formal methods meet. In: Principles of Verification (3). Lecture Notes in Computer Science, vol. 15262, pp. 126–154. Springer (2024)
48. Suilen, M., Simão, T.D., Parker, D., Jansen, N.: Robust anytime learning of Markov decision processes. In: NeurIPS (2022)
49. Sutton, R.S., McAllester, D.A., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: NIPS. pp. 1057–1063. The MIT Press (1999)
50. Teh, Y.W., Bapst, V., Czarnecki, W.M., Quan, J., Kirkpatrick, J., Hadsell, R., Heess, N., Pascanu, R.: Distral: Robust multitask reinforcement learning. In: NIPS. pp. 4496–4506 (2017)
51. Towers, M., Kwiatkowski, A., Terry, J.K., Balis, J.U., Cola, G.D., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J.J., Tan, H., Younis, O.G.: Gymnasium: A standard interface for reinforcement learning environments. CoRR **abs/2407.17032** (2024)
52. Wang, Y., Zou, S.: Online robust reinforcement learning with model uncertainty (2021)
53. Wiesemann, W., Kuhn, D., Rustem, B.: Robust Markov decision processes. Math. Oper. Res. **38**(1), 153–183 (2013)
54. Wilson, E.B.: Probable inference, the law of succession, and statistical inference. Journal of the American Statistical Association (1927)
55. Wolff, E.M., Topcu, U., Murray, R.M.: Robust control of uncertain markov decision processes with temporal logic specifications. In: CDC. IEEE (2012)
56. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV. Lecture Notes in Computer Science, vol. 2404, pp. 223–235. Springer (2002)

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

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

Quantum and GPU

AutoQ 2.0: From Verification of Quantum Circuits to Verification of Quantum Programs

Yu-Fang Chen[1(✉)], Kai-Min Chung[1], Min-Hsiu Hsieh[2], Wei-Jia Huang[2], Ondřej Lengál[3(✉)], Jyun-Ao Lin[4(✉)], and Wei-Lun Tsai[1(✉)]

[1] Institute of Information Science, Academia Sinica, Taipei, Taiwan
gulu0724@gmail.com, alan23273850@gmail.com
[2] Hon Hai Quantum Computing Research Center, Taipei, Taiwan
[3] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
lengal@fit.vutbr.cz
[4] Innovation Frontier Institute of Research for Science and Technology & Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei, Taiwan
jyalin@gmail.com

Abstract. We present a verifier of quantum programs called AutoQ 2.0. Quantum programs extend quantum circuits (the domain of AutoQ 1.0) by classical control flow constructs, which enable users to describe advanced quantum algorithms in a formal and precise manner. The extension is highly non-trivial, as we needed to tackle both theoretical challenges (such as the treatment of measurement, the normalization problem, and lifting techniques for verification of classical programs with loops to the quantum world), and engineering issues (such as extending the input format with a support for specifying loop invariants). We have successfully used AutoQ 2.0 to verify two types of advanced quantum programs that cannot be expressed using only quantum circuits: the *repeat-until-success* (RUS) algorithm and the weak-measurement-based version of Grover's search algorithm. AutoQ 2.0 can efficiently verify all our benchmarks: all RUS algorithms were verified instantly and, for the weak-measurement-based version of Grover's search, we were able to handle the case of 100 qubits in ~20 minutes.

1 Introduction

Quantum *programs* are an extension of quantum *circuits* that provide users with greater control over quantum computing by allowing them to use more complex programming constructs like branches and loops. Some of the most advanced quantum algorithms cannot be defined by quantum circuits alone. For example, certain class of programs, such as the *repeat-until-success* (RUS) algorithms [41] (which are commonly used in generating special quantum gates) and the weak-measurement-based version [6] of Grover's search algorithm [31], use a loop with the condition being a classical value (0 or 1) obtained by measuring a particular qubit. This added expressivity presents new challenges, particularly in terms of verification. The additional complexity comes from the *measurement* operation, where a particular qubit is measured to obtain a classical value (and the quantum state is partially collapsed, which might require *normalization*), and reasoning about control flow induced by *branches* and *loops*.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 87–108, 2025.
https://doi.org/10.1007/978-3-031-90660-2_5

In classical program verification, a prominent role is played by *deductive verification* [30,34,32], represented, e.g., by the tools Dafny [37], KeY [4], Frama-C [8], VeriFast [36], VCC [23], and many more. These tools only require the users to provide specifications in the form of pre- and post-conditions, along with appropriate loop invariants. The rest of the proving process is entirely (in the ideal case) automated. Unfortunately, in the realm of quantum computing, similar fully automated deductive verification tools are, to the best of our knowledge, missing. Advanced tools for analysis and verification of quantum programs—based on, e.g., *quantum Hoare logic* and the tool CoqQ [50] or the *path-sum* formalism [5] and the tool Qbricks [14]—are quite powerful but require a significant amount of human effort.

To bridge this gap, we present AutoQ 2.0, a major update over AutoQ 1.0 [19] with an added support for *quantum programs* (AutoQ 1.0 only supported quantum *circuits*). In AutoQ 1.0, given a triple $\{P\}\, C\, \{Q\}$, where P and Q are the pre- and post-conditions recognizing sets of (pure) quantum states (represented by *tree automata*) and C is a quantum circuit, we can verify if all quantum states in P reach some state in Q after executing C. In AutoQ 2.0, we addressed several key challenges to make the support of quantum programs possible. First, we need to handle *branch* statements. The key issue here is to handle *measurement* of quantum states whose value is used in a branch condition. For this we developed automata-based algorithms to compute the quantum states after the measurement (Section 5). The second challenge is the handling of *loop* statements. Similarly to deductive verification of classical programs, we require the users to provide *an invariant* for each loop. With the loop invariant provided, we developed a framework handling the rest of the verification process fully automatically. Moreover, we show that a naive implementation of the measurement operation will encounter the *probability amplitude normalization* problem. This is handled by designing a new algorithm for *entailment testing* (Section 6).

Under this framework, the preconditions, postconditions, and invariants are all described with a new automata model called *level-synchronized tree automata (LSTAs)* [1]. LSTAs are specifically designed to efficiently encode quantum states and gate operations. As the core data structure of the tool, we provide a formal definition of LSTAs in Section 2.2 to facilitate the presentation of our new entailment testing approach.

We used AutoQ 2.0 to verify various quantum programs using the *repeat-until-success* (RUS) paradigm [41], as well as the weak-measurement-based version [6] of Grover's search [31] (Section 7). AutoQ 2.0 can efficiently verify all our benchmarks. The verification process for all RUS algorithms was instantaneous and for the weakly measured versions of Grover, we were able to handle the case of 100 qubits in ~20 min. To the best of our knowledge, AutoQ 2.0 is currently the only tool for verification of quantum programs with such a degree of automation.

Related work. Our work aligns with Hoare-style verification of quantum programs, a topic extensively explored in prior studies [51,43,48,27,39]. This approach, inspired by D'Hondt and Panangaden, utilizes diverse Hermitian operators as quantum predicates, resulting in a robust and comprehensive proof system [25]. However, specifying properties with Hermitian operators is often non-intuitive and difficult for automation due to their vast matrix sizes. Consequently, these methods are typically implemented us-

ing proof assistants like Coq [9], Isabelle [45], or standalone tools built on top of Coq, like CoqQ [50]. These tools require substantial manual effort in the proof search. The Qbricks approach [15] addresses the challenge of proof search by combining cutting-edge theorem provers with decision procedures, leveraging the Why3 platform [29]. Nevertheless, this approach still demands considerable human intervention.

In the realm of automatic quantum software analysis tools, *circuit equivalence checkers* [5,22,33,47,23] prove to be efficient but less flexible in specifying desired properties, primarily focusing on equivalence. These tools are valuable in compiler validation, with notable examples being QCEC [13], Feynman [5], and SliQEC [17,44]. *Quantum model checking*, supporting a rich specification language (various temporal logics [28,40,46]), is, due to its limited scalability, more suited for verifying high-level protocols [7]. QPMC [28] stands out as a notable tool in this category. Quantum abstract interpretation [49,42] over-approximates the reachable state space to achieve better scalability, but so far handles only circuits. The work in [52,26] aims at the verification of *parameterized quantum programs* like *variational quantum eigensolver (VQE)* or *quantum approximate optimization algorithm (QAOA)*. However, the correctness properties they focused are very different from what AutoQ 2.0 can handle. While the mentioned tools are fully automated, they serve different purposes or address different phases of the development cycle compared to AutoQ 2.0.

2 Background

Before we start, we first provide a minimal background needed for this work.

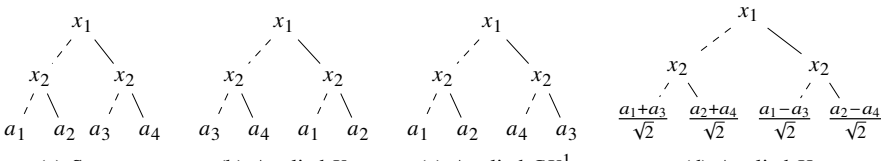
2.1 The Tree-View of Quantum States

In a traditional computer system with n bits, a state is represented by n Boolean values 0 or 1. An n-qubit *quantum state* can be viewed as a "probabilistic distribution" over n-bit classical states. Here we often refer to each classical state as a *computational basis state* or *basis state* for short. Hence a quantum state can be represented by a *binary tree* whose branches correspond to the computational basis states and leaves correspond to *complex probability amplitudes*[5].

In Fig. 1(a), we can see an example of a 2-qubit state q that maps basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ to probability amplitudes a_1, a_2, a_3, and a_4, respectively. The left-going dashed line denotes the 0-branch, and the right-going solid line denotes the 1-branch. A quantum state can also be represented as a formal sum of basis states multiplied by their amplitudes, e.g., we can represent the state q as $a_1|00\rangle + a_2|01\rangle + a_3|10\rangle + a_4|11\rangle$.

Quantum gates are fundamental operations performed on quantum states. Basic quantum gates and their effects on the state q from Fig. 1(a) are shown in Figs. 1(b) to 1(d). To specify the target qubit to which a single qubit gate U is applied, a subscript number i is used. For example, X_i denotes the application of the X gate, which is also known as the quantum "negation" gate, to the i-th qubit. The effect of this gate is to swap the 0- and 1-subtrees under all x_i nodes (cf. Fig. 1(b)). On the other hand, for a controlled gate, a superscript number i is used to indicate the control qubit, while a subscript number j is used for the target qubit. The most notable example is the controlled-X gate CX^i_j, which

[5] A state with complex amplitude $a+bi$ has the probability $|a+bi|^2 = a^2+b^2$ of being observed.



(a) State q (b) Applied X_1 (c) Applied CX_2^1 (d) Applied H_1

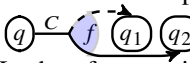
Fig. 1. The effect of applying selected quantum gates to state q

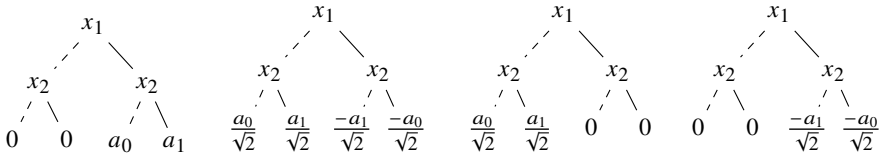
applies the X_j gate to 1-subtrees under (assuming $i < j$) all x_i nodes (cf. Fig. 1(c)). Observe that after applying an H gate (the *Hadamard* gate, which creates a quantum superposition; cf. Fig. 1(d)), there is a *normalization factor* $\frac{1}{\sqrt{2}}$ on all leaves to keep the sum of probabilities one. This normalization factor can be derived from the tree leaf values $(a_1 + a_3)$, $(a_2 + a_4)$, $(a_1 - a_3)$, and $(a_2 - a_4)$ and the fact that $\sum_{i=1}^{4} |a_i|^2 = 1$.

2.2 Level-Synchronized Tree Automata

As we mentioned in Section 1, the new algorithms introduced in this work are built on top of LSTAs, making it essential to provide a formal definition. Readers may choose to skim this section initially and refer back to it for details as needed later.

Trees. Our framework is based on the concept of perfect binary trees. A *perfect binary tree* T is a map from *tree nodes* $\bigcup_{0\le i\le n}\{0,1\}^i$, for some $n \in \mathbb{N}_0 := \mathbb{N}\cup\{0\}$, to a nonempty set of symbols Σ, i.e., $T\colon \bigcup_{0\le i\le n}\{0,1\}^i \to \Sigma$. All nodes $v \in \bigcup_{0\le i<n}\{0,1\}^i$ are *internal* and have children nodes $v.0$ (left) and $v.1$ (right) where '.' denotes concatenation (we denote the empty string by ϵ). All nodes $v \in \{0,1\}^n$ are *leaves* and have no children. A node v's *height* is its word length, denoted $|v|$. A node v is at tree level i when $|v| = i$. We denote T's height by n. Perfect binary trees can be used to represent quantum states or vectors of the size 2^n. For instance, the quantum state of Fig. 1(a) corresponds to a perfect binary tree $T = \{\epsilon \mapsto x_1, 0 \mapsto x_2, 1 \mapsto x_2, 00 \mapsto a_1, 01 \mapsto a_2, 10 \mapsto a_3, 11 \mapsto a_4\}$ of height 2. Children of the node 1 are 10 and 11, and the leaf node 10 has no children.

LSTAs. A *(symbolic) level-synchronized tree automaton (LSTA)* [1] is a tuple $\mathcal{A} = \langle Q, \mathbb{N}\cup\mathsf{term}(\mathbb{C},\mathbb{X}), \Delta, \mathcal{R}, \varphi\rangle$ where Q is a finite set of *states*, $\mathcal{R} \subseteq Q$ is a set of *root states*, $\mathsf{term}(\mathbb{C},\mathbb{X})$ is a set of terms obtained from complex numbers $\mathbb{C}$ and a set of *complex variables* $\mathbb{X}$ using function symbols from some fixed theory (in this paper, we will use $\mathbb{N}$ for internal node symbols and $\mathsf{term}(\mathbb{C},\mathbb{X})$ for leaf symbols). Δ is a set of *transitions* of the form $\delta_i = q \xrightarrow{f|C} (q_1, q_2)$ (*internal transitions*) and $\delta_\ell = q \xrightarrow{f|C} ()$ (*leaf transitions*), where $q, q_1, q_2 \in Q$, $f \in \mathsf{term}(\mathbb{C},\mathbb{X})$, and $C \subseteq \mathbb{N}$ is a finite set of *choices*. In the rest of the paper, we also draw the internal transition δ_i and leaf transition δ_ℓ as  and  , respectively, to provide a more intuitive presentation. In the aforementioned form, we call q, f, C, q_1, q_2, and $\{q_1, q_2\}$ the *top*, *symbol*, *choices*, *left*, *right*, and *bottom*, respectively, of the transition δ_i, and denote them by $\mathtt{top}(\delta_i)$, $\mathtt{sym}(\delta_i)$, $\mathtt{ch}(\delta_i)$, $\mathtt{left}(\delta_i)$, $\mathtt{right}(\delta_i)$, and $\mathtt{bot}(\delta_i)$, respectively. Needless to



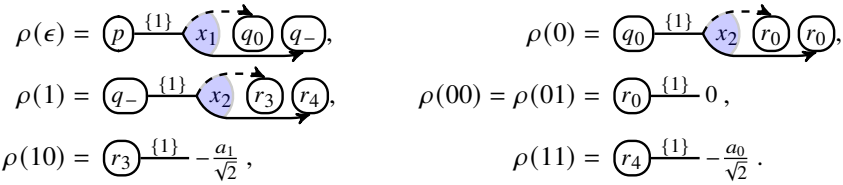
(a) The initial state (b) Applied $H_1; CX_2^1$ (c) $M_1 = 0$ (d) $M_1 = 1$

Fig. 2. Intermediate states during the execution of Algorithm 1

say, $\texttt{bot}(\delta_\ell) = \emptyset$. We further require the choices of different transitions with the same top state to be disjoint, i.e., $\forall \delta_1 \neq \delta_2 \in \Delta\colon \texttt{top}(\delta_1) = \texttt{top}(\delta_2) \implies \texttt{ch}(\delta_1) \cap \texttt{ch}(\delta_2) = \emptyset$. Finally, the *global constraint* φ is a formula used to restrict the values of $\mathbb{X}$ to those that satisfy φ (if not stated, it is assumed to be "true"). For instance, when $\mathbb{X} = \{a, b\}$, we can set $\varphi = |a| > |b|$ to restrict the allowed valuation of a and b.

Semantics of LSTAs. A *run* of an LSTA $\mathcal{A}$ on a tree T is a map $\rho\colon \mathrm{dom}(T) \to \Delta$ from tree nodes to transitions of $\mathcal{A}$ such that for each node $v \in \mathrm{dom}(T)$, when v is an internal node, $\rho(v)$ is of the form $q \xrightarrow{T(v)|C} (\texttt{top}(\rho(v.0)),\ \texttt{top}(\rho(v.1)))$. When v is a leaf node, $\rho(v)$ is of the form $q \xrightarrow{T(v)|C} ()$.

We give a *run* ρ of the LSTA $\mathcal{A}$ in Fig. 3 on the tree T in Fig. 2(d) (used later in Algorithm 1) as follows:



A run ρ is *accepting* if $\texttt{top}(\rho(\epsilon)) \in \mathcal{R}$ and all transitions used at the same level share some common choice, i.e., $\forall 0 \leq i \leq n\colon \bigcap\{\texttt{ch}(\delta) \mid \delta \in \{\rho(v) \mid v \in \{0,1\}^i\}\} \neq \emptyset$ (this is the essential difference from standard tree automata that gives LSTAs the power to compactly represent some classes of quantum states). The *language* of $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$ is a set of trees T over $\mathbb{N} \cup \mathsf{term}(\mathbb{C}, \mathbb{X})$ such that there exists an accepting run of $\mathcal{A}$ over T. Given a tree T over $\mathbb{N} \cup \mathsf{term}(\mathbb{C}, \mathbb{X})$ and an assignment $\sigma\colon \mathbb{X} \to \mathbb{C}$, we use $T[\sigma]$ to denote the tree obtained from T by (i) substituting all occurrence of variables $x \in \mathbb{X}$ in T by $\sigma(x)$ and (ii) evaluating all terms in the resulting tree into values $c \in \mathbb{C}$.

3 Overview

In this section, we provide an overview of automata-based quantum program verification with a running example (chosen for its simplicity). In the example, the quantum program creates the effect of a non-standard quantum gate "$-X$" (applying the X gate and negating all amplitude values) using the standard gates X, H, and CX (Algorithm 1). The program operates on a 2-qubit system and performs the "$-X$" gate on the second qubit when the

first qubit is measured to be 1; otherwise, (the first qubit is 0 after measurement) the state stays unchanged.

For all a_0, $a_1 \in \mathbb{C}$, we verify Algorithm 1 against the precondition $\{a_0 |10\rangle + a_1 |11\rangle\}$, which allows the state with the first qubit $|1\rangle$ and the second qubit $a_0 |0\rangle + a_1 |1\rangle$, and the postcondition $\{a_0 |10\rangle + a_1 |11\rangle\,,\ -a_1 |10\rangle - a_0 |11\rangle\}$, which includes the original state and the state after the "$-X_2$" gate.

Algorithm 1: "$-X_2$" if $M_1 = 1$

```
1 Pre: {a0 |10⟩ + a1 |11⟩};
2 H1; CX2^1;
3 if M1 = 0 then {X1};
4 Post: {a0 |10⟩ + a1 |11⟩ ,
5        −a1 |10⟩ − a0 |11⟩};
```

Our approach first constructs two LSTAs P and Q that can recognize the states (binary trees) of the pre- and post-conditions, respectively, and then computes another LSTA by executing the gates $H_1; CX_2^1$ from P (see Fig. 2(a) for the only quantum state accepted by P) with the gate algorithm introduced in [1]. This results in an LSTA P_1 that recognizes the state shown in Fig. 2(b). After applying the operator M_1 to measure x_1 (Line 3), P_1 splits into two copies. One copy, P_2, accepts the only quantum state shown in Fig. 2(c), where the first qubit is measured to be 0. The other copy, P_3, accepts the only quantum state shown in Fig. 2(d), where the first qubit is measured to be 1.

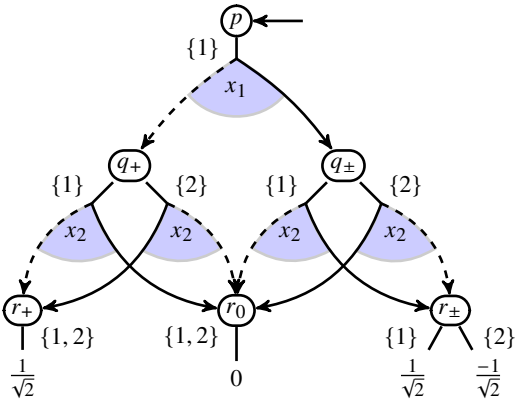
It is important to note that the probability amplitudes of the quantum states from Figs. 2(c) and 2(d) have not been normalized yet. To do that, we need to multiply all leaves with the normalization factor $\sqrt{2}$. This will ensure that the square sum of their absolute amplitude values becomes 1. Although the quantum states are not yet normalized, we, however, still have sufficient information to obtain the corresponding normalized states. In AutoQ 2.0, we choose to ignore all normalization factors and design a new entailment testing algorithm (Section 6) that can detect the equivalence of two non-normalized states. After both the true and false paths of the **if** statement in the example are processed, we obtain two LSTAs P_2 and P_3 capturing all reachable states. We then construct their union and test if all states in the union are included in the post-condition (recognized by Q) by testing entailment.

A drawback of Algorithm 1 is that the desired effect "$-X$" manifests only if $M_1 = 1$. In the case $M_1 = 0$, we need to run the same algorithm again until we get a measurement of 1. To achieve this, we can use a **while** loop statement, as shown in Algorithm 2. The loop allows us to repeatedly execute the same branch statement until the desired outcome is achieved.

Algorithm 2: "$-X_2$"

```
1 Pre: {a0 |10⟩ + a1 |11⟩};
2 H1; CX2^1;
3 Inv: {a0/√2 |00⟩ + a1/√2 |01⟩ − a1/√2 |10⟩ − a0/√2 |11⟩};
4 while M1 = 0 do {X1; H1; CX2^1};
5 Post: {−a1 |10⟩ − a0 |11⟩};
```

To verify that the loop works correctly, we require the user to provide a loop invariant in the form of an LSTA. The invariant here is $\{\frac{a_0}{\sqrt{2}} |00\rangle + \frac{a_1}{\sqrt{2}} |01\rangle - \frac{a_1}{\sqrt{2}} |10\rangle - \frac{a_0}{\sqrt{2}} |11\rangle\}$ (cf. Fig. 2(b)). The verification process then involves checking if the invariant is *inductive*, covers all reachable states before entering the loop, and does not contain any state that would violate the post-condition. More details on the verification process will be given in Section 5.3. With the loop invariant provided, we can ensure that the algorithm ends up with a state where the "$-X$" gate is applied to the second qubit when it terminates.



(a) LSTA

```
Constants
c+ := 1 / sqrt2
c0 := 0
c- := -1 / sqrt2
Root States p
Transitions
[1,{1}](q+, q+-) -> p
[2,{1}](r+, r0) -> q+
[2,{2}](r0, r+) -> q+
[2,{1}](r0, r+-) -> q+-
[2,{2}](r+-, r0) -> q+-
[c+,{1,2}] -> r+
[c0,{1,2}] -> r0
[c+,{1}] -> r+-
[c-,{2}] -> r+-
```

(b) The `.lsta` file.

Fig. 4. The LSTA for Bell states and its textual description

In AutoQ 2.0, preconditions, postconditions, and invariants are represented as sets of quantum states, encoded using the LSTA model. Therefore, it is important for users to understand how to encode a set of quantum states with an LSTA. Below, we provide two examples to give a basic understanding of the process. In the first example, we show how to encode the postcondition of Algorithm 1, $\{a_0 |10\rangle + a_1 |11\rangle\,,\ -a_1 |10\rangle - a_0 |11\rangle\}$.

The corresponding LSTA is shown in Fig. 3. The LSTA constructs trees that depict quantum states beginning from the initial state p at the root. It continues to build the tree by choosing transitions to explore new child states at each step, and this process continues until it reaches the leaves. For instance, the tree in Fig. 2(c) can be generated by first picking the transition $p \xrightarrow{\{1\}} x_1(q_+, q_0)$, then the two transitions $q_+ \xrightarrow{\{1\}} x_2(r_1, r_2)$ and $q_0 \xrightarrow{\{1\}} x_2(r_0, r_0)$, and ending with the three leaf transitions $r_1 \xrightarrow{\{1\}} a_0/\sqrt{2}$, $r_2 \xrightarrow{\{1\}} a_1/\sqrt{2}$, and $r_0 \xrightarrow{\{1\}} 0$. Similar to the conventional tree automata (TAs) model, LSTAs utilize disjunctive branches to represent various states that share a common structure. In Fig. 3, the state p has two possible outgoing transitions. If one picks the other transition $p \xrightarrow{\{2\}} x_1(q_0, q_-)$ at the beginning, we can generate the tree shown in Fig. 2(d).

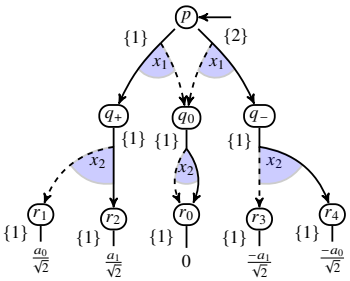


Fig. 3. The LSTA recognizing the postcondition of Algorithm 1

The previous example does not fully demonstrate why incorporating a set of choices (the numbers in the curly brackets) into the design of LSTAs is beneficial. Let us consider another well-known example: the set of Bell states $\{|00\rangle \pm |11\rangle\,, |01\rangle \pm |10\rangle\}$, generated by the LSTA in Fig. 4(a). Without the restriction that all transitions at the same level

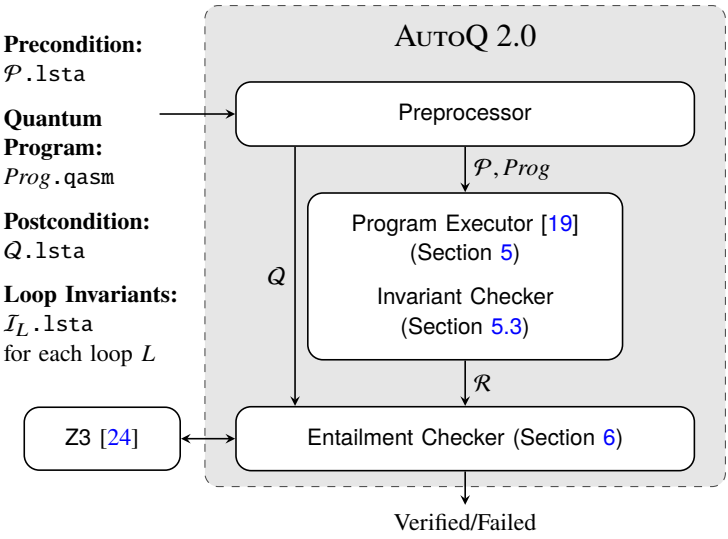


Fig. 5. The architecture of AutoQ 2.0

must share a common choice, this LSTA would generate eight different trees (since q_+, $q_\pm$, and $r_\pm$ each have two outgoing transitions), which correspond to the quantum states $\{|00\rangle \pm |11\rangle, |01\rangle \pm |10\rangle, |00\rangle \pm |10\rangle, |01\rangle \pm |11\rangle\}$. However, only four of these trees correspond to the Bell states, meaning the others are unintended. The LSTA uses the labeled choices to rule out the unintended trees. More specifically, at level 2, the two transitions labeled $\{1\}$ can be used simultaneously, as they share the common choice 1. Similarly, the two transitions labeled $\{2\}$ can be used together due to their common choice 2. In contrast, a transition labeled $\{1\}$ cannot be used alongside one labeled 2, as their choice sets are disjoint. At level 3, the transitions from $r_\pm$ can be used freely with those from r_+ and r_0, since $\{1\}$ (and $\{2\}$) $\cap \{1, 2\} \neq \emptyset$. There are two valid combinations of transitions at levels 2 and 3, and this LSTA generates exactly the four Bell states using the nine transitions shown in the figure. The corresponding `.lsta` file, which illustrates the input format for AutoQ 2.0, is shown in Fig. 4(b). In `.lsta` files, transitions are labeled with pairs $[a, b]$, where a indicates the symbol x_a and b is the set of choices.

4 System Architecture

We present the architecture of AutoQ 2.0 in Fig. 5. The tool is written in C++ and uses the SMT solver Z3 [24] for satisfiability and entailment checking of constraints. We allow the use of any theory that is supported by Z3. In our experiments, we used `NIRA` (non-linear integer and real arithmetic). While this logic is generally undecidable, Z3 always quickly solved the formulae we presented to it in our experiments.

Similar to verifiers for classical programs, in order to use AutoQ 2.0, the user needs to provide the following: (i) a *quantum program* in the OpenQASM 3.0 format, (ii) *pre- and post-conditions* in the `.lsta` format along with SMT formulae in the `.smt` format

Algorithm 3: Algorithm for measurement

Input: LSTA $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{R}, \varphi \rangle$, target qubit x_i, measurement outcome b
Output: The LSTA $M_i^{=b}(\mathcal{A})$

$Q' := \{q' \mid q \in Q\}$;
$\Delta' := \{q' \xrightarrow{f|C} (q'_1, q'_2) \mid q \xrightarrow{f|C} (q_1, q_2) \in \Delta\} \cup \{q' \xrightarrow{0|C} () \mid q \xrightarrow{f|C} () \in \Delta\}$;
$\Delta^{\neq x_i} := \{q \xrightarrow{f|C} (q_0, q_1) \mid q \xrightarrow{f|C} (q_0, q_1) \in \Delta \wedge f \neq x_i\} \cup \{q \xrightarrow{f|C} () \mid q \xrightarrow{f|C} () \in \Delta\}$;
if $b = 0$ **then** $\Delta^{=x_i} := \{q \xrightarrow{x_i|C} (q_0, q'_1) \mid q \xrightarrow{x_i|C} (q_0, q_1) \in \Delta\}$;
    else $\Delta^{=x_i} := \{q \xrightarrow{x_i|C} (q'_0, q_1) \mid q \xrightarrow{x_i|C} (q_0, q_1) \in \Delta\}$;
return $M_i^{=b}(\mathcal{A}) = \langle Q \cup Q', \Sigma, \Delta^{=x_i} \cup \Delta^{\neq x_i} \cup \Delta', \mathcal{R}, \varphi \rangle$;

to constrain the terms, and (iii) *invariant* for each loop in the `.lsta` format together with an SMT formula. The specification (pre- and post-conditions) and the invariant (for each loop) can be written as an LSTA (an `.lsta` file). Once these files are provided, AutoQ 2.0 will process them and report either "Verified" or "Failed".

Compared to AutoQ 1.0, there are several major changes in AutoQ 2.0. Firstly, it features a new input interface to facilitate the use of quantum programs (instead of only circuits) and uses LSTA as the back-end model (instead of standard tree automata). Additionally, Program Executor now supports measurement and branch statements. Another significant addition is the new Invariant Checker component, which handles loop invariants. The Invariant Checker also uses the Entailment Checker to verify the *inductiveness* of the invariant, which we do not explicitly show in the figure.

5 Handling Branch, Measurement, and Loop

We will start by presenting the syntax of quantum programs that AutoQ 2.0 can handle and then informally describe their semantics. We use a flavor of quantum programs that is similar to the one in [48], which is captured by the following grammar:

$$P \;::=\; U \;\mid\; P; P \;\mid\; \textbf{while}\ (M_i = b)\ \textbf{do}\ \{P\} \;\mid\; \textbf{if}\ (M_i = b)\ \textbf{then}\ \{P\}\ \textbf{else}\ \{P\}$$

where P is a quantum program, U is a quantum gate annotated with its control and target qubits (e.g., CX_1^2), $b \in \{0, 1\}$, and M_i is the measured value of the i-th qubit. AutoQ 2.0 supports standard non-parameterized quantum gates that allow (approximate) universal computation [12,3], including Clifford gates (H, S, and CX), T, Z, $SWAP$, Toffoli, etc.

The execution of quantum gate U updates a quantum state (tree) in the standard way [20]. The language allows sequential composition ($P; P$) of gate operations, branches (**if** ... **else** ...), and loops (**while** ...). When using **if** and **while** statements, the condition $M_i = b$ (denoting that the value obtained from measuring x_i was b) is used to determine in which path to continue.

5.1 Handling Measurement

The key part of handling branch statements in AutoQ 2.0 is how measurement changes the quantum states and how we should update the LSTA capturing the set of reachable

states. As mentioned in Section 3, if the measured value of x_i is 1, then we should update the probability of the 0-subtrees below all x_i nodes to 0. Examples can be found in Fig. 2(b) (before measuring x_1) and Figs. 2(c) and 2(d) (after measuring x_1 as 0 and 1 respectively). An important design choice was that we do not normalize the probability amplitudes and simply just make all leaves of one of the subtrees zero in this step, leaving the task of *matching non-normalized states* to the entailment test (cf. Section 6).

In some cases, the measurement can generate an LSTA whose language contains a tree where all leaves are 0. This can happen, e.g., when we compute the tree representing the quantum state obtained from the state in Fig. 2(a) by measuring $x_1 = 0$. We do not consider such a tree to represent a quantum state. To handle such cases, our entailment test $\mathcal{R} \models^{uts} \mathcal{Q}$ (formally defined in Section 6) adds a 0-*labeled tree* to the language of $\mathcal{Q}$ before the test. We use $M_i^{=b}(\mathcal{A})$ to represent the LSTA obtained from $\mathcal{A}$ after measuring x_i for the outcome b. The procedure for computing $M_i^{=b}(\mathcal{A})$ is given in Algorithm 3.

The goal of the algorithm is to update all leaf values of $\bar{b}$-subtrees under x_i to 0, where $\bar{b} = 1 - b$. Lines 1 and 2 of Algorithm 3 create a primed copy of the input LSTA and update all leaf values to 0 (Line 2). Lines 3 to 5 construct the new transition system: only transitions labeled with x_i are modified (Lines 4 and 5), while others remain unchanged (Line 3). The key steps are in Lines 4 and 5, which control the subtrees of the measured qubit. When $b = 0$ (Line 4), all leaves of the 1-subtree are modified to 0, and thus, we update q_1 in the original transition to q_1' (symmetrically for $b = 1$ in Line 5).

5.2 Handling Branch Statements

Given an LSTA $\mathcal{A}$ that recognizes a set of quantum states, we can precisely compute the set of states that result from executing a branch statement **if** $(M_i = b)$ **then** $\{P_1\}$ **else** $\{P_0\}$ as follows (assuming that P_0 and P_1 do not involve loops): (i) Create two LSTAs $M_i^{=1}(\mathcal{A})$ and $M_i^{=0}(\mathcal{A})$. (ii) Compute the result after executing P_b from $M_i^{=b}(\mathcal{A})$ for $b \in \{0, 1\}$, following the gates' semantics and recursively trigger the procedure for branches. We use $\mathcal{A}^0$ and $\mathcal{A}^1$ to denote the LSTAs after executing P_0 and P_1, respectively. (iii) Construct an LSTA recognizing the union of $\mathcal{A}^0$ and $\mathcal{A}^1$ and return it as the final result of this procedure. In principle, our approach can handle nested control flow. We are, however, not aware of any real-world quantum program that uses a nested control structure, and, therefore, for simplicity, AutoQ 2.0 now only supports programs with non-nested control flow.

5.3 Handling Loop Statements

If we come across a loop statement **while** $(M_i = b)$ **do** $\{B\}$ with B being the loop body, we require the user to provide a loop invariant in the form of an LSTA. We refer to the invariant as $\mathcal{I}$; it needs to satisfy the following properties: (i) It contains all reachable states, captured by an LSTA $\mathcal{R}$, before entering the loop. That is, $\mathcal{R} \models^{uts} \mathcal{I}$. (ii) It is *inductive*, i.e., $B(M_i^{=b}(\mathcal{I})) \models^{uts} \mathcal{I}'$, where $B(\mathcal{A})$ denotes an LSTA recognizing the set of quantum states after executing B from the quantum states in $\mathcal{A}$ and $\mathcal{I}'$ is an LSTA obtained from $\mathcal{I}$ whose variables and constraints are updated to a primed version. The inductiveness guarantees that if we take any state accepted by $\mathcal{I}$ and perform B on the

state, the result will also be accepted by $\mathcal{I}$. Together with the condition that $\mathcal{I}$ covers all reachable states before entering the loop, $\mathcal{I}$ *over-approximates* all reachable states at the loop entrance. We can, therefore, use $M_i^{\neq b}(\mathcal{I})$ to over-approximate all reachable states at the loop exit.

6 Testing Entailment up to Scaling

In our approach, we use a special entailment test at some points, which we call *entailment up to scaling*, denoted as $\mathcal{A} \models^{uts} \mathcal{B}$. The reason for a special entailment relation is that—as mentioned before—at measurements, we do not perform normalization. Intuitively, given two LSTAs $\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma, \Delta_{\mathcal{A}}, \mathcal{R}_{\mathcal{A}}, \varphi_{\mathcal{A}} \rangle$ and $\mathcal{B} = \langle Q_{\mathcal{B}}, \Sigma, \Delta_{\mathcal{B}}, \mathcal{R}_{\mathcal{B}}, \varphi_{\mathcal{B}} \rangle$, the relation $\mathcal{A} \models^{uts} \mathcal{B}$ holds if and only if for every tree $T_{\mathcal{A}}$ in the language of $\mathcal{A}$ and assignment to the variables occurring in $T_{\mathcal{A}}$, we can find a linearly scaled copy of $T_{\mathcal{A}}$ in the semantics of $\mathcal{B}$ (such that the values of variables occurring in both $T_{\mathcal{A}}$ and $T_{\mathcal{B}}$ match). Formally,

$$\begin{aligned} \mathcal{A} \models^{uts} \mathcal{B} \iff\ & (\forall T_{\mathcal{A}} \in \mathcal{L}(\mathcal{A}))(\forall \sigma_{\mathcal{A}} \colon vars(T_{\mathcal{A}}) \to \mathbb{C}) \colon \\ & (\exists T_{\mathcal{B}} \in \mathcal{L}(\mathcal{B}))(\exists \sigma_{\mathcal{B}} \colon (vars(T_{\mathcal{B}}) \setminus vars(T_{\mathcal{A}})) \to \mathbb{C}) \colon \\ & (\exists r \in \mathbb{R} \setminus \{0\}) \colon T_{\mathcal{A}}[\sigma_{\mathcal{A}}] = r \cdot T_{\mathcal{B}}[\sigma_{\mathcal{A}} \cup \sigma_{\mathcal{B}}], \end{aligned}$$

where $r \cdot T$ denotes the tree with the same structure as T with all numbers in leaves multiplied by r and $vars(\gamma)$ for any mathematical object γ (a term, a tree with terms in leaves, a set of terms, etc.) denotes the set of free variables occurring in the object. The $\models^{uts}$ relation is central to our approach.

Enumerating all trees of $\mathcal{A}$ and looking for their scaled copies in $\mathcal{B}$ would be too inefficient and even impossible in the case of LSTAs recognizing infinitely many finite trees (such as those modelling invariants of parameterized quantum programs [1]). Therefore, we modified the algorithm for LSTA language inclusion test presented in [1]. We note that language inclusion testing for LSTAs is more involved than for standard TAs (cf. [11,2,35,38]). In the modification, we allow to relate the leaf values with a linear factor for scaling (in contrast to only by identity as done in the original inclusion testing algorithm), so that it tests the entailment $\mathcal{A} \models^{uts} \mathcal{B}$.

The algorithm makes use of the following essential property of trees generated by an LSTA $\mathcal{A}$: if two nodes at the same level of a tree T are labelled by the same state in an accepting run of $\mathcal{A}$ on T, then the subtrees rooted in these nodes are identical (this follows from the semantics of LSTAs and the restriction on transitions, cf. Section 2.2).

Intuitively, the algorithm works as follows. It starts in the root states of $\mathcal{A}$ and $\mathcal{B}$ and performs a downward traversal through the LSTAs, level by level, remembering, at each level, how states from $\mathcal{A}$ can map to the states in $\mathcal{B}$. Moreover, the algorithm also remembers how the terms in the leaves of the tree from $\mathcal{A}$ map to the terms in the leaves of the tree from $\mathcal{B}$. The downward successors of each level are computed from transitions leaving states at the level that need to be synchronized on their choice. The algorithm explores the space of all of the reachable mappings until it reaches a point such that the tree from $\mathcal{A}$ has all branches terminated. At this moment, we check whether the terms from the leaf transitions of $\mathcal{A}$ can be mapped (up to scaling) to the corresponding terms from the leaf transitions in $\mathcal{B}$. If not, conclude that the entailment does not hold.

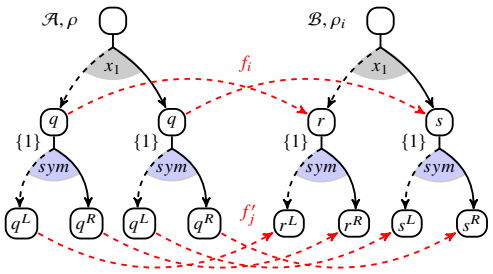


Fig. 6. Computing f'_j from f_i

Formally, the algorithm performs a search in the directed graph (V, E) (constructed on the fly), where each vertex $v \in V$ is of the form $v = (D, \{(f_1, g_1), \ldots, (f_m, g_m)\})$ for some m where $D \subseteq Q_{\mathcal{A}}$ is called the *domain*, each $f_i \colon D \to 2^{Q_{\mathcal{B}}}$ is a map that assigns every state of $\mathcal{A}$ from the domain D a set of states of $\mathcal{B}$, and each $g_i \colon \mathsf{term}(\mathbb{C}, \mathbb{X}) \to 2^{\mathsf{term}(\mathbb{C},\mathbb{X})}$ is a (partial) mapping from terms to sets of terms. Intuitively, D represents the set of states of $\mathcal{A}$ at one level of $\mathcal{A}$'s run ρ, and every f_i represents the same level of some possible run ρ_i of $\mathcal{B}$ on the same tree and the way it can match the run ρ of $\mathcal{A}$. For instance, in Section 6, the state q of $\mathcal{A}$ corresponds to the states r, s of $\mathcal{B}$ because they are used in the same tree level and the same tree nodes, so we have $f_i(q) = \{r, s\}$. Due to the property that all occurrences of a state at the same level in a run generate the same subtree mentioned above, we only need to maintain encountered states and their alignment with each another. The term mappings g_i are used to remember how the terms from the leaves of $\mathcal{A}$ are mapped to terms in the leaves of $\mathcal{B}$. For instance, if some term t from a leaf of $\mathcal{A}$ is mapped by two different terms t_1 and t_2 of $\mathcal{B}$, we will later need to check whether there is a scaling factor r such that $t = r \cdot t_1$ and, at the same time, $t = r \cdot t_2$ (and the global constraints of $\mathcal{A}$ and $\mathcal{B}$ are satisfied). We give a general algorithm here; for LSTAs that accept only perfect trees (as is the case for the ones encoding quantum states), all branches of the accepted trees terminate at the same time so there is no need to remember the term mappings g across different levels and the scaling compatibility could be checked only locally.

The graph search begins from the *source* vertices, one for each state $q \in \mathcal{R}_{\mathcal{A}}$, of the form $(\{q\}, \{(\{q \mapsto \{r_1\}\}, \emptyset), \ldots, (\{q \mapsto \{r_k\}\}, \emptyset)\})$, where $\{r_1, \ldots, r_k\} = \mathcal{R}_{\mathcal{B}}$, corresponding to the root states of both $\mathcal{A}$ and $\mathcal{B}$ (the $\emptyset$'s denote empty term mappings). If the search finds a *terminal* vertex $(\emptyset, F)$, where $(\emptyset, g) \notin F$, meaning that an accepting run of $\mathcal{A}$ has been found, but there is no corresponding matching run of $\mathcal{B}$ (for any g), we can conclude that the entailment test failed (it represents the case when $\mathcal{A}$ finished reading all branches of the tree but $\mathcal{B}$ did not). On the other hand, if there is $(\emptyset, g) \in F$, we still need to check that the set of terms g is compatible. The graph's edges represent generating the next level of runs for both $\mathcal{A}$ and $\mathcal{B}$ and how the respective states align with each other. The specific construction of the edges from a vertex $v = (D, \{(f_1, g_1), \ldots, (f_m, g_m)\})$, where $D \neq \emptyset$, to each lower-level vertex $v' = (D', \{(f'_1, g'_1), \ldots, (f'_n, g'n)\})$ follows.

First, we compute possible successors of the D-component of v. To do this, we need to explore all *feasible* sets $\Gamma_{\mathcal{A}}$ of transitions from D in $\mathcal{A}$. More concretely, in each set $\Gamma_{\mathcal{A}}$, we select exactly one downward transition $\delta_{q_{\mathcal{A}}}$ originating from each $q_{\mathcal{A}} \in D$, such that all transitions in $\Gamma_{\mathcal{A}}$ share a common choice (as required by the definition of an accepting run (cf. Section 2.2). Formally, given $D = \{q_1, \ldots, q_k\}$, we consider all sets of transitions $\Gamma_{\mathcal{A}} = \{\delta_1, \ldots, \delta_k\}$ such that the following formula holds:

$$(\forall 1 \leq i \leq k \colon \delta_i \in \Delta \wedge \mathsf{top}(\delta_i) = q_i) \quad \wedge \quad \bigcap \{\mathsf{ch}(\delta_i) \mid 1 \leq i \leq k\} \neq \emptyset. \qquad (1)$$

Algorithm 4: FindAllMappings($\Gamma_{\mathcal{A}}, f$)

Input: Set of transitions $\Gamma_{\mathcal{A}} \subseteq \Delta_{\mathcal{A}}$, map $f\colon \{\mathtt{top}(\delta) \mid \delta \in \Gamma_{\mathcal{A}}\} \to 2^{Q_{\mathcal{B}}}$
Output: The set of pairs (f', g') of mappings compatible with $\Gamma_{\mathcal{A}}$ obtained from f

1 $Q \leftarrow \bigcup \mathrm{img}(f)$; $F' \leftarrow \emptyset$;
2 **foreach** $\Gamma_{\mathcal{B}} \in \mathtt{FTrans}(\mathcal{B}, Q)$ **do**
3 $f' \leftarrow \emptyset$; $g' \leftarrow \emptyset$; *failed* $\leftarrow$ **false**;
4 **foreach** $\delta_{\mathcal{A}} \in \Gamma_{\mathcal{A}}$ and $q \in f(\mathtt{top}(\delta_{\mathcal{A}}))$ **do**
5 $\{\delta_{\mathcal{B}}\} \leftarrow \{\delta_{\mathcal{B}} \in \Gamma_{\mathcal{B}} \mid q = \mathtt{top}(\delta_{\mathcal{B}})\}$;
6 **if** $a = \mathtt{sym}(\delta_{\mathcal{A}})$ and $b = \mathtt{sym}(\delta_{\mathcal{B}})$ are both leaf symbols **then** $g'(a)$.insert(b) ;
7 **else if** $\mathtt{sym}(\delta_{\mathcal{A}}) = \mathtt{sym}(\delta_{\mathcal{B}})$ is an internal symbol **then**
8 $f'(\mathtt{left}(\delta_{\mathcal{A}}))$.insert($\mathtt{left}(\delta_{\mathcal{B}})$); $f'(\mathtt{right}(\delta_{\mathcal{A}}))$.insert($\mathtt{right}(\delta_{\mathcal{B}})$);
9 **else** *failed* $\leftarrow$ **true**; **break**;
10 **if** $\neg$*failed* **then** F'.insert((f', g')) ;
11 **return** F';

We will denote the set of all such $\Gamma_{\mathcal{A}}$'s from a set of states D as $\mathtt{FTrans}(\mathcal{A}, D)$.

Next, we will show how to construct the set of all feasible pairs of mappings $\{(f'_1, g'_1), \ldots (f'_n, g'_n)\}$ for some $\Gamma_{\mathcal{A}}$ and a pair of upper level mappings (f, g). Each pair of mappings (f'_j, g'_j) at the lower level is derived from some pair of mappings (f_i, g_i) at the upper level and a set of downward transitions $\Gamma_{\mathcal{B}}$ from $\mathcal{B}$. The construction process is described in Algorithm 4, where we use (f, g) to denote an upper level state and term mapping respectively and (f', g') to denote their lower level counterparts.

The basic idea of the algorithm is simple: (i) we use the upper level mapping f and transitions $\Gamma_{\mathcal{A}}$ to compute the set of top states Q (Line 1), (ii) then, we find all feasible transitions $\Gamma_{\mathcal{B}}$ from Q (Line 2), and, finally, (iii) for each pair $(\Gamma_{\mathcal{A}}, \Gamma_{\mathcal{B}})$, we construct one pair of lower level state and term mappings (f', g') (Lines 4–9). Specifically, in step (iii), it must hold that for each transition $\delta_{\mathcal{A}} \in \Gamma_{\mathcal{A}}$, every state $q \in f(\mathtt{top}(\delta_{\mathcal{A}}))$ needs to be able to match $\delta_{\mathcal{A}}$ by a transition from $\Gamma_{\mathcal{B}}$. To check this, for every such q we select from $\Gamma_{\mathcal{B}}$ the transition $\delta_{\mathcal{B}}$, which is the transition of $\Gamma_{\mathcal{B}}$ with q as its top (it follows from FTrans that there is exactly one). There are three possible cases:

- *Both $\delta_{\mathcal{A}}$ and $\delta_{\mathcal{B}}$ are leaf transitions (Line 6):* we remember in g' that the symbol of $\delta_{\mathcal{B}}$ need to be able to match the symbol of $\delta_{\mathcal{A}}$, which will be checked later for all such matchings together.
- *The symbols of both $\delta_{\mathcal{A}}$ and $\delta_{\mathcal{B}}$ are internal (Line 7):* in this case, we add new entries to the lower level mapping f'.
- *One transition is internal while the other is leaf (Line 9):* the pair $(\Gamma_{\mathcal{A}}, \Gamma_{\mathcal{B}})$ cannot form a feasible lower level mapping.

Finally, the main entailment testing routine is summarized in Algorithm 5. Line 2 creates the set of source vertices of the explored graph. Lines 3–4 pick a vertex (D, F) that has not been processed yet. Lines 6–10 check whether (D, F) is a terminal vertex and conclude that the entailment test fails when such a vertex is reached. This check consists

Algorithm 5: Checking if $\mathcal{A} \models^{uts} \mathcal{B}$

Input: LSTAs $\mathcal{A} = \langle Q_{\mathcal{A}}, \mathbb{N} \cup \mathsf{term}(\mathbb{C}, \mathbb{X}), \Delta_{\mathcal{A}}, \mathcal{R}_{\mathcal{A}}, \varphi_{\mathcal{A}} \rangle$,
$\mathcal{B} = \langle Q_{\mathcal{B}}, \mathbb{N} \cup \mathsf{term}(\mathbb{C}, \mathbb{X}), \Delta_{\mathcal{B}}, \mathcal{R}_{\mathcal{B}} = \{r_1, \ldots, r_k\}, \varphi_{\mathcal{B}} \rangle$.
Output: **true** if $\mathcal{A} \models^{uts} \mathcal{B}$, **false** otherwise

1 $processed \leftarrow \emptyset$;
2 $workset \leftarrow \{(\{q\},\ \{(\{q \mapsto \{r_1\}\}, \emptyset), \ldots, (\{q \mapsto \{r_k\}\}, \emptyset)\}) \mid q \in \mathcal{R}_{\mathcal{A}}\}$;
3 **while** $\exists (D, F) \in workset$ **do**
4 $\quad workset$.remove$((D, F))$;
5 $\quad processed$.insert$((D, F))$;
6 $\quad$ **if** $D = \emptyset$ **then**
7 $\quad\quad \mathbb{Y} \leftarrow \{vars(u_1) \mid (u_1 \mapsto U_2) \in g \wedge (\emptyset, g) \in F\} \cup vars(\varphi_{\mathcal{A}})$;
8 $\quad\quad \mathbb{Z} \leftarrow \{vars(U_2) \mid (u_1 \mapsto U_2) \in g \wedge (\emptyset, g) \in F\} \cup vars(\varphi_{\mathcal{B}}) \setminus \mathbb{Y}$;
9 $\quad\quad$ **if** $\neg\forall \mathbb{Y} \colon \varphi_{\mathcal{A}} \implies \exists \mathbb{Z} \colon \varphi_{\mathcal{B}} \wedge \bigvee_{(\emptyset,g) \in F} \exists r \in \mathbb{R} \setminus \{0\} \colon \bigwedge_{\substack{(u_1 \mapsto U_2) \in g \\ u_2 \in U_2}} u_1 = r \cdot u_2$ **then**
10 $\quad\quad\quad$ **return false**;
11 $\quad$ **foreach** $\Gamma_{\mathcal{A}} \in \mathtt{FTrans}(\mathcal{A}, D)$ **do**
12 $\quad\quad D' \leftarrow \{q \in \mathtt{bot}(\delta) \mid \delta \in \Gamma_{\mathcal{A}}\}$; $F' \leftarrow \emptyset$;
13 $\quad\quad$ **foreach** $(f, g) \in F$ **do**
14 $\quad\quad\quad U \leftarrow \mathtt{FindAllMappings}(\Gamma_{\mathcal{A}}, f)$;
15 $\quad\quad\quad F' \leftarrow F' \cup \{(f', g \uplus g') \mid (f', g') \in U\}$;
16 $\quad\quad$ **if** $(D', F') \notin processed \cup workset$ **then** $workset$.insert$((D', F'))$;
17 **return true**;

of looking at all pairs $(\emptyset, g)$ in F and checking whether there is a way how the $\mathcal{B}$-terms from the g's can together cover (modulo a scaling factor) the behaviour of the $\mathcal{A}$-terms. Lines 11–16 are the edge construction procedure. Lines 11–12 enumerate all feasible $\Gamma_{\mathcal{A}}$ and use them to create the next vertex (D', F'). Specifically, D' are the bottom states from $\Gamma_{\mathcal{A}}$ (Line 12), and F' are the union of all feasible mappings of $\Gamma_{\mathcal{A}}$ (Lines 13–16). The successor term mapping is computed from the upper-level one and from the one returned by `FindAllMappings` by, for each term of $\mathcal{A}$, merging corresponding sets of terms of mappings: $g \uplus g' = \{x \mapsto Y \mid x \in \mathrm{dom}(g \cup g'), Y = g(x) \cup g'(x)\}$ (Line 15).

A crucial part of the algorithm is the term mapping g check between the terms from $\mathcal{A}$ and the terms from $\mathcal{B}$ (Lines 7–10). Here, we want to check that for every possible value that we can obtain from a term t on the left-hand side of the entailment (satisfying $\mathcal{A}$'s global constraint $\varphi_{\mathcal{A}}$) and every term t_i from $g(t) = \{t_1, \ldots, t_k\}$, there is a way to obtain a value (satisfying $\mathcal{B}$'s global constraint $\varphi_{\mathcal{B}}$) such that for all these values, there is a common scaling factor r to make them equal to the value from t. We emphasize the way how we need to deal with the quantified variables. Variables $\mathbb{Y}$ occurring on the left-hand side (and possibly also on the right-hand side, since we may need to synchronize the values) are quantified universally, while variable $\mathbb{Z}$ occuring only on the right-hand side of the entailment are quantified existentially. This scaling check requires invoking an SMT solver with a formula in the `NIRA` (non-linear integer and real arithmetic) logic. In the special case where all leaf symbols are constants, r is the only variable, and global constraints are **true**, the problem reduces to a simple `QF_LRA` (quantifier-free linear real arithmetic) formula.

Theorem 1 (Soundness). *When Algorithm 5 terminates, it returns true iff* $\mathcal{A} \models^{uts} \mathcal{B}$.

Theorem 2 (Termination). *When the terms in leaf symbols and the global constraints of* $\mathcal{A}$ *and* $\mathcal{B}$ *use a decidable theory, the algorithm always terminates.*

Proof. Since the number of states and terms occurring in $\mathcal{A}$ and $\mathcal{B}$ is finite, the constructed graph is also finite. Further, since the underlying theory for the terms and global constraints is assumed to be decidable, the check at Line 9 always terminates. □

7 Experimental Results

We demonstrate the use of AutoQ 2.0 [21] on two real-world use cases consisting of quantum programs with loops that were proposed in [41,6]. We ran all experiments on a server running Ubuntu 22.04.3 LTS with an AMD EPYC 7742 64-core processor (1.5 GHz), 1,152 GiB of RAM, and a 1 TB SSD.

7.1 The Weakly Measured Version of Grover's Algorithm

Grover's algorithm [31], introduced in 1996, is a quantum algorithm that performs an unstructured search. Given an *oracle function* (which can say whether a particular binary assignment is a solution), Grover's algorithm can efficiently find a solution (with high probability). The algorithm requires approximately $O(\sqrt{N/k})$ evaluations of the oracle function, where N is the size of the function's domain (usually 2^n for n qubits), and k is the number of solutions. The number of solutions is, however, not always known, making it difficult to determine the algorithm's parameters (the algorithm is sensitive to the number of evaluations; in particular, doing more evaluations may make the probability of finding the solution smaller). To address this issue, a variation of Grover's search, called the *weakly measured* version (cf. Algorithm 6), was recently proposed [6]. The weakly measured version eliminates the need for knowing the number of solutions, making the algorithm more applicable.

Algorithm 6: A Weakly Measured Version of Grover's algorithm (solution $s = 0^n$)

```
1 Pre: {1 |0^{n+2}⟩ + 0 |∗⟩};
2 H_3; H_4; ...; H_{n+2};
3 O_{2,...,(n+2)}; CK_1^2; O_{2,...,(n+2)};
4 Inv: {v_soll |000^n⟩ + v_k |000^{n-1}1⟩ + ··· +
5        v_k |001^n⟩ + v_sol2 |100^n⟩ + 0 |∗⟩};
6 while M_1 = 0 do
7 |   {G_{2,...,(n+2)}; O_{2,...,(n+2)}; CK_1^2; O_{2,...,(n+2)}};
8 Post: {1 |10s⟩ + 0 |∗⟩};
```

To explain the algorithm, we first introduce some of its key components. The algorithm works over qubits $x_1, \ldots, x_{n+2}$. Line 2 first applies multiple *Hadamard* gates in parallel to obtain the superposition on all qubits other than x_1 and x_2 (which are two ancillas), obtaining the state in Fig. 7(a). The *oracle* circuit, denoted as $O_{2,\ldots,(n+2)}$, works from qubits x_2 to x_{n+2}, where x_2 is the ancilla qubit and x_3 to x_{n+2} are the working qubits. As shown from Figs. 7(a) and 7(b) (and also from Figs. 7(c) and 7(d)), the effect of the oracle circuit is to flip the ancilla qubit of the computational bases corresponding to the solutions. That is, it swaps the amplitude values of $|0s\rangle$ and $|1s\rangle$, for all solutions s. The oracle circuit can be constructed using gates supported in AutoQ 2.0.

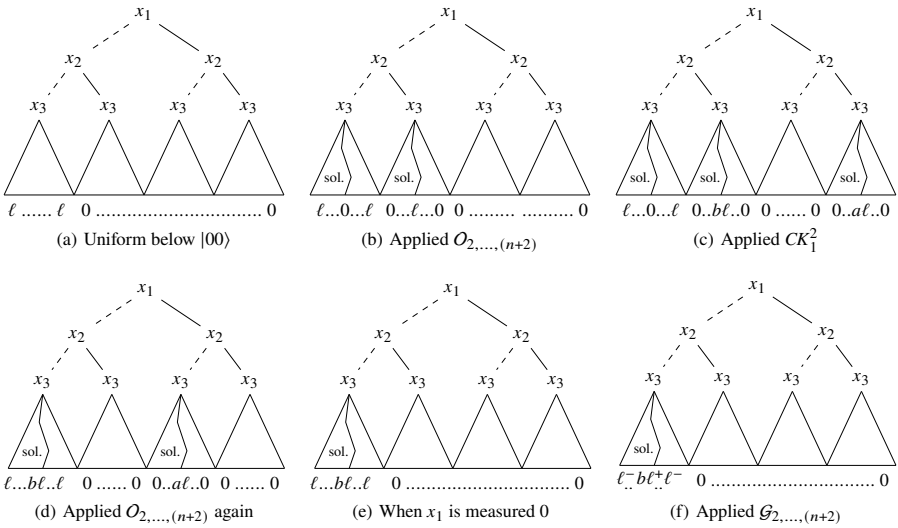


Fig. 7. Intermediate states of Algorithm 6

The controlled rotation gate CK_j^i is a special gate supported in AutoQ 2.0. In this algorithm, the gate always applies to a target qubit whose value is $|0\rangle$ when the controlled qubit is $|1\rangle$, and it updates the target qubit to $a\,|1\rangle + b\,|0\rangle$ with $a^2 + b^2 = 1$, for some very small b. In AutoQ 2.0, we use $a = \frac{21}{221}$ and $b = \frac{220}{221}$. We demonstrate the behavior of CK_1^2 from Figs. 7(b) and 7(c), where a small portion ($\frac{21^2}{221^2} \approx 1\,\%$) of the probability under the branch $|01\rangle$ is moved to the branch $|11\rangle$, as shown in Fig. 7(c). After applying $O_{2,\ldots,(n+2)}$ again, we obtain the state in Fig. 7(d) (this state is captured by the loop invariant). Here, we can already measure the qubit x_1 and if the result is 1, this collapses the probability of the left sub-tree of x_1 in Fig. 7(d) to 0, so the only non-zero probability basis is the solution $|10s\rangle$.

Otherwise (the result of measuring x_1 was 0), we enter the loop, which contains the *Grover iteration* circuit, denoted as $\mathcal{G}_{2,\ldots,(n+2)}$, which also uses $O_{2,\ldots,(n+2)}$ as a component. The effect of $\mathcal{G}_{2,\ldots,(n+2)}$ is to increase the probabilities of basis states for the solutions and decrease others, as shown in Figs. 7(e) and 7(f). After $\mathcal{G}_{2,\ldots,(n+2)}$, we execute the same sequence *oracle-rotation-oracle* as above to obtain a state resembling Fig. 7(d). We keep repeating the above procedure until we measure $x_1 = 1$, in which case we terminate with a solution.

The results of the verification of weakly measured Grover's search are in the left-hand side of Table 1: AutoQ 2.0 was able to verify the program w.r.t. the specification even for larger numbers of qubits in reasonable time.

7.2 Unitaries as Repeat-Until-Success Circuits

Repeat-until-success programs are a general framework that was introduced to simplify quantum circuit decomposition (we introduced an example of generating the "$-X$" gate via the RUS framework in Section 3). RUS programs have been shown to be

Table 1. Results of verifying some real-world examples with AutoQ 2.0. The number x in WMGrover(x) indicates that the number of items to be searched is 2^x.

Weakly Measured Grover's Search [6]						*Repeat-Until-Success* [41]					
program	**qubits**	**gates**	**result**	**time**	**memory**	**program**	**qubits**	**gates**	**result**	**time**	**memory**
WMGrover (03)	7	50	OK	0.0s	42MB	$(2X+\sqrt{2}Y+Z)/\sqrt{7}$	2	29	OK	0.0s	7MB
WMGrover (10)	21	169	OK	0.2s	42MB	$(I+i\sqrt{2}X)/\sqrt{3}$	2	17	OK	0.0s	7MB
WMGrover (20)	41	339	OK	0.8s	42MB	$(I+2iZ)/\sqrt{5}$	2	27	OK	0.0s	6MB
WMGrover (30)	61	509	OK	2.3s	43MB	$(3I+2iZ)/\sqrt{13}$	2	43	OK	0.0s	7MB
WMGrover (40)	81	679	OK	5.4s	43MB	$(4I+iZ)/\sqrt{17}$	2	77	OK	0.0s	6MB
WMGrover (50)	101	849	OK	11s	44MB	$(5I+2iZ)/\sqrt{29}$	2	69	OK	0.0s	7MB

more efficient (in terms of circuit depth) than ancilla-free techniques when it comes to synthesizing single-qubit gates (cf. [41,10]). We present the results of verification of RUS programs for generating various non-standard gates in the right-hand side of Table 1. Note that AutoQ 2.0 can verify these programs instantaneously.

8 Conclusion and Future Work

We presented a major extension of AutoQ 1.0 [19] with an added support for control flow constructs and evaluated its feasibility on a family of programs for the weak-measurement-based version of Grover's algorithm and on implementations of a number of non-standard quantum gates using repeat-until-success circuits. In the future, we wish to extend the framework with automating invariant generation (e.g., using a modification of the symbolic-execution-based technique from [16]) and add support for dealing with more complex loops that give rise to mixed states.

Acknowledgements.

We thank the reviewers for their useful remarks that helped us improve the quality of the paper. This work was supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation project 25-18318S, the FIT BUT internal project FIT-S-23-8151, the National Science and Technology Council, Taiwan, projects under Grant no. NSTC 113-2222-E-027-007- and 112-2222-E-001-002-MY3, Foxconn research project 05T-1120327-1C-, Academia Sinica Grand Challenge Seeding Project, and Academia Sinica Principal Investigator Project no. AS-IV-114-M07-ASSA.

Data Availability Statement.

An environment with the tools and data used for the experimental evaluation in the current study is available at [18].

References

1. Abdulla, P.A., Chen, Y.G., Chen, Y.F., Holík, L., Lengál, O., Lin, J.A., Lo, F.Y., Tsai, W.L.: Verifying quantum circuits with level-synchronized tree automata. Proceedings of the ACM on Programming Languages **9**(POPL), 923–953 (2025)

2. Abdulla, P.A., Chen, Y., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. 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. 158–174. Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_14, https://doi.org/10.1007/978-3-642-12002-2_14
3. Aharonov, D.: A simple proof that Toffoli and Hadamard are quantum universal (2003). https://doi.org/10.48550/arxiv.quant-ph/0301040, https://arxiv.org/abs/quant-ph/0301040
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6, https://doi.org/10.1007/978-3-319-49812-6
5. Amy, M.: Towards large-scale functional verification of universal quantum circuits. In: Quantum Physics and Logic (2018)
6. Andrés-Martínez, P., Heunen, C.: Weakly measured while loops: peeking at quantum states. Quantum Science and Technology **7**(2), 025007 (feb 2022). https://doi.org/10.1088/2058-9565/ac47f1, https://dx.doi.org/10.1088/2058-9565/ac47f1
7. Anticoli, L., Piazza, C., Taglialegne, L., Zuliani, P.: Towards quantum programs verification: From Quipper circuits to QPMC. In: Devitt, S.J., Lanese, I. (eds.) Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9720, pp. 213–219. Springer (2016). https://doi.org/10.1007/978-3-319-40578-0_16, https://doi.org/10.1007/978-3-319-40578-0_16
8. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
9. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
10. Bocharov, A., Roetteler, M., Svore, K.M.: Efficient synthesis of universal repeat-until-success quantum circuits. Phys. Rev. Lett. **114**, 080502 (Feb 2015). https://doi.org/10.1103/PhysRevLett.114.080502, https://link.aps.org/doi/10.1103/PhysRevLett.114.080502
11. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Ibarra, O.H., Ravikumar, B. (eds.) Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5148, pp. 57–67. Springer (2008). https://doi.org/10.1007/978-3-540-70844-5_7, https://doi.org/10.1007/978-3-540-70844-5_7
12. Boykin, P.O., Mor, T., Pulver, M., Roychowdhury, V.P., Vatan, F.: A new universal and fault-tolerant quantum basis. Inf. Process. Lett. **75**(3), 101–107 (2000). https://doi.org/10.1016/S0020-0190(00)00084-3, https://doi.org/10.1016/S0020-0190(00)00084-3
13. Burgholzer, L., Wille, R.: Advanced equivalence checking for quantum circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **40**(9), 1810–1824 (2020)

14. Chareton, C., Bardin, S., Bobot, F., Perrelle, V., Valiron, B.: An automated deductive verification framework for circuit-building quantum programs. In: Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 30. pp. 148–177. Springer International Publishing (2021)
15. Chareton, C., Bardin, S., Bobot, F., Perrelle, V., Valiron, B.: An automated deductive verification framework for circuit-building quantum programs. In: Yoshida, N. (ed.) ESOP. Lecture Notes in Computer Science, vol. 12648, pp. 148–177. Springer International Publishing, Cham (March 2021)
16. Chen, T., Chen, Y., Jiang, J.R., Lengál, O., Jobranová, S.: Accelerating quantum circuit simulation with symbolic execution and loop summarization. In: Proc. of ICCAD'24. ACM (2024)
17. Chen, T., Jiang, J.R., Hsieh, M.: Partial equivalence checking of quantum circuits. In: 2022 IEEE International Conference on Quantum Computing and Engineering (QCE). pp. 594–604 (2022). https://doi.org/10.1109/QCE53715.2022.00082
18. Chen, Y.F., Chung, K.M., Hsieh, M.H., Huang, W.J., Lengal, O., Lin, J.A., Tsai, W.L.: AutoQ 2.0: An automata-based quantum program verifier (TACAS artifact evaluation record). https://zenodo.org/records/14114791 (2024)
19. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L.: AutoQ: An automata-based quantum circuit verifier. In: International Conference on Computer Aided Verification. pp. 139–153. Springer (2023)
20. Chen, Y., Chung, K., Lengál, O., Lin, J., Tsai, W., Yen, D.: An automata-based framework for verification and bug hunting in quantum circuits. In: 44th ACM SIGPLAN Conference on Programming Language Design and Implementation—PLDI'23. ACM (2023)
21. Chen, Y.F., Lengál, O., Tsai, W.L.: AutoQ 2.0: An automata-based C++ tool for quantum program verification (Nov 2024), https://github.com/alan23273850/AutoQ/tree/tacas25
22. Coecke, B., Duncan, R.: Interacting quantum observables: categorical algebra and diagrammatics. New Journal of Physics **13**(4), 043016 (apr 2011). https://doi.org/10.1088/1367-2630/13/4/043016
23. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_2, https://doi.org/10.1007/978-3-642-03359-9_2
24. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14. pp. 337–340. Springer (2008)
25. D'Hondt, E., Panangaden, P.: Quantum weakest preconditions. Mathematical Structures in Computer Science **16**(3), 429–451 (2006)
26. Fang, W., Ying, M., Wu, X.: Differentiable quantum programming with unbounded loops. ACM Trans. Softw. Eng. Methodol. **33**(1) (nov 2023). https://doi.org/10.1145/3617178, https://doi.org/10.1145/3617178
27. Feng, Y., Ying, M.: Quantum Hoare logic with classical variables. ACM Transactions on Quantum Computing **2**(4), 1–43 (2021)

28. Feng, Y., Yu, N., Ying, M.: Model checking quantum Markov chains. J. Comput. Syst. Sci. **79**(7), 1181–1198 (2013). https://doi.org/10.1016/j.jcss.2013.04.002, https://doi.org/10.1016/j.jcss.2013.04.002
29. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22. pp. 125–128. Springer (2013)
30. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science. pp. 19–32. Proceedings of Symposia in Applied Mathematics, AMS (1967), https://mathscinet.ams.org/mathscinet/article?mr=235771
31. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Miller, G.L. (ed.) Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996. pp. 212–219. ACM (1996). https://doi.org/10.1145/237814.237866, https://doi.org/10.1145/237814.237866
32. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18, https://doi.org/10.1007/978-3-319-91908-9_18
33. Hietala, K., Rand, R., Hung, S.H., Wu, X., Hicks, M.: Verified optimization in a quantum intermediate representation. arXiv preprint arXiv:1904.06319 (2019)
34. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259, https://doi.org/10.1145/363235.363259
35. Holík, L., Lengál, O., Simácek, J., Vojnar, T.: Efficient inclusion checking on explicit and semi-symbolic tree automata. In: Bultan, T., Hsiung, P. (eds.) Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6996, pp. 243–258. Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_18, https://doi.org/10.1007/978-3-642-24372-1_18
36. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6461, pp. 304–311. Springer (2010). https://doi.org/10.1007/978-3-642-17164-2_21, https://doi.org/10.1007/978-3-642-17164-2_21
37. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20, https://doi.org/10.1007/978-3-642-17511-4_20
38. Lengál, O., Šimáček, J., Vojnar, T.: VATA: A library for efficient manipulation of non-deterministic tree automata. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 79–94. Springer (2012)
39. Liu, J., Zhan, B., Wang, S., Ying, S., Liu, T., Li, Y., Ying, M., Zhan, N.: Formal verification of quantum algorithms using quantum Hoare logic. In: International conference on computer aided verification. pp. 187–207. Springer (2019)

40. Mateus, P., Ramos, J., Sernadas, A., Sernadas, C.: Temporal Logics for Reasoning about Quantum Systems, p. 389–413. Cambridge University Press (2009). https://doi.org/10.1017/CBO9781139193313.011
41. Paetznick, A., Svore, K.M.: Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. Quantum Info. Comput. **14**(15–16), 1277–1301 (nov 2014)
42. Perdrix, S.: Quantum entanglement analysis based on abstract interpretation. In: International Static Analysis Symposium. pp. 270–282. Springer (2008)
43. Unruh, D.: Quantum Hoare logic with ghost variables. In: 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–13. IEEE (2019)
44. Wei, C.Y., Tsai, Y.H., Jhang, C.S., Jiang, J.H.R.: Accurate bdd-based unitary operator manipulation for scalable and robust quantum circuit verification. In: Proceedings of the 59th ACM/IEEE Design Automation Conference. p. 523–528. DAC '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3489517.3530481, https://doi.org/10.1145/3489517.3530481
45. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21. pp. 33–38. Springer (2008)
46. Xu, M., Fu, J., Mei, J., Deng, Y.: Model checking QCTL plus on quantum Markov chains. Theor. Comput. Sci. **913**, 43–72 (2022). https://doi.org/10.1016/j.tcs.2022.01.044, https://doi.org/10.1016/j.tcs.2022.01.044
47. Xu, M., Li, Z., Padon, O., Lin, S., Pointing, J., Hirth, A., Ma, H., Palsberg, J., Aiken, A., Acar, U.A., et al.: Quartz: superoptimization of quantum circuits. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 625–640 (2022)
48. Ying, M.: Floyd-Hoare logic for quantum programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(6), 1–49 (2012)
49. Yu, N., Palsberg, J.: Quantum abstract interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 542–558 (2021)
50. Zhou, L., Barthe, G., Strub, P.Y., Liu, J., Ying, M.: Coqq: Foundational verification of quantum programs. Proceedings of the ACM on Programming Languages **7**(POPL), 833–865 (2023)
51. Zhou, L., Yu, N., Ying, M.: An applied quantum Hoare logic. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1149–1162 (2019)
52. Zhu, S., Hung, S.H., Chakrabarti, S., Wu, X.: On the principles of differentiable quantum programming languages. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 272–285. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386011, https://doi.org/10.1145/3385412.3386011

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

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

Parallel Equivalence Checking of Stabilizer Quantum Circuits on GPUs*

Muhammad Osama(✉), Dimitrios Thanos(✉), and Alfons Laarman(✉)

Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, The Netherlands
{m.o.mahmoud,d.thanos,a.w.laarman}@liacs.leidenuniv.nl

Abstract. Equivalence checking plays a crucial role in quantum circuit compilation, optimization, and verification. Stabilizer circuits can be simulated classically by tracking the so-called stabilizer operators in linear time. But the simulation of large stabilizer circuits with thousands of qubits and gates, arising e.g. in the study of novel quantum error-correction protocols, still poses a challenge. In this work, we propose a GPU-based deterministic algorithm for equivalence checking of stabilizer circuits using the stabilizer tableau formalism. We explore various design choices and implement the most efficient version. Our algorithm significantly outperforms the state-of-the-art CCEC checker (which relies on the Stim simulator) in terms of time, memory, and energy. Our approach demonstrates up to two orders of magnitude speedup over existing methods. Notably, previous attempts at GPU acceleration in this area were unsuccessful, making this the first effective implementation.

1 Introduction

Motivation. We are currently in the noisy intermediate-scale quantum (NISQ) era of quantum computing [25] and progressing towards machines capable of handling circuits with many qubits and gates of quantum circuits [2,4,29]. To build large-scale quantum computers, efficient compilation, optimization, and verification of quantum circuits are essential. Equivalence Checking (EC), a critical subroutine, plays a vital role in these tasks. Therefore, it is crucial to perform EC as efficiently as possible.

This work focuses on stabilizer circuits—an important class of quantum circuits that are classically simulatable [1,12]. Stabilizer circuits are integral to the development of error-corrected quantum computers, serving as the foundation for the most prominent and scalable error-correcting codes. A leading example is the *surface codes* [7,10–12,15,17–20,28]. The stabilizer circuits necessary for fault-tolerant quantum computing are exceptionally large, spanning millions of qubits and gates [7,10,17,19], surpassing the capabilities of current multi-core architectures like CPUs.

* This work was supported by the Dutch National Growth Fund (NGF), as part of the Quantum Delta NL programme.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 109–128, 2025.
https://doi.org/10.1007/978-3-031-90660-2_6

To further enhance scalability and sustainability of EC, we propose harnessing the Graphics Processing Units (GPUs) for parallel processing. By utilizing the abundant cores of GPUs, we achieve substantial accelerations in EC computations and in the process improve their energy efficiency [13]. Our novel algorithm efficiently performs EC for stabilizer circuits on GPUs, outperforming the state-of-the-art *Clifford-Circuits Equivalence Checking* (CCEC) framework (which uses Stim as a backend) [29] by an acceleration factor of 186×, while reducing energy consumption by up to 98% of energy. Stim [9] is the state-of-the-art for CPU-based simulation of stabilizer quantum circuits. It excels in simulating large circuits by utilizing the vectorized instructions in modern CPUs, enabling simultaneous execution of arithmetic and logical operations on large scale. While Stim is tailored for CPU-based simulations, the growing complexity of stabilizer circuits demands more powerful hardware solutions, such as GPUs. However, previous attempts to use GPUs for these circuits were unsuccessful. Notably, an earlier attempt by the author of Stim to accelerate *tableau formalism* [12] using GPUs did not achieve the desired results.[1] With this work, we demonstrate that GPUs can yield significant gains in performing this task.

Related Work. The state-of-the-art tool for EC of stabilizer circuits is CCEC [29], a framework that uses Stim simulator [9] as a backend. CCEC leverages a theorem that reduces the problem of quantum circuit equivalence to a computationally tractable statement for stabilizer circuits. Specifically, evaluating the equivalence of two stabilizer circuits can be achieved using techniques designed for their efficient simulation. By replacing CCEC's Stim components with an independent GPU-based algorithm, we can develop a more efficient EC tool.

We now discuss previous efforts to parallelize the simulation of universal quantum circuits—generally a more complex task than simulating stabilizer circuits. Accelerating the simulation of universal circuits does not inherently improve performance for special classes, such as stabilizer circuits; instead, algorithms specifically tailored for stabilizer circuits are often necessary. For instance, while Stim can efficiently handle stabilizer circuits with millions of qubits, tools designed to handle *universal* circuits—such as Qiskit-Aer [16], which utilizes GPUs, can scale in practice to hundreds of qubits.

Early attempts at parallelizing the simulation of quantum circuits, primarily focused on algebraic operations like matrix multiplication, which are fundamental to quantum simulation [6, 24]. While straightforward, these methods are restricted to circuits with hundreds of qubits or fewer. Others [27] use *ZX*-diagrams and parallelization to rewrite and optimize circuits to reduced versions with fewer T gates.[2] Authors in [8] introduce *stabilizer frames*, a formalism to reduce the simulation of universal circuits to the simulation of stabilizer states. Albeit, the number of frames increases exponentially with the number of T gates.

ECMC [21] is another relevant tool that, although not parallelized, utilizes theorem 1 to perform EC. By leveraging weighted model counting, ECMC can process universal quantum circuits with hundreds of qubits. This is on par

[1] Refer to section 5.3 (GPU Experiment Failure) in [9, p.14].
[2] T gates are notoriously *hard* to simulate [22].

with other simulators such as [26] which uses symbolic binary decision diagrams (CFLOBDD) to simulate universal circuits.

Contributions. In summary, our contributions are as follows:

- ⋆ Exploring the design space and analyzing various choices to derive an efficient data structure for *stabilizers tableau formalism* [12] on the GPU.
- ⋆ A parallel algorithm on two levels: First, sweeping all parallel gates simultaneously within a specific time *window*, and second, updating the tableau concurrently on a GPU.
- ⋆ An implementation for checking equivalence of stabilizer circuits on GPUs, leveraging *concurrent kernel execution* [23].
- ⋆ Thorough evaluation of our tool QuASARQ compared to CCEC [29] and ECMC [21] which uses the same theorem.

Outline. We give the formal definitions of the quantum basics and the EC of the stabilizer circuits in the next section. In section 3, we analyze various design choices for the data structure of tableau formalism and present our algorithms for EC. Section 4 provides a thorough evaluation of our algorithms on the GPU. Finally, in section 5, we conclude and present some directions for future work.

2 Preliminaries

2.1 Quantum States and Fundamental Concepts

A qubit is described by a linear combination of the *basis states* $|0\rangle$ and $|1\rangle$ that can be identified with the 2-vectors $\begin{bmatrix}1 & 0\end{bmatrix}^T$ and $\begin{bmatrix}0 & 1\end{bmatrix}^T$, respectively. These states are analogous to the classical bits 0 and 1 but in the quantum case, a single qubit can take on the value of the linear combination $|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ where α_0, α_1 are complex numbers and $|\psi\rangle$, like any quantum state, must be normalized to be a unit vector (i.e., $|\alpha_0|^2 + |\alpha_1|^2 = 1$). The linear combination of basis states is also known as *superposition*.

The operation for combining n single-qubit states $|\psi\rangle_1, \cdots, |\psi\rangle_n$ into an n-qubit state $|\psi\rangle_1 \otimes \cdots \otimes |\psi\rangle_n$ is known as the *tensor product*. It is denoted by $\otimes$ and is typically known in linear algebra as the Kronecker product. The n-qubit states have the form $\sum_{\ell \in \{0,1\}^n} \alpha_\ell |\ell\rangle$ where $|\ell\rangle$ is the conventional way of denoting a string of multiple tensored single-qubit states, i.e., we suppress the $\otimes$ symbol and say $|1011\rangle$ instead of $|1\rangle \otimes |0\rangle \otimes |1\rangle \otimes |1\rangle$. If a quantum state cannot be factored into a tensor product of its individual subsystems, then we say it is an *entangled* state. For example, the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is entangled, while the state $|1001\rangle$ is not.

In general, a quantum state $|\psi\rangle$ of n qubits can be written as a vector of 2^n complex numbers: $|\psi\rangle = [\alpha_0, \alpha_1, \cdots, \alpha_{2^n}]^T$. For every state, there is an *adjoint* defined as $(|\psi\rangle^*)^T$ and denoted by $\langle\psi|$. The "$*$" operator is a complex conjugation, meaning that we replace every entry with its complex conjugate, then transpose the vector (i.e., transforming it into a row vector). The normalization requirement, can be now reformulated into $\langle\psi| \cdot |\psi\rangle = 1$. This product is also known as *inner product* and can be performed between distinct states.

2.2 Stabilizer Circuits

Quantum states can be manipulated by two types of operations: quantum gates and quantum measurements. Quantum gates are *unitary* matrices, representing reversible linear transformations denoted as $U \in \mathbb{C}^{2^n \times 2^n}$. These gates preserve both the norm and the inner product of quantum states. A key property of unitary gates is that their inverse, denoted $U^\dagger$, satisfies $UU^\dagger = U^\dagger U = I$, where $U^\dagger = (U^*)^T$ represent the conjugate transpose of U. The effect of gate U acting on a state $|\psi\rangle$, can be calculated using matrix-vector multiplication $U \cdot |\psi\rangle$.

Two important (but non-universal) gate sets are the Pauli gates, defined as the following unitary matrices:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

and the Clifford gate set (CX is also called $CNOT$):

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Stabilizer circuits, also known as *Clifford circuits* are quantum circuits composed of Clifford gates. The quantum states generated by applying these circuits to the computational basis $|0\rangle^{\otimes n}$ are called *stabilizer states*. Interestingly, stabilizer circuits can be simulated efficiently by a classical algorithm [1,12].

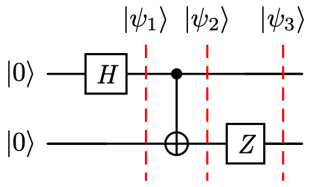


Fig. 1: A 2-qubit stabilizer circuit.

Example 1. Consider the circuit shown in fig. 1, which starts with the initial state $|00\rangle$. The horizontal wires represent the qubits, and the dashed lines indicate the intermediate states. The final state $|\psi_3\rangle$ is obtained by sequentially applying the gates to the initial state $|00\rangle$ from left to right.

1. Apply the Hadamard gate (H) to the first qubit: $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The circuit's state becomes: $|\psi_1\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$.
2. Apply the Controlled-X gate (CX) to both qubits. The effect of CX is to flip the target qubit when the control qubit is $|1\rangle$. Thus, $|\psi_2\rangle = CX|\psi_1\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.
3. Apply the Pauli-Z gate (Z) to the second qubit. Since $Z|0\rangle = |0\rangle$ and $Z|1\rangle = -|1\rangle$, the final state is: $|\psi_3\rangle = Z|\psi_2\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$.

Equivalently, we could represent the entire circuit in the previous example as a unitary operator $U = (I \otimes Z) \cdot CX \cdot (H \otimes I)$, and then apply it to $|00\rangle$ to compute the final state. Generally, every circuit corresponds to a unitary operator in $\mathbb{C}^{2n \times 2n}$.

The Pauli group $\mathcal{P}$ on n qubits is defined as the set of operators of the form $\mathcal{P} = \pm\mathcal{P}_1 \otimes \cdots \otimes \mathcal{P}_n$ with $\mathcal{P}_i \in \{I, X, Y, Z\}$. For simplicity, the tensor product

is often suppressed, and n-Pauli strings are written in shorthand. For example, $X \otimes I \otimes Y \otimes X \otimes Z$ can be compactly written as $XIYXZ$. The *Clifford group* C on n qubits consists of the unitary operators V that normalize the *Pauli group* $\mathcal{P}$, that is, $C = \{V \in \mathbb{C}^{2n \times 2n} \mid V\mathcal{P}V^\dagger = \mathcal{P}\}$. The Clifford group is generated by the Clifford gate set [22]. Substituting V with any Pauli gate satisfies the normalization condition $V\mathcal{P}V^\dagger = \mathcal{P}$. This shows that Pauli gates belong to the Clifford group, and thus they can be written as a combination of gates from the Clifford gate set. Each stabilizer state, generated by a stabilizer circuit, can be uniquely encoded by a subgroup of the Clifford group, requiring exactly n independent generators, where each generator is an n-Pauli string.

Stabilizer states are called this way because they are *stabilized* by certain operators. A unitary operator U stabilizes a state iff $U|\psi\rangle = |\psi\rangle$. For example, the Pauli string IZ stabilizes the state $|0\rangle \otimes |0\rangle = |00\rangle$. A state can be stabilized by multiple Pauli strings; for instance, $|00\rangle$ is also stabilized by II, ZI and ZZ. These Pauli strings generate the maximal commutative subgroup $\mathcal{H}_{\max} = \{II, IZ, ZI, ZZ\}$, which is the stabilizer group of $|00\rangle$. Each maximal commutative subgroup of the Pauli group stabilizes exactly one stabilizer state [22].

Although the stabilizer group for an n-qubit stabilizer state has 2^n elements, only n Pauli strings are needed to generate such a group. For example, $\mathcal{H}_{\max}$ can be generated by the two generators $\{IZ, ZZ\}$. This property allows stabilizer states to be efficiently and succinctly represented by a data structure such as the tableau formalism.

Tableau formalism. A stabilizer state's generators can be described by a binary table known as a *stabilizer tableau* [12]. We refer to the tableau as the tuple $\mathcal{T} = (X_M, Z_M, S_M)$, where the matrices X_M and Z_M together represent the Pauli strings and S_M encodes their signs (see eq. (1)). Each row encodes a string of n Pauli operators and has size $2n + 1$, where n is the number of qubits. If the (i, j)-th entry of X_M is 1 and the corresponding entry in Z_M is 0, then the j-th term of the i-th Pauli string corresponds to an X operator. Analogously, a 0 in X_M and 1 in Z_M correspond to a Z operator. If both entries are 0, it represents the identity operator I. Conversely, if both entries are 1, it represents a Y operator. For example, the tableau below encodes the generators $\{-ZII, IZI, IXY\}$.

$$\mathcal{T} = \left[\begin{array}{ccc|ccc|c} 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}\right] \cong \left\langle \begin{array}{c} -ZII \\ IZI \\ IXY \end{array} \right\rangle \tag{1}$$

$X_M \quad Z_M \quad S_M$

To update $\mathcal{T}$ into a new state, after the action of a Clifford gate on a qubit q, one needs to apply the following updating rules for all $i \in \{1, \ldots, n\}$ [12]:

$$
\begin{aligned}
H_q &: S_M(i) := S_M(i) \oplus \big(X_M(i,q) \wedge Z_M(i,q)\big),\\
&\quad \mathrm{Swap}\big(X_M(i,q), Z_M(i,q)\big);\\
P_q &: S_M(i) := S_M(i) \oplus \big(X_M(i,q) \wedge Z_M(i,q)\big),\\
&\quad Z_M(i,q) := Z_M(i,q) \oplus X_M(i,q);\\
CX_{q_1,q_2} &: S_M(i) := S_M(i) \oplus \big(X_M(i,q_1) \wedge Z_M(i,q_2) \wedge \neg\big(X_M(i,q_2) \oplus Z_M(i,q_1)\big)\big),\\
&\quad Z_M(i,q_1) := Z_M(i,q_1) \oplus Z_M(i,q_2),\\
&\quad X_M(i,q_2) := X_M(i,q_2) \oplus X_M(i,q_1);
\end{aligned}
\tag{2}
$$

Observe that collapsing the signs involves the *sequential* XOR operation ($\oplus$), which poses a challenge for parallelization. Algorithm 2 addresses this issue, though it increases the algorithm's complexity. The following example shows how gates can be applied in parallel to $\mathcal{T}$ when acting on independent qubits, hence affecting independent columns.

Example 2. Given a 3-qubit system initialized to the $|000\rangle$ state, assume that the gates H_1 and P_3 are applied in order, to the qubits 1 and 3 respectively. The affected columns according to eq. (2), are updated and highlighted by colours:

$$
\left[\begin{array}{ccc|ccc|c} 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0 \end{array}\right]
\overset{H_1}{\Longrightarrow}
\left[\begin{array}{ccc|ccc|c} 1&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0 \end{array}\right]
\overset{P_3}{\Longrightarrow}
\left[\begin{array}{ccc|ccc|c} 1&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&1&0 \end{array}\right]
\tag{3}
$$

2.3 Equivalence Checking

As each quantum circuit corresponds to a unitary operator, a naive approach to EC of two circuits U, V would be to compare the exact values of the operators that are given in a decomposed form as sequences of gates. Specifically, one would compose the gates by matrix multiplication to obtain the two operators and check if $U = cV$, where $c \in \mathbb{C}$ with norm 1. The constant c is called *global phase* and has no physical significance—It is unobservable [22]. While this approach would work in principle, the matrix will grow exponentially w.r.t. the number of qubits.

Fortunately, there exists a more efficient approach, particularly suited for stabilizer circuits. It exploits a mathematical result (theorem 1 below) that enables equivalence evaluation by computing in the stabilizer basis. As we will see, the tableau formalism rigorously facilitates this task. To evaluate if two stabilizer circuits U, V are equivalent, it suffices to check whether their outputs result in the exact same tableau when their simulation is initialized with the I_Z tableau.[3] Additionally, their tableaux should be identical when initialized with the I_X tableau.[4] It is important to emphasize that this is not the same as comparing the output states of two simulated circuits. When a tableau is used for

[3] Tableau with the Z_M matrix filled with 1s in its diagonal and 0s everywhere else.
[4] Tableau with the X_M matrix filled with 1s in its diagonal and 0s everywhere else.

simulation, the generators that are used to generate the group which encodes some state, are not unique. Therefore, in the case of simulation, you may have different tableaux which encode the same state. But in our case, since we are not interested in comparing states but the actual tableaux, the outputted generators, and thus the tableaux, must be identical. This process can efficiently decide equivalence up to a global phase factor. See theorem 1 below and its proof in [29].

Theorem 1. *Let U,V be unitaries, each of them corresponding to an n-qubit circut. Then $U \cong V$ if and only if for all $\mathcal{P} \in \{X, Z\}$ and $i \in [n]$, the condition $U\mathcal{P}_i U^\dagger = V\mathcal{P}_i V^\dagger$ holds, where $\mathcal{P}_i = I^{\otimes i-1} \otimes \mathcal{P} \otimes I^{\otimes n-i}$.*

When referring to the equivalence of two circuits $\mathcal{C}, \mathcal{C}'$ that are not considered as single unitary operators but as a circuit (i.e. a sequence of gates), by abuse of notation, we will use $\mathcal{C} \cong \mathcal{C}'$ when $\mathcal{C}$ and $\mathcal{C}'$ encode the decomposition of equivalent unitary operators.

2.4 GPU Programming

CUDA [23] is a programming interface that enables general-purpose programming for a GPU. It has been developed and continues to be maintained by NVIDIA since 2007. In this work, we use CUDA with C++. Therefore, we use CUDA terminology when we refer to thread and memory hierarchies.

A GPU consists of a finite number of *streaming multiprocessors* (SM), each containing hundreds of *cores*. For instance, the RTX 4090, which we used for this work, has 128 SMs containing 16,384 cores. A GPU computation can be launched in a program by the *host* (CPU side of a program), which calls a GPU function called a *kernel*, executed by the *device* (GPU side of a program).

When a kernel is called, the number of threads needed to execute it is specified. According to the CUDA paradigm [23], these threads are partitioned into thread *blocks*, i.e., 3-dimensional (3D) vectors grouping threads up to 1,024. Each thread block is assigned to an SM. All threads together form a 3D *grid* where threads and blocks can be indexed by a 1D, 2D, or 3D unique identifier (ID) accessible within the kernel. This ID works similarly as the (x, y, z) coordinates in 3D space. With this ID, different threads in the same block can process multi-dimensional data (e.g., X_M and Z_M).

Memory Hierarchy. A GPU has multiple types of memory, with the largest being the *global memory*. This memory can be used to transfer data between the host and the device. It is accessible by all GPU threads and offers high bandwidth but also high latency. Other types are *shared memory* and *registers*. The former is on-chip memory with a low latency, used as block-local memory. Threads within a block can share data via this memory. Registers are the fastest and are used to store thread-local data. Yet, their size is very small. Allocating too much memory for thread-local variables can cause data to spill over into global memory, dramatically reducing performance.

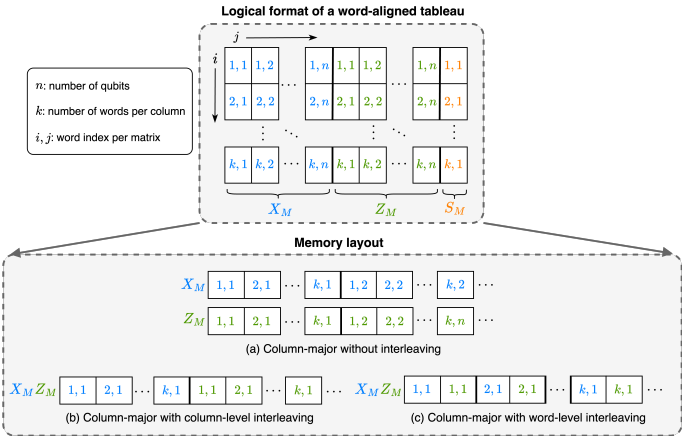


Fig. 2: Logical and physical formats of a word-aligned tableau.

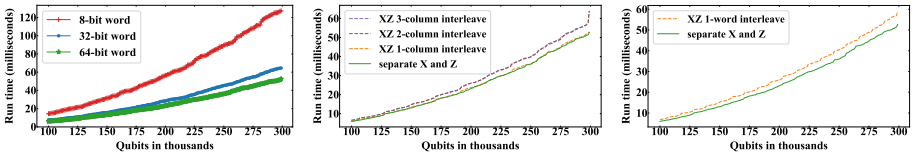
GPU threads can use atomic operations to manipulate data atomically in both global and shared memory. For example, ATOMICXOR(`addr`, `val`) atomically XORs the element stored at `addr` with the 32-bit or 64-bit value `val`.

3 Parallel Equivalence Checking on GPUs

In this section, we begin with evaluating the design decisions for the storing and updating of the data structure that we later use for efficient EC. The data structure is based on the stabilizer tableau but optimized for GPU usage. Next, we introduce the implementation for EC on the GPU, taking advantage of our optimized data structure. Our proposed implementation incorporates two levels of parallelism. First, we process different generators (Pauli strings) independently in parallel. Second, we maximize parallelism by processing as many *parallel gates* as possible simultaneously. However, calculating the signs of the Pauli strings in parallel remains a challenge. We present a tree-based approach for collapsing signs on-the-fly.

3.1 Data Structure Design

Upon designing the data structure for storing and updating the stabilizer tableau on a GPU, several key considerations were made to ensure high efficiency of both the GPU memory and execution performance. GPUs operate on aligned memory, meaning data are stored and accessed in *words* (e.g., 8, 32, or 64 bits). This allows multiple bits to be processed simultaneously with a single instruction (e.g. XOR), which is crucial for fast tableau updates. To represent the stabilizer tableau, we encode the binary form of the tableau as described in section 2.2, eq. (1). Figure 2 visualizes the logical format of tableau $\mathcal{T}$. Words in X_M, Z_M, and S_M are indexed by $i \in [k]$ and $j \in [n]$, where k is the number of generators divided by the word length in bits. The tableau size is thus $k \cdot (2n + 1)$ words.



(a) Different word sizes of 8, 32, and 64 bits (b) Column-level interleaving vs no-interleaving (c) Word-level interleaving vs no-interleaving

Fig. 3: Performance impact of different architectural choices.

Memory Alignment. The tableau is stored in 64-bit words, the maximum size supported by the GPU's architecture, enabling the processing of 64 bits in parallel with a single instruction. This reduces the overall number of operations and boosts performance, as demonstrated in our experiments in fig. 3a. The experiments were conducted using benchmarks introduced in section 4. Notably, the 64-bit word size outperforms its counterparts significantly. The main reason for the performance peak lies in the fact that a 64-bit word allows for the execution of a single instruction on 64 different bits simultaneously. This minimizes the computational steps, leading to faster tableau updates.

Memory Layout. Given the nature of the tableau updates when applying Clifford gates (cf. example 2 in section 2.2), it is logical to store the X_M and Z_M in a column-major format (See fig. 2 in the bottom). This ensures each column can be accessed contiguously in memory, improving locality during tableau updates.

In addition, we experimented with *interleaving* the X_M and Z_M matrices at both the word and column level. The rationale behind interleaving is to improve memory locality by storing data that will be accessed together (i.e., from the same qubit) close to each other in memory. However, our results showed that interleaving did not provide a performance benefit despite the fact that the gate updates require communication between the X_M and Z_M columns. We hypothesize that the increased complexity in memory indexing mitigates any potential benefits from interleaving. As shown in fig. 3b and fig. 3c, separate storage of X_M and Z_M matrices yielded better performance. This finding is consistent with previous results from Stim [9] simulator where word interleaving was ruled out, albeit no performance analysis was provided.

Tableau Slicing. For circuits with hundreds of thousands of qubits and more, storing the entire tableau in memory can become infeasible. To tackle this, we implemented a tableau slicing strategy. The tableau is sliced horizontally into smaller, non-square blocks (not to be confused with thread blocks in CUDA), each of which can be processed independently (recall that generators are independent as demonstrated in example 2). This allows us to check the equivalence of circuits with large qubit counts using only the available GPU memory.

Each block is initialized to the same state as the full tableau. For example, in a circuit with 1 million qubits, the tableau would require approximately 250GB of memory. By slicing the tableau into smaller sections, we can reduce this memory requirement to fit within the available GPU memory (24GB in our experiments), as shown in table 1.

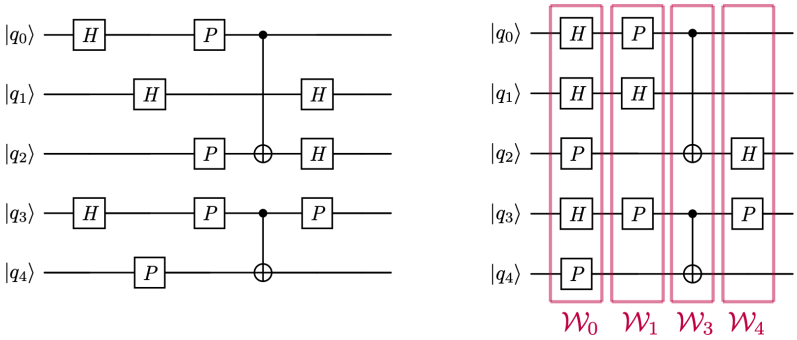


Fig. 4: Input circuit on the left. Scheduled circuit on the right.

Example 3. For a system with 256 qubits, if we store the tableau in 64-bit words, the full tableau would require $k \cdot (2n + 1) = (256/64) \cdot 513 = 2,052$ words in memory. By slicing the tableau's generators into two halves, each block will require 1,026 words, and the update operations can be performed independently on each block. This strategy ensures that memory is fully utilized while maintaining computational consistency across blocks.

3.2 The Main Algorithm for Parallel Equivalence Checking

Given two circuits, denoted as $\mathcal{C}$ and $\mathcal{C}'$, our primary objective is to verify whether ($\mathcal{C} \cong \mathcal{C}'$) by deploying the GPU massive resources in the tableau formalism. To achieve this, we must define gate parallelism and determine how to schedule their execution. Informally, *parallelizable gates* are those that can be executed concurrently because their corresponding operations commute and are independent (i.e., not causally related). We formalize this in terms of causal dependency as follows.

Definition 1 (Gate Dependency Relation). *Given a circuit $\mathcal{C}$, the causal dependency between gates defines a partial order "$\preceq$" in the set of gates $g \in \mathcal{C}$, such that $g \preceq g'$ iff g' is causally dependent on g.*

The transitivity of the relation guarantees that the following definition of *parallel gates* is well defined with no dependencies between gates.

Definition 2 (Parallel gates). *Given a circuit $\mathcal{C}$, a set $W \subseteq \mathcal{C}$, is a set of* parallel gates *(or* window*) iff $\forall g, g' \in W,\ g \not\preceq g' \ \wedge \ g' \not\preceq g$.*

Definition 3 (Maximal Window). *A window $W \subseteq \mathcal{C}$ is* maximal *iff there is no other window $W' \subseteq \mathcal{C}$ such that $W \subsetneq W'$.*

Henceforward, by a *window*, we mean a *maximal window* and denote it by $\mathcal{W}$.

The above definitions form the basis for the scheduleWindows routine of Algorithm 1, which outlines the main steps for EC on the GPU. Details of this routine will be discussed later. Given input circuits $\mathcal{C}$ and $\mathcal{C}'$, the algorithm checks their equivalence and returns the outcome in the Boolean variable

Algorithm 1: Parallel Equivalence Checking.

Input: $\mathcal{C}, \mathcal{C}', n$, `stream1`, `stream2` // stream1, stream2 are CUDA-related variables.
Output: `isEquivalent` // $\mathcal{C} \cong \mathcal{C}'$?
1 $\mathcal{R}, \mathcal{R}' \leftarrow$ scheduleWindows$(\mathcal{C}, \mathcal{C}')$ // Implement definitions 2 and 3
2 $\mathcal{R}_d, \mathcal{R}'_d \leftarrow$ copyToDeviceAsync$(\mathcal{R}, \mathcal{R}')$ // $\mathcal{R}_d, \mathcal{R}'_d$ reside in GPU memory.
3 $\mathcal{T}, \mathcal{T}'$, k, `numTableauBlocks` $\leftarrow$ allocateTableaux$(2n(2n+1))$
4 **foreach** $b := 1, 2, 3, \ldots,$ `numTableauBlocks` **do** // Loop over all tableau blocks
5 `isEquivalent` $\leftarrow$ checkEquivalence$(I_Z, \mathcal{R}_d, \mathcal{R}'_d, \mathcal{T}, \mathcal{T}', b, k,$ `stream1`, `stream2`$)$
6 **if** `isEquivalent` **then** // If equivalent, then we check for the I_X tableau.
7 `isEquivalent` $\leftarrow$ checkEquivalence$(I_X, \mathcal{R}_d, \mathcal{R}'_d, \mathcal{T}, \mathcal{T}', b, k,$ `stream1`, `stream2`$)$
8 **return** `isEquivalent` // Return equivalence outcome of $\mathcal{R}$ and $\mathcal{R}'$.
9 *function* checkEquivalence(`initState`, $\mathcal{R}_d, \mathcal{R}'_d, \mathcal{T}, \mathcal{T}', b, k,$ `stream1`, `stream2`$)$:
10 $\mathcal{T} \leftarrow$ initTableauAsync(`initState`, $(b-1)\cdot k$, `stream1`) // Offset: $(b-1)\cdot k$.
11 $\mathcal{T}' \leftarrow$ initTableauAsync(`initState`, $(b-1)\cdot k$, `stream2`)
12 **forall** $\mathcal{W}_d, \mathcal{W}'_d \in \{\mathcal{R}_d, \mathcal{R}'_d\}$ **do**
13 $\mathcal{T} \leftarrow$ updateTableauAsync$(\mathcal{T}, \mathcal{W}_d, k,$ `stream1`$)$
14 $\mathcal{T}' \leftarrow$ updateTableauAsync$(\mathcal{T}', \mathcal{W}'_d, k,$ `stream2`$)$
15 synchronizeStreams(`stream1`, `stream2`)
16 **return** $\mathcal{T} = \mathcal{T}'$ // Perform tableaux matching ($\mathcal{T} = \mathcal{T}'$) on the GPU.

`isEquivalent`. The CPU schedules the circuits $\mathcal{R}, \mathcal{R}'$ and transfers them to the GPU memory, while the GPU handles the updates of the tableaux $\mathcal{T}, \mathcal{T}'$ and checks for equivalence. Typically, the input circuit is stored as a sequence of symbols, one for each gate, along with the associated qubits on which they act. Standard examples are OpenQASM [5] or the Stim [9] format. Figure 5 illustrates the workload distribution between the CPU and GPU for those tasks.

The scheduleWindows procedure (l.1) maximizes parallelism by organizing the gates into maximally independent windows (cf. definition 3). This scheduling ensures efficient utilization of GPU resources during tableau updates. When a maximal window is constructed, we schedule additional windows until all the gates in $\mathcal{C}$ are assigned to $\mathcal{R}$. The procedure works as follows: starting with the input circuit (left part of fig. 4) stored in a queue, we fill an empty window with parallel gates (removed) from the queue until either the window reaches its maximal size $|\mathcal{W}| = n$, or no more parallel gates can be added due to dependencies (definition 1). This process is repeated, scheduling new windows until the queue is empty (see the right part of fig. 4 for the end result).

The number of windows in $\mathcal{R}$ is proportional to the circuit depth (i.e., maximum number of gates per wire). In circuits with minimal gate dependencies, more gates can be scheduled into a single window, reducing the effective depth and improving performance by allowing more parallel updates to the tableau.

Next, at l.2, $\mathcal{R}, \mathcal{R}'$ are copied asynchronously to the device. The subscript d in $\mathcal{R}_d$ denotes the device version of $\mathcal{R}$. Asynchronous data transfers can be performed concurrently w.r.t. the host. For example, the tableau allocation and slicing can proceed at l.3 while the circuit transfer is pending. The allocateTableaux function blocks execution until all prior data transfers synchronize. It takes $2n(2n+1)$ bits to allocate enough space for two tableaux (one per circuit). Tableaux are allocated physically in arrays of 64-bit words as explained in section 3.1. Words are accessed in column-major. If the tableaux fit entirely into the device memory, the values of `numTableauBlocks` and k will be

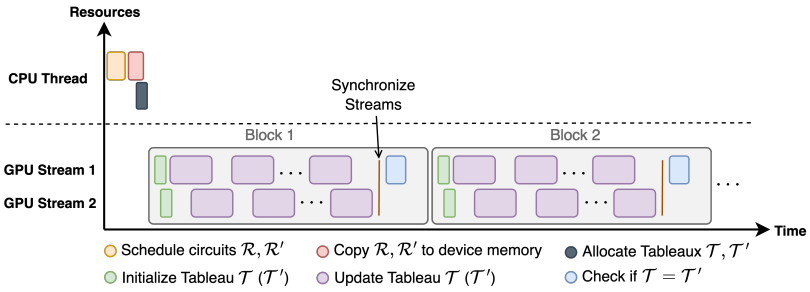


Fig. 5: Workload of algorithm 1.

1 resp. $n/64$. Otherwise, `numTableauBlocks` is some value greater than 1 such that $0 < k \leq (n/64)$. Both tableaux have identical dimensions as the number of qubits is the same for both circuits. For simplicity, we use $\mathcal{T}$ to refer to either the entire tableau or any of its individual blocks, depending on the context. As illustrated in fig. 5, the CPU thread's workload is small compared to the GPU tasks and does not add any overhead to the overall runtime.

The loop at l.4 checks $\mathcal{R}_d$ and $\mathcal{R}'_d$ for equivalence for every block b via the routine checkEquivalence. At l.5, we first check under the initial value of the tableau I_Z (cf. section 2.3 for more details). If the circuits are equivalent (l.6), we try again with the tableau I_X (l.7).

checkEquivalence is defined at lines 9-16. Operations highlighted in violet are processed by the GPU. At lines 10-11 tableaux are set to their initial value `initState` concurrently via the streams `stream1` and `stream2`. If there are more than one block, i.e., $b > 1$, the next block must be offset by $((b-1) \cdot k)$ words. Similarly, the loop at l.12 launches two concurrent kernels updateTableauAsync via the streams `stream1` and `stream2` to update the tableaux for each *window* in both circuits. A stream is a sequence of instructions that are executed in issue-order on the GPU [23]. Using streams allows concurrent execution of kernels, provided there are enough unoccupied SMs. See fig. 5 for the distribution of our kernels over `stream1` and `stream2`. Observe that these kernels may not completely overlap depending on the availability of SMs. Our experiments showed that kernels overlapping leads to 25% gain in kernel performance compared to non-overlapping. Once all windows are processed, both streams are synchronized at l.15 to ensure the stabilizers in both tableaux are fully built. At 16, we check the words in $\mathcal{T}$ against $\mathcal{T}'$ and return the outcome of the comparison. Next, we explain how updateTableauAsync is implemented.

3.3 Tableau Manipulation on GPU

The kernel in algorithm 2 updates the tableau $\mathcal{T}$ by applying parallel gates stored in $\mathcal{W}_d$. Within this process, threads in the y-dimension (l.2) are responsible for fetching new words per column, whereas threads in the x-dimension (l.3) handle the retrieval of parallel gates. The variables $\texttt{tid}_y$ and $\texttt{tid}_x$ hold the global

Algorithm 2: Tableau Manipulation on GPU.

```
   Input: 𝒯, 𝒲_d, k, sh, stream                                  // sh is a shared-memory array
   Output: 𝒯
1  device kernel updateTableauAsync(𝒯 := (X_M, Z_M, S_M), 𝒲_d, k, stream):
2      for all tid_y ∈ [0, k) do in parallel        // tid_y fetches a new word per column.
3          for all tid_x ∈ [0, |𝒲_d|) do in parallel          // tid_x fetches a new gate.
4              switch 𝒲_d[tid_x] do                          // Switch over parallel gates.
5                  case H do
6                      q ← getQubits(H)
7                      sh[t_x] ← S[tid_y] ⊕ (X_M[q · k + tid_y] ∧ Z_M[q · k + tid_y])
8                      swap(X_M[q · k + tid_y], Z_M[q · k + tid_y])        // Update X_M, Z_M.
9                  case ... do ...    // Extendable to other gates like CX, CY, etc.
10         synchronizeThreads( )                            // Synchronize shared memory.
11         collapseSigns(sh, |𝒲_d|)                  // Collapse signs in shared memory.
   /* A function to collapse all thread-local signs using tree structure.      */
12 device function collapseSigns(sh, |𝒲_d|):
13     for l := B_x/2, B_x/4, ..., 1 do                          // Loop over log(B_x) steps
14         if t_x < l then sh[t_x] ← sh[t_x] ⊕ sh[t_x + l]
15         synchronizeThreads( )
16     if t_x = 0 ∧ sh[0] = 1 then        // Collapse root sh[0] with old signs S[tid_y].
17         atomicXOR(S[tid_y], sh[0])
```

thread ID in y- and x-dimension, respectively. To branch over all supported gates by our algorithm, a switch statement is used at l.4. Here, we only show the H gate but many more Clifford gates are directly supported without decomposition. At lines 5-9, we implement the update rules executed for each gate modifying both the words encoding the Pauli strings and their signs. Recall that $\mathcal{T}$ is stored in a column-major array; thus, given a qubit q, and the column length k, a column in the X_M or Z_M matrix can be accessed in position $(q \cdot k)$. To access words per column in parallel, the thread ID ($\texttt{tid}_y$) is used to offset the column index.

The function getQubits at l.6 maps the input gate to its connected wires or qubits. Qubit values are assumed to start from 0 to $n-1$ to align with the array index. At l.7, the signs are calculated and temporarily stored in shared memory (denoted as `sh`). The variable $\texttt{sh}[\texttt{t}_x]$ holds the thread-local sign per thread block in the x-dimension. To ensure each thread block has written its signs to `sh` before proceeding, we need to synchronize at l.10. Once synchronized, these signs need to be collapsed into a final result, which will be used to update the global sign in the tableau. This collapsing process is handled by a specialized procedure, collapseSigns, which we will describe in more detail next. The main update to the tableau occurs at l.8, where the words in X_M and Z_M (representing the Pauli strings) are swapped in global memory.

Signs Collapsing. In our initial attempt to parallelize the signs collapsing process, we employed two kernels. The first kernel handled the tableau updates and the local sign calculations, storing the results in global memory. The second kernel read these local signs from global memory and performed the collapsing using the atomic operation atomicXOR. Unfortunately, this trial proved to be considerably slow due to the excessive global memory accesses (approximately $k \cdot n$) and the high usage of atomic instructions ($k \cdot |\mathcal{W}_d|$).

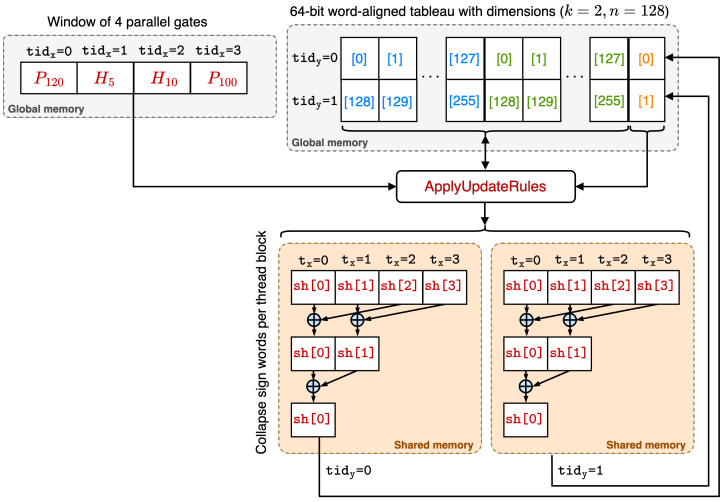


Fig. 6: Running example of algorithm 2 on 128-qubit system.

To remedy the global memory access issue, we made a second attempt. In this trial, we eliminated the temporary storage of local signs in global memory. Instead, we collapsed all local signs atomically on-the-fly during the tableau update within the whole window. While this approach saved roughly $(k \cdot n)$ memory accesses compared to the initial attempt, the number of atomic operations remained a bottleneck. This is where our tree-like approach comes into play. The CollapseSigns routine in algorithm 2 effectively reduces the number of atomics by collapsing signs per thread block using a bottom-up binary tree. The downside is the excessive usage of global memory during the tree build-up, which we mitigate by utilizing the fast on-chip shared memory.

The loop at l.13 performs the collapsing using the XOR operator in shared memory by sweeping the binary tree in a bottom-up manner. Initially, l is set to $B_x/2$, where $B_x = 2^w$ and $w \in \{1, 2, \ldots, 10\}$. At l.14, we XOR in parallel the first half in $\texttt{sh}[\texttt{t}_x]$ with the second half $\texttt{sh}[\texttt{t}_x + l]$. Afterwards, threads are synchronized and l is halved until only one element remains at $\texttt{sh}[0]$ (resembles the root of the tree), which holds the collapsed block-local sign. Finally, at l.17, we collapse the block-local signs stored at $\texttt{sh}[0]$ atomically with the old global signs at $S[\texttt{tid}_y]$. This approach introduces an additional $log(B_x)$ steps to jump between tree levels (l.13). Albeit, the number of atomic operations is reduced from $k \cdot |\mathcal{W}_d|$ to $k \cdot |\mathcal{W}_d|/B_x$.

Figure 6 depicts a running example of algorithm 2 executed on the window $\mathcal{W} = \{P_{120}, H_5, H_{10}, P_{100}\}$ with $(n = 128)$. For that, we require a tableau of dimensions $(k \cdot (2n+1) = 2 \cdot (256+1))$. As a result, the number of threads in the y-dimension is 2, and 4 in the x-dimension. In that case, one thread block for both dimensions is sufficient (e.g. $B_x = 4$, $B_y = 2$). Once the collapsing process is complete and threads are synchronized per block, the root of the tree, denoted as $\texttt{sh}[0]$, holds the final collapsed word. If multiple thread blocks are involved

in the x-dimension, their results are combined using an atomicXOR operation (as shown in l.17). In this specific example, as we only have one thread block in the x-dimension, `sh`[0] can be directly written to $S[\texttt{tid}_y]$ non-atomically.

3.4 Complexity Analysis

The overall complexity of algorithm 2 is influenced by two factors: the number of words per column k and the number of gates $|\mathcal{W}_d|$. Given that the upper bound of both factors is the number of qubits n, the worst-case parallel complexity can be expressed as $\mathcal{O}(\frac{n}{N_y}\ \frac{n}{N_x} + \log(B_x))$. Here, $\log(B_x)$ indicates the number of sequential steps at l.13, while N_y and N_x represent the total threads launched in the y- and x-dimensions, respectively. Note that the upper bound of the logarithmic steps is limited to B_x, not n, since the maximum size of a thread block is 1,024 threads across all dimensions [23].

The circuit depth (denoted by d) also impacts the EC performance overall. As the depth increases and more gates become causally dependent, fewer gates can be executed in parallel, leading to more sequential tableau updates (cf. the loop at l.12 in algorithm 1). Thus, the total time complexity considering the circuit depth becomes $\mathcal{O}(d \cdot \frac{n}{N_y}\ \frac{n}{N_x} + \log(B_x))$. For shallow circuits with highly parallelizable gates, the runtime is significantly reduced, while deeper circuits with more dependencies between gates lead to a closer-to-linear scaling in depth.

4 Experimental Evaluation

Setup. We implemented algorithm 1 in a new tool called QuaSARQ using CUDA C++. The code was compiled with CUDA 12.4 targeting compute capability 8.9. The GPU experiments were conducted on a machine running Ubuntu 22.04, equipped with an RTX 4090 GPU, featuring 16,384 cores at 2.23 GHz and 24GB of global memory. We compare QuaSARQ with CCEC,[5] which is based on Stim—the state-of-the-art simulator for stabilizer circuits. Additionally, we evaluate our tool against ECMC,[6] which employs the GPMC [14] model counter[7] and supports equivalence checking for both Clifford and universal circuits using theorem 1. CCEC and ECMC experiments are conducted separately on the compute nodes of DAS-6 [3] cluster to dedicate our computing hours on the GPU machine to QuaSARQ experiments. Each node of DAS-6 had an AMD EPYC 7282 CPU (2.8 GHz) with 256GB of memory. Circuits were tested in isolation on a separate computing node, with a timeout of 5,000 seconds. It is worth mentioning that DAS-6's CPU clock is 20% faster than the GPU clock running QuaSARQ.

Since our EC algorithm is deterministic, runtimes can be extrapolated based solely on gate counts. Therefore, evaluating the algorithm on a diverse set of

[5] https://github.com/System-Verification-Lab/CCEC
[6] https://github.com/System-Verification-Lab/Quokka-Sharp [21]
[7] We attempted to use gpusat2, but it failed on all circuits due to memory overflow.

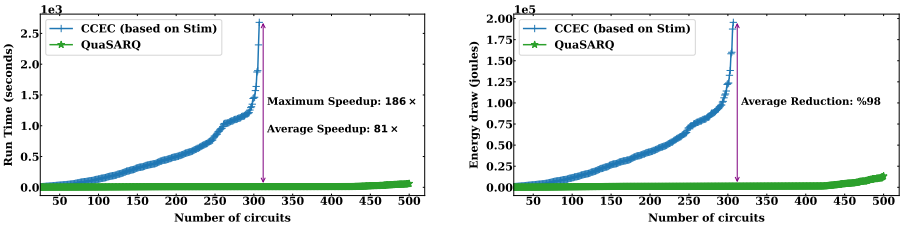


Fig. 7: EC Performance of QuaSARQ against CCEC across 500 circuits (qubits count: 1K–500K, circuit depth: 100)

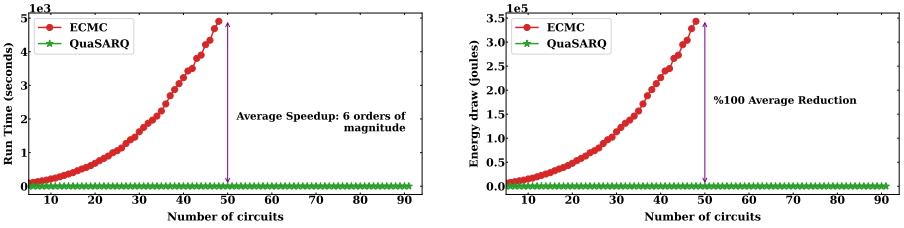


Fig. 8: EC Performance of QuaSARQ against ECMC across 91 circuits (qubits count: 1K–10K, circuit depth: 1)

quantum algorithms is unnecessary; testing with randomly generated circuits suffices. To evaluate our algorithm, we generated 500 random circuits using Qiskit[8] in OpenQASM format [5]. The number of qubits in these circuits ranges from 1K to 500K, increasing in steps of 1K, with a fixed depth of 100. We chose this value to test the effectiveness of our algorithms in circuits of moderate depth. The total number of gates ranges from 68K to 34M. By default, Qiskit generates uniformly distributed gates in the Clifford group $\{X, Y, Z, H, P, P^\dagger, CX, CY, CZ, SWAP, ISWAP\}$, which are supported by our tool and Stim. These gates are common Clifford gates and they can be decomposed into a combination of the basic gates $\{H, P, CX\}$. To simulate erroneous circuits for EC, we randomly selected a gate in each circuit and replaced it with another randomly chosen gate in the Clifford group. Thus, there is a $\frac{1}{11}$ chance of the gate remaining unchanged.

To complement these benchmarks and evaluate ECMC's performance against QuaSARQ under reasonable computational resources, we generated a smaller set of 91 circuits. In this reduced dataset, the number of qubits was scaled down to 10K, and the circuit depth was limited to a single level, ensuring the tests could be performed within practical time and memory limits.

Experiments. Figures 8a and 8b show cactus plots of the runtime and the energy consumption of CCEC vs. QuaSARQ, respectively. Similarly, figures 8a

[8] https://www.ibm.com/quantum/qiskit

Table 1: Statistics for a selection of ten circuits.

Qubits	Initial time (s)	Schedule time (s)	Simulation time (s)	Energy draw (J)	Tableau blocks	Tableau memory (GB)	Circuit memory (GB)	Clifford gates
50,000	0.18	0.12	0.31	24	1	1.17	0.05	3,437,182
100,000	0.34	0.23	2.38	542	1	4.66	0.10	6,874,333
150,000	0.58	0.36	2.59	589	1	10.48	0.14	10,311,694
200,000	0.82	0.48	4.64	1068	1	18.63	0.19	13,749,773
250,000	0.99	0.59	4.85	1114	2	19.41	0.23	17,186,737
300,000	1.17	0.70	4.70	1074	3	18.62	0.28	20,624,343
350,000	1.37	0.82	4.32	983	4	16.91	0.32	24,063,024
400,000	1.50	0.95	5.64	1280	4	22.08	0.37	27,499,531
450,000	1.69	1.10	4.76	1092	5	18.87	0.41	30,937,687
500,000	1.84	1.20	30.24	6900	6	19.42	0.46	34,375,212

and 8b compares QuaSARQ to ECMC. QuaSARQ's time includes the initial time (memory allocations, parsing, etc.), the scheduling time to obtain $\mathcal{R}$ and $\mathcal{R}'$, the transfer time to send $\mathcal{R}$ to the device, and the checking time of algorithm 1. To measure the energy draw, we multiplied the runtime to the average power consumed by the processing unit over the life cycle of both applications. Overall, QuaSARQ achieves an average speedup of 81×, and saves energy by 98% compared to CCEC. Against ECMC, QuaSARQ delivers an acceleration of six orders of magnitude for EC of stabilizer circuits. This is not surprising, as ECMC targets universal quantum circuits. The results highlight the effectiveness of QuaSARQ in achieving unparalleled speedups for smaller circuits with shallow depth, consistent with our complexity analysis in section 3.4.

Table 1 reports statistics for a selection of ten test cases. Evidently, the initial and scheduling times are negligible compared to the simulation time. Regarding tableau allocation, QuaSARQ, with its tableau slicing mechanism, occupies at most 22GB of memory regardless of the number of tableaux or qubits being simulated. For EC, two tableaux are required (one per circuit). Refer to *tableau memory* in Table 1 for precise numbers. Even when QuaSARQ supports measurements, 24GB is sufficient for the additional required tableau, the *destabilizer* tableu, albeit with more blocks. The destabilizer tableau is for keeping track of anti-commutators necessary for simulating measurements [1]. In contrast, Stim in CCEC and ECMC consumed up to 256GB of memory before eventually running out. This underscores QuaSARQ 's ability to reduce memory usage by up to 90%.

5 Conclusions and Future Work

We presented a parallel algorithm for equivalence checking of stabilizer circuits on GPUs and developed a new tool called QuaSARQ. Experimental results show that QuaSARQ significantly outperforms the current state of the art, with a 186× speedup compared to CCEC, which leverages Stim—a simulator that utilizes the SSE2/AVX2 vectorized instructions. To the best of our knowledge, this is the first GPU-based equivalence checker that surpasses CCEC in terms of time, memory, and energy.

Looking ahead, we aim to enable measurements on GPUs to support the full simulation of quantum stabilizer circuits.

References

1. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. Physical Review A **70**(5) (nov 2004). https://doi.org/10.1103/physreva.70.052328
2. Amy, M.: Towards Large-scale Functional Verification of Universal Quantum Circuits. Electronic Proceedings in Theoretical Computer Science **287**, 1–21 (Jan 2019). https://doi.org/10.4204/eptcs.287.1
3. Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., Wijshoff, H.: A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. IEEE Computer **49**(5), 54–63 (2016). https://doi.org/10.1109/MC.2016.127
4. Berent, L., Burgholzer, L., Wille, R.: Towards a SAT Encoding for Quantum Circuits: A Journey From Classical Circuits to Clifford Circuits and Beyond. In: Meel, K.S., Strichman, O. (eds.) SAT 2022. LIPIcs, vol. 236, pp. 18:1–18:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.SAT.2022.18
5. Cross, A.W., Bishop, L.S., Smolin, J.A., Gambetta, J.M.: Open Quantum Assembly Language (2017). https://doi.org/10.48550/arXiv.1707.03429
6. De Raedt, H., Jin, F., Willsch, D., Willsch, M., Yoshioka, N., Ito, N., Yuan, S., Michielsen, K.: Massively parallel quantum computer simulator, eleven years later. Computer Physics Communications **237**, 47–61 (2019). https://doi.org/https://doi.org/10.1016/j.cpc.2018.11.005
7. Fowler, A.G., Mariantoni, M., Martinis, J.M., Cleland, A.N.: Surface codes: Towards practical large-scale quantum computation. Phys. Rev. A **86**, 032324 (Sep 2012). https://doi.org/10.1103/PhysRevA.86.032324
8. Garcia, H.J., Markov, I.L.: Simulation of Quantum Circuits via Stabilizer Frames. IEEE Transactions on Computers **64**(08), 2323–2336 (aug 2015). https://doi.org/10.1109/TC.2014.2360532
9. Gidney, C.: Stim: a fast stabilizer circuit simulator. Quantum **5**, 497 (Jul 2021). https://doi.org/10.22331/q-2021-07-06-497
10. Gidney, C., Ekerå, M.: How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. Quantum **5**, 433 (Apr 2021). https://doi.org/10.22331/q-2021-04-15-433
11. Google Quantum AI: Suppressing quantum errors by scaling a surface code logical qubit. Nature **614**(7949), 676–681 (Feb 2023). https://doi.org/10.1038/s41586-022-05434-1
12. Gottesman, D.: Stabilizer Codes and Quantum Error Correction. arXiv:quant-ph/9705052 (1997). https://doi.org/10.48550/arXiv.quant-ph/9705052
13. Harris, D.: Sustainable Strides: How AI and Accelerated Computing Are Driving Energy Efficiency (2024), https://blogs.nvidia.com/blog/accelerated-ai-energy-efficiency, accessed on July 31, 2024
14. Hashimoto, K.: GPMC (2020), https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/GPMC
15. Hein, M., Dür, W., Eisert, J., Raussendorf, R., den Nest, M.V., Briegel, H.J.: Entanglement in Graph States and its Applications (2006). https://doi.org/10.48550/arXiv.quant-ph/0602096

16. Javadi-Abhari, A., Treinish, M., Krsulich, K., Wood, C.J., Lishman, J., Gacon, J., Martiel, S., Nation, P.D., Bishop, L.S., Cross, A.W., Johnson, B.R., Gambetta, J.M.: Quantum computing with Qiskit (2024). https://doi.org/10.48550/arXiv.2405.08810
17. Kitaev, A.Y.: Quantum computations: algorithms and error correction. Russian Mathematical Surveys **52**(6), 1191 (dec 1997). https://doi.org/10.1070/RM1997v052n06ABEH002155
18. Litinski, D.: A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery (Mar 2019). https://doi.org/10.22331/q-2019-03-05-128
19. Marques, J.F., Varbanov, B.M., Moreira, M.S., Ali, H., Muthusubramanian, N., Zachariadis, C., Battistel, F., Beekman, M., Haider, N., Vlothuizen, W., Bruno, A., Terhal, B.M., DiCarlo, L.: Logical-qubit operations in an error-detecting surface code. Nature Physics **18**(1), 80–86 (Dec 2021). https://doi.org/10.1038/s41567-021-01423-9
20. McEwen, M., Bacon, D., Gidney, C.: Relaxing Hardware Requirements for Surface Code Circuits using Time-dynamics. Quantum **7**, 1172 (Nov 2023). https://doi.org/10.22331/q-2023-11-07-1172
21. Mei, J., Coopmans, T., Bonsangue, M., Laarman, A.: Equivalence Checking of Quantum Circuits by Model Counting. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds.) Automated Reasoning. pp. 401–421. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-63501-4_21
22. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press (2010)
23. NVIDIA: CUDA C Programming Guide (2024), https://docs.nvidia.com/cuda/cuda-c-programming-guide
24. Obenland, K.M., Despain, A.M.: A Parallel Quantum Computer Simulator (1998). https://doi.org/10.48550/arXiv.quant-ph/9804039
25. Preskill, J.: Quantum Computing in the NISQ era and beyond. Quantum **2**, 79 (Aug 2018). https://doi.org/10.22331/q-2018-08-06-79
26. Sistla, M., Chaudhuri, S., Reps, T.: Symbolic quantum simulation with quasimodo. In: Enea, C., Lal, A. (eds.) Computer Aided Verification. pp. 213–225. Springer Nature Switzerland, Cham (2023)
27. Sutcliffe, M., Kissinger, A.: Fast classical simulation of quantum circuits via parametric rewriting in the ZX-calculus (2024)
28. Terhal, B.M.: Quantum error correction for quantum memories. Rev. Mod. Phys. **87**, 307–346 (Apr 2015). https://doi.org/10.1103/RevModPhys.87.307
29. Thanos, D., Coopmans, T., Laarman, A.: Fast Equivalence Checking of Quantum Circuits of Clifford Gates. In: André, É., Sun, J. (eds.) Automated Technology for Verification and Analysis. pp. 199–216. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-45332-8_10

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

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

SliQSim: A Quantum Circuit Simulator and Solver for Probability and Statistics Queries

Tian-Fu Chen[1,4,5] and Jie-Hong R. Jiang[1,2,3,4,5](✉)

[1] Graduate School of Advanced Technology, National Taiwan University, Taipei, Taiwan
[2] Graduate Inst. of Electronics Engineering, National Taiwan University, Taipei, Taiwan
[3] Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
[4] Center for Quantum Science and Engineering, National Taiwan University, Taipei, Taiwan
[5] Physics Division, National Center for Theoretical Sciences, Taipei, Taiwan
{d11k42001,jhjiang}@ntu.edu.tw

Abstract. `SliQSim`, originally developed as the first exact quantum circuit simulator, is extended in this paper to provide capabilities for the analysis and verification of quantum states. It provides an interface for users to specify interested quantum states for querying the exact probability or expectation value of a user-defined property. Case studies on three quantum algorithms show the unique capability of `SliQSim` benefiting exact quantum circuit analysis and verification, beyond the support by other tools.

1 Introduction

Quantum computation shows promise in its advantage of solving certain important problems, which are classically hard. Simulation and verification of quantum circuits play a crucial role in quantum program compilation, ensuring quantum algorithms are correctly executed. To date, many classical simulators have been proposed to efficiently simulate given quantum circuits, such as decision-diagram-based `SliQSim` [19], `DDSim` [25], `Quasimodo` [17], and `MEDUSA` [5], ZX-calculus-based `QuiZX` [12], tensor-network-based `TensorCircuit` [24], and model-counting-based [14] methods. To the best of our knowledge, none of the existing simulators allow users to query the probability for a customizable set of quantum states.

Current simulation tools mainly provide two kinds of interfaces for returning measurement outcomes. The first one is providing the whole quantum state information, i.e., the full spectrum of probability distribution. In this case, to know the probability corresponding to the set of states satisfying certain properties, the user has to iterate over and sum up the probabilities of all states under the computational basis that satisfies the properties. However, this brute-force method is barely scalable due to the exponential quantum state blow-up in the

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 129–138, 2025.
https://doi.org/10.1007/978-3-031-90660-2_7

qubit number. The second one is providing random sampling results of a specified number of shots. However, the Monte-Carlo-based methods are far from exact and may lead to incorrect conclusions when one requires a precise answer.

For example, given a Grover's circuit with a certain search criterion, to verify the formula of amplitude amplification in Grover's algorithm, we may be interested in the probability that the measurement result satisfies the specified search criterion after certain iterations. Although current simulation tools may finish simulating the circuit fast, there is no way to quickly extract such probability from the simulation result. With the first interface, one may need to enumerate through an exponential number of quantum states for proper computation of the interested probability or expectation value. On the other hand, with the second interface, the precision of the estimated probability is sensitive to the shot number and can be far from exact. As to be demonstrated later in Example 1, the precision loss can severely affect the property checking results. Therefore, an exact and efficient property query interface is necessary.

In this work, we extend the state-of-the-art quantum circuit simulator `SliQSim` [19], which achieves exact complex number representation without numerical precision loss via algebraic methods and high computation efficiency by partitioned BDD representation and manipulation for implicit matrix-vector multiplication, to further support effective and exact queries about probabilities or expectation values of user-specified properties. Utilizing the Boolean-based data structure, `SliQSim` can easily mask out the computational components that do not satisfy a property, and the probabilities of the remaining parts are then summed over and reported. Our tool allows users to specify any set of quantum states for exact probability and expectation value queries of customizable properties. The provided query functions allow users to inspect the simulation results from different perspectives and verify the functionality of quantum algorithms.

We note that there are prior efforts on quantum circuit verification. However, they target different goals from ours. For instance, some verifiers focus on checking the equivalence or similarity between two quantum circuits or their output states [1,4,7,20], Hoare-logic style reasoning over program execution [8], quantum abstract interpretation [23], first-order specifications of quantum programs [2], model-checking quantum automata or quantum Markov chains [22], etc. These verification tools do not handle our considered tasks.

2 Tool Architecture

`SliQSim` [19] exploits an algebraic representation to exactly specify an n-qubit quantum state $|\psi\rangle = [\alpha_0, \alpha_1, \ldots, \alpha_{2^n-1}]^{\mathrm{T}}$ without any numerical precision loss as follows. Each probability amplitude, a complex number α_i, in the state vector is represented by $\alpha_i = \frac{1}{\sqrt{2}^k}(a_i\omega^3 + b_i\omega^2 + c_i\omega + d_i)$, where $\omega = e^{i\pi/4}$, k is an integer shared by all α_i's, and a_i, b_i, c_i, and d_i are each an r-bit integer. Then $|\psi\rangle$ can be represented by $4r$ n-variable Boolean functions, and each Boolean function can be represented by a standard reduced ordered binary decision diagram (ROBDD) [3]. We denote these $4r$ BDDs by F_j^a, F_j^b, F_j^c, and F_j^d for $j = 1, 2, \ldots, r$. The

functionality of the $4r$ ROBDDs is defined by letting the truth value of F_j^x under binary assignments of the index t be equal to the j^{th} bit of x_t, $x \in \{a, b, c, d\}$. Therefore, the value of α_t can be recovered from the $4r$ ROBDDs by substituting the n-qubit variables with the binary values of index $t \in \{0, 1, \ldots, 2^n - 1\}$. In such a data structure, gate applications can be done by Boolean operations on the BDDs.

We extend `SliQSim` as shown in Figure 1. First, a circuit file for simulation or analysis is taken (❶). `SliQSim` will calculate the output state of the circuit and, depending on users' requirements, do random samplings or return the output state vector (❷), which are the functions of the original `SliQSim`. After that, the newly equipped verifier core will read the list of user-defined properties (❸). In the verifier core, each property will be translated into a Boolean function F^p, where $F^p = 1$ indicates that the property is satisfied. This property function F^p is sent to the Boolean function handler (❹), which will conduct the conjunction of F^p and the Boolean functions that represent the output state (i.e., F_j^x's). In other words, we will obtain a "pseudo-state" copied from the output state, but for the computation bases that do not satisfy the property, their probability amplitudes are equivalently set to 0. After the pseudo-state is obtained and returned, the verifier will send the pseudo-state to the probability calculator (❺), which will calculate the total probability of the state. Finally, the probability or truth value of each property will be returned, depending on the types of properties (❻).

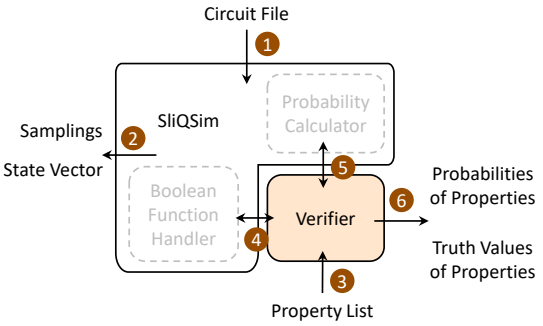


Fig. 1. Tool architecture of `SliQSim`.

To understand why `SliQSim` can easily include the probability query function, we note that `SliQSim` takes advantage of representing quantum states as Boolean formulas. This allows it to filter out components that do not meet the required properties by a simple conjunction with the property's function. Therefore, our framework can be easily adapted for simulators that use decision diagrams or Boolean functions as their underlying data structures, such as `DDSim` [25], `Quasimodo` [17], and `MEDUSA` [5].

We note that a similar idea is mentioned in `Quasimodo` [17], which claims to have the potential capability of probability query. However, it does not implement an interface that accepts user-defined functions, while `SliQSim` provides various kinds of query functions and can automatically transform them into corresponding Boolean formulas. More importantly, even if `Quasimodo` implements the probability query function, it suffers from the numerical precision loss problem and cannot achieve exact analysis as `SliQSim` does.

3 Features and Demo

`SliQSim` supports various types of queries for user-defined properties. The users can use the expressions listed below to specify their desired properties.

1. `bf {formula}`: Returns the probability that the measurement result satisfies the specified Boolean formula `formula`. We notice that any state set can be generally characterized by a Boolean formula, so solely the `bf` function is complete for arbitrary state-set specification. Moreover, we provide some common state-set expressions below (Expressions 2 and 3) for users' convenience.
2. `hweq/hwneq/hwgt/hwlt {qubits} {num}`: Returns the probability that the Hamming weight of the measured bit-string of the specified qubits `qubits` is equal to, nonequal to, greater than, lower than the specified number `num`.
3. `inteq/intneq/intgt/intlt {qubits} {num}`: Returns the probability that the binary integer represented by the specified qubits `qubits` is equal to, non-equal to, greater than, lower than the specified number `num`.
4. `expt {qubits} {Pauli(-value)_string}`: Returns the expectation value of the observable defined by the specified Pauli string with optionally specified measured-values (0 or 1) for each individual Pauli operator over the specified qubits `qubits`, commonly used in variational quantum circuits [11,10,21] and wire-cut circuits [15,18]. Specifically, the returned value is calculated by the product of the probability of obtaining the specified measured values and the expectation value of the Pauli operators without measured values with respect to the post-measurement state.
5. `weightedsum {weight_1} {expression_1} ... {weight_k} {expression_k} endweighteedsum`, for some $k \geq 1$: Returns the weighted sum of the query results of the specified expressions. This query statement is commonly used in variational quantum circuits [11,10,21]. Each weight `weight` and its corresponding expression `expression` are stated in an independent line, and the lines are clipped by keywords `weightedsum` and `endweightedsum`. The expression can be `bf`, `hweq`, `hwneq`, `hwgt`, `hwlt`, `inteq`, `intneq`, `intgt`, `intlt`, or `expt`.
6. `assign {var_name} {expression}`: Does not return values, but rather stores the specified expression `expression` in the specified variable name `var_name`, which can be further utilized in the `bf` expression. The expression can be `bf`, `hweq`, `hwneq`, `hwgt`, `hwlt`, `inteq`, `intneq`, `intgt`, or `intlt`.
7. `between/outof/leq/geq {range or threshold}`: Is specified before the expressions: `bf`, `hweq`, `hwneq`, `hwgt`, `hwlt`, `inteq`, `intneq`, `intgt`, `intlt`, `expt`, `weightedsum`. The predicate returns true or false according to whether or not the probability returned by its subsequent expression function is between, out of, less than or equal to, or greater than or equal to the specified range or threshold.
8. `amp {compt_basis}`: Returns the probability amplitude (as a complex number) of the specified computational basis `compt_basis`.
9. `dist {qubits}`: Returns the exact spectrum of the probability distribution upon measuring the specified qubits `qubits`.

We notice that `expt` is a special case since it is not a "property" that directly corresponds to truth values. To get the expectation value of a Pauli string over some qubits, we first convert the Pauli string to computation basis (i.e., I and Z) by updating the quantum state according to the equalities $Y = (SH)Z(HS^\dagger)$ and $X = HZH$. Then we define the property P_a to be the XNOR operation over negations of all qubits whose Pauli string correspondence is the expectation value of Z, and the property P_b to be the AND operation over qubits (resp. negations of qubits) whose Pauli string correspondence is obtaining 1 (resp. 0) on Z basis. Then the expectation value of the original string is $\textsc{Prob}(P_b \wedge P_a) - \textsc{Prob}(P_b \wedge \neg P_a)$, where $\textsc{Prob}(P)$ is the probability of P.

To demonstrate the usage and application of our tool, we give three examples as follows.

Example 1 *Consider an m-bit oracle that returns true if and only if there are w non-zero bits among the m input bits. We may use Grover's algorithm [9] to search for satisfying assignments of the oracle. Let p be the probability of obtaining a satisfying assignment after one Grover iteration. Then p should follow the relation $p = \sin^2(3 \cdot \arcsin\sqrt{t/2^m})$, where t is the number of satisfying assignments. We can use a simulator to verify this point. For example, let $m = 10, w = 2$. Then, we construct a Grover's algorithm circuit with one Grover iteration and specify the desired property as*

```
hweq q[0] q[1] q[2] q[3] q[4] q[5] q[6] q[7] q[8] q[9] 2.
```

Then `SliQSim` helps us to calculate the probability $p = 0.350517$. (Note that the real-number output of `SliQSim` is represented in an (inexact) floating-point number converted from the exact algebraic representation in the last step.) Accordingly, we can infer that there are $t = 2^m * \sin^2((\arcsin\sqrt{p})/3) = 45$ satisfying assignments, which equals the expected result $\binom{10}{2}$.

We notice that the exact value of the probability is important in this example application. For instance, if there is a 2%-error to mistake p as 0.345, then the inferred t would become 44, which is incorrect.

Example 2 *Suppose an oracle $U_f : |x\rangle|0\rangle \mapsto |x\rangle|f(x)\rangle$ is designed for Simon's algorithm [16]. To be able to apply Simon's algorithm, f should be a two-to-one function satisfying $f(x) = f(y)$ if and only if $x = y$ or $x = y \oplus s$, where s is a secret key. Let t be the measurement outcome of Simon's algorithm circuit. Then it is promised that*

$$t \cdot s \bmod 2 = 0 \tag{1}$$

We can use a simulator to verify this point. For example, let $m = 5$ be the variable number of f and $s = 11001$. Then we construct a Simon's algorithm circuit and specify the desired property as

```
bf q[0]^q[1]^q[4].
```

Then `SliQSim` can help us to calculate the probability p that $t \cdot s \bmod 2 \neq 0$, which should equal 0.

However, if due to a compilation error, the oracle does not satisfy the two-to-one constraint for two input assignments, then Eq. (1) will not always hold. To be precise, the probability p would be $p = 1/2^m$. In this case, `SliQSim` can still detect the small probability $p = 0.03125$ for $m = 5$ in the above example.

In this example, the error in the oracle is so small that the random sampling method can hardly detect the small probability. Averagely, $1/p = 2^m$ number of samplings are expected to detect such error, which is too large to be practical as m grows. In contrast, `SliQSim` provides an efficient way to obtain the exact value of p.

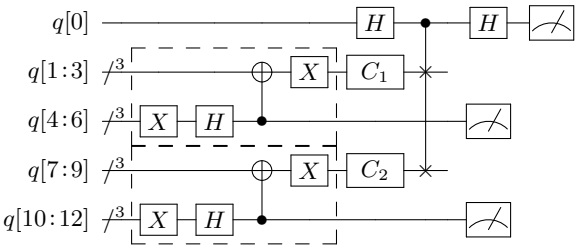


Fig. 2. The circuit of N-I equivalence checking with delayed initialization.

Example 3 *The N-I equivalence Boolean matching problem [6], which asks whether two given reversible circuits can be equivalent under some input negation conditions, has recently been shown tractable by quantum computation, but intractable by classical computation [6]. The reader is referred to [6] for the detailed algorithm on finding the input negation condition that makes two oracle circuits C_1 and C_2 equivalent. Figure 2 shows the quantum circuit for deciding the N-I equivalence between C_1 and C_2. The circuit is equipped with the ancilla-assisted initialization technique [13] (shown in the dashed boxes with ancilla qubits $q[4:6]$ and $q[10:12]$), so that the initialization of $q[i]$ to $|0\rangle$ (resp. $|+\rangle$) is delayed and transformed into letting $q[i+3]$ collapse to $|1\rangle$ on Z-basis (resp. $|-\rangle$ on X-basis), for $i \in \{1, 2, 3, 7, 8, 9\}$. (This will introduce a probability factor of constant 0.5 for each i as side effects.)*

For m-bit oracles ($m = 3$ here), it needs to iterate through m types of measurements on $q[0], q[4:6], q[10:12]$. For the i^{th} iteration, we compute the probability of obtaining $|0\rangle$ on $q[0]$, $|1\rangle$ on $q[i]$ and $q[6+i]$, and $|-\rangle$ on other qubits in $q[4:6]$ and $q[10:12]$. This can be specified as checking properties as follows.

```
expt q[0] q[4] q[5] q[6] q[10] q[11] q[12] z0z1x1x1z1x1x1
expt q[0] q[4] q[5] q[6] q[10] q[11] q[12] z0x1z1x1x1z1x1
expt q[0] q[4] q[5] q[6] q[10] q[11] q[12] z0x1x1z1x1x1z1
```

If the i^{th} qubits of oracles are matched without a negation, the i^{th} resulting probability should be $1/2^{2m}$. Otherwise, the i^{th} resulting probability should be $1/2^{2m+1}$.

We notice that the above example conducts multiple property queries on a single circuit, and it requires the capability of X-basis measurements, which can be easily handled by `SliQSim`. Otherwise, one may need to execute the circuit many

times and manually modify the circuit to translate X-basis measurements to computational bases for each execution. Moreover, the exactness of `SliQSim` helps distinguish the subtle difference between $1/2^{2m}$ and $1/2^{2m+1}$.

4 Case Studies

We performed empirical case studies on the examples mentioned in Section 3 for demonstration. All experiments were conducted on a server with Intel Core i7-8700 CPU at 3.20GHz and 32 GB RAM. The timeout (TO) limit is set to 600 seconds. Note that, because there are no other tools that can handle our considered verification tasks, only the results of `SliQSim` were reported.

4.1 Grover's Algorithm

We generated Grover's algorithm circuits with different numbers of qubits. The bit number m of the oracle ranges from $\{20, 40, 60\}$, and the search criterion is set to have $m/10$ non-zero bits among the m input bits. The qubit number of the circuit is set to $n = m + \lceil \log_2 m \rceil + 1$. We intend to simulate the circuit to get the probability of obtaining a satisfying assignment after one Grover iteration, which is utilized to infer the number of satisfying assignments of the oracle.

Table 1. Results on Grover's circuits.

m	cnt	SliQSim		10^5 shots		vector	
		cnt	time	cnt	time	cnt	time
20	190	190	0.080	173	21.870	TO	
30	4060	4060	0.348	1193	32.647	TO	
40	91390	91390	1.434	0	43.034	TO	

Table 1 shows the results. In the first column, "m" denotes the bit number of the oracle. In the second column, "cnt" denotes the number of satisfying assignments. The following columns show the result from different methods: using `SliQSim` for exact query, using SliQSim with 10^5 random samplings for estimation, or using SliQSim to report the whole state vector. For each method, the "cnt" column shows the number of satisfying assignments inferred from the calculated exact or estimated approximate probability. The runtime in seconds is also reported.

As one can see in Table 1, `SliQSim` is precise enough to always infer the correct number of satisfying assignments. In contrast, the results estimated from random samplings suffer from precision loss and have longer runtime. This situation becomes more significant as the circuit size grows. For $m = 40$ and 60, random sampling with 10^5 shots cannot even detect the small probability, and the runtime is much longer than `SliQSim`. As for the interface that provides the whole quantum state information, it fails to finish any circuit because there are tens of qubits, leading to millions of entries in the state vector.

4.2 Simon's Algorithm

Table 2. Results on Simon's circuits.

m	SliQSim		10^5 shots		vector	
	prob	time	prob	time	prob	time
10	9.77E-04	0.017	1.02E-03	13.663	9.77E-04	469.850
15	3.05E-05	0.034	2.00E-05	28.436	TO	
20	9.54E-07	0.066	0	37.255	TO	

We design Simon's algorithm circuits with different numbers of qubits. The variable number m of the oracle function ranges from $\{10, 15, 20\}$, and the erroneous oracle has exactly two input assignments that do not satisfy the two-to-one constraint. The qubit number of the circuit is $n = 2m$. We then simulate the circuit to check if the simulator can find the probability of violating Eq. (1).

Table 2 shows the results. In the first column, "m" denotes the variable number of the oracle, and the following columns show the results from different methods. For each method, the "prob" column shows the calculated exact or estimated approximate probability of violating Eq. (1), and the "time" column reports the runtime in seconds.

As one can see in Table 2, SliQSim can always detect the error of the oracle by reporting the small probability of violating Eq. (1). In contrast, a lot of sampling shots are required to detect the small probability, and the runtime is also longer. For $m = 20$, 10^5 shots are not even enough while taking a much longer runtime. As for the interface that provides the whole quantum state information, although it can report the exact probability, it is not scalable enough to finish $m \geq 15$ in the time limit, showing the infeasibility of brute-force methods.

4.3 N-I Equivalence Boolean Matching

Table 3. Results on N-I equivalence checking circuits.

m	n	SliQSim time	1-shot time
3	13	0.035	0.015
4	17	0.334	0.078
5	21	3.355	0.435
6	25	38.489	1.447
7	29	336.656	10.578

We generate N-I equivalence-checking circuits with different numbers of qubits. The input number of the oracle function m ranges from 3 to 7, and the qubit number of the whole circuit is $n = 4m+1$. We generate the first oracle as a random logic circuit with $5m$ gates, and the second oracle is produced by adding random negation gates at the input. We then simulate the circuit to find the input negation gates required to make the oracles equivalent.

The experiment results are shown in Table 3. In the first column, "m" denotes the variable number of the oracle, and in the second column, "n" denotes the qubit number of the whole circuit. The third column reports the runtime of SliQSim querying the m properties, while the fourth column reports the runtime of SliQSim obtaining one random sampling result in seconds for reference. Here, we omit to compare with the interface that provides the whole quantum state information because X-basis measurement is required in this algorithm, so summing up probability amplitudes in a state vector does not work.

Although `SliQSim` seems to take much longer time than the one-shot simulation, we notice that more than 2^m shots are required to achieve enough precision to decide whether a negation gate exists, and a total of m different circuits are required to be sampled. Therefore, `SliQSim` is still much more efficient than sampling-based methods.

Acknowledgments

This work was supported in part by the National Science and Technology Council of Taiwan under grants 113-2119-M-002-024, the NTU Center of Data Intelligence: Technologies, Applications, and Systems under grant NTU-113L900903, and the NTU Core Consortium Project NTU-CC114L895002. The authors thank IBM Q Hub at NTU and Quantum Technology Cloud Computing Center at NCKU for supporting access to quantum devices.

Data Availability Statement

The source code and benchmarks used for the experiments are available at https://figshare.com/articles/software/SliQSim-TACAS2025_zip/27239637?file=49820907, and the `SliQSim` project is publicly available at https://github.com/NTU-ALComLab/SliQSim.

References

1. Amy, M.: Towards large-scale functional verification of universal quantum circuits. arXiv preprint arXiv:1805.06908 (2018)
2. Bauer-Marquart, F., Leue, S., Schilling, C.: symQV: Automated symbolic verification of quantum programs. In: Int. Symp. on Formal Methods. pp. 181–198. Springer (2023)
3. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comput. **100**(8), 677–691 (1986)
4. Burgholzer, L., Wille, R.: Advanced equivalence checking for quantum circuits. IEEE Trans. on Comput.-Aided Design Integr. Circuits Syst. **40**(9), 1810–1824 (2021)
5. Chen, T.F., Chen, Y.F., Jiang, J.H.R., Jobranová, S., Lengál, O.: Accelerating quantum circuit simulation with symbolic execution and loop summarization. In: Proc. ICCAD (2024)
6. Chen, T.F., Jiang, J.H.R.: Boolean matching reversible circuits: Algorithm and complexity. In: Proc. DAC (2024)
7. Chen, T.F., Jiang, J.H.R., Hsieh, M.H.: Partial equivalence checking of quantum circuits. In: Proc. QCE. pp. 594–604 (2022)
8. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L.: AutoQ: An automata-based quantum circuit verifier. In: Proc. CAV. pp. 139–153. Springer (2023)
9. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proc. STOC. pp. 212–219 (1996)
10. Incudini, M., Tarocco, F., Mengoni, R., Di Pierro, A., Mandarino, A.: Computing graph edit distance on quantum devices. Quantum Mach. Intell. **4**(2), 24 (2022)

11. Kandala, A., Mezzacapo, A., Temme, K., Takita, M., Brink, M., Chow, J.M., Gambetta, J.M.: Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. Nature **549**(7671), 242–246 (2017)
12. Kissinger, A., van de Wetering, J.: Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. Quantum Sci. and Technol. **7**(4), 044001 (2022)
13. Li, X., Kulkarni, V., Chen, D.T., Guan, Q., Jiang, W., Xie, N., Xu, S., Chaudhary, V.: Efficient circuit wire cutting based on commuting groups. In: Proc. QCE (2024)
14. Mei, J., Bonsangue, M., Laarman, A.: Simulating quantum circuits by model counting. In: Proc. CAV. pp. 555–578. Springer (2024)
15. Peng, T., Harrow, A.W., Ozols, M., Wu, X.: Simulating large quantum circuits on a small quantum computer. Physical review letters **125**(15), 150504 (2020)
16. Simon, D.R.: On the power of quantum computation. SIAM J. on Comput. **26**(5), 1474–1483 (1997)
17. Sistla, M., Chaudhuri, S., Reps, T.: Symbolic quantum simulation with Quasimodo. In: Proc. CAV. pp. 213–225. Springer (2023)
18. Tang, W., Tomesh, T., Suchara, M., Larson, J., Martonosi, M.: CutQC: using small quantum computers for large quantum circuit evaluations. In: Proc. ASPLOS. pp. 473–486 (2021)
19. Tsai, Y.H., Jiang, J.H.R., Jhang, C.S.: Bit-slicing the Hilbert space: Scaling up accurate quantum circuit simulation. In: Proc. DAC. pp. 439–444 (2021)
20. Wei, C.Y., Tsai, Y.H., Jhang, C.S., Jiang, J.H.R.: Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In: Proc. DAC. pp. 523–528 (2022)
21. Wierichs, D., Gogolin, C., Kastoryano, M.: Avoiding local minima in variational quantum eigensolvers with the natural gradient optimizer. Physical Review Research **2**(4), 043246 (2020)
22. Ying, M., Feng, Y.: Model checking quantum systems: Principles and algorithms. Cambridge University Press (2021)
23. Yu, N., Palsberg, J.: Quantum abstract interpretation. In: Proc. PLDI. pp. 542–558 (2021)
24. Zhang, S.X., Allcock, J., Wan, Z.Q., Liu, S., Sun, J., Yu, H., Yang, X.H., Qiu, J., Ye, Z., Chen, Y.Q., et al.: TensorCircuit: a quantum software framework for the NISQ era. Quantum **7**, 912 (2023)
25. Zulehner, A., Wille, R.: Advanced simulation of quantum computations. IEEE Trans. on Comput.-Aided Des. of Integr. Circuits and Syst. **38**(5), 848–859 (2018)

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

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

GPUExplore^{PROB}: Markov Chain State Space Construction and Verification with GPUs

Jan Heemstra and Anton Wijs(✉)

Eindhoven University of Technology, Eindhoven, The Netherlands
{j.h.heemstra,a.j.wijs}@tue.nl

Abstract. GPUExplore^{PROB} is an extension of GPUExplore that constructs state spaces of Markov Chains and performs probabilistic model checking entirely on a GPU. It can construct the state space of a Discrete-Time Markov Chain and verify that it satisfies a given Probabilistic Computation-Tree Logic formula. We present the tool, and experimentally compare with Storm, demonstrating its effectiveness.

1 Introduction

Graphics Processing Units (GPUs) have great potential to accelerate model checking, both explicit-state [8,10,24,26,29,30,35] and symbolic, based on SAT solving [25]. Initially, GPUs were used to accelerate parts of the model checking procedure, such as successor generation [12], property checking using a given in-memory state space [3], and counter-example construction [34], but soon, tools were developed to perform model checking entirely on a GPU [4,8,10,27,31–33,35]. One of these tools, GPUExplore, has been actively developed since 2013, and was recently equipped with an algorithm to verify Linear-Time Temporal Logic formulae [24]. In this paper, we present GPUExplore^{PROB}, an extension of GPUExplore aimed at *probabilistic* model checking (PMC) [2]. The tool is available on Zenodo [14].

In the earliest work on GPU accelerated model checking, PMC was already targeted, since the matrix-vector multiplications required to verify probabilistic properties can be ideally performed by GPUs [5,6,19,28]. However, *state space exploration*, i.e., the identification of all states reachable from a defined initial state, is much harder to parallelise, and formed the bottleneck of the computation, as it was still performed by a single CPU thread. Once the state space was explored, the resulting transition matrix, storing all the transitions of the state space, was copied to the GPU device memory, after which the multiplications could be performed. Using the experience gained in parallelising state space exploration for GPUExplore [29–31,33], we have developed the first GPU tool that constructs the state spaces of Markov Chains and performs explicit-state PMC *entirely* on a GPU. To verify probabilistic systems, GPUExplore^{PROB} currently supports checking whether a *Discrete-Time Markov Chain* [22] (DTMC) satisfies

This work is supported by NWO grant OCENW.M.21.061 for the GAP project.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 139–147, 2025.
https://doi.org/10.1007/978-3-031-90660-2_8

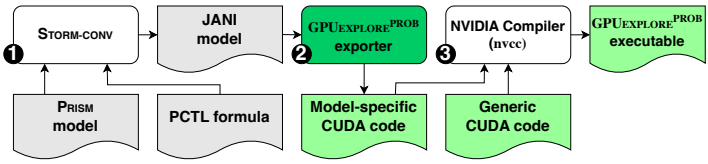


Fig. 1: Workflow from model to executable.

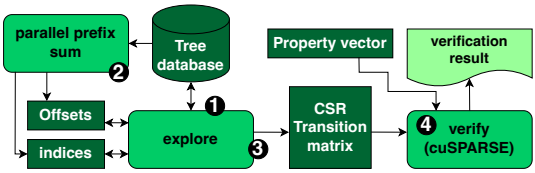


Fig. 2: Workflow from state space construction to property checking.

a *Probabilistic Computation-Tree Logic* (PCTL) formula [2]. For each state in a DTMC, a single probability distribution is defined over its successor states, i.e., the probabilities of its outgoing transitions add up to 1. Support for other model types, such as Continuous-Time Markov Chains (CTMCs) and Markov Decision Processes (MDPs), is planned for future work. These types are more expressive, but offer similar parallelisation challenges, hence GPUexplore^PROB currently already demonstrates the effectiveness of GPU accelerated PMC in general.

2 Construction and Verification of DTMCs

From model to transition matrix. GPUexplore^PROB has been developed to run on NVIDIA GPUs with at least Compute Capability 7.0, and is implemented in CUDA C++. It currently accepts DTMC models in the JANI format [7]. Prism [20] models, together with a PCTL formula, can first be converted to JANI, using the Storm-conv tool of the Storm model checker [11, 15] (see step 1 in Fig. 1). A JANI model is given to the GPUexplore^PROB exporter tool, written in Python, that produces CUDA C++ code necessary to identify all the system states reachable from the specified initial state of the model (step 2). In particular, the exporter generates CUDA functions that accept a state as input, and, when executed, produce successor states of the given state, in accordance with the model, as output. Instead of interpreting the model, GPUexplore^PROB therefore *executes* the model to create new states. The code produced by the exporter can be combined with GPUexplore^PROB generic code, which implements the main data structures, such as hash table, vector and matrix, and CUDA functions that are common for all input models, such as finding and storing states in the hash table. The combined code can be compiled using NVIDIA's nvcc compiler (step 3) resulting in an executable, specific for the input model.

Once an executable has been obtained, state space construction and verification can commence. Fig. 2 shows the workflow of this, and Fig. 3 visualises how GPUexplore^PROB runs on a GPU, in particular which data is stored in which

type of memory. First, an exploration kernel, i.e., CUDA function, is executed repeatedly to identify, construct and store all states reachable from the initial state (step 1 in Fig. 2). We assume that there is a finite number of reachable states. The explore operation is performed in a massively parallel way by the GPU threads. These threads run in *blocks* of 512 threads on *Streaming Multiprocessors* (SM), see Fig. 3. By default, 3,240 of these blocks are running, with the SMs interleaving between the blocks. Each block has some fast, on-chip memory, called *shared memory*, which is used as a state cache to store the states that will be explored by the block, and the states constructed during the exploration. Once all the states in this cache scheduled for exploration have been processed, the newly constructed states are stored in a large global *tree database*, which is a hash table residing in the GPU device memory, unless they are already there. A newly stored state can immediately be claimed for exploration by the thread block storing it, but, as there is a predefined limit to the number of states a block can explore in parallel, in practice, often many states will be left to be picked up by other blocks. This ensures that in many cases, depending on the state space structure, many blocks will perform work simultaneously. An important feature of GPUEXPLORE^{PROB}, new w.r.t. GPUEXPLORE, is that it uses NVIDIA's *unified memory*. This allows combining CPU memory with device memory into one address space. Because of this feature, GPUEXPLORE^{PROB} can use more of the available device memory for its tree database, and hence explore larger state spaces. After the exploration stage, the tree database can be moved to CPU memory to make room for the transition matrix while still remaining accessible.

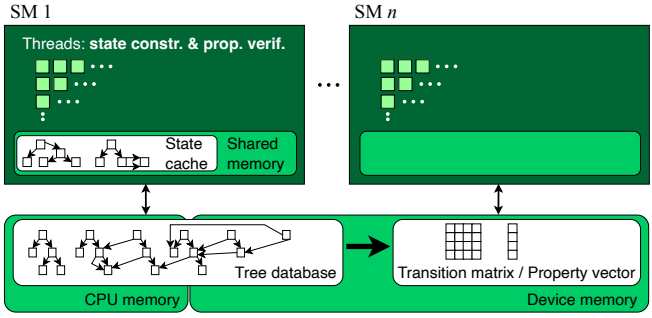


Fig. 3: GPUEXPLORE^{PROB} running on a GPU.

To make better use of the relatively small amount of device memory, states are stored as *binary trees*, with the leaves containing the actual state information, i.e., the values of the variables in the model and the current state of each process in the model. By doing so, common subtrees can be shared by states, which in practice means that 2–6× less memory is required [29]. In addition, the tree roots are stored in the tree database in a compressed form by means of *Cleary compression* [9, 29]. The result is that roots can be stored as 32-bit integers, while non-roots are stored as 64-bit integers.

Once all the reachable states have been stored, the threads perform several combined *parallel prefix sums* [16] (step 2 of Fig. 2). This is the first step of the procedure to derive a *transition matrix* from the filled tree database. In such a matrix, a non-zero probability $p \in \langle 0, 1]$ at position (i, j) indicates that from state i, a transition with probability p exists to state j. Deriving this matrix from

the tree database in a massively parallel way is non-trivial, and our procedure has been specifically designed for GPUEXPLORE$^{\text{PROB}}$. As transition matrices tend to be very sparse, the matrix is stored in *Compressed Sparse Row* (CSR) format, so only the non-zero entries are actually stored. To achieve this, every stored state needs to be assigned a unique ID in the range $[0, n\rangle$, with n the number of reachable states, and for each state, the number of outgoing transitions, and therefore successor states, needs to be recorded in device memory. During exploration, when a state has been explored, we store its number of successors at its location in the tree database. After exploration, these numbers are gathered together into an array using *stream compaction* [18], and they are added up with a parallel prefix sum, as illustrated in Fig. 4. Initially, the successor numbers are gathered, see the top of the figure. The algorithm works in rounds. In each round i, the first one being 0, the GPU threads scan this array, and the value in each cell j is updated to be the sum of its old value plus the value in cell $j - 2^i$, if it exists. After $\log n$ rounds, the algorithm terminates. After all resulting values have been shifted one position to the right, and the first cell has been assigned 0, the offsets have been obtained. A very similar operation can be performed to assign the states consecutive IDs. Essentially, for every state, the number of states preceding it must be counted. Our procedure performs both prefix sums in a single procedure, i.e., their rounds are combined.

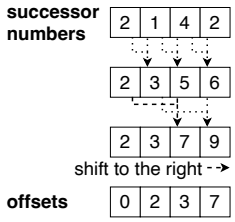


Fig. 4: Computing offsets.

Once this step has finished, the non-zero probabilities can be stored in an array called *values*. For each state i, the probabilities of its outgoing transitions need to be stored in *values* from position *offsets*$[i]$ to position *offsets*$[i + 1]$. Likewise, the IDs of the states reached via these transitions, i.e., the columns of the matrix, are stored in an array *succs*. Storage of the probabilities and the successor IDs is again performed in a massively parallel way, by running the explore kernel again (step 3). In the kernel, the successors of each state are explored twice: once to construct the successor and retrieve its global id from global memory into shared memory, and once more to relate the retrieved global id to its predecessor. This is needed because the state storage and retrieval mechanism to and from global memory is agnostic of the source state.

Verifying a PCTL property. GPUEXPLORE$^{\text{PROB}}$ has full support for the temporal logic PCTL. A PCTL formula is a *state formula*, in which atomic propositions are combined with logical operators plus the probabilistic operator $\mathbb{P}_{\bowtie b}(\varphi)$, with $\bowtie \in \{\leq, <, >, \geq\}$ and $b \in [0.0, 1.0]$. Here, φ is a *path formula*, in which state formulae are combined using the operators X (next), U (until) and $\mathsf{U}^{\leq k}$ (bounded until). A state s satisfies $\mathbb{P}_{\bowtie b}(\varphi)$ iff the probability of following a path from s that satisfies φ is in the interval represented by $\bowtie b$. A derived operator $\mathbb{P}_{=?}(\varphi)$ can be used to request the probability of following a path satisfying φ.

We refrain from explaining how exactly PCTL formulae can be verified, the interested reader is referred to [2]. What is important to note here is that the verification procedure (step 4 in Fig. 2) relies on matrix-vector multiplications.

For instance, for a formula of the form $\varphi = \mathbb{P}_{\bowtie b}(\Phi_1 \,\mathsf{U}\, \Phi_2)$, a system of linear equations must be solved, which can be done by using an iterative method, such as Jacobi or Gauss-Seidel. The used matrix is the transition matrix, and the initial vector v is derived from the PCTL formula. For instance, to verify the formula φ given above, $v[i] = 1.0$ iff state i satisfies Φ_2. To perform matrix-vector multiplications, GPUexplore^{PROB} uses NVIDIA's cuSPARSE library [23].

To store intermediate verification results, GPUexplore^{PROB} associates with each state a 64-bits bit-vector; each bit can be used to store the result of verifying a particular subformula of the given PCTL formula. Initially, the states in the tree database are inspected to determine the result for each atomic proposition in the given PCTL formula, and these results are stored in the bit-vectors, such that from that point on, the tree database does not need to be inspected anymore.

3 Experiments

Experimental setup. We compared GPUexplore^{PROB} with Storm. Storm 1.9.0, built from commit 5d5ebe4 of the master branch of the Storm GitHub repository, was run on an AMD Ryzen™ 7 5800X from 2020, installed on an MSI MPG x570 Gaming Pro Carbon Wi-Fi motherboard, with 32GB of DDR4 RAM. This machine is equipped with an RTX 3090 GPU from 2020, with 10,496 CUDA cores and 24GB of GDDR6X VRAM. GPUexplore^{PROB} was run on this GPU.

Benchmarks. From the Quantitative Verification Benchmark Set [13] (QComp), all DTMC models were selected with 1) at least one instance having a state space with the number of states between 5,000 and 3,000,000 states (to exclude very small state spaces, for which GPU acceleration is not useful, and to exclude state spaces too large to fit in GPUexplore^{PROB}'s 20GB tree database), and 2) at least one formula in PCTL, not involving a reward extension [1], as this is not yet supported by GPUexplore^{PROB}.[1] In addition, benchmarks created for the Rubicon tool [17] and for [21] have been used, taken from their accompanying artifacts. Table 1 presents a list of benchmarks, referring to how our naming scheme for instances (see Table 2) provides the values of the involved constants. In addition, the verified PCTL formula is given.

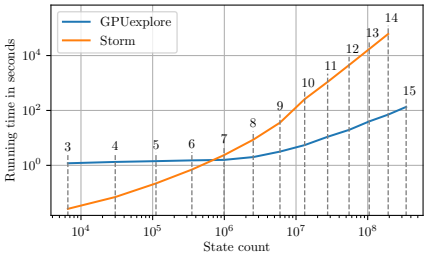


Fig. 5: Scalability w.r.t. Crowds.

Fig. 5 shows how GPUexplore^{PROB} and Storm (with its *sparse* engine) scale when verifying Crowds instances. `TotalRuns` ranges from 3 to 15, `CrowdSize` is 10. For `TotalRuns`=15, Storm ran out of memory (32GB). GPUexplore^{PROB}

[1] An exception is the EGL model, which is excluded due to a current technical limitation of our tool, related to the number of outgoing transitions a state may have.

Table 1: Benchmark descriptions.

Case	Instance name	Property
BRP [13]	BRP.N.MAX	$\mathbb{P}_{=?}[\mathbf{true}\,\mathsf{U}\,\mathtt{s}=5]$
Coupon [13]	Coupon.N.DRAWS (B=5)	$\mathbb{P}_{=?}[\mathbf{true}\,\mathsf{U}\,\mathtt{\_ret0\_}]$
Crowds [13]	Crowds.TotalRuns.CrowdSize	$\mathbb{P}_{=?}[\mathbf{true}\,\mathsf{U}\,\mathtt{observe0}>1]$
Nand [13]	Nand.N.K	$\mathbb{P}_{=?}[\mathbf{true}\,\mathsf{U}\,\mathtt{s}=4\wedge\mathtt{z/N}<0.1]$
Lumbroso [21]	Lumbroso.N	$\mathbb{P}_{=?}[\mathbf{true}\,\mathsf{U}^{\leq 250}\,\mathtt{term}]$
WeatherFactory [17]	WeatherFactory.⟨nr. of factories⟩	$\mathbb{P}_{=?}[\mathbf{true}\,\mathsf{U}^{\leq 100,000}\,\mathtt{allStrike}]$
Factory [17]	Factory.⟨nr. of factories⟩	$\mathbb{P}_{=?}[\mathbf{true}\,\mathsf{U}^{\leq 100,000}\,\mathtt{allStrike}]$

Table 2: Runtimes (secs.) of GPUEXPLORE$^{\text{PROB}}$ and STORM. OOM: out of mem.

			GPUEXPLORE$^{\text{PROB}}$			STORM			speedup		
Instance	states	trans.	explore	check	total	explore	check	total	explore	check	total
BRP.64.5	5192	6781	0.809	0.378	1.188	0.020	0.000	0.020	0.025	0.000	0.017
Coupon.7.3	338K	586K	1.100	0.211	1.310	0.255	0.010	0.265	0.232	0.047	0.202
Coupon.9.4	28M	56M	1.641	11.493	13.134	27.055	2.660	29.715	16.487	0.231	2.262
Crowds.12.10	55M	132M	3.204	17.007	20.211	81.420	4,422.589	4,504.009	25.412	260.045	222.849
Crowds.13.10	105M	253M	5.100	35.123	40.222	152.738	17,083.549	17,236.287	29.949	486.392	428.529
Crowds.14.10	193M	466M	8.575	63.090	71.665	277.658	61,048.680	61,326.338	32.380	967.644	855.736
Crowds.15.10	344M	831M	14.265	117.147	131.412	683.681	OOM	OOM	47.927	OOM	OOM
Factory.8	256	66K	0.779	4.367	5.146	0.065	3.528	3.593	0.083	0.808	0.698
Factory.9	512	262K	0.762	4.705	5.467	0.221	15.293	15.514	0.290	3.250	2.838
Factory.10	1024	1M	0.870	6.086	6.956	0.976	63.162	64.138	1.122	10.378	9.221
Lumbroso.10000	19M	19M	2.665	3.290	5.954	13.418	9.013	22.431	5.035	2.740	3.767
Nand.20.K4	308K	476K	1.312	0.207	1.519	0.357	0.054	0.411	0.272	0.261	0.271
Nand.40.K4	4M	6M	1.521	2.000	3.521	4.588	1.253	5.841	3.016	0.627	1.659
Nand.60.K4	19M	30M	2.147	13.315	15.462	21.404	6.377	27.781	9.969	0.479	1.797
WeatherFactory.8	512	262K	0.802	4.697	5.500	0.230	15.188	0.230	0.287	3.234	0.042
WeatherFactory.9	1024	1M	0.893	6.111	7.004	1.022	63.275	1.022	1.144	10.354	0.146
WeatherFactory.10	2048	4M	1.319	11.625	12.944	4.553	266.765	271.318	3.452	22.948	20.961

could verify this instance, thanks to unified memory, demonstrating its usefulness. For the 14-instance, GPUEXPLORE$^{\text{PROB}}$ achieved an 856× speedup.

Table 2 presents the runtime results for both GPUEXPLORE$^{\text{PROB}}$ and STORM (sparse engine). For each case, the number of reachable states and transitions is given. The runtimes are broken down into exploration / construction time of the state spaces and checking the PCTL formula. The last three columns compare the results of the two tools in the relevant columns, i.e., exploration time, checking time, and total time. The BRP case shows that for models with very small state spaces, both in terms of number of states *and* number of transitions, using a GPU is not promising, which is to be expected; to keep many GPU threads busy, large state spaces are required. The results for the other models show examples where exploration starts to pay off, with the best result being 47× faster than STORM (Crowds.15.10). Regarding formula checking, GPUEXPLORE$^{\text{PROB}}$ is particularly fast when many matrix-vector multiplications are required, which is always the case when the formula contains the bounded until operator with a large bound. The best recorded result is 968× faster than STORM (Crowds.14.10). For unbounded until, STORM is often able to apply preprocessing steps to avoid the multiplications (almost) altogether. As these steps have not yet been implemented in GPUEXPLORE$^{\text{PROB}}$, it tends to be slower in those cases. However, there is no fundamental reason preventing us from adding support for those steps in the tool, and we plan to add them in the near future.

Concluding, the results show the potential of GPU accelerated PMC, both for state space construction and verification. For future work, we will further optimise verification, and add support for CTMCs and MDPs. Finally, we will conduct research on using Multi-Terminal Binary Decision Diagrams.

References

1. Andova, S., Hermanns, H., Katoen, J.P.: Discrete-time rewards model-checked. In: FORMATS. LNCS, vol. 2791, pp. 88–104. Springer (2003). https://doi.org/10.1007/978-3-540-40903-8_8
2. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
3. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. J. Parall. Distrib. Comput. **72**, 1083–1097 (2012). https://doi.org/10.1016/J.JPDC.2011.10.015
4. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a GPGPU-parallel SPIN Model Checker. In: SPIN 2014. pp. 87–96. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2632362.2632379
5. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphics Processors. STTT **13**(1), 21–35 (2011). https://doi.org/10.1007/s10009-010-0176-4
6. Bošnački, D., Edelkamp, S., Sulewski, D.: Efficient probabilistic model checking on general purpose graphics processors. In: SPIN. LNCS, vol. 3925, pp. 32–49. Springer (2006). https://doi.org/10.1007/978-3-642-02652-2_7
7. Budde, C., Dehnert, C., Hahn, E., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative Model and Tool Interaction. In: TACAS, Part 2. LNCS, vol. 10206, pp. 151–168. Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_9
8. Bussi, L., Ciancia, V., Gadducci, F.: Towards a Spatial Model Checker on GPU. In: FORTE. LNCS, vol. 12719, pp. 188–196. Springer (2021). https://doi.org/10.1007/978-3-030-78089-0_12
9. Cleary, J.: Compact Hash Tables Using Bidirectional Linear Probing. IEEE Trans. on Computers **c-33**(9), 828–834 (1984). https://doi.org/10.1109/TC.1984.1676499
10. DeFrancisco, R., Cho, S., Ferdman, M., Smolka, S.A.: Swarm model checking on the GPU. Int. J. Softw. Tools Technol. Transf. **22**(5), 583–599 (2020). https://doi.org/10.1007/s10009-020-00576-x
11. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is Coming: A Modern Probabilistic Model Checker. In: CAV, Part II. LNCS, vol. 10427, pp. 592–600. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_31
12. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: SPIN. LNCS, vol. 6349, pp. 106–123. Springer (2010). https://doi.org/10.1007/978-3-642-16164-3_8
13. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS. LNCS, vol. 11427, pp. 344–350. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_20
14. Heemstra, J., Wijs, A.: GPUexplore-prob (2025). https://doi.org/10.5281/zenodo.14771103
15. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. STTT **24**(4), 589–610 (2022). https://doi.org/10.1007/s10009-021-00633-z
16. Hillis, W., Jr., G.S.: Data parallel algorithms. Communications of the ACM **29**(12), 1170–1183 (1986). https://doi.org/10.1145/7902.7903
17. Holtzen, S., Junges, S., Vazquez-Chanlatte, M., Millstein, T., Seshia, S., Van den Broeck, G.: Model Checking Finite-Horizon Markov Chains with Probabilistic Inference. In: CAV. LNCS, vol. 12760, pp. 577–601. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_27

18. Horn, D.: Stream Reduction Operations for GPGPU Applications. In: GPU Gems 2, chap. 36. Pearson Education (2005)
19. Khan, M., Hassan, O., Khan, S.: Accelerating SpMV Multiplication in Probabilistic Model Checkers Using GPUs. In: ICTAC. LNCS, vol. 12819, pp. 86–104. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_6
20. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: TOOLS. LNCS, vol. 2324, pp. 200–204. Springer (2002). https://doi.org/10.1007/3-540-46029-2_13
21. Mertens, H., Katoen, J.P., Quatmann, T., Winkler, T.: Accurately Computing Expected Visiting Times and Stationary Distributions in Markov Chains. In: TACAS. LNCS, vol. 14571, pp. 237–257. Springer (2024). https://doi.org/10.1007/978-3-031-57249-4_12
22. Norris, J.: Markov Chains. Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press (1998)
23. NVIDIA Corporation: cuSPARSE, the CUDA sparse matrix library. `https://docs.nvidia.com/cuda/cusparse`, last visited on 08/10/24
24. Osama, M., Wijs, A.: Hitching a Ride to a Lasso: Massively Parallel On-The-Fly LTL Model Checking. In: TACAS, Part 2. LNCS, vol. 14571, pp. 23–43. Springer (2024). https://doi.org/10.1007/978-3-031-57249-4_2
25. Osama, M., Wijs, A.: GPU Acceleration of Bounded Model Checking with ParaFROST. In: CAV, Part II. LNCS, vol. 12760, pp. 447–460. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_21
26. Wei, H., Chen, X., Ye, X., Fu, N., Huang, Y., Shi, J.: Parallel Model Checking on Pushdown Systems. In: ISPA/IUCC/BDCloud/SocialCom/SustainCom. pp. 88–95. IEEE (2018). https://doi.org/10.1109/BDCloud.2018.00026
27. Wei, H., Ye, X., Shi, J., Huang, Y.: ParaMoC: A Parallel Model Checker for Pushdown Systems. In: ICA3PP. LNCS, vol. 11945, pp. 305–312. Springer (2019). https://doi.org/10.1007/978-3-030-38961-1_26
28. Wijs, A., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: SPIN. LNCS, vol. 7385, pp. 98–116. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_9
29. Wijs, A., Osama, M.: A GPU Tree Database for Many-Core Explicit State Space Exploration. In: TACAS. LNCS, vol. 13993, pp. 684–703. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_35
30. Wijs, A., Osama, M.: GPUexplore 3.0: GPU Accelerated State Space Exploration for Concurrent Systems with Data. In: SPIN. LNCS, vol. 13872, pp. 188–197. Springer (2023). https://doi.org/10.1007/978-3-031-32157-3_11
31. Wijs, A., Bošnački, D.: GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In: TACAS. LNCS, vol. 8413, pp. 233–247 (2014). https://doi.org/10.1007/978-3-642-54862-8_16
32. Wijs, A., Bošnački, D.: Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs. STTT **18**(2), 169–185 (2016). https://doi.org/10.1007/s10009-015-0379-9
33. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking. In: FM. LNCS, vol. 9995, pp. 694–701. Springer (2016). https://doi.org/10.1007/978-3-319-48989-6_42
34. Wu, Z., Liu, Y., Liang, Y., Sun, J.: GPU Accelerated Counterexample Generation in LTL Model Checking. In: ICFEM. LNCS, vol. 8829, pp. 413–429. Springer (2014). https://doi.org/10.1007/978-3-319-11737-9_27

35. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU Accelerated On-the-Fly Reachability Checking. In: ICECCS. pp. 100–109 (2015). https://doi.org/10.1109/ICECCS.2015.21

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

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

14th Competition on Software Verification (SV-COMP 2025)

Improvements in Software Verification and Witness Validation: SV-COMP 2025

Dirk Beyer [1]✉ and Jan Strejček [2]

[1] LMU Munich, Munich, Germany
[2] Masaryk University, Brno, Czech Republic

Abstract. The 14th edition of the *Competition on Software Verification (SV-COMP 2025)* evaluated 62 verification tools and 18 witness validation tools, making it the largest comparison of its kind so far. Out of these, 35 verification and 13 validation tools participated with an active support of teams led by 33 different representatives from 12 countries. The verification track of the competition was executed on a benchmark set of 33 353 verification tasks with C programs and 6 different specifications (reachability, memory safety, memory cleanup, overflows, termination, and data races) and 674 verification tasks with Java programs checked for assertion validity. Additionally, we considered 673 verification tasks with Java programs checked for runtime exceptions as a demo category. The validation track analyzed the witnesses generated in the verification track and newly also 103 handcrafted witnesses. To handle the increasing complexity of the competition, the organization committee has been established.

Keywords: Formal Verification · Program Analysis · Competition · Software Verification · Verification Tasks · Verification Witnesses · Witness Validation · Benchmark · Specification · C Language · Java Language · SV-COMP · SV-Benchmarks · BenchExec · CoVeriTeam

1 Introduction

This report presents the objectives, processes, rules, participants, and results of SV-COMP 2025. It extends the series of competition reports (see footnote). This year, we focus on a more precise description of the current category structure, competition workflow, scoring schema, and changes in the competition done in 2025. The 14th Competition on Software Verification (`https://sv-comp.sosy-lab.org/2025`) is again the largest comparative evaluation ever in this area. The objectives of the competitions were discussed earlier (1–4 [23]) and extended over the years (5–6 [24] and 7 [30]):

1. provide an overview of the state of the art in software-verification technology and increase visibility of the most recent software verifiers,

This report extends previous reports on SV-COMP [17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 30]. Reproduction packages are available on Zenodo (see Table 5).

✉ `dirk.beyer@sosy.ifi.lmu.de`

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 151–186, 2025.
https://doi.org/10.1007/978-3-031-90660-2_9

2. establish a repository of software-verification tasks that is publicly available for free use as a standard benchmark suite for evaluating verification software,
3. establish standards that make it possible to compare different verification tools, including a property language and formats for the results,
4. accelerate the transfer of new verification technology to industrial practice by identifying the strengths of the various verifiers on a diverse set of tasks,
5. educate PhD students and others on performing reproducible benchmarking, packaging tools, and running robust and accurate research experiments,
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 participation, and other meta data in a central repository.

The SV-COMP 2020 report [24] 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 area of formal methods [16]. 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 [84], VerifyThis [70], Test-Comp [29], and TermCOMP [77]. The SV-COMP 2020 report [24] provides a more detailed discussion.

Quick Summary of Changes. We aim to keep the setup of the competition stable. Still, there are always improvements and developments. For the 2025 edition, the following changes were made:

- The organization committee was established.
- The number of considered verification tasks again increased in both C and Java languages: the number of C tasks increased from 30 300 in 2024 to 33 353 and the number of Java tasks increased from 587 in 2024 to 674 (not counting the demo category mentioned below).
- We newly considered the property saying that a Java program does not cause a runtime exception. We used 673 tasks with this property as a demo category.
- In the validation track, we newly used 103 handcrafted witness validation tasks as a complement to the validation tasks generated by verifiers.
- The category structure was extended by three new base categories with C programs added to meta category *SoftwareSystems*, namely *SoftwareSystems-Intel-TDX-Module-ReachSafety*, *SoftwareSystems-uthash-MemCleanup*, and *SoftwareSystems-DeviceDriversLinux64-Termination*, one new base category *RuntimeException-Java* with Java programs (ran as a demo category due to late announcement) added to *JavaOverall*, and one new meta category *ValidationCrafted* of handcrafted validation tasks divided into two base categories *CorrectnessWitnesses-Loops* and *ViolationWitnesses-ControlFlow*.
- The definition of the *memory safety* subproperty expressing that all allocated memory is tracked was reformulated more precisely.

- The validation of violation witnesses in version 2.0 turned into a regular category due to a higher number of participating tools.
- The normalization formula computing the scores in meta categories was modified to ignore void tasks and empty categories.
- We now officially recognize two types of *hors-concours* participants: *meta verifiers* and *inactive* participants, which are tools without active team support.
- The medals can be assigned only for positive scores.

2 Organization and Processes

Organization. The competition was established and ran for the first 13 years by Dirk Beyer. The SV-COMP community supported the competition mostly by maintaining the collection of verification tasks. Executing the competition in its current form would not be possible without the benchmarking framework BenchExec [42] and the system for distributed benchmark execution BenchCloud [35], which are developed and maintained by his research group. Since the beginning, the numbers of benchmarks and participants are substantially growing. Moreover, completely new layers of complexity were added with witness validation, the whole validation track, and a second format of witnesses. To distribute the effort necessary to smoothly run of the competition, the organization committee was established before the SV-COMP 2025. The competition has now two chairs (Dirk Beyer and Jan Strejček) and members in charge of benchmark quality (Zsófia Ádám, Raphaël Monat, Simmo Saan, and Frank Schüssele), category structure (Thomas Lemberger), infrastructure development (Philipp Wendler, Po-Chun Chien, Marek Jankola, Henrik Wachowitz, Matthias Kettl, and Marian Lingsch-Rosenfeld), and reproducibility (Levente Bajczi).

Procedure. SV-COMP is an open competition (also known as comparative evaluation). Verification tasks and handcrafted validation tasks are publicly available in a repository (Table 4) where anyone can contribute. The competition has basically two phases: *training* and *evaluation*. During the training phase, participating teams can repeatedly submit new versions of their tool. The organizers run the tool on relevant tasks and provide the results to the whole community. The participants can inspect the results, fix bugs in their tools and submit a new version or report an issue with some tasks. The set of verification tasks and handcrafted validation tasks is frozen approximately two weeks before the the end of the training phase. In the evaluation phase, the tools are again executed on all relevant tasks, the participants are asked for an inspection of the results and they can challenge the validity of some tasks. After removing the invalid tasks (they are marked as *void*), the results are announced on the competition web site.

Competition Jury. The competition jury reviews the competition contribution papers and helps the organizer with resolving any disputes that might occur (cf. competition report of SV-COMP 2013 [18]). The tasks of the jury were described in more detail in the report of SV-COMP 2022 [27]. The jury consists of the SV-COMP chairs and one representative of each actively participating

tool. The representatives of participating tools circulate every year after the tool-submission deadline. The current representatives are listed in Tables 6 and 7. The jury includes one additional member, P. Darke (TCS, India), representing a tool that ultimately did not actively participate due to license issues. The current jury is also listed on the web site (`https://sv-comp.sosy-lab.org/2025/committee.php`).

3 Tasks, Workflow, and Scoring

Verification and Validation Tasks. A *verification task* consists of a program, a property to be verified, and an expected verification result. A *validation task* is a program, a property, and a witness of the program correctness or property violation to be validated. SV-COMP 2025 supports witnesses in two versions of the witness format, namely 1.0 [37] and 2.0 [7]. Some validation tasks (e.g., the handcrafted ones) contain also the expected validation result. SV-COMP 2025 used the task-definition format in version 2.1 to denote the verification and validation tasks.

Properties. In connection with C programs, we consider 6 different properties: *unreachability of a given function* (referenced as `unreach-call` in the competition), *memory safety* (`valid-memsafety`) composed of three subproperties saying that all pointer dereferences are valid (`valid-deref`), all memory deallocations are valid (`valid-free`), and all allocated memory is tracked (`valid-memtrack`), *memory cleanup* (`valid-memcleanup`) saying that all allocated memory is deallocated before the program terminates, *no overflow* (`no-overflow`) saying that operations on signed integers never overflow, *no data race* (`no-data-race`) saying that the program does not contain any data race, and *termination* (`termination`) saying that the program always terminates. Note that a program is considered memory safe only if it satisfies all three subproperties. The subproperties are used in particular to report what is violated if a program is not memory safe. For Java programs, we consider the property *assertion validity* (`assert_java`) and newly also *no runtime exception* (`runtime-exception`). We refer to the conference web page for precise definition of these properties (`https://sv-comp.sosy-lab.org/2025/rules.php`). We note that the precise definition of tracked memory was modified for SV-COMP 2025 as the previous definition was ambiguous.

Categories. The verification tasks are divided into base categories loosely reflecting the programming language, program features, the considered property, and the source of the benchmarks. Base categories are accumulated into meta categories. For the C language, there are two levels of meta categories. The top level contains the category *Overall* of all C verification tasks and the category *FalsificationOverall*, which is rather specific. The category *FalsificationOverall* was originally introduced to support bug-finding tools. It has basically the same structure as *Overall*, but it contains only the tasks with safety properties, which are all but *termination*. In other words, *FalsificationOverall* does not contain the whole meta category *Termination* and base category *SoftwareSystems-DeviceDriversLinux64-Termination*. It also uses a specific scoring schema as explained below. For Java language, there is only a single meta category called *JavaOverall*. The category

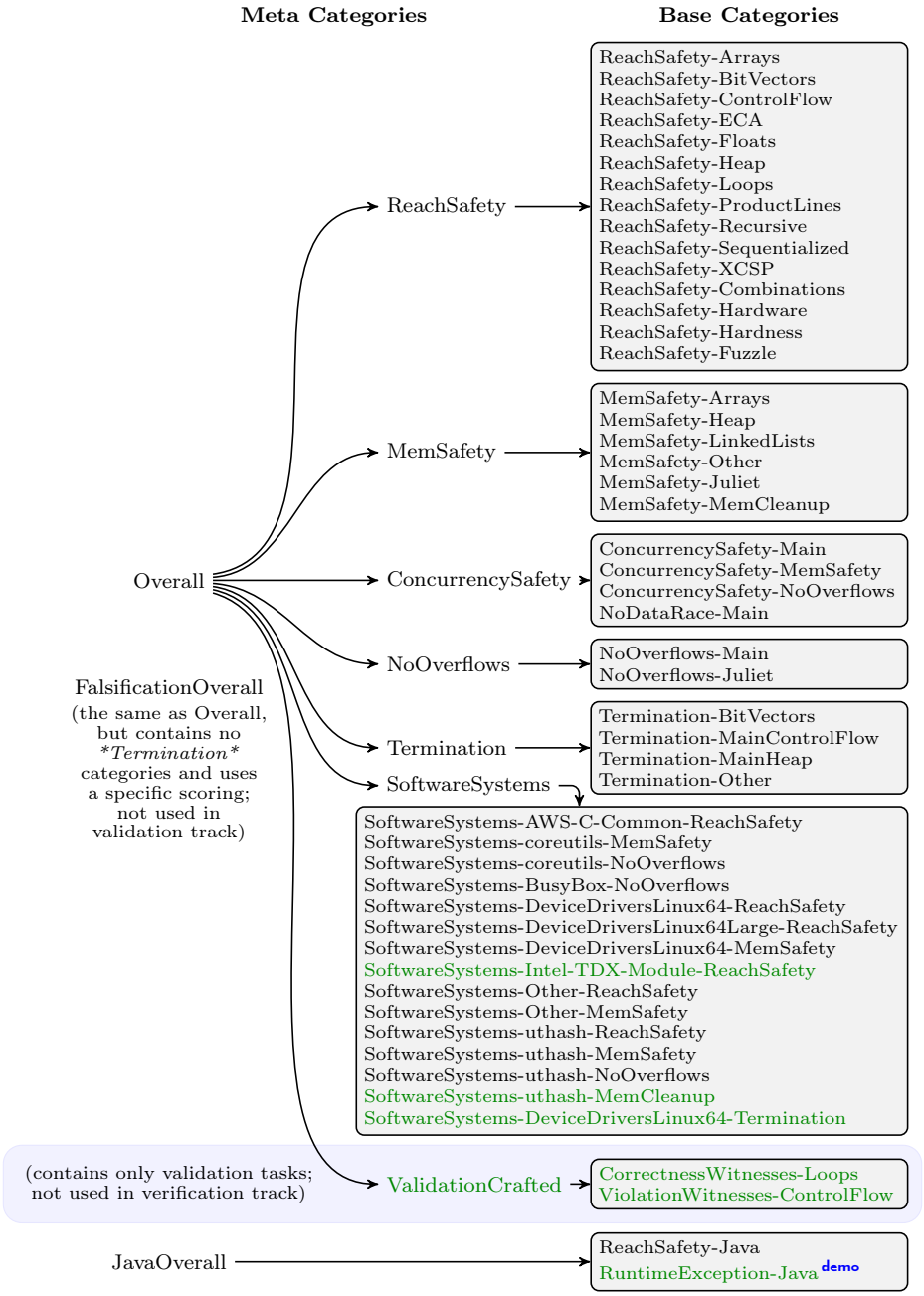


Fig. 1: Category structure for SV-COMP 2025; demo marks the demo category, new categories are green

structure was significantly updated for SV-COMP 2024. For SV-COMP 2025, we added 3 new base categories to the category *SoftwareSystems*. Further, the category *JavaOverall* was extended with the base category *RuntimeException-Java* of verification tasks with the property *no runtime exception*. This base category ran only as a demo category because it was announced shortly before the competition deadlines. As a consequence, this base category is not reflected in the score computations of *JavaOverall* The current category structure including the new categories is shown in Fig. 1. The meta categories and the number of verification tasks they contain are shown in Tables 10, 11, and 12 presenting the results of the verification track. Finally, the categories are described in detail on the competition web site (`https://sv-comp.sosy-lab.org/2025/benchmarks.php`).

All validation tasks are basically divided into 4 groups: the witnesses of program correctness are separated from the witnesses of property violation and each of these two groups is again divided according to the used witness format (version 1.0 or 2.0). In each of the four groups, the validation tasks are organized basically in the same category structure: each validation task generated by a verifier is in the base category determined by the corresponding verification task. The handcrafted validation tasks are stored in two new base categories united in a new meta category called *ValidationCrafted* as shown in Fig. 1. The meta category *FalsificationOverall* is not considered in the validation track. As the witness formats and current validators have certain limitations, SV-COMP 2025 supports witnesses in individual formats only for selected properties and categories, as shown in Table 1. Some categories in some groups are empty, for example because SV-COMP does not support the particular witness format for the category. All empty categories are removed from the category structures of the groups.

Competition Workflow. Roughly speaking, the inputs of the competition are (a) *verification tasks* and, for each verifier and witness validator, the participating teams provide (b) *benchmark definition* listing the base categories the tool is supposed to be applied on (i.e., the tool opts-out from the categories that are not listed there), (c) *tool-info module* that specifies the interface for running the tool and interpreting its results, and (d) *tool archive* on Zenodo from where the tool is downloaded. The inputs provided by participating teams are described more precisely in the report for Test-Comp 2021 [26] (SV-COMP and Test-Comp use similar workflow and components).

The verification track proceeds as follows. The competition scripts run each verifier on all relevant verification tasks given by its *benchmark definition*. If the verifier solves a task, it returns either TRUE meaning that the program satisfies the given property or FALSE meaning that the program violates the property. The output TRUE is accompanied with a *correctness witness* and the output FALSE with a *violation witnesses* in some of the formats supported by SV-COMP for the corresponding property and category (see Table 1). When SV-COMP supports both formats, the verifier can actually produce two witnesses, one in each format. If no format is supported, the witness is not required. This is the case of correctness witnesses in categories **-Arrays*, **-Floats*, **-Heap*, **MemSafety**, *ConcurrencySafety-**, **NoDataRace**, **Termination-**, and **-Java*. The generated

Table 1: Support of witnesses in individual formats based on property and category; '–' indicates that the property is not relevant for the witness type in the column

		Correctness Witnesses		Violation Witnesses	
Property	**Categories**	v1.0	v2.0	v1.0	v2.0
`unreach-call`	all except **-Arrays*, **-Floats*, **-Heap*, and *ConcurrencySafety-**	✓	✓	✓	✓
`unreach-call`	**-Arrays*, **-Floats*, and **-Heap*			✓	✓
`unreach-call`	*ConcurrencySafety-**			✓	
`valid-memsafety`	all			–	–
`valid-deref`	all except *ConcurrencySafety-**	–	–	✓	✓
`valid-deref`	*ConcurrencySafety-**	–	–	✓	
`valid-free`	all except *ConcurrencySafety-**	–	–	✓	✓
`valid-free`	*ConcurrencySafety-**	–	–	✓	
`valid-memtrack`	all	–	–	✓	
`valid-memcleanup`	all			✓	
`no-overflow`	all except *ConcurrencySafety-**	✓	✓	✓	✓
`no-overflow`	*ConcurrencySafety-**			✓	
`no-data-race`	all			✓	
`termination`	all			✓	
`assert-java`	all			✓	
`runtime-exception`	all			✓	

witnesses are turned into validation tasks. If the program of the validation task is written in C, we run WitnessLint to check that the witness adheres to the format. If the answer is negative, the witness is *syntactically invalid* and we never consider such a witness as confirmed. We do not apply this step for validation tasks with programs in Java as there is no witness linter for these witnesses. Competition scripts then run all suitable validators on each validation task, where suitability is determined by the *benchmark definition* of validators. Each validator can either confirm the task, refute it, or fail to solve it. A validator confirms a correctness witness by returning True and refutes it by returning False. A violation witness is confirmed by False and refuted by True.

The validation track uses the same inputs, only the verification tasks are replaced by *validation tasks*. These are the handcrafted validation tasks and all validation tasks generated by the verifiers in the verification track and successfully checked by WitnessLint. A validation task contains an expected result if it is handcrafted or if it was generated with an incorrect verification result (i.e., it contains a correctness witness for a verification task with property violation or a violation witness for a verification task with a correct program). In the latter case, the expected validation result corresponds to refutation. Each validator is executed on all validation tasks specified by the corresponding *benchmark definition*.

Computing Resources. The computing resources for each tool execution were the same as in SV-COMP 2024. Each verifier run was limited by 15 GB of memory

Table 2: Scores per individual verification results used since SV-COMP 2021

Result	Points	Description
True correct	+2	Program correctly reported to satisfy the property and the correctness witness confirmed (or not required)
True incorrect	−32	Incorrect program reported as correct (wrong proof)
False correct	+1	Property violation was correctly found and the violation witness was confirmed
False incorrect	−16	Violation reported but the property holds (false alarm)

and 15 min of cumulative CPU time on 4 processing units. Each run of a validator was limited to 2 processing units, 7 GB of memory, and 1.5 min of CPU time for violation witnesses and 15 min of CPU time for correctness witnesses. The machines for running the experiments are part of a computer cluster at the SoSy-Lab at LMU, which consists of 168 machines, where each machine has one Intel Xeon E3-1230 v5 CPU with 8 processing units, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 24.04 with Linux kernel 6.8). We used BenchExec [42] to measure and control computing resources (CPU time, memory) and BenchCloud [35] to distribute, install, run, and clean-up verification runs, and to collect the results.

Scoring Schema. For verification track, the scoring schema of SV-COMP 2025 in base categories was the same as for SV-COMP 2021. Table 2 lists all the cases when verification results are awarded with non-zero points, where a verification result is

- *incorrect* if it does not agree with the expected result of the verification task,
- *correct* if it agrees with the expected result and the produced witness (or at least one of them if there are two) is confirmed by some validator. In the categories where correctness witnesses are not required, a verification results is *correct* whenever it agrees with the expected result.

When a verification result agrees with the expected result but no validator confirms the witness (although it is required), we call the result *correct-unconfirmed*. The score of a verifier in a base category is simply the sum of the points for individual verification tasks. The score for a meta category is computed from the scores of all contained (meta or base) categories on the next level and the number of tasks in these categories. Formally, if a meta category contains k categories of the next level and the i-th contained category has the score s_i and consists of n_i verification tasks, then the meta category gets the score $(\Sigma_{i=1}^{k} s_i/n_i) \cdot (\Sigma_{i=1}^{k} n_i)/k$, i.e., the sum of scores in each category normalized by the number of tasks in the category multiplied by the average number of tasks in the contained categories. Note that in previous years, the numbers n_i included also void tasks that are technically in categories but are not used in the competition (a verification task is marked as *void* typically because it was changed for some serious reason after the task freezing deadline). Since SV-COMP 2025, these void tasks are not included in n_i.

Table 3: Scores per individual validation results used since SV-COMP 2024

Result	Points	Description for correctness witnesses Description for violation witnesses
TRUE correct	+2	The correctness witness was correctly confirmed The violation witness was correctly refuted
TRUE incorrect	−32	The correctness witness was confirmed but it is not correct The violation witness was refuted but it is correct
FALSE correct	+1	The correctness witness was correctly refuted The violation witness was correctly confirmed
FALSE incorrect	−16	The correctness witness was refuted but it is correct The violation witness was confirmed but it is not correct

The scoring in *FalsificationOverall* and its subcategories is slightly different. As this meta category was motivated by tools that can only find a bug, the scores in its base categories are sums of the points for results FALSE (both correct and incorrect), while the results TRUE are ignored. The computation of scores for meta categories remains unchanged.

For validation track, the scoring schema of SV-COMP 2025 in base categories was the same as for SV-COMP 2024. The biggest difference from the verification track comes from the fact that many validation tasks do not contain the expected validation result. In fact, it is contained only in handcrafted validation tasks and in the tasks generated by verifiers that solve some verification task incorrectly (and thus the witness they produce should be refuted). For the remaining tasks, the expected results are determined by voting. For each such a task, we collect the results from all validators that solved (i.e., confirmed or refuted) the task. If we have at least two such results and at least 75 % of them agree on their decision, then the expected result is set by the majority vote. In all other cases, the expected result is not determined and the validation task has no influence on the validation track results as it is considered *void*. Tool developers can inspect the results and convince the community that a voted expected results is in fact wrong. In such a case, the corresponding validation task is removed from the competition.

Table 3 lists all the cases when validation results are awarded with non-zero points. A validation result TRUE or FALSE is considered *correct* if it agrees with the expected result and *incorrect* otherwise. To compute the scores in base categories, we virtually divide each base category C into two subcategories C_c, C_w, where

- C_c contains the witnesses that are expected to be *correct* (i.e., correctness witnesses with the expected result TRUE and violation witnesses with the expected result FALSE), and
- C_w contains the witnesses that are expected to be *wrong* (i.e., correctness witnesses with the expected result FALSE and violation witnesses with the expected result TRUE).

The score of a validator in each subcategory C_c, C_w is the sum of the points for individual validation tasks in the subcategory. If some of the subcategories C_c, C_w

Table 4: Publicly available components for reproducing SV-COMP 2025

Component	Repository	Version
Verification Tasks	gitlab.com/sosy-lab/benchmarking/sv-benchmarks	svcomp25
Benchmark Definitions	gitlab.com/sosy-lab/sv-comp/bench-defs	svcomp25
Tool-Info Modules	github.com/sosy-lab/benchexec	3.29
Verifiers	gitlab.com/sosy-lab/benchmarking/fm-tools	svcomp25
BenchExec (Benchmarking)	github.com/sosy-lab/benchexec	3.29
BenchCloud (Distribution)	gitlab.com/sosy-lab/software/benchcloud	1.3.0
Witness Format	gitlab.com/sosy-lab/benchmarking/sv-witnesses	2.0.3
CoVeriTeam for CI	gitlab.com/sosy-lab/software/coveriteam	1.2.1
Processing Scripts	gitlab.com/sosy-lab/benchmarking/competition-scripts	svcomp25

Table 5: Artifacts published for SV-COMP 2025

Content	DOI	Reference
Verification Tasks	10.5281/zenodo.15012096	[33]
Competition Results	10.5281/zenodo.15012085	[32]
FM-Tools (Verifiers and Validators)	10.5281/zenodo.15055359	[31]
Verification Witnesses	10.5281/zenodo.15012077	[34]
BenchExec	10.5281/zenodo.15007216	[133]
CoVeriTeam	10.5281/zenodo.11193690	[47]

is empty, the score for C is directly the score for the non-empty subcategory. If both subcategories C_c, C_w are non-empty and have respective scores s_c, s_w, then the score for C is computed as for a meta category in the verification track, i.e., $(\frac{s_c}{|C_c|} + \frac{s_w}{|C_w|}) \cdot \frac{|C_c|+|C_w|}{2}$. The score of a validator in a meta category is then computed by the same process as in the verification track. Note that all the scores presented in this paper and on the competition web are rounded to integers before printing, but their computation is done with a higher precision.

Ranking. As before, the rank of a verifier in each category was decided based on the achieved score. In case of a tie, we used the *success run time* as the secondary criterion, which is the total CPU time of the verifier over all tasks in a given category for which the verifier reported a correct verification result. Ranking in validation track works in the same way. Recall that a tool participates only in the categories specified in its *benchmark definition*. In contrast to the previous year, we assign medals only to tools with a positive score.

Reproducibility. SV-COMP results must be reproducible. Hence, all major components are maintained in public version-control repositories. Table 4 lists these components with the links to the repositories and their versions used in SV-COMP 2025. Most of these components are described with more details in the SV-COMP 2016 report [21]. Later, BenchCloud was introduced to distribute the benchmarking jobs in an elastic cloud and collect results. Moreover, CoVeriTeam is used to continuously check (GitLab CI pipeline) whether tools can be executed in

the competition environment. The processing scripts to execute the experiments and post-process the data into tables, scores, and rankings are also publicly released. The competition artifacts are published at Zenodo (see Table 5) with the relevant tools and data to guarantee their long-term availability and immutability.

For the reproducibility reasons, SV-COMP requires since 2018 that the verifiers and validators must be publicly available for download and has a license that

- allows reproduction and evaluation by anybody (incl. results publication),
- does not restrict the usage of the verifier output (log files, witnesses), and
- allows (re-)distribution of the unmodified verifier archive via SV-COMP repositories and archives.

The verification and handcrafted validation tasks used in the competition are also accompanied by a license. In this case, the stated license must allow to

- view, understand, investigate, and reverse engineer the algorithm or system,
- change the program (in particular, pre-process and adopt the programs to be useful for a verification task),
- (re-)distribute the (original and changed) program under the same terms (in particular, in replication packages for research projects or as regression tests),
- compile and execute the program (in particular, for the purpose of verifying that a specification violation exists),
- and commercially take advantage of the program (in particular, to not exclude developers of commercial verifiers).

4 Participating Verifiers and Validators

In total, SV-COMP 2025 evaluates 62 verification and 18 validation tools. Besides 35 verifiers and 13 validators registered to and supported in the competition by development teams, we also evaluated some tools participating in previous years but not actively registered and supported this year. These tools are called *inactive*, clearly marked with ∅ in all tables, and they do not appear in rankings. Further, we clearly distinguish *meta verifiers* according to the following characterization approved by the community of SV-COMP 2023.

> A *meta verifier* is a combination of at least two existing verification components such that each result produced by the combination can be computed by some of its components alone. A verifier is the result of research and engineering in verification algorithms and approaches, while a typical meta verifier selects a verification component to run, sets up its parameters, and potentially post-processes its output.

Meta verifiers are annotated with meta in all tables and also excluded from rankings. Note that before SV-COMP 2025, both inactive tools and meta verifiers were marked as *hors-concours* participants and not properly distinguished.

Tables 6 and 7 list all validation and verification tools evaluated in SV-COMP 2025, respectively. The tables contain the tool name (with hyperlink),

Table 6: Participating validators with tool references, representing jury members, their affiliations, and indications of supported witnesses depending on format version and type; ∅ for inactive, new for first-time participants, *Cor.* for correctness witnesses, *Vio.* for violation witnesses, and ✓ for newly added support

				Witness Format			
				v1.0		v2.0	
Validator	**Ref.**	**Jury Member**	**Affiliation**	**Cor.**	**Vio.**	**Cor.**	**Vio.**
ConcurrentW2T	[13]	Z. Ádám	BME Budapest		✓		
CPAchecker	[40, 41]	M. Lingsch-Rosenfeld	LMU Munich	✓	✓	✓	✓
CPA-w2t∅	[37, 39]	–	–		✓		
CProver-w2t∅	[37, 39]	–	–		✓		
Dartagnan	[116]	H. Ponce de León	Huawei Dresden		✓		
Goblint	[122]	S. Saan	U. Tartu			✓	
GWIT∅	[85]	–	–		✓		
JCWIT∅	[56]	–	–	✓			
LIV	[45]	M. Lingsch-Rosenfeld	LMU Munich	✓		✓	
MetaVal	[43]	M. Lingsch-Rosenfeld	LMU Munich	✓	✓	✓	✓
MetaVal++new		M. Lingsch-Rosenfeld	LMU Munich			✓	
Mopsa	[106]	R. Monat	Inria & U. Lille			✓	
NITWIT∅	[136]	–	–		✓		
Symbiotic-Witch	[8]	P. Ayaziová	Masaryk U., Brno		✓		
UAutomizer	[36, 38]	M. Ebbinghaus	U. Freiburg	✓	✓	✓	✓
UReferee new		F. Schüssele	U. Freiburg	✓		✓	
Wit4Java	[135]	T. Wu	U. Manchester		✓		
Witch	[7, 9]	P. Ayaziová	Masaryk U., Brno				✓

Table 7: Participating verifiers with tool references, representing jury members and their affiliations; ∅ for inactive, meta for meta verifiers, and new for first-time participants

Verifier	**Ref.**	**Jury Member**	**Affiliation**
2LS	[48, 103]	V. Malík	BUT, Czechia
aise	[99, 132]	Z. Chen	NUDT, China
APROVE	[100]	N. Lommen	RWTH Aachen, Germany
BRICK	[49]	L. Bu	Nanjing U., China
Bubaak	[51, 53]	M. Chalupa	ISTA, Austria
Bubaak-SpLit	[52]	M. Chalupa	ISTA, Austria
CBMC∅	[58, 95]	–	–
COASTAL∅	[129]	–	–
CoOpeRace meta new		V. Vojdani	U. Tartu, Estonia
CPAchecker	[10, 11]	M. Lingsch-Rosenfeld	LMU Munich, Germany
CPALockator∅	[5, 6]	–	–

(continues on next page)

Table 7: Participating verifiers (continued)

Verifier	Ref.	Jury Member	Affiliation
CPA-bam-bnb ∅	[4, 131]	–	–
CPA-bam-smg ∅		–	–
CPV	[57]	P.-C. Chien	LMU Munich, Germany
Crux ∅	[69, 123]	–	–
CSeq ∅	[63, 89]	–	–
Dartagnan	[76, 115]	H. Ponce de León	Huawei Dresden, Germany
Deagle	[80]	F. He	Tsinghua U., China
DIVINE ∅	[15, 96]	–	–
EBF ∅	[3]	–	–
EmergenTheta	[12, 108]	L. Bajczi	BME Budapest, Hungary
ESBMC-incr	[59, 62]	T. Wu	U. Manchester, UK
ESBMC-kind	[75, 134]	T. Wu	U. Manchester, UK
Frama-C-SV ∅	[44, 64]	–	–
Gazer-Theta ∅	[1, 79]	–	–
GDart	[110]	F. Howar	TU Dortmund, Germany
GDart-LLVM ∅		–	–
Goblint	[121, 130]	S. Saan	U. Tartu, Estonia
Graves-CPA ∅ meta	[97]	–	–
Hornix new		M. Blicha	U. Lugano, Switzerland
Infer ∅	[50, 93]	–	–
Java-Ranger	[86, 125]	S. Hussein	Ain Shams U., Egypt
JayHorn ∅	[92, 124]	–	–
JBMC	[60, 61]	P. Schrammel	Diffblue, UK
JDart ∅	[102, 109]	–	–
Korn	[72, 73]	G. Ernst	LMU Munich, Germany
Lazy-CSeq ∅	[87, 88]	–	–
LF-checker ∅		–	–
Locksmith ∅	[117]	–	–
MLB		L. Bu	Nanjing U., China
Mopsa	[91, 107]	R. Monat	Inria & U. Lille, France
Nacpa meta new	[98]	H. Wachowitz	LMU Munich, Germany
PeSCo-CPA ∅ meta	[119, 120]	–	–
PIChecker ∅	[126]	–	–
Pinaka ∅	[55]	–	–
PredatorHP ∅	[83, 114]	–	–
Proton	[105, 111]	R. Metta	TCS, India
RacerF new	[65]	T. Dacík	BUT, Czechia
SPF ∅	[112, 118]	–	–
SVF-SVC new	[104]	M. Richards	U. New South Wales, AU
sv-sanitizers		S. Saan	U. Tartu, Estonia
SWAT	[101]	N. Loose	U. Luebeck, Germany
Symbiotic	[54, 90]	M. Jonáš	Masaryk U., Czechia

(continues on next page)

Table 7: Participating verifiers (continued)

Verifier	Ref.	Jury Member	Affiliation
Theta	[127, 128]	L. Bajczi	BME Budapest, Hungary
Thorn new		L. Bajczi	BME Budapest, Hungary
UAutomizer	[81, 82]	M. Heizmann	U. Freiburg, Germany
UGemCutter	[74, 94]	D. Klumpp	U. Freiburg, Germany
UKojak	[71, 113]	M. Bentele	U. Freiburg, Germany
UTaipan	[68, 78]	D. Dietsch	U. Freiburg, Germany
VeriAbs ∅	[2, 66]	–	–
VeriAbsL ∅	[67]	–	–
VeriOover ∅		–	–

Table 8: Algorithms and techniques used by the participating tools;
∅ for inactive, meta for meta verifiers, and new for first-time participants

Tool	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	Task Translation
2LS				✓	✓			✓	✓		✓						✓				
aise			✓																		
AProVE																	✓				
BRICK	✓		✓	✓				✓								✓					
Bubaak			✓								✓					✓	✓			✓	
Bubaak-SpLit			✓		✓						✓				✓	✓	✓		✓	✓	
CBMC ∅				✓							✓					✓					
COASTAL ∅			✓																		
ConcurrentW2T																✓					
CoOpeRace meta new																✓			✓	✓	
CPAchecker	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	
CPALockator ∅	✓	✓					✓				✓	✓	✓	✓		✓					
CPA-bam-bnb ∅	✓	✓					✓				✓	✓	✓	✓							
CPA-bam-smg ∅																					
CPA-w2t ∅						✓						✓			✓						
CProver-w2t ∅				✓																	
CPV	✓	✓		✓	✓	✓					✓			✓					✓	✓	✓
Crux ∅			✓																		
CSeq ∅				✓							✓					✓					

(continues on next page)

Table 8: Algorithms and techniques (continued)

Tool	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	Task Translation
DARTAGNAN				✓							✓					✓					
DEAGLE				✓							✓					✓					
DIVINE[∅]			✓				✓				✓					✓			✓	✓	
EBF[∅]				✓																	
EMERGENTHETA	✓	✓		✓	✓						✓			✓					✓	✓	✓
ESBMC-INCR				✓	✓						✓					✓					
ESBMC-KIND				✓	✓		✓	✓			✓					✓					
FRAMA-C-SV[∅]								✓													
GAZER-THETA[∅]	✓	✓		✓			✓				✓	✓	✓	✓						✓	
GDART			✓								✓									✓	
GDART-LLVM[∅]			✓								✓										
GOBLINT								✓								✓			✓		
GRAVES-CPA[∅] meta	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓	
GWIT[∅]			✓								✓									✓	
HORNIX new						✓															
INFER[∅]								✓	✓	✓										✓	
JAVA-RANGER			✓								✓										
JAYHORN[∅]	✓	✓				✓		✓					✓	✓							
JBMC				✓							✓					✓					
JCWIT[∅]				✓																	
JDART[∅]			✓								✓									✓	
KORN	✓	✓	✓				✓													✓	
LAZY-CSEQ[∅]				✓							✓					✓					
LF-CHECKER[∅]																					
LIV																					✓
LOCKSMITH[∅]																✓					
METAVAL																			✓		✓
METAVAL++ new																					✓
MLB			✓								✓									✓	
MOPSA								✓													
NACPA meta new																			✓	✓	
NITWIT[∅]							✓														
PESCO-CPA[∅] meta	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓	
PICHECKER[∅]	✓	✓									✓	✓		✓		✓			✓		

(continues on next page)

Table 8: Algorithms and techniques (continued)

Tool	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	Task Translation
PINAKA∅			✓	✓							✓										
PREDATORHP∅									✓												
PROTON				✓																	
RACERF new																✓					
SPF∅			✓						✓							✓					
SVF-SVC new			✓																		
SV-SANITIZERS																✓					
SWAT			✓																		
SYMBIOTIC			✓		✓			✓	✓		✓					✓				✓	
SYMBIOTIC-WITCH			✓																		
THETA	✓	✓		✓			✓				✓	✓		✓		✓			✓	✓	✓
THORN new		✓																	✓	✓	
UAUTOMIZER	✓	✓									✓		✓	✓	✓	✓	✓		✓	✓	
UGEMCUTTER	✓	✓									✓		✓	✓	✓	✓			✓	✓	
UKOJAK	✓	✓									✓		✓	✓							
UREFEREE new	✓	✓									✓		✓	✓	✓	✓	✓		✓	✓	
UTAIPAN	✓	✓					✓	✓			✓		✓	✓	✓	✓			✓	✓	
VERIABS∅	✓			✓	✓		✓	✓										✓	✓	✓	
VERIABSL∅	✓			✓	✓		✓	✓										✓	✓	✓	
VERIOOVER∅																					
WIT4JAVA																					
WITCH			✓																		

references to papers that describe the tool, the representing jury member and the affiliation. The listings are also available on the competition web site at https://sv-comp.sosy-lab.org/2025/systems.php. Table 6 additionally indicates the witness formats and witness kinds supported by individual validators and whether they were supported in SV-COMP 2025 for the first time (✓) or not (✓).

Table 8 lists the algorithms and techniques used by the verification and validation tools (some techniques were omitted due to limited space). Further, Table 9 gives an overview of common solver libraries and frameworks used by these tools. Note that both tables are based on information provided by teams that registered individual tools to SV-COMP 2025 or to some previous editions in the case of currently inactive tools. The web site https://fm-tools.sosy-lab.org

Table 9: Solver libraries and frameworks used as components by at least three participating tools; tools that did not declare any used library or framework are omitted, ∅ for inactive, meta for meta verifiers, and new for first-time participants

Tool	CPAchecker	CProver	Esbmc	Jpf	Ultimate	JavaSMT	MathSAT	Cvc	SMTinterpol	Z3	MiniSAT	Apron
2LS		✓									✓	
aise								✓		✓		
APRoVE										✓		
BRICK										✓	✓	
Bubaak										✓		
CBMC∅		✓		✓							✓	
COASTAL∅						✓	✓					✓
CPAchecker	✓					✓	✓					
CPALockator∅	✓					✓	✓					
CPA-bam-bnb∅	✓					✓	✓					
CPA-bam-smg∅	✓					✓	✓					
CPV											✓	
Crux∅										✓		
CSeq∅		✓									✓	
Dartagnan						✓						
Deagle											✓	
EBF∅			✓				✓					
ESBMC-incr			✓				✓					
ESBMC-kind			✓				✓					
GDart								✓		✓		
GDart-LLVM∅										✓		
Goblint												✓
Graves-CPA∅ meta	✓					✓	✓					
Hornix new										✓		
Java-Ranger				✓								
JBMC		✓									✓	
JDart∅				✓				✓		✓		
Korn										✓		
Lazy-CSeq∅		✓									✓	
Mopsa												✓
Nacpa meta new	✓											
PeSCo-CPA∅ meta	✓					✓	✓					
PIChecker∅	✓					✓	✓		✓			
SPF∅				✓								
SWAT										✓		

(continues on next page)

Table 9: Solver libraries and frameworks (continued)

Tool	CPAchecker	CProver	Esbmc	Jpf	Ultimate	JavaSMT	MathSAT	Cvc	SMTInterpol	Z3	MiniSAT	Apron
Symbiotic										✓		
UAutomizer					✓		✓	✓	✓	✓		
UGemCutter					✓		✓	✓	✓	✓		
UKojak					✓				✓	✓		
UReferee [new]					✓		✓	✓	✓	✓		
UTaipan					✓		✓	✓	✓	✓		
VeriAbs [∅]	✓	✓								✓	✓	
VeriAbsL [∅]	✓	✓								✓	✓	

provides more information about all tools evaluated in SV-COMP and Test-Comp since 2023 in a uniform way.

Figure 2 shows the evolution of the number of verifiers and validators participating in individual editions of SV-COMP. It seems that the number of actively participating tools, in particular verifiers, is now stable.

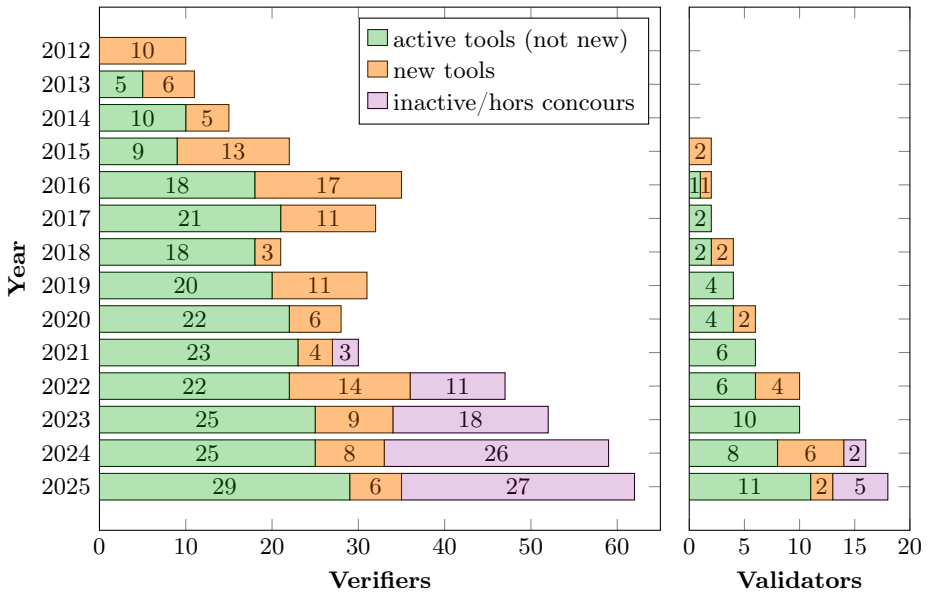


Fig. 2: Number of evaluated verifiers and validators in each year of SV-COMP; three new hors concours tools of 2022 counted only as new, not as hors concours

5 Results

The results of the competition represent 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* (the number of verification tasks that can be solved and correctness of the results, as accumulated in the score) and the *efficiency* (resource consumption in terms of CPU time). 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 provided to the participants in advance and their objections have been settled.

Consumed Resources. Before we present the competition results, we report some statistics to give an impression of the overall computation work: One complete execution of all verifiers in the competition consisted of 942 284 verification runs (each verifier runs on each verification task of the categories listed in the verifier's *benchmark definition*), consuming 2 312 days of CPU time (without validation). Witness validation required 21.8 million validation runs (each validator runs on each validation task of the categories listed in the validator's *benchmark definition*) consuming 2 573 days of CPU time. Moreover, each tool was executed several times, in order to make sure no installation issues occur during the execution.

Verification track. Tables 10 and 11 present the quantitative overview of all tools and all meta categories except the meta categories included in *FalsificationOverall*. We split the presentation into two tables, one for the verifiers that participate with an active team support (Table 10), and one for the inactive verifiers (Table 11). The head row lists meta categories with the number of valid tasks and the maximal score for each category. The tools are listed in alphabetical order; every table row lists the scores of one verifier. An empty table cell means that the verifier did not participate in the corresponding meta category. In Table 10, we indicate the top three candidates in each category by formatting their scores in bold face and in larger font size. We recall that inactive tools and meta verifiers are excluded from rankings. 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/2025/results`) and in the results artifact (see Table 5). Note that the results for subcategories of *FalsificationOverall* are not explicitly presented neither in this report nor on the web due to their marginal significance. The results can be obtained from the detailed results of the corresponding categories in *Overall* presented on the web.

Table 12 shows the medalists for each meta category. The column 'Solved Tasks' shows the number of tasks that the respective verifier solved correctly and produced a witness that was confirmed (or not required). The cumulative run time of the verifier on these tasks is presented in the column 'CPU Time'. The column 'Unconf. Tasks' indicates the number of tasks for which the verifier returned a correct answer, but the corresponding witness was not confirmed by any validator. The columns 'False Alarms' and 'Wrong Proofs' provide the number of verification tasks for which the verifier reported wrong results, i.e., reporting a property

Table 10: Overview of the results of all actively participating verifiers; empty cells indicate opt-outs, meta for meta verifiers, new for first-time participants

Participant	**ReachSafety** 11268 tasks max. score 17860	**MemSafety** 4042 tasks max. score 6409	**ConcurrencySafety** 3175 tasks max. score 5733	**NoOverflows** 8211 tasks max. score 13297	**Termination** 2328 tasks max. score 4079	**SoftwareSystems** 4329 tasks max. score 7131	**FalsificationOverall** 30758 tasks max. score 10675	**Overall** 33353 tasks max. score 55561	**JavaOverall** 673 tasks max. score 926
2LS	6053	686	0	6887	1703	1	1938	12658	
AISE									
APROVE					**2219**				
BRICK									
BUBAAK	6684	3355	-190	6572	1491	1697	**5565**	17768	
BUBAAK-SPLIT	6053	3349	-189	6556	1122	1649	5562	16497	
COOPERACE meta new									
CPACHECKER	**10368**	**4892**	1770	8777	1301	**2178**	**6999**	**26786**	
CPV	**7755**								
DARTAGNAN			**3385**						
DEAGLE			**4604**						
EMERGENTHETA	2106			-361	620				
ESBMC-INCR			2155						
ESBMC-KIND	**8717**	3158	2155	8668	1115	-1948	3741	18444	
GOBLINT	2427	2198	2448	8486	969	545		17266	
HORNIX new									
KORN									
MOPSA	2807	2697	0	8491	0	**2086**		13521	
NACPA meta new	10270	4887	1453	8843	1290	2127	6997	26131	
PROTON					**3685**				
RACERF new									
SV-SANITIZERS		861		1723					
SVF-SVC new	-68717	-10965		-19469		0			
SYMBIOTIC	7097	**4479**	60	7704	1411	**1822**	**6459**	**20691**	
THETA	3277	600	2275	-170	712				
THORN new	-2519		298	-286	536				
UAUTOMIZER	5666	**3909**	2993	**11074**	**3334**	654	4762	**29710**	
UGEMCUTTER			**3144**						
UKOJAK	4935	2939	0	**8878**	0	300	3848	12872	
UTAIPAN	6007	3711	2593	**10736**	0	288	4695	20244	
GDART									**627**
JAVA-RANGER									**676**
JBMC									**628**
MLB									579
SWAT									508

Table 11: Overview of the results of all inactively participating verifiers; empty cells indicate opt-outs, ∅ for inactive, meta for meta verifiers

Verifier	**ReachSafety** 11268 tasks max. score 17860	**MemSafety** 4042 tasks max. score 6409	**ConcurrencySafety** 3175 tasks max. score 5733	**NoOverflows** 8211 tasks max. score 13297	**Termination** 2328 tasks max. score 4079	**SoftwareSystems** 4329 tasks max. score 7131	**FalsificationOverall** 30758 tasks max. score 10675	**Overall** 33353 tasks max. score 55561	**JavaOverall** 673 tasks max. score 926
CBMC ∅	1330	1885	819	7200	1199	-2586	-3581	9100	
CPA-bam-bnb ∅						-2370			
CPA-bam-smg ∅		3249				-4079			
CPALockator ∅			-4967						
Crux ∅	2133			608					
CSeq ∅			-12720						
DIVINE ∅	4629	502	351	0	0	72	352	3680	
EBF ∅			360						
Frama-C-SV ∅				1573					
Gazer-Theta ∅									
GDart-LLVM ∅									
Graves-CPA ∅ meta	4041					-670	-820	4508	
Infer ∅	-96489		-8970	-76213		-32987			
Lazy-CSeq ∅			-15153						
LF-checker ∅			396						
Locksmith ∅									
PeSCo-CPA ∅ meta	6269					-1598	2435	16328	
PIChecker ∅			459						
Pinaka ∅	2448			958	922				
PredatorHP ∅		4733							
VeriAbs ∅	11012								
VeriAbsL ∅	11224								
VeriOover ∅									
WitnessMap									
COASTAL ∅									-3960
JayHorn ∅									248
JDart ∅									-1224
SPF ∅									184

violation when the property holds (false alarm) and claiming that the program satisfies the property although it actually violates it (wrong proof), respectively.

Score-Based Quantile Functions. We use score-based quantile functions [18, 42] because these visualizations make it easier to understand the results of the

Table 12: Verification: Overview of the medalists in each meta category; values for CPU time in hours and rounded to two significant digits

Category Rank	Verifier	Score	CPU Time	Solved Tasks	Unconf. Tasks	False Alarms	Wrong Proofs
ReachSafety (11268 tasks, max. score 17860)							
1	**CPAchecker**	**10368**	150	6 653	230	2	
2	ESBMC-kind	8717	69	6 830	599		**14**
3	CPV	7755	160	6 235	438	27	
MemSafety (4042 tasks, max. score 6409)							
1	**CPAchecker**	**4892**	18	3 818	1		
2	Symbiotic	4479	2.3	3 671	0		**1**
3	UAutomizer	3909	37	2 280	2		**1**
ConcurrencySafety (3175 tasks, max. score 5733)							
1	**Deagle**	**4604**	3.1	2 500	38	1	**4**
2	Dartagnan	3385	17	2 012	30	3	**3**
3	UGemCutter	3144	50	1 805	48		
NoOverflows (8211 tasks, max. score 13297)							
1	**UAutomizer**	**11074**	68	6 724	13		
2	UTaipan	10736	74	6 622	11	1	**2**
3	UKojak	8878	54	5 910	2		
Termination (2328 tasks, max. score 4079)							
1	**Proton**	**3685**	24	1 942	159	1	
2	UAutomizer	3334	18	1 667	4		
3	AProVE	2219	32	1 006	43		
SoftwareSystems (4329 tasks, max. score 7131)							
1	**CPAchecker**	**2178**	30	2 022	55		
2	Mopsa	2086	20	2 212	0		
3	Symbiotic	1822	6.1	1 487	231		**1**
FalsificationOverall (30758 tasks, max. score 10675)							
1	**CPAchecker**	**6999**	100	7 100	67	2	
2	Symbiotic	6459	28	6 379	37		
3	Bubaak	5565	18	5 739	236	9	
Overall (33353 tasks, max. score 55561)							
1	**UAutomizer**	**29710**	270	16 677	196		**8**
2	CPAchecker	26786	240	20 506	372	6	**1**
3	Symbiotic	20691	63	14 324	628		**3**
JavaOverall (673 tasks, max. score 926)							
1	**Java-Ranger**	**676**	5.7	491	13		**1**
2	JBMC	628	0.32	430	91		
3	GDart	627	2.1	460	15		

comparative evaluation. The results archive (see Table 5) and the web site (`https://sv-comp.sosy-lab.org/2025/results`) include such a plot for each category. As an example, we show the plot for category *Overall* (all verification tasks in C) in Fig. 3. A total of 16 verifiers participated in category *Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [18]). A more detailed discussion of score-based quan-

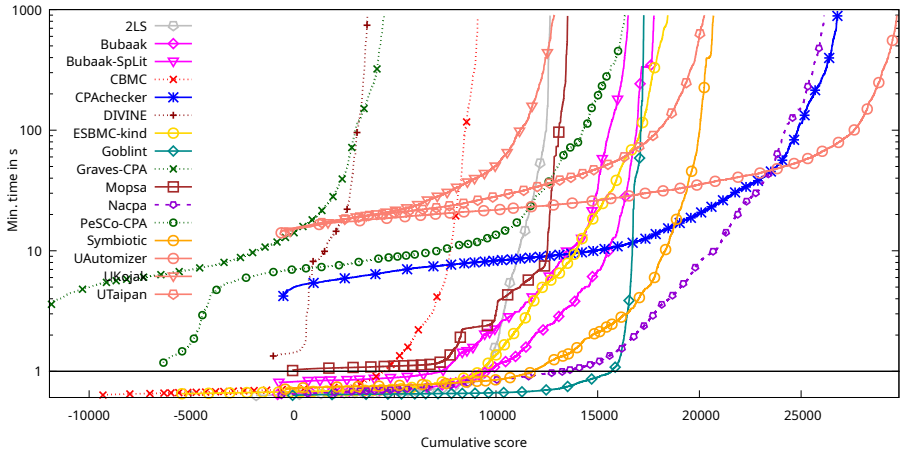


Fig. 3: Quantile functions for category *Overall*. Each quantile function illustrates the quantile (x-coordinate) of the scores obtained by correct verification runs below a certain run time (y-coordinate), minus the overall penalty for incorrect results. More details were given previously [18]. A logarithmic 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.

tile plots, including examples of what insights one can obtain from the plots, is provided in previous competition reports [18, 21]. Since this year we use different lines to distinguish regular active participants (solid line), active meta verifiers (dashed line), and inactive verifiers (dotted line).

The graph shows that the *Overall* winner is UAutomizer as its graph end most to the right. The graph for UAutomizer also starts left from $x = 0$ because the verifier produced 8 wrong proofs and therefore received some negative points. Also other verifiers whose graphs start with a negative cumulative score produced wrong results.

Validation Track. Validation of verification witnesses was pioneered by SV-COMP in 2015 (by the tools CPAchecker and CBMC$^{\varnothing}$, see the documentation for the early attempts: `https://sv-comp.sosy-lab.org/2015/witnesses/`). Shortly after that, verification witnesses become more and more important for various reasons: they do not only justify and help to understand and interpret verification results, but they also serve as exchange object for intermediate results and allow to make use of imprecise verification techniques (e.g., via machine learning). However, a case study on the quality of the results of witness validators [46] published in 2022 revealed a great potential for improvements. To stimulate further advances in verification witnesses and their verifiers, the study suggested that witness validators should also undergo a periodical comparative evaluation and proposed a scoring schema for witness-validation results. This materializes in the validation track run by SV-COMP since 2023.

Table 13: Validation of correctness witnesses (version 1.0): Overview of the medalists in each meta category; values for CPU time in hours and rounded to two significant digits, new for first-time participants

Rank	Validator	Score	CPU Time	Solved Tasks	Wrong Confirm.	Wrong Refut.
Category						
ReachSafety (39573 tasks, max. score 70152)						
1	UAutomizer	**52303**	390	16 576		4
2	MetaVal	49340	750	24 887		53
3	CPAchecker	40427	460	33 426		
NoOverflows (38846 tasks, max. score 77692)						
1	UAutomizer	**77578**	280	38 812		
2	CPAchecker	75470	140	38 181		
3	LIV	16438	5.6	4 920		
SoftwareSystems (13913 tasks, max. score 25507)						
1	UAutomizer	**20171**	230	13 572		
2	MetaVal	15487	290	13 783		5
3	UReferee new	14072	48	5 399		
Overall (92332 tasks, max. score 172540)						
1	UAutomizer	**146763**	900	68 960		4
2	CPAchecker	113930	620	76 991		
3	MetaVal	72634	1 000	38 670		58

Thanks to the development and adoption of the version 2.0 of the witness format [7], the validation track has now four regular subtracks:

1. validation of correctness witnesses in format version 1.0 (see Table 13),
2. validation of violation witnesses in format version 1.0 (see Table 14),
3. validation of correctness witnesses in format version 2.0 (see Table 15), and
4. validation of violation witnesses in format version 2.0 (see Table 16).

The subtrack with violation witnesses 2.0 ran for the first time as a regular part of the competition as it was only a demonstration subtrack in SV-COMP 2024. The tables present only the medalists for all non-empty meta categories of each subtrack. If some category has less than 3 medalists, it means that less than three validators reached a positive score. The column 'Wrong Confirm.' gives the number of cases when the respective validator confirmed a witness that was incorrect. The column 'Wrong Refut.' shows the number of cases when the respective validator refuted a correct witness. We recall that the correctness or incorrectness of many witnesses used in the validation track is determined by voting and thus the numbers of wrong confirmations and refutation do not have to be completely objective.

The complete results of all validators in all relevant categories of all subtracks are available in the results artifact (see Table 5) and on the SV-COMP web site (`https://sv-comp.sosy-lab.org/2025/results/results-validated/`).

The limited support of some properties and program features by the witness formats (see Table 1), missing medalists in some categories, and the negative scores

Table 14: Validation of violation witnesses (version 1.0): Overview of the medalists in each meta category; values for CPU time in hours and rounded to two significant digits

Category Rank	Validator	Score	CPU Time	Solved Tasks	Wrong Confirm.	Wrong Refut.
ReachSafety (20391 tasks, max. score 29907)						
1	ConcurrentW2T	**1931**	2.2	2 734	4	
MemSafety (8810 tasks, max. score 13215)						
1	CPAchecker	**4455**	19	8 638	22	27
ConcurrencySafety (755 tasks, max. score 1132)						
1	Dartagnan	**777**	2.9	693		
2	CPAchecker	511	1.6	531		3
3	UAutomizer	478	4.9	567		104
NoOverflows (17697 tasks, max. score 26546)						
1	Symbiotic-Witch	**3926**	14	13 326	185	2
Termination (2057 tasks, max. score 2314)						
1	CPAchecker	**2314**	9.5	2 057		
2	UAutomizer	2314	16	2 057		
SoftwareSystems (1334 tasks, max. score 2001)						
1	CPAchecker	**1356**	5.8	1 086		1
2	Symbiotic-Witch	586	0.38	653	1	3
3	MetaVal	535	5.6	633	9	
Overall (51044 tasks, max. score 73092)						
-	demo category (less than three participants)					

in the detailed results on the web site: all of these show the need of further research and development in the are of software verification witnesses and their validation.

6 Conclusion

The 14th edition of the Competition on Software Verification (SV-COMP 2025) compared 62 automatic tools for software verification (including 6 new ones and 27 tools without active team support) and 18 automatic tools for validation of verification witnesses (including 2 new and 5 tools without active team support). The overall numbers of evaluated verifiers and validators were historically the highest (see Fig. 2). The total number of verification tasks in SV-COMP 2025 for both C and Java programs was significantly increased to precisely 34 700 (including one demo category).

The results of the competition show a progress in both verification and witness validation area, especially in the adoption of the witness format 2.0. However, even the best verification and validation tools still produce some incorrect results. This motivates further improvements of verifiers, but also extensions of the witness format to support more properties and program features, and development of validators.

Table 15: Validation of correctness witnesses (version 2.0): Overview of the medalists in each meta category; values for CPU time in hours and rounded to two significant digits

Category Rank	Validator	Score	CPU Time	Solved Tasks	Wrong Confirm.	Wrong Refut.
ReachSafety (19365 tasks, max. score 28241)						
1	UAutomizer	**16085**	200	7 719	2	10
2	CPAchecker	10378	180	14 467	2	
3	LIV	6829	50	12 667	16	
NoOverflows (26686 tasks, max. score 40029)						
1	UAutomizer	**32637**	270	23 890		93
2	CPAchecker	26940	60	18 931		
3	Mopsa	25022	12	21 430		
SoftwareSystems (7581 tasks, max. score 11913)						
1	Mopsa	**7751**	29	6 948		
2	CPAchecker	5092	12	2 474		
3	MetaVal	3182	60	2 880	2	
ValidationCrafted (3 tasks, max. score 6)						
1	UAutomizer	**6**	0.043	3		
2	CPAchecker	6	0.16	3		
Overall (53635 tasks, max. score 87557)						
1	UAutomizer	**57804**	580	37 976	3	134
2	CPAchecker	56546	260	35 875	2	
3	Mopsa	30743	95	32 857		

Data-Availability Statement. The verification tasks and results of the competition are published at Zenodo, as described in Table 5. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Table 4. For easy access, the results are presented also online on the competition web site `https://sv-comp.sosy-lab.org/2025`. The main results of SV-COMP 2025 were reproduced in an independent reproduction report [14].

Funding Statement. SV-COMP 2025 was supported by Huawei – Dresden Research Center, Germany. Some participants of this competition were funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY). Jan Strejček has been supported by the Czech Science Foundation grant GA23-06506S.

Acknowledgments. We thank the organization committee for their help in running the competition, the jury for their work on improving the quality of the verification tasks and for their advice in refining and apply the competition rules, and the verification community for contributing their tools to the evaluation.

Table 16: Validation of violation witnesses (version 2.0): Overview of the medalists in each meta category; values for CPU time in hours and rounded to two significant digits

Category Rank	Validator	Score	CPU Time	Solved Tasks	Wrong Confirm.	Wrong Refut.
ReachSafety (3548 tasks, max. score 4494)						
1	Witch	**3259**	6.1	3 305		
2	CPAchecker	3081	15	3 083	8	
MemSafety (107 tasks, max. score 147)						
1	UAutomizer	**147**	0.53	107		
2	Witch	104	0.040	84		
3	CPAchecker	43	0.043	23		
NoOverflows (4706 tasks, max. score 5882)						
1	UAutomizer	**5147**	27	4 691	1	
2	CPAchecker	5010	11	4 475		
3	Witch	3900	4.7	4 584		
SoftwareSystems (148 tasks, max. score 222)						
1	Witch	**222**	0.12	148		
2	CPAchecker	74	0.70	140		
3	MetaVal	73	0.97	135		
ValidationCrafted (100 tasks, max. score 150)						
1	Witch	**150**	0.042	100		
Overall (8609 tasks, max. score 11866)						
1	Witch	**9850**	11	8 221		
2	MetaVal	162	11	1 846	2	51

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. pp. 184–203. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_11
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 (3). 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., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
11. 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 (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
12. 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 (3). pp. 371–375. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_23
13. Bajczi, L., Ádám, Z., Micskei, Z.: ConcurrentWitness2Test: Test-harnessing the power of concurrency (competition contribution). In: Proc. TACAS (3). pp. 330–334. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_16
14. Bajczi, L., Ádám, Z., Micskei, Z.: SV-COMP'25 reproduction report (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
15. 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
16. 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
17. 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
18. 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
19. 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
20. 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
21. 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

22. 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
23. 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
24. 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
25. 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
26. 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
27. 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
28. 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
29. 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
30. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
31. Beyer, D.: FM-Tools Release 2.2: Data set of metadata about tools for formal methods (SV-COMP 2025, Test-Comp 2025). Zenodo (2025). https://doi.org/10.5281/zenodo.15055359
32. Beyer, D., Strejček, J.: Results of the 14th Intl. Competition on Software Verification (SV-COMP 2025). Zenodo (2025). https://doi.org/10.5281/zenodo.15012085
33. Beyer, D., Strejček, J.: SV-Benchmarks: Benchmark set for software verification (SV-COMP 2025). Zenodo (2025). https://doi.org/10.5281/zenodo.15012096
34. Beyer, D., Strejček, J.: Verification witnesses from verification tools (SV-COMP 2025). Zenodo (2025). https://doi.org/10.5281/zenodo.15012077
35. Beyer, D., Chien, P.C., Jankola, M.: BenchCloud: A platform for scalable performance benchmarking. In: Proc. ASE. pp. 2386–2389. ACM (2024). https://doi.org/10.1145/3691620.3695358
36. 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
37. 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
38. 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
39. 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

40. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_26
41. Beyer, D., Lingsch-Rosenfeld, M.: CPAchecker Validator 4.0 (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
42. 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
43. 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
44. 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
45. 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
46. 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
47. Beyer, D., Wachowitz, H.: Coveriteam Release 1.2.1. Zenodo (2024). https://doi.org/10.5281/zenodo.11193690
48. 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
49. 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
50. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
51. 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
52. Chalupa, M., Richter, C.: Bubaak-SpLit: Split what you cannot verify (competition contribution). In: Proc. TACAS (3). pp. 353–358. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_20
53. Chalupa, M., Richter, C.: Bubaak: Dynamic cooperative verification (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
54. 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
55. 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
56. Cheng, Z., Wu, T., Schrammel, P., Tihanyi, N., de Lima Filho, E.B., Cordeiro, L.C.: JCWIT: A correctness-witness validator for Java programs based on bounded model checking. In: Proc. ISSTA. pp. 1831–1835. ACM (2024). https://doi.org/10.1145/3650212.3685303

57. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
58. 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
59. 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
60. 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
61. 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
62. 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
63. 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
64. 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
65. Dacík, T., Vojnar, T.: RacerF: Data race detection with Frama-C (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
66. 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
67. 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
68. 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
69. 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
70. 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
71. 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
72. 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

73. 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
74. 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
75. 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
76. 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
77. 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
78. 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
79. 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
80. 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
81. 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 (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
82. 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
83. 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
84. 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
85. 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
86. 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
87. 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

88. 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
89. 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
90. 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 (3). pp. 406–411. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_29
91. 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)
92. 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
93. 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
94. 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
95. 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
96. 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
97. 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
98. Lemberger, T., Wachowitz, H.: Nacpa: Native checking with parallel-portfolio analyses (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
99. Lin, Y., Chen, Z., Wang, J.: AISE v2.0: Combining loop transformations (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
100. Lommen, N., Giesl, J.: AProVE (KoAT + LoAT) (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
101. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT: Modular dynamic symbolic execution for Java applications using dynamic instrumentation (competition contribution). In: Proc. TACAS (3). pp. 399–405. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_28
102. 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
103. 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

104. McGowan, C., Richards, M., Sui, Y.: SVF-SVC: Software verification using SVF (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
105. Metta, R., Karmarkar, H., Madhukar, K., Venkatesh, R., Chakraborty, S.: Proton: Probes for non-termination and termination (competition contribution). In: Proc. TACAS (3). pp. 393–398. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_27
106. 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 (3). pp. 387–392. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_26
107. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-C with trace partitioning and autosuggestions (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
108. Mondok, M., Bajczi, L., Szekeres, D., Molnár, V.: EmergenTheta: Variations on symbolic transition systems (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
109. 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
110. 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
111. Mukhopadhyay, D., Metta, R., Karmarkar, H., Madhukar, K.: PROTON 2.1: Synthesizing ranking functions via fine-tuned locally hosted LLM (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
112. 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
113. 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
114. 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
115. 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
116. 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
117. 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
118. 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

119. 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
120. 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
121. 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 (3). pp. 381–386. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_25
122. 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 (3). pp. 335–340. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_17
123. 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
124. 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
125. 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
126. 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
127. Telbisz, C., Bajczi, L., Szekeres, D., Vörös, A.: Theta: Various approaches for concurrent program verification (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
128. 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
129. 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
130. 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
131. 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
132. Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 347–352. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_19
133. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.29. Zenodo (2025). https://doi.org/10.5281/zenodo.15007216

134. Wu, T., Li, X., Manino, E., Menezes, R., Gadelha, M., Xiong, S., Tihanyi, N., Petoumenos, P., Cordeiro, L.: ESBMC v7.7: Efficient concurrent software verification with scheduling, incremental SMT and partial order reduction (competition contribution). In: Proc. TACAS (3). LNCS 15698, Springer (2025)
135. 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
136. 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

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

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

SV-COMP'25 Reproduction Report (Competition Contribution)

Levente Bajczi(✉), Zsófia Ádám, and Zoltán Micskei

Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics, Budapest, Hungary
{bajczi,adamzsofi,micskeiz}@mit.bme.hu

Abstract. The International Competition on Software Verification (SV-COMP) has been an important driver of progress in the formal verification community, fostering tool development, benchmarking, and reproducibility. As the competition grows in scale and complexity, a reproducibility study is essential to evaluate its robustness across environments, uncover hidden dependencies, and ensure long-term sustainability. This work aims to reaffirm the reliability of SV-COMP's results, provide insights for similar competitions, and facilitate the adoption of its infrastructure beyond the competition. We reproduced the verification and validation results of active participants, including score and ranking calculations for the verification track. We found several problems prohibiting reusability and reproducibility of some participating tools, but we did not find serious issues with the competition infrastructure itself.

1 Introduction and Goals

The International Competition on Software Verification (SV-COMP) [2] is a long-running driving force in the software verification community, with a continuously expanding portfolio of benchmarks and tools.

Reproducibility, reusability, and availability were always a key priority. However, the size and complexity of SV-COMP mean a complex technological stack for benchmarking. Compared to just running a verifier on a program, instead, it is multiple tens of tools, each run on a subset of thousands of programs, multiplied by the number of validators that have to be executed on the results of these verifiers. This necessitates scaling up to a large cluster of machines. With each step, technological complexity grows.

There has been a partial reproduction of SV-COMP'23 [4], but we believe that a full reproducibility study of SV-COMP is due, to:

- evaluate reproducibility across different environments,
- provide an overview of the tasks and solutions involved, offering insights valuable to other competition organizers,
- facilitate the adoption of this technological stack for applications beyond SV-COMP, e.g., benchmarking by researchers and tool developers,
- identify hidden issues, e.g., hidden dependencies or missing documentation,
- ensure that the competition can endure changes in organization,
- enhance trust in SV-COMP's results by reaffirming their reliability.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 187–191, 2025.
https://doi.org/10.1007/978-3-031-90660-2_10

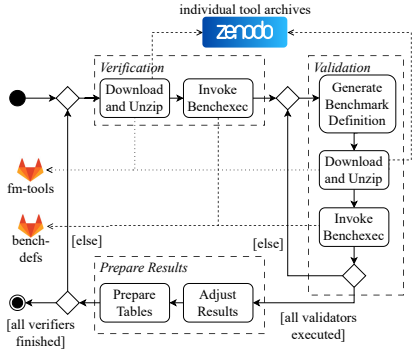


Fig. 1: Reproduction Workflow

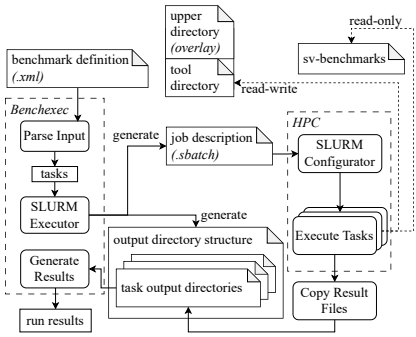


Fig. 2: Benchmarking Workflow

2 Reproduction Workflow

We did not use the official reproduction packages for SV-COMP'25, as at the time of writing, they are not yet finalized. Instead, we executed our experiments in parallel with the competition. This should not influence the validity of our findings, but some highlighted problems might have been already solved.

We relied on the official execution/reproduction guide for SV-COMP[1], which provides scripts to run the verification tasks, run the validation tasks, and prepare the tables for the results. It is referenced by the benchmark definition repository[2], which contains the `.xml` files used for running BenchExec [3] for both the verifiers and the validators; and the `category-structure.yml` file, which defines the participation and role of each tool in the competition. This repository also references the Formal-Methods Tools repository[3], which contains further metadata (such as necessary packages) as well as a DOI referencing each version of each tool. The execution scripts use this DOI to download the tool binaries and execute them against the benchmarks[4]. The reproduction workflow can be seen (summarized) in Figure 1.

We used the Hungarian *Komondor* cluster[5], which consists of computation nodes with two 64-core AMD EPYC 7763 CPUs and 256 GB of system memory each, running Red Hat Enterprise Linux 8.6. We requested array jobs via the SLURM Workload Manager with resource allocations outlined in SV-COMP's rules (i.e., 15 GB memory, 4 CPU cores, and 15 CPU-minutes for each verification task), with multiple tasks aggregated in the same job both sequentially and in parallel, in a Singularity[6] container based on Ubuntu 24.04. RUNEXEC was used to sequester the tasks to non-overlapping CPU cores.

We used the `fuse-overlayfs` package to create an overlay file system in the containers. For performance and stability reasons, we found that only mapping

[1] benchmarking/competition-scripts
[2] sv-comp/bench-defs
[3] benchmarking/fm-tools
[4] benchmarking/sv-benchmarks
[5] hpc.kifu.hu/komondor
[6] sylabs/singularity

	2ls	aise	aprove	brick	bubaak	bubaak-split	cpachecker	cpv	dartagnan	deagle	emergentheta	esbmc-incr	esbmc-kind	gdart	goblint	hornix	java-ranger	jbmc	korn	mlb	mopsa	proton	racerf	svf-svc	sv-sanitizers	swat	symbiotic	theta	thorn	uautomizer	ugemcutter	ukojak	utaipan
Conc.Safety	!4				?	?	?		!1	?		✓	✓		!2						!13						?	✓	?	✓	✓	?	✓
MemSafety	!4				?	?	✓						✓		!2						!13			!3	!4		✓	✓		?		?	?
NoOverflows	!4				✓	✓	✓				!5		✓		!2						!13			!3	!4		✓	!5	!5	✓		✓	✓
ReachSafety	!4	?		!12	✓	✓	✓	!7			✓		✓		!2	✓			!1		!13			!3			✓	✓	?	✓		✓	✓
Termination	!4		!9		✓	✓	✓				!5		✓		!2						!13	✓					✓	!5	!5	✓		?	?
JavaOverall														✓			!10	✓		!10 !13					!11								

Missing packages:
[1]`java` [2]`libmpfr` [3]`clang` [4]`gcc` [5]`clang-format` [6]`pandas` [7]`ply,pycparser` [8]`libfile-copy-link-perl` [9]`libbsd` [10]`javac` [11]`python3` [12]`libgfortran5`

Other:
[13]*read-only* sv-benchmarks

Fig. 3: Possible causes of failure. "?" denotes uninterpretable logs. Tools with "!" or "?" failed to run in the category, tools with ✓ were at least partly successful.

the current working directory as read-write is the best option, as we expect to find a witness there. Mapping the entire directory hierarchy (including sv-benchmarks with almost 200 000 files) as read-write caused instability problems when run on multiple hundreds of nodes from the same network filesystem, often causing `fuse` to become stuck in uninterruptible sleep. Therefore, all directories above the tool directory (the working directory in SV-COMP) are mapped read-only. The benchmarking workflow can be seen summarized in Figure 2.

We only ran non-*hors-concours* and non-inactive tools participating in SV-COMP'25. This resulted in running 33 verifiers (for a list, see Figure 3) and 33 validators. The results are published as a Zenodo artifact [1]. The tables are hosted at `home.mit.bme.hu/~bajczi/sv-comp-repro-25`. The ranking summary is available at `results-verified/scoretable.html`. The rankings correspond to the official rankings of verifiers (excluding tools with problems, see section 3). Together, 38 358 tasks have mismatching verdict categories out of the more than 495 828 tasks (19 222 due to errors detailed in section 3, but 19 136 due to other factors, mostly by reaching resource limits at different times). 1 officially correct result flipped to wrong (`mlb`), and 24 officially wrong results flipped to correct (`mlb`, `swat`). Compared to the official results, tasks correctly solved in both environments used 24% less walltime, 32% less cputime, and 11% more memory in the reproduction environment. Detailed statistics and differences are published on Zenodo [1].

3 Conclusion and Recommendations

Besides some minor issues that we submitted solutions to via merge requests (`benchmarking/competition-scripts!160`, `!161`), we did not find any problems in the competition scripts that would impede the reproduction workflow. Our only recommendation is to extend the documentation with details on how a competition can be run – e.g., how to run ranking calculations or run pre-runs by restricting resource allocation.

However, multiple tools produced no correct results, misrepresenting their actual SV-COMP performance. This stems from differences between SV-COMP's

execution environment and our reproduction attempt. Solving these issues is key to enhancing SV-COMP's reliability and simplifying independent tool execution.

Missing Packages. Tool developers declare the packages they need alongside the tool archives, but many tools leave out necessary packages, or declare non-existent ones. In the reproduction environment, we installed all the packages explicitly declared by the tools. Consequently, some tools had access to missing dependencies if they were declared by others. To uncover how widespread this issue is, we ran a separate experiment, executing each tool in each major category (opt-ins were left out) with just the declared packages of the respective tools installed. The results can be seen in Figure 3, and in the Zenodo archive [1].

Assumed Read-Write Access. Some tools tried to overwrite input files. While this is not against the rules of SV-COMP, we feel this goes against the purpose of making verification tools accessible to people outside of SV-COMP: if a verification tool modifies the input files it verifies, users may (justifiably) avoid using it. Such problems are also denoted in the table in Figure 3.

Result Files Pattern Mismatch. The benchmark definition files specify which files to keep after verification. Some tools do not include their own witness filenames (e.g., DEAGLE produces an `error-witness.graphml` file, but the definition only keeps `**/witness.*` files). The competition environment presumably disregards this directive and keeps every file (otherwise, the tools in question could not get their witnesses validated). However, this means that the benchmark definition file could not be reused outside of SV-COMP.

Working Directory Must Be Tool Directory. Some tool developers presume that BenchExec will always be executed from the tool directory. Offending tools mostly rely on relative paths holding from the working directory rather than prepending the directory of the tool archive to these paths. This makes it hard to use the tool binaries outside of SV-COMP and benchmarking.

3.1 Recommendations

In order to address the uncovered problems, we recommend to make the following modifications to SV-COMP:

- Enforce package declaration by a CI-check for the Formal-Methods Tools repository (preferred) or part of the tool qualification review (otherwise);
- Encode in the rules which files the verification tools may modify/create (e.g., all files within the current working directory);
- Enforce the result files pattern in the competition environment
- Encode in the rules that tools must be able to execute when called from outside their respective directories, and enforce this rule by introducing a CI-check (preferred) or a manual review (otherwise).

While we understand that rules will never cover all corner cases that would make tools SV-COMP-specific, these modifications cover a large set of problems that make the reuse of participating tools difficult. We believe that making participating verifiers accessible to anyone (and not just as benchmarking baselines, but as actual verification tools) is one of the most significant purposes of SV-COMP, and these modifications would enhance its impact.

Funding. This research was partially funded by the EKÖP-24-3 New National Excellence Program under project numbers EKÖP-24-3-BME-288 and EKÖP-24-3-BME-213, and the Doctoral Excellence Fellowship Programme under project numbers 400434/ 2023 and 400433/2023; funded by the NRDI Fund of Hungary. We also acknowledge KIFÜ (Governmental Agency for IT Development, Hungary, ror.org/01s0v4q65) for awarding us access to the Komondor HPC facility based in Hungary.

References

1. Bajczi, L.: SV-COMP25 Reproduction Data (Dec 2024). https://doi.org/10.5281/zenodo.14913865
2. Beyer, D., Strejček, J.: Improvements in Software Verification and Witness Validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
3. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and Resource Measurement. In: Fischer, B., Geldenhuys, J. (eds.) Model Checking Software. pp. 160–178. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_12
4. Gerhold, M., Hartmanns, A.: Reproduction report for SV-COMP 2023. Tech. rep., University of Twente (2023). https://doi.org/10.48550/arXiv.2303.06477

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

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

CPAchecker 4.0 as Witness Validator
(Competition Contribution)

Dirk Beyer(✉) and Marian Lingsch-Rosenfeld*

http://cpachecker.sosy-lab.org

LMU Munich, Munich, Germany

Abstract. CPAchecker is a tool for software verification, witness validation, and test-case generation, based on the concept of *configurable program analysis*. One of its main applications is to validate correctness and violation witnesses in versions 1.0 and 2.0. The witness validation is achieved by strengthening a selection of verification algorithms using the information from the witness. Due to the modular approach of CPAchecker, extending its verification analyses for witness validation can be easily done. Similar to CPAchecker's verification approach, witness validation uses a selection of analyses dependent on the witness type, the specification, and program features. To validate correctness witnesses, CPAchecker uses k-induction and predicate abstraction to verify that the invariants from the witness hold and the correctness of the program can be proven. To validate violation witnesses, CPAchecker uses predicate abstraction, value analysis, SMGs, and BDDs. CPAchecker's many verification algorithms make it a versatile and successful tool for witness validation.

1 Software Architecture

CPAchecker [13] is a tool for automatic software verification, *witness validation*, and test-case generation. It is possible for it to cover these different use-cases due to its modular architecture, based on the concept of *Configurable Program Analysis* (CPA) [12]. Each CPA represents information relevant to an analysis, for example an abstract domain, control-flow information, or an automaton, and can be combined with other CPAs in a modular manner. Exchanging information with other CPAs is done through a well-defined interface called the *strengthening* operator, which is used to incorporate information from one abstract state into the another. This modular approach allows any analysis, of which many exist [2, 4, 5, 7, 8, 15, 16], to be used for witness validation. The information from the witness can be encoded as a CPA, simplifying the adaptation of existing analyses for witness validation. One major challenge for witness validation is relating the input program to its internal representation as a control-flow automaton (CFA). In particular, this challenge occurs when validating witnesses in version 2.0, which are defined purely on the input program [1]. CPAchecker analyses such witnesses by transparently keeping track of the abstract-syntax-tree elements generated during parsing and their correspondence to the CFA nodes and edges.

* Jury member

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 192–198, 2025.
https://doi.org/10.1007/978-3-031-90660-2_11

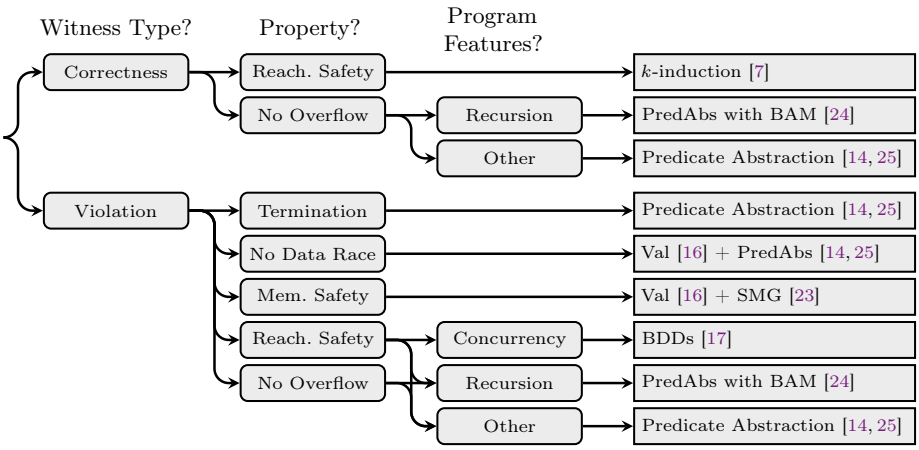


Fig. 1: Algorithm selection based on the witness type, property, and program

2 Validation Approach

When participating as a verifier, CPAchecker uses different verification algorithms [3, 6], since each has specific strengths and supports different properties. To validate witnesses, we choose only the most mature analyses and adapt the selection process to consider the witness type (violation or correctness), specification, and program features for the selected analyses. Figure 1 shows the selection process. The chosen verification algorithm is strengthened with the information from the witness. The analyses and how they interact with the witness are described in the following sections for all validation categories.

2.1 Correctness Witnesses

Validation of correctness witnesses is based on strengthening the chosen analysis using the invariants provided by the witness. Additionally, the invariants are added as proof-goals in order to validate their correctness.

Reachability Safety. CPAchecker uses one of its best-performing and mature analyses for reachability safety, k-Induction [7] augmented with invariants from the witness. First the invariants are matched to their corresponding nodes of the control-flow automaton (CFA). For witnesses 1.0 we do this by a reachability analysis only dependent on the control-flow. For witnesses 2.0 we directly match the location of invariants to their corresponding CFA nodes. Afterwards, k-Induction first verifies that the invariants are correct and then uses them to prove the safety property, while increasing k as necessary. This means that all invariants in the witness need to be k-inductive for some k in order to be validated.

No Overflow. To validate witnesses for proofs showing the absence of overflows, CPAchecker uses *predicate abstraction* [14]. The information in the witness is encoded using an invariant-injection automaton, which is traversed simultaneously to the state-space exploration and strengthens the analysis. Recursive programs are

handled by block-abstraction memoization (BAM) [9, 10, 24, 26], which summarizes the input-output behavior of recursive functions.

2.2 Violation Witnesses

Due to the modular nature of CPAchecker, any analysis, except for the validation of non-termination witnesses (see below), can easily be combined with an automaton, represented as a CPA, that *restricts* the state space and *strengthens* the analysis with assumptions. The automaton is based on the witness and guides the analysis towards the violation by exploring it simultaneously to the state-space. We *restrict* the state space by avoiding any transition that does not match the witness. For example, if the `then` branch is desired by the witness, then the analysis stops the exploration of the `else` branch. We *strengthen* the analysis by adding the assumptions to the abstract state. The assumptions are added through a common interface, the strengthening operator [8], which each analysis needs to implement to successfully make use of the information.

Termination. To validate a non-termination witness, we need to show that (1) the stem of the witness can reach the recurrence condition and that (2) whenever a state fulfills the recurrence condition, it returns to a state at the loop-head fulfilling the recurrence condition. Both of these checks are expressed as reachability analyses performed by predicate abstraction [14].

No Data Race. To validate a witness showing that there is a data race, we use an analysis based on value analysis [16] and predicate abstraction [14]. The value analysis uses the assumptions in the witness to determine additional concrete values to be tracked. The predicate abstraction adds the assumptions to the path formula to compute what predicates are valid in the successor state. The same strengthening is done whenever information from the witness needs to be added to either the value analysis or predicate abstraction for any other property.

Memory Safety. For memory safety, a combination of value analysis and symbolic memory graphs (SMGs) is used to explore the state-space while keeping track of the memory structure. In this case, the assumptions in the witness are used to strengthen the abstract state of the value analysis.

Reachability Safety. Predicate abstraction [14] is used to validate violation witnesses for reachability safety. It is aided by BAM [10, 24, 26] for tasks with recursive functions. The assumptions inside the witness are used to strengthen the predicate abstract state. If the task contains concurrency, an analysis based on BDDs [11, 18] is used, which currently ignores the assumptions in the witness.

No Overflow. To validate no-overflow violation witnesses, predicate abstraction [14] is used. The assumptions are used to strengthen the predicate abstract state. To handle tasks with recursion we add BAM [10, 24, 26].

3 Strengths and Weaknesses

CPAchecker validator performed well in SV-COMP 2025 [19], consistently ranking in the top three in almost all categories with a well-defined witness format, the only exceptions being validation of handcrafted violation witnesses in version 2.0,

validation of software-systems correctness witnesses in version 1.0, and validation of reach-safety and no-overflow violation witnesses in version 1.0. In particular, it is one of the only three validators participating in all categories for which a witness format exists, together with UAutomizer and MetaVal. In general, CPAchecker performed similarly when validating witnesses in version 1.0 and in version 2.0, since both are encoded in the same manner as a CPA to combine them with the verification analyses.

Notably CPAchecker performed best in the validation of correctness witnesses, but lagged behind other tools when validating violation witnesses. This is due to CPAchecker using primarily the predicate analysis for validation, which is better in finding proofs than bugs. In general, since the analyses used for validation are the same as for verification, only strengthened with the witnesses, they have the same strengths and weaknesses.

We do not use the parallel portfolio of different analyses that boosts CPAchecker 4.0 as a verifier, we rather select one mature analyses, in order to increase the confidence in the validation result. In the future, a portfolio of other mature analyses could be used, improving the performance without compromising the confidence in the results.

4 Setup and Configuration

CPAchecker validator in version 4.0 [20] was used in SV-COMP 2025. It runs on any system with a Java 17 compatible runtime environment, though its default SMT solver MathSAT5 [22] is bundled only for Linux. For platforms other than GNU/Linux, we recommend using the provided container image [21]. To validate a witness using CPAchecker, execute the following command:

```
bin/cpachecker --witnessValidation --benchmark --heap 10000M
    --timelimit limit --32 --witness witness.{yml,graphml}
    --spec property.prp program.i
```

SV-COMP uses a timelimit of 900 s for correctness witnesses and 90 s for violation witnesses, though it will work with other/no time limits. Replace `--32` by `--64` for programs that assume a 64-bit memory model. More information on how to run CPAchecker is available on its website https://cpachecker.sosy-lab.org and tutorial paper [2].

5 Project and Contributors

CPAchecker validator builds upon existing verification algorithms with additional support for handling witnesses. The success of CPAchecker is the result of contributions from over 100 developers, primarily from institutions such as LMU Munich, TU Darmstadt, University of Paderborn, University of Passau, TU Prague, University of Oldenburg, TU Vienna, ISP RAS, and numerous other universities and research institutes. We extend our utmost gratitude to all contributors to CPAchecker which made it possible to have such a successful validator. A complete list of contributors and further details about the project can be found at https://cpachecker.sosy-lab.org.

Data-Availability Statement. The tool is available at https://cpachecker.sosy-lab.org and the version used in SV-COMP 2025 is archived at Zenodo [20].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 496588242 (IdeFix).

References

1. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. pp. 184–203. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_11
2. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
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 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
4. Beyer, D., Chien, P.C., Jankola, M., Lee, N.Z.: A transferability study of interpolation-based hardware model checking for software verification. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3660797
5. 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
6. 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. https://www.sosy-lab.org/research/pub/2018-ISoLA.Strategy_Selection_for_Software_Verification_Based_on_Boolean_Features.pdf
7. 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
8. 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
9. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE. pp. 634–644. ACM (2018). https://doi.org/10.1145/3238147.3238195
10. Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Proc. ESEC/FSE. pp. 50–62. ACM (2020). https://doi.org/10.1145/3368089.3409718
11. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_26
12. 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

13. 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
14. 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
15. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning **69** (2025). https://doi.org/10.1007/s10817-024-09702-9, preprint: https://doi.org/10.48550/arXiv.2208.05046
16. 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. https://www.sosy-lab.org/research/pub/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf
17. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPAchecker. In: Proc. MEMICS. pp. 1–11. LNCS 7721, Springer (2013). https://doi.org/10.1007/978-3-642-36046-6_1. https://www.sosy-lab.org/research/pub/2013-MEMICS.BDD-Based_Software_Model_Checking_with_CPAchecker.pdf
18. Beyer, D., Stahlbauer, A.: BDD-based software verification: Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transfer **16**(5), 507–518 (2014). https://doi.org/10.1007/s10009-014-0334-1
19. Beyer, D., Strejček, J.: Report on SV-COMP 2025. In: Proc. TACAS. LNCS , Springer (2025)
20. Beyer, D., Wendler, P.: CPAchecker release 4.0. Zenodo (2024). https://doi.org/10.5281/zenodo.14203369
21. Beyer, D., Wendler, P.: CPAchecker release 4.0 (image). Zenodo (2024). https://doi.org/10.5281/zenodo.14209310
22. 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
23. 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
24. 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
25. 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
26. 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

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

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

AISE v2.0: Combining Loop Transformations (Competition Contribution)

Yao Lin[1,2], Zhenbang Chen[1,2(✉)]*, and Ji Wang[1,2]

[1] College of Computer Science and Technology, National University of Defense Technology, Changsha, China
[2] State Key Laboratory of Complex and Critical Software Environment, National University of Defense Technology, Changsha, China
{linyao23,zbchen,wj}@nudt.edu.cn

Abstract. AISE is a C program verifier that synergizes symbolic execution and abstract interpretation. This year, AISE v2.0 introduces a loop transformation scheme based on recurrence analysis to handle programs involving nonlinear arithmetic. By combining loop transformations, AISE v2.0 achieved a score of 1031 and won first place in the *ReachSafety-Loops* category, demonstrating the effectiveness of the methods employed in AISE v2.0.

1 Verification Approach

The key idea of AISE [19] is to synergize abstract interpretation [12] and symbolic execution [7,15]. This synergy[1] enables AISE to handle the program with loops effectively. However, when the program involves nonlinear arithmetic, AISE still suffers from the path explosion problem. To address this, we introduce a loop abstraction method, which *over-approximates* the loop in the program based on the invariant generated by the recurrence analysis [17,18]. Additionally, we propose several other loop transformation methods to address different scenarios.

Figure 1 illustrates the workflow of AISE v2.0. Given a C program $\mathcal{P}$, we first compile it into LLVM IR format and then preprocess it using LLVM's passes. Next, AISE will be used to verify $\mathcal{P}$. If AISE cannot finish within 60 seconds, the loops in $\mathcal{P}$ will be transformed, generating four variants, *i.e.*, NL, A, ANs, and E. AISE then sequentially verify NL, A, and ANs to determine whether the properties in $\mathcal{P}$ hold. If an unknown or violation occurs during this process, we will revert to verifying $\mathcal{P}$ or E until timeout. Below, we introduce each loop transformation and its role in verification.

Not entering the loop (NL). Figure 2b illustrates a scenario where the loop in $\mathcal{P}$ is not entered. Since symbolic execution often "gets stuck" in loops, prioritizing the analysis of this scenario may help reduce unnecessary overhead.

Loop abstraction (A). Inspired by the loop abstraction method [13], we use an invariant-based loop abstraction here to tackle the challenge brought by loops. Figure 2c shows the abstracted program A. Intuitively, A represents the set of

* Jury member
[1] Due to space limitations, further technical details can be found in [19].

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 199–204, 2025.
https://doi.org/10.1007/978-3-031-90660-2_12

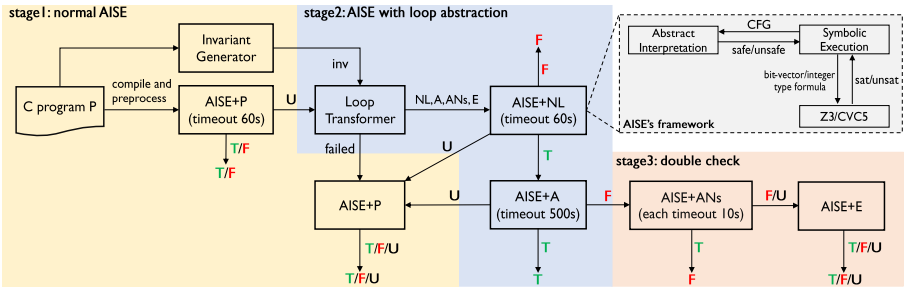
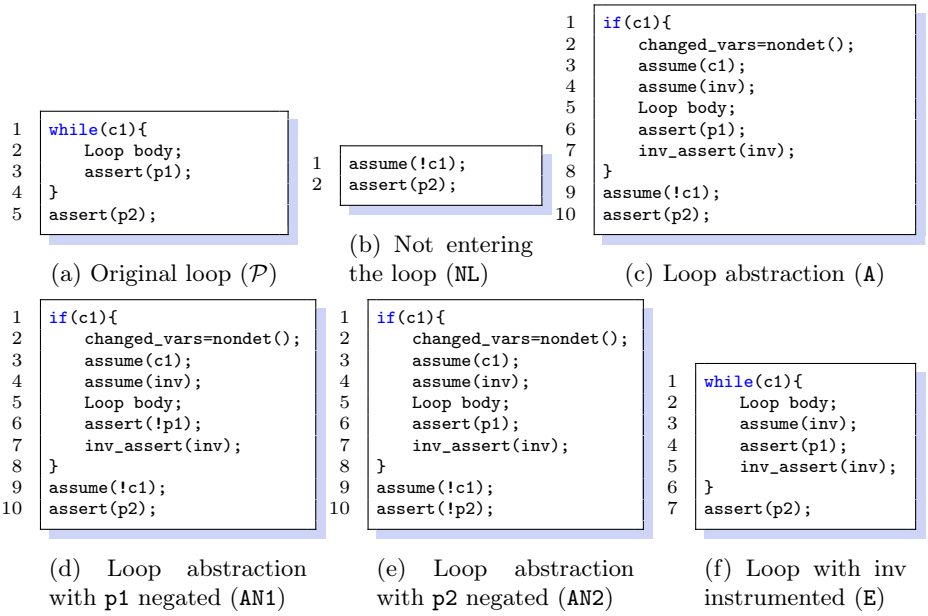


Fig. 1: The workflow of `AISE` v2.0 ($\mathcal{P}$: the original program, `NL`: not entering the loop, `A`: loop abstraction, `ANs`: loop abstractions with one negated property in each, `E`: the program with `inv` instrumented, **T**: True, **F**:False, **U**:Unknown)

all possible behaviors that could occur during any iteration of the loop in $\mathcal{P}$. In Figure 2c, Line 2 represents the non-deterministic assignments to the variables modified in the `Loop body` (denoted as `changed_vars`), which describes all possible states of all variables in the `Loop body` at the beginning of the loop's any iteration. Line 3 and Line 9 ensure the correct semantics of entering and exiting the loop, respectively. To improve the precision of abstraction, we use recurrence analysis [17,18], which captures the numerical relationships of variables using real arithmetic, to generate the loop invariant (`inv`) and add it in Line 4. However, when the loop involves integer division, rounding issues may cause the `inv` to fail to satisfy the inductiveness. Therefore, additional checks[2] (Line 7) are needed to verify the inductiveness of `inv`. If `inv` is violated, we will remove it from the transformed program and generate another invariant (which is included in each analysis step in Figure 1). Notably, recurrence analysis [17,18] only plays the role of generating `inv`, and `inv` can also be produced by other loop invariant generation techniques. Due to Line 1, the symbolic execution of `A` also covers the case not entering the loop (*i.e.*, `NL`).[3] For nested loops, we abstract the innermost loop first and then progressively abstract each outer loop before verification. By verifying that `p1` and `p2` hold in `A`, we can conclude that these properties also hold in $\mathcal{P}$. However, when verification fails, whether the property in $\mathcal{P}$ is violated cannot be determined, as it may be a false positive due to the *over-approximation* of the loop abstraction in `A`.

Loop abstraction with one negated property (`AN`). `AN` is generated from `A` with only one property negated. If a violation occurs when verifying `A`, we will first confirm it by verifying `ANs`. `AN1` (Figure 2d) and `AN2` (Figure 2e) are the `ANs` generated from `A` (Figure 2c). If we ignore assertions, we can also say that `ANs` are the *over-approximation* of $\mathcal{P}$. If the assertions in `AN1` and `AN2` are all reachable in $\mathcal{P}$ and the verification result of either is true, we can conclude that one property in $\mathcal{P}$ will be violated while the remaining properties will hold. For example, if

[2] `inv_assert` here functions like `assert`, but with a different name for distinction.
[3] Because of development time limitations, the scenarios described by `NL` and `A` overlap, which will be optimized in the future.



(a) Original loop ($\mathcal{P}$) (b) Not entering the loop (`NL`) (c) Loop abstraction (`A`)

(d) Loop abstraction with `p1` negated (`AN1`) (e) Loop abstraction with `p2` negated (`AN2`) (f) Loop with inv instrumented (`E`)

Fig. 2: Illustration of the loop transformation scheme

the assertions in `AN1` are all reachable in $\mathcal{P}$ and we can prove that `!p1` and `p2` hold in `AN1`, then we can conclude that `!p1` and `p2` also hold in $\mathcal{P}$, that is to say, the `assert(p1)` in $\mathcal{P}$ will be violated. During preprocessing, we use `LLVM`'s passes to remove the unreachable basic blocks in $\mathcal{P}$, ensuring that all assertions in `ANs` are reachable in $\mathcal{P}$. If unreachable assertions still remain in `ANs` after transformation, this approach may result in false positives.

Loop with invariant (`E`). This variant in Figure 2f is the last step of confirming the violation, *i.e.*, employing `AISE` to check the violation. However, different from $\mathcal{P}$, we add the invariant `inv` in Line 3, which improves the efficiency of symbolic execution by leveraging the invariant to accelerate constraint solving process.

2 Implementation

The `Loop Transformer` of `AISE` v2.0 is based on `LLVM` [1]. `AISE`'s abstract interpretation module is `CLAM` [2,14], configured as in [19]. `AISE`'s symbolic execution module is based on `BUBAAK-LEE` [3] (a fork of `KLEE` [10] used in the `BUBAAK` [11]) with some improvements. We improve the solver part by supporting floating-point programs and employing the theory of integer as an additional option (`KLEE` uses bit-vector theory by default) for solving path conditions. The symbolic execution module of `AISE` first uses `Z3` [16] to check formulas' satisfiability, and switches to `CVC5` [8] if `Z3` times out. The `Invariant Generator` is based on `c_convertor` [4], which summarizes loops by computing closed-form solutions for each variable in the loop body [17] or for the polynomial expressions

Table 1: The contribution of each method

	total	AISE+P	AISE+NL	AISE+A	AISE+ANs	AISE+E
total correct	603	528	0	66	8	1
correct true	428	362	0	66	0	0
correct false	175	166	0	0	8	1
total correct-unconfirmed	73	49	0	23	0	1
total incorrect	0	0	0	0	0	0
score	1031	890	0	132	8	1

extracted from the loop body [18]. Besides fixing some bugs, we have optimized the `Invariant Generator` by considering branch conditions and the initial assignments of variables.

3 Result and Discussion

This year, by combining (1) the synergy between abstract interpretation and symbolic execution and (2) a recurrence analysis based loop transformation scheme, `AISE` v2.0 scored 1031 points and won first place in the *ReachSafety-Loops* category, demonstrating its capability in verifying the programs with loops. Table 1 presents the contribution of each method. As shown by the table, the synergy in `AISE` is effective for most programs, and when it fails within 60 seconds, loop transformations can further enhance `AISE`'s ability. Due to lack of invariant in correctness witnesses, we still have 73 unconfirmed results. `AISE` v2.0 could not produce any results in the `AISE+NL` phase due to the program's small size, and it may perform better on larger programs. When handling programs with arrays or loops containing multiple branches, both `CLAM` and `Invariant Generator` may not work, causing `AISE` v2.0 to still suffer from the path explosion problem. Additionally, the SMT solvers encounter difficulties when solving nonlinear constraints, further limiting the verification capabilities of `AISE` v2.0. Since we only consider whether the program is correct when it terminates, `AISE`'s verify method currently cannot handle programs with non-terminating loops.

4 Software Project, Tool setup and Contributors

`AISE` v2.0 can run on the Ubuntu 22.04/24.04 LTS and is licensed under GPL 3.0. This year, `AISE` v2.0 participates in the *ReachSafety-Loops* category of SV-COMP 2025 [9]. The execution command of `AISE` v2.0 is as follow:

```
./bin/aise <data_model> <program>
```

Where `<data_model>` is the program's data model (32-bit or 64-bit) and `<program>` is the input program. The contributors of `AISE` v2.0 are all from the National University of Defense Technology. A complete list of contributors is available at [5].

Data-Availability Statement The artifact for `AISE` v2.0 has been archived and is available on Zenodo [6].

Acknowledgement This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62032024).

References

1. LLVM. https://llvm.org, accessed 2024-12-11
2. CLAM repository. https://github.com/seahorn/clam (2022)
3. BUBAAK-LEE repository. https://github.com/mchalupa/bubaak-lee (2022)
4. c_convertor repository. https://github.com/psy054duck/c_convertor (2024)
5. Contributors list of AISE. https://github.com/zbchen/aise-verifier/blob/master/Contributors.txt, accessed 2025-1-23
6. AISE artifact. https://zenodo.org/records/14203693 (2024)
7. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques **51**(3) (May 2018). https://doi.org/10.1145/3182657
8. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
9. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
10. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association
11. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers: (competition contribution). p. 535–540. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30820-8_32
12. Cousot, P.: Abstract interpretation. ACM Comput. Surv. **28**(2), 324–328 (Jun 1996). https://doi.org/10.1145/234528.234740
13. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1407–1412 (2015). https://doi.org/10.7873/DATE.2015.0245
14. Gurfinkel, A., Navas, J.A.: Abstract interpretation of llvm with a region-based memory model. p. 122–144. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-95561-8_8
15. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (Jul 1976). https://doi.org/10.1145/360248.360252
16. de Moura, L.M., Bjørner, N.S.: Z3: An efficient smt solver. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems (2008)
17. Wang, C., Lin, F.: Solving conditional linear recurrences for program verification: The periodic case. Proc. ACM Program. Lang. **7**(OOPSLA1) (Apr 2023). https://doi.org/10.1145/3586028

18. Wang, C., Lin, F.: On polynomial expressions with c-finite recurrences in loops with nested nondeterministic branches. In: Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I. p. 409–430. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-65627-9_20
19. Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 347–352. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_19

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

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

AProVE (KoAT + LoAT)

(Competition Contribution)

Nils Lommen(✉) and Jürgen Giesl(✉)

RWTH Aachen University, Aachen, Germany
{lommen,giesl}@cs.rwth-aachen.de

Abstract. To (dis)prove termination of C programs, AProVE uses symbolic execution to transform the program's LLVM code into an integer transition system (ITS). These ITSs are analyzed by our backend tools KoAT (for termination) and LoAT (for non-termination) which we integrated into our novel framework to replace previously used external backend tools. In this way, we benefit from the recent improvements in the backend tools KoAT and LoAT. The transformation steps in AProVE and the tools in the backend produce sub-proofs which are then combined automatically in order to generate a complete termination proof. If non-termination is proved, then a witness for a non-terminating path in the original C program is returned.

1 Verification Approach and Software Architecture

AProVE (KoAT + LoAT) is a framework combining our three tools AProVE, KoAT, and LoAT to prove or disprove termination of C programs automatically. To this end, the C program is compiled into the intermediate representation of the LLVM framework [28] by the Clang compiler [1]. Afterwards, the LLVM program is processed by AProVE and transformed into a *symbolic execution graph* (SEG, see [24, 26, 36, 37] for more details). Finally, the SEG is either analyzed directly by AProVE (see [23–25, 37] for more details on the internal (non-)termination proofs in AProVE) or transformed into *integer transition systems* (ITSs). These ITSs are analyzed by our tools KoAT [8, 20, 29–32] (for termination) and LoAT [14–17] (for non-termination). In case of non-termination, the proofs by AProVE and LoAT are transformed into non-termination proofs for the original C program.

Earlier versions of AProVE that participated in *SV-COMP* until 2022 used the external tool T2 [7] for proving termination of ITSs (and both T2 and LoAT for proving non-termination). In contrast, instead of T2, our new version uses our own ITS analyzer KoAT in the backend which allows us to benefit from the numerous recent improvements that we developed for KoAT. A bird's-eye view of our approach is sketched in Fig. 1.

Termination Analysis by KoAT: In the following, we briefly describe how our tool KoAT proves termination of ITSs which result from the transformation of C programs. LoAT can be used to *disprove* termination and infer *lower* run-

N. Lommen—Jury member representing AProVE (KoAT + LoAT) at *SV-COMP 2025*

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 205–211, 2025.
https://doi.org/10.1007/978-3-031-90660-2_13

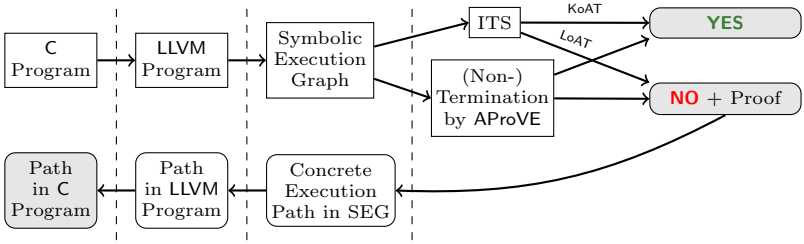


Fig. 1. AProVE (KoAT + LoAT) Framework for Proving and Disproving Termination

time bounds for ITSs, and its integration in order to prove non-termination of C programs was described in [25]. In contrast, KoAT is a tool to automatically *prove* termination and infer *upper* complexity bounds for ITSs based on a modular analysis of separate program parts [8, 32]. To prove termination of sub-programs, it uses *multiphase-linear ranking functions* (MΦRFs) [5, 20]. In contrast to classical ranking functions, MΦRFs can also represent bounds on programs with multiple "phases" of executions.

Moreover, we embedded a technique [21] to analyze termination of so-called *triangular weakly non-linear loops* (twn-loops) in our tool KoAT [29, 32]. In particular, this approach also allows us to analyze programs with *non-linear* arithmetic. An example for a terminating twn-loop, which may result from a sub-program within a larger C program, is:

$$\textbf{while } (x_2 - x_1^2 > 0 \wedge x_1 > 0) \textbf{ do } (x_1, x_2, x_3) \leftarrow (4 \cdot x_1,\ 9 \cdot x_2 - 8 \cdot x_3^3,\ x_3) \qquad (1)$$

This loop does not admit a MΦRF over $\mathbb{R}$ (see [22]). The guard of such twn-loops are propositional formulas over (possibly non-linear) polynomial inequations. The update is *triangular*, i.e., we can order the variables such that the update of any x_i does not depend on the variables $x_1, \ldots, x_{i-1}$ with smaller indices. So the restriction to triangular updates prohibits "cyclic dependencies" of variables (e.g., where the new values of x_1 and x_2 both depend on the old values of x_1 and x_2). For example, a loop whose body consists of the assignment $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_2 + 1)$ is triangular, whereas a loop with the body $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_1 + 1)$ is not triangular. From a practical point of view, the restriction to triangular loops seems quite natural. For example, in [15], 1511 polynomial loops were extracted from the *Termination Problems Data Base* [38], the benchmark collection which is used at the annual *Termination and Complexity Competition* [19], and only 26 of them were non-triangular. Furthermore, the update of a twn-loop is *weakly non-linear*, i.e., no variable x_i has a non-linear occurrence in its own update. So for example, a loop with the body $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_2 + 1)$ is weakly non-linear, whereas a loop with the body $(x_1, x_2) \leftarrow (x_1 \cdot x_2, x_2 + 1)$ is not. With triangularity and weak non-linearity, one can compute a *closed form* which corresponds to applying the loop's update n times. For example, the closed forms of (1) are $x_1 \cdot 4^n$, $(x_2 - x_3^3) \cdot 9^n + x_3^3$, and x_3 for x_1, x_2, and x_3, respectively.

Using these closed forms, termination can be reduced to a formula over $\mathbb{Z}$ [21] (whose satisfiability is decidable for linear arithmetic and where SMT solvers often also prove (un)satisfiability in the non-linear case). The idea of the reduction is to consider each atom of the loop guard individually, instantiate the variables in the atoms by their closed forms, and order the summands of the resulting expressions w.r.t. their growth rate in n. For example, for the atom $x_2 - x_1^2 > 0$ of (1), we obtain $\alpha_1 \cdot 16^n + \alpha_2 \cdot 9^n + \alpha_3 > 0$ where $\alpha_1 = -x_1^2$, $\alpha_2 = x_2 - x_3^3$, and $\alpha_3 = x_3^3$. If the initial values of the variables satisfy $\alpha_1 > 0$, then the addend $\alpha_1 \cdot 16^n$ would dominate all other addends at some point and the atom $x_2 - x_1^2 > 0$ would hold for all large enough n, assuming that there are no overflows. (However, in our example $\alpha_1 = -x_1^2 > 0$ is unsatisfiable.) Otherwise, if $\alpha_1 = -x_1^2 = 0$, then the second addend $\alpha_2 \cdot 9^n$ is the fastest growing summand. Applying this idea to all addends subsequently, i.e., checking if the first polynomials $\alpha_1, \ldots, \alpha_{j-1}$ are 0 and α_j is positive, we can reduce non-termination to an existential first-order (FO) problem. For our example, we obtain the reduction $\mathrm{red}(x_2 - x_1^2 > 0) = \alpha_1 > 0 \vee (\alpha_1 = 0 \wedge \alpha_2 > 0) \vee (\alpha_1 = 0 \wedge \alpha_2 = 0 \wedge \alpha_3 > 0)$ for the atom $x_2 - x_1^2 > 0$. Similarly, we have $\mathrm{red}(x_1 > 0) = (x_1 > 0)$ for the second atom of the loop guard. Replacing the atoms of the guard by their reduction results in the overall FO problem $\psi = \mathrm{red}(x_2 - x_1^2 > 0) \wedge \mathrm{red}(x_1 > 0)$. Thus, (1) is non-terminating over $\mathbb{Z}$ iff ψ is satisfiable over $\mathbb{Z}$. In our example, the formula ψ is unsatisfiable since $\mathrm{red}(x_2 - x_1^2 > 0)$ is only satisfiable if $x_1 = 0$. However, this violates $\mathrm{red}(x_1 > 0)$. Thus, the twn-loop (1) is terminating.

KoAT's novel approach of analyzing some sub-programs with ranking functions and other sub-programs with our technique for twn-loops increases the power of automated termination analysis substantially, in particular also for programs containing non-linear arithmetic. Nevertheless, there still exist programs whose termination is difficult to analyze and where it would help to gain "more information" on the values of variables in order to determine the infeasibility of certain paths in the program. For example, one could transform a loop whose body contains an if-else-instruction where either always the if-branch or the else-branch are executed into two separate single-path loops. This transformation explicitly removes infeasible paths which contain both branches. Thus, the idea is to transform an ITS $\mathcal{P}$ into a new ITS $\mathcal{P}'$ which is "easier" to analyze. Of course, we ensure that the runtime of $\mathcal{P}'$ is *at least* the runtime of $\mathcal{P}$. Then it is sound to prove termination for $\mathcal{P}'$ instead of $\mathcal{P}$. Such transformations can be performed by *control-flow refinement* via partial evaluation [11], which we integrated into KoAT's approach [20, 31].

2 Discussion of Strengths and Weaknesses

An underlying design concept of AProVE (KoAT + LoAT) is its modular structure. It allows us to use different backends and techniques to analyze programs and benefit from their individual strengths. Hence, our new framework can make use of recent progress in software verification tools. Furthermore, both AProVE

and our backend tools KoAT and LoAT are modular themselves (see, e.g., KoAT's modular analysis of sub-programs in Sect. 1).

One of AProVE's main weaknesses was that it essentially relied on variants of (linear) ranking functions for termination proofs. However, this problem has now been solved by integrating our tool KoAT into AProVE. Besides ranking functions, KoAT also uses a technique for termination analysis of twn-loops which allows us to analyze programs with non-linear arithmetic on which the other sound tools participating in the termination category of *SV-COMP* fail. For example, the C program which just instantiates all variables non-deterministically and consists of the loop (1) and even its following linear, simplified variant from [22]

$$\textbf{while } (x_1 < x_2 \wedge x_1 > 0) \textbf{ do } (x_1, x_2) \leftarrow (3 \cdot x_1, 2 \cdot x_2)$$

can both not be shown terminating by Ultimate Automizer [10] and 2LS [35], which were the two most powerful sound[1] tools at the termination category of last year's *SV-COMP*. In contrast, AProVE proves their termination using the implementation of our termination technique for twn-loops in KoAT.

Concerning the proofs of non-termination, currently AProVE (KoAT + LoAT) uses a version of LoAT which disproves termination using the loop acceleration technique of [16]. In [17], a new version of LoAT was developed that is based on an acceleration driven clause learning calculus which improves LoAT's power for non-termination proofs substantially. We plan to integrate this new version in our AProVE (KoAT + LoAT) framework in the future, using a new format to ease the automated handling of LoAT's non-termination proofs. Moreover, as KoAT and LoAT can also analyze complexity of programs, it would be interesting to extend AProVE (KoAT + LoAT) to complexity analysis as well.

3 Setup and Configuration

AProVE (KoAT + LoAT) is developed in the "*Programming Languages and Verification*" group at RWTH Aachen University. A list of present and past contributors can be accessed on the website [2]. In *SV-COMP 2025*, AProVE (KoAT + LoAT) only participates in the category "*Termination*". All files from the submitted archive [33] must be extracted into one folder. AProVE is implemented in Java and needs a Java 17 Runtime Environment. To analyze the resulting ITSs in the backend, the tools KoAT [20, 29–32] and LoAT [14–17] are used. Furthermore, it applies the satisfiability checkers Z3 [9], Yices [12], and MiniSAT [13], see [18]. Our archive contains all these tools. Using the wrapper script `aprove.py` in the BenchExec repository, AProVE (KoAT + LoAT) can be invoked, e.g., on the benchmarks defined in `aprove.xml` in the *SV-COMP* repository. The most recent version of our tool for *SV-COMP 2025* [6] can be downloaded at [33]. Moreover, C programs for all examples from the paper are available at [33] and [34].

[1] The tool PROTON [27] (which won the termination category last year) outputs termination for both these loops, but also for the *non-terminating* variants of these loops where the guard $x_1 > 0$ is missing. In contrast, both Ultimate Automizer and 2LS prove non-termination of these variants.

Data Availability Statement. Our tools are available at their respective websites (for AProVE [2], KoAT [3], and LoAT [4]). The version of AProVE (KoAT + LoAT) used for *SV-COMP 2025* is archived at Zenodo [33].

References

[1] Clang: https://clang.llvm.org.

[2] AProVE Website: https://aprove.informatik.rwth-aachen.de/.

[3] KoAT Website: https://koat.verify.rwth-aachen.de/.

[4] LoAT Website: https://loat-developers.github.io/LoAT/.

[5] A. M. Ben-Amram and S. Genaim. "On Multiphase-Linear Ranking Functions". In: *Proc. CAV '17*. LNCS 10427. 2017, pp. 601–620. DOI: 10.1007/978-3-319-63390-9_32.

[6] D. Beyer and J. Strejček. "Improvements in Software Verification and Witness Validation: *SV-COMP 2025*". In: *Proc. TACAS '25*. LNCS. 2025.

[7] M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. "T2: Temporal Property Verification". In: *Proc. TACAS '16*. LNCS 9636. 2016, pp. 387–393. DOI: 10.1007/978-3-662-49674-9_22.

[8] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. "Analyzing Runtime and Size Complexity of Integer Programs". In: *ACM Transactions on Programming Languages and Systems* 38 (2016), pp. 1–50. DOI: 10.1145/2866575.

[9] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. TACAS*. LNCS 4963. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[10] D. Dietsch, M. Bentele, M. Ebbinghaus, M. Heizmann, D. Klumpp, A. Podelski, and F. Schüssele. Ultimate Automizer *SV-COMP 2025*. Zenodo. 2024. DOI: 10.5281/zenodo.14209043.

[11] J. J. Doménech, J. P. Gallagher, and S. Genaim. "Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis". In: *Theory and Practice of Logic Programming* 19.5-6 (2019), pp. 990–1005. DOI: 10.1017/S1471068419000310.

[12] B. Dutertre and L. de Moura. *System Description:* Yices 1.0. https://yices.csl.sri.com/papers/yices-smtcomp06.pdf. 2006.

[13] N. Eén and N. Sörensson. "An Extensible SAT-solver". In: *Proc. SAT '03*. LNCS 2919. 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[14] F. Frohn and J. Giesl. "Proving Non-Termination via Loop Acceleration". In: *Proc. FMCAD '19*. 2019, pp. 221–230. DOI: 10.23919/FMCAD.2019.8894271.

[15] F. Frohn and C. Fuhs. "A Calculus for Modular Loop Acceleration and Non-Termination Proofs". In: *International Journal on Software Tools for Technology Transfer* 24.5 (2022), pp. 691–715. DOI: 10.1007/S10009-022-00670-2.

[16] F. Frohn and J. Giesl. “Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)”. In: *Proc. IJCAR ’22*. LNCS 13385. 2022, pp. 712–722. DOI: 10.1007/978-3-031-10769-6_41.

[17] F. Frohn and J. Giesl. “Proving Non-Termination by Acceleration Driven Clause Learning (Short Paper)”. In: *Proc. CADE ’23*. LNCS 14132. 2023, pp. 220–233. DOI: 10.1007/978-3-031-38499-8_13.

[18] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. “SAT Solving for Termination Analysis with Polynomial Interpretations”. In: *Proc. SAT ’07*. LNCS 4501. 2007, pp. 340–354. DOI: 10.1007/978-3-540-72788-0_33.

[19] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. “The Termination and Complexity Competition”. In: *Proc. TACAS ’19*. LNCS 11429. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[20] J. Giesl, N. Lommen, M. Hark, and F. Meyer. “Improving Automatic Complexity Analysis of Integer Programs”. In: *The Logic of Software. A Tasting Menu of Formal Methods*. LNCS 13360. 2022, pp. 193–228. DOI: 10.1007/978-3-031-08166-8_10.

[21] M. Hark, F. Frohn, and J. Giesl. “Termination of Triangular Polynomial Loops”. In: *Formal Methods in System Design* (2023). DOI: 10.1007/s10703-023-00440-z.

[22] M. Heizmann and J. Leike. “Ranking Templates for Linear Loops”. In: *Logical Methods in Computer Science* 11.1 (2015). DOI: 10.2168/LMCS-11(1:16)2015.

[23] J. Hensel, F. Emrich, F. Frohn, T. Ströder, and J. Giesl. “AProVE: Proving and Disproving Termination of Memory-Manipulating C Programs (Competition Contribution)”. In: *Proc. TACAS ’17*. LNCS 10206. 2017, pp. 350–354. DOI: 10.1007/978-3-662-54580-5_21.

[24] J. Hensel, J. Giesl, F. Frohn, and T. Ströder. “Termination and Complexity Analysis for Programs with Bitvector Arithmetic by Symbolic Execution”. In: *Journal of Logical and Algebraic Methods in Programming* 97 (2018), pp. 105–130. URL: https://doi.org/10.1016/j.jlamp.2018.02.004.

[25] J. Hensel, C. Mensendiek, and J. Giesl. “AProVE: Non-Termination Witnesses for C Programs (Competition Contribution)”. In: *Proc. TACAS ’22*. LNCS 14132. 2022, pp. 403–407. DOI: 10.1007/978-3-030-99527-0_21.

[26] J. Hensel and J. Giesl. “Proving Termination of C Programs with Lists”. In: *Proc. CADE ’23*. LNCS 14132. 2023, pp. 266–285. DOI: 10.1007/978-3-031-38499-8_16.

[27] H. Karmarkar, M. Kumar, R. Metta, and D. Mukhopadhyay. PROTON *SV-COMP 2025*. Zenodo. 2024. DOI: 10.5281/zenodo.14209458.

[28] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proc. CGO ’04*. 2004, pp. 75–88. URL: https://doi.org/10.1109/CGO.2004.1281665.

[29] N. Lommen, F. Meyer, and J. Giesl. “Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops”. In: *Proc.*

IJCAR '22. LNCS 13385. 2022, pp. 734–754. DOI: 10.1007/978-3-031-10769-6_43.

[30] N. Lommen and J. Giesl. "Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs". In: *Proc. FroCoS '23*. LNCS 14279. 2023, pp. 3–22. DOI: 10.1007/978-3-031-43369-6_1.

[31] N. Lommen, É. Meyer, and J. Giesl. "Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT (Short Paper)". In: *Proc. IJCAR '24*. 2024, pp. 233–243. DOI: 10.1007/978-3-031-63498-7_14.

[32] N. Lommen, É. Meyer, and J. Giesl. "Targeting Completeness: Automated Complexity Analysis of Integer Programs". In: *CoRR* abs/2412.01832 (2024). DOI: 10.48550/arXiv.2412.01832.

[33] N. Lommen and J. Giesl. AProVE (KoAT + LoAT) Download. Zenodo. 2024. DOI: 10.5281/zenodo.13937818.

[34] N. Lommen and J. Giesl. AProVE (KoAT + LoAT) Website. 2025. URL: https://koat.verify.rwth-aachen.de/svcomp25.

[35] V. Malík, F. Nečas, P. Schrammel, and T. Vojnar. 2LS. Zenodo. 2023. DOI: 10.5281/zenodo.10184626.

[36] T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. "AProVE: Termination and Memory Safety of C Programs (Competition Contribution)". In: *Proc. TACAS '15*. LNCS 9035. 2015, pp. 417–419. DOI: 10.1007/978-3-662-46681-0_32.

[37] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. "Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic". In: *Journal of Automated Reasoning* 58.1 (2017), pp. 33–65. DOI: 10.1007/s10817-016-9389-x.

[38] TPDB (Termination Problems Data Base). URL: https://github.com/TermCOMP/TPDB.

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

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

Bubaak: Dynamic Cooperative Verification (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. Cooperative verification is gaining momentum in recent years. The usual setup in cooperative verification is that a verifier A is run with some pre-defined resources, and if it is not able to verify the program, the verification task is passed to a verifier B together with information learned about the program by verifier A, then the chain can continue to a verifier C, and so on. This scheme is static: tools run one after another in a fixed pre-defined order and fixed parameters and resource limits (the scheme may differ for properties to be analyzed, though).
Bubaak is a program analysis tool that allows to run multiple program verifiers in a dynamically changing combination of parallel and sequential portfolios. Bubaak starts the verification process by invoking an initial set of tasks; every task, when it is done (e.g., because of hitting a time limit or finishing its job), rewrites itself into one or more successor tasks. New tasks can be also spawned upon events generated by other tasks. This all happens dynamically based on the information gathered by finished and running tasks. During their execution, tasks that run in parallel can exchange (partial) verification artifacts, either directly or with Bubaak as an intermediary.

1 Verification Approach

The original idea of Bubaak [7] was to run multiple verifiers in parallel and apply *runtime monitoring on the verifiers* to gather information about their progress and their findings (e.g., found loop invariants). Based on the observed data, Bubaak then would instruct particular tools (at runtime) to stop their search at some points in the program, or to assume an invariant in other points.

The current version of Bubaak[3] generalizes this idea and allows to execute verifiers in a *dynamically changing* combination of parallel and sequential portfolio. What verifiers (or tools in general) are executed next and with what parameters is determined from the information gathered during the verification

M. Chalupa—Jury member and the corresponding author
[3] This paper covers the development of Bubaak since SV-COMP 2023.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 212–216, 2025.
https://doi.org/10.1007/978-3-031-90660-2_14

process. The verifiers are instrumented to share information about what they are doing and what they have found (this part amounts to runtime monitoring). They can also be instrumented to receive and use such information from outside: either from Bubaak or directly from other verifiers. An example situation can be that verifier A sends information about reachable states to verifier B, while verifier B informs Bubaak about found invariants and Bubaak distributes the invariants to other verifiers that may benefit from it.

Allowing the information exchange classifies Bubaak as a *cooperative verification* [4,3,2] tool. In cooperative verification, multiple verification tools help each other to increase their strength and verify more than each of the tools can verify alone. Cooperative verification tools usually employ only a static scheme that is fixed before the verification starts (the scheme is usually parametrized by the analyzed property, though). To the extent of our knowledge, Bubaak is the first tool to implement what we call *dynamic cooperative verification*: verifiers are spawned and stopped dynamically during the verification process based on the information discovered by verifiers that are currently running or that have finished running previously. At the same time, verifiers may exchange information. This information exchange includes passing artifacts to tools when they are invoked, but also direct messages (e.g., via sockets) between tools, and messages between Bubaak and tools.

The tool that is closest to Bubaak in the way how it allows tools to cooperate is CoVeriTeam [2] that allows a kind of meta-programming where verifiers are first-class objects in its domain-specific language. However, the language of CoVeriTeam is still very restricted and does not allow, e.g., to immediately react on a message in the standard output of a tool, which is something that Bubaak can do. CoVeriTeam focuses on combining off-the-shelf tools that are treated as black box. While using off-the-shelf tools has its benefits, it means that CoVeriTeam has access only to the output of the tools – standard (error) output and artifacts generated *after* the tool has finished its job. There is also no possibility to control the verifiers while they are running. In Bubaak, on the other hand, we assume the verifiers to be integrated and possibly manually modified to allow their monitoring and interception, which allows us to gain control over their execution. Naturally, off-the-shelf tools can be used by Bubaak too, but without the benefits that are brought by monitoring of their internals.

2 Software Architecture

The architecture of Bubaak is centered around *tasks* and their *rewriting*. Internally, a task executes a process, like a compiler or a verification tool, monitors its execution, and acts on events that occurred in the process or its outputs. There are also special tasks that do not execute any processes and, for example, only wait for other tasks to finish and aggregate their results.

Bubaak 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 should also

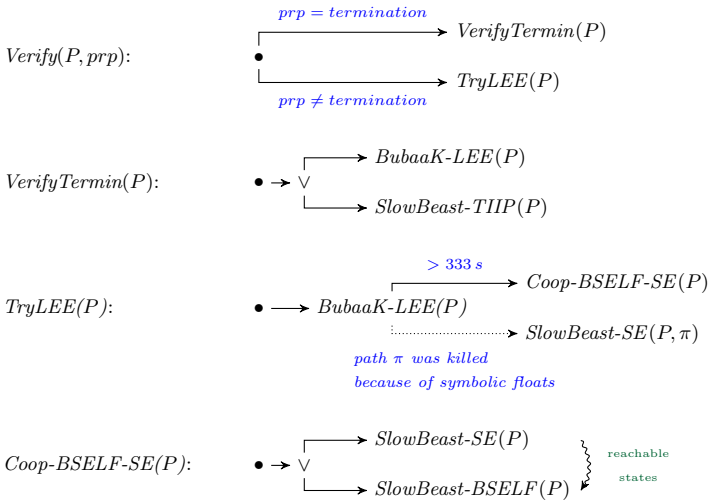


Fig. 1. The workflow of BUBAAK for SV-COMP 2025. For brevity, the scheme does not show errors handling and the result propagation.

specify how the results of the new tasks should be aggregated into a single result. A task is also allowed to spawn new sub-tasks before it has finished. Generating new tasks is not fixed in a static scheme: a task can spawn new tasks or rewrite itself into new tasks based on the context its has at hand and the information that was gathered so far from the finished and running tasks.

What tasks are executed and how they rewrite is defined by a selected *workflow*. The simplified workflow of BUBAAK in SV-COMP 2025 [1] is in Figure 1. The figure hides details like compilation into LLVM [9] (which is implemented also as a task), errors handling and the result propagation. The workflow starts with the task *Verify* that only rewrites itself into the task *VerifyTermin* if the analyzed property is *termination*, and into the task *TryLEE* otherwise.

Task *VerifyTermin* aggregates the results from two other tasks that it invokes; these tasks run *symbolic execution (SE)* provided by BUBAAK-LEE, and SLOWBEAST with *Termination with Inductive Invariants with Progress (TIIP)* [7]. The aggregated result is the one returned by the first tool that gives a conclusive result, or it ends up being *unknown/error*.

Task *TryLEE* runs BUBAAK-LEE for 333 seconds and if no result is reached by that time, the task rewrites itself into *Coop-BSELF-SE*. There may also occur the event that BUBAAK-LEE *killed* the search on a path that hit a computation involving symbolic floating point expressions. BUBAAK detects this event and spawns SLOWBEAST (in parallel to BUBAAK-LEE which keeps running) that replays the killed path and continues search from where BUBAAK-LEE stopped. This way, BUBAAK selectively combines the strengths of more mature and optimized implementation of symbolic execution in BUBAAK-LEE and the support of symbolic floats in SLOWBEAST.

Task *Coop-BSELF-SE* aggregates the results of two new tasks that it spawns. These tasks each run Slowbeast, one runs classical SE and the other runs *backward symbolic execution with loop folding (BSELF)* [8]. The algorithms run in parallel and during runtime, SE sends information about reached states into BSELF via a pipe-based channel. BSELF uses the information about reachable states to quickly filter out invalid candidates for invariants.

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.

Note that the workflow of Bubaak in SV-COMP 2025 is substantially different from the workflow in SV-COMP 2023, where Bubaak just ran BubaaK-LEE and Slowbeast in parallel without any cooperation. As a side note, there is also another workflow that competed in SV-COMP 2025 under the name Bubaak-SpLit. This workflow implements *dynamic program splitting*: the input program is split into two „smaller" programs and each of the two *splits* is analyzed by BubaaK-LEE for 16 s. If BubaaK-LEE is unable to finish within 16 s on a split, splitting is applied again on this split and the whole process continues until at most 128 splits are generated, at which point all unverified splits are analyzed by running two instances of Slowbeast in parallel: one that runs SE and the other BSELF (without any information exchange). It is very easy to implement such kind of workflows in Bubaak.

3 Strengths and Weaknesses

The workflow of Bubaak in SV-COMP 2025 builds on a combination of SE, which is very efficient in finding bugs, and BSELF that can prove programs correct. TIIP is in fact based on the very same cornerstones [7].

The dynamic task-based architecture allows Bubaak to implement many known cooperative verification schemes, be it cooperation through residual programs [4] or through dynamic information exchange. This architecture brings a plethora of possibilities to create powerful algorithms. For SV-COMP, however, we do not use the full strength of the architecture as a simple scheme described in Section 2 proved to be working well.

The main disadvantage for Bubaak in SV-COMP is that running multiple verifiers in parallel rapidly consumes CPU time. Also, none of the verifiers we use at the moment can properly deal with concurrency.

Results of Bubaak at SV-COMP 2025 In SV-COMP 2025, Bubaak took the 3rd place in the category *FalsificationOverall*. It has also scored well in some sub-categories. Mainly in the category *SoftwareSystems*, it performed well on the *ASW-C-Common* and *uthash* benchmarks.

Spawning Slowbeast to continue killed paths from BubaaK-LEE took place in 4708 benchmarks and 1681 were successfully decided by the joint forces of the two tools within the time limit 333 s for the task *TryLEE*.

The tool gave 19 wrong answers, mostly because of a restricted support for command-line arguments of the main function (the `argv` argument) and an unhandled failure of the SMT solver in Slowbeast.

Acknowledgments. This work was in part supported by the ERC-2020-AdG 10102009 grant, and in part by the German Research Foundation (DFG) – WE2290/13-2 (Coop2).

Data Availability Statement. The version of Bubaak that competed at SV-COMP 2025 is available at Zenodo [5]. The source code of Bubaak is available at GitLab [6].

References

1. Beyer, D., Strejček, J.: Report on SV-COMP 2025. In: Proc. TACAS. LNCS , Springer (2025)
2. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS 2022, Part I. LNCS, vol. 13243, pp. 561–579. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
3. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM 2022. LNCS, vol. 13550, pp. 111–128. Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
4. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA 2020, Part I. LNCS, vol. 12476, pp. 143–167. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
5. Bubaak artifact. Zenodo (2024). https://doi.org/10.5281/zenodo.14205712
6. Bubaak repository. `https://gitlab.com/mchalupa/bubaak` (2022)
7. Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers - (competition contribution). In: Proc. TACAS 2023, Part II. LNCS, vol. 13994, pp. 535–540. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
8. Chalupa, M., Strejček, J.: Backward symbolic execution with loop folding. In: Proc. SAS 2021. LNCS, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
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

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

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

EmergenTheta: Variations on Symbolic Transition Systems (Competition Contribution)

Milán Mondok, Levente Bajczi(✉), Dániel Szekeres, and Vince Molnár

Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics, Budapest, Hungary
{mondok,bajczi,szekeres,molnarv}@mit.bme.hu

Abstract. EmergenTheta is our sandbox for experimental analyses. After its successful debut in SV-COMP'24, we kept some well-performing but still under-tested configurations, and complemented them with a new saturation algorithm over decision diagrams, and two ways of extending their verification power: wrapping them in a lightweight, counterexample-guided abstraction refinement (CEGAR) loop based on implicit predicate abstraction; and backwards traversal of the state space. All such analyses now rely on a common interface to the underlying symbolic transition system, integrating seamlessly into the existing Theta framework. Using this combination of proven analyses and novel extensions, EmergenTheta outperformed our expectations in SV-COMP'25.

Funding. This research was partially funded by the EKÖP-24-3 New National Excellence Program under project numbers EKÖP-24-3-BME-78, EKÖP-24-3-BME-213, and EKÖP-24-3-BME-159, and the Doctoral Excellence Fellowship Programme under project numbers 400434/2023 and 400443/2023; funded by the NRDI Fund of Hungary.

1 Verification Approach

EmergenTheta parses the C programs and transforms them into the extended control flow automaton (XCFA) formalism through a series of preprocessing steps. XCFA are then transformed into the low-level symbolic transition system (STS) formalism, which captures the model behavior using two SMT formulas, one characterizing the initial states and the other characterizing the transition relation.

Bounded model checking (BMC) [8] can prove the faultiness of a model by constructing path constraints that characterize all possible counterexamples of a given length k and checking the satisfiability of such constraints using an SMT solver. The bound k is incremented until either the model is proven unsafe, or

L. Bajczi—Jury member representing EmergenTheta at SV-COMP 2025.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 217–222, 2025.
https://doi.org/10.1007/978-3-031-90660-2_15

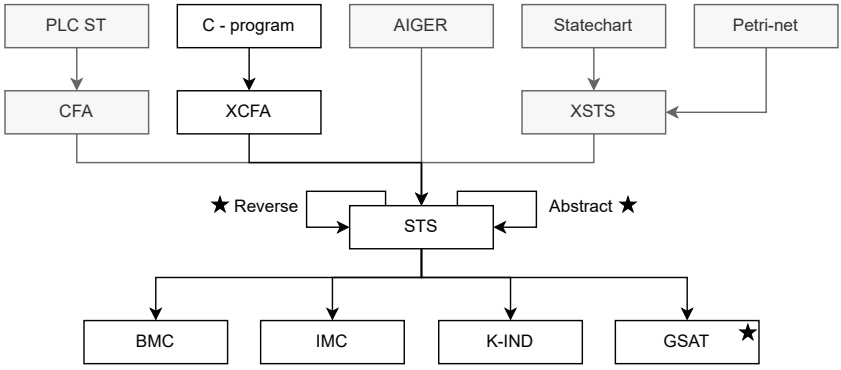


Fig. 1: Overview of the verification approach of EmergenTheta. 2025 additions are denoted with the ★ symbol. Parts irrelevant to SV-COMP are grayed out.

until the available resources allow. *K-induction* (K-IND) [17] and *interpolation-based model checking* (IMC) [13] extend BMC with additional checks in an attempt to prove that the model is safe. 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.

Decision-diagram-based algorithms are a novel addition to the set of available algorithms this year. *Substitution diagrams* [15] allow us to handle SMT formulas as multi-valued decision diagrams (MDDs) in a top-down approach supported by efficient caching and lazy evaluation for the presence of edges. Using substitution diagrams, we can represent the states and transitions of the system with MDDs constructed from the SMT formulas of the STS representation. The *saturation* algorithm [9] can compute the set of reachable states on decision diagrams in a bottom-up manner. It decomposes the exploration into smaller explorations on submodels exploiting the locality of the system's events. The *generalized saturation* (GSAT) [14] algorithm employs a weaker notion of locality to further enhance the saturation effect.

The modular architecture of EmergenTheta also allows us to wrap the aforementioned analyses into a CEGAR loop based on *implicit predicate abstraction* [18] and to traverse the state space in a *reversed* manner.

We run the aforementioned analyses in a sequential portfolio. We start with the GSAT algorithm (because it can find a safety proof fairly quickly, or report the task not solvable with the configuration), then move on to BMC and K-Induction. Afterwards, we intended to start the *abstracted* and *reversed* IMC, but ended up using K-Induction due to a bug (see Sect. 3).

Configuration	unreach-call safe	unreach-call unsafe	termination safe	termination unsafe	no-overflow safe	no-overflow unsafe
Preprocessing	1	0	51	0	232	0
GSAT	152	33	0	0	0	0
BMC	209	134	11	38	34	143
K-Induction	380	34	4	1	6	4
Abstract K-Induction	37	0	112	0	63	1
Reversed K-Induction	2	0	0	0	0	0
Sum	781	201	178	39	335	148
Wrong	2	0	20	0	15	0

Fig. 2: Successful verification results per configuration

2 Software Architecture

EmergenTheta is a JVM-based tool, written primarily in Kotlin (and legacy parts of the code are written in Java).

EmergenTheta is part of the Theta framework [12], and can therefore use its integrated frontend- and solver-infrastructure. Parsing C files is done using a custom ANTLR grammar [2]. Satisfiability Modulo Theories (SMT) solvers can be invoked three different ways:

- Z3 [16] is natively integrated, its Java API is a dependency of Theta
- Other solvers are integrated via SMT-LIBv2 [11], such as MathSAT5 [10] or CVC5 [5]
- JavaSMT [1] has been integrated since last year's SV-COMP

In SV-COMP'25, we use Z3, MathSAT5, and CVC5.

3 Discussion of Strengths and Weaknesses of the Approach

EmergenTheta took part in SV-COMP'25 with 3 checkable properties: the conventional unreach-call (as previously [3]); and we debuted with termination and no-overlow this year using a pre-processing step using TransVer [7]. This broadened verification reach is a definitive strength of our tool, but cannot be attributed to its architecture, as the pre-processing step is transparent to EmergenTheta. In the future we plan to integrate it into our portfolio more, as experiences with reachability need not translate to other properties when it comes to verification efficacy (based on results in Figure 2 [6]).

It is also worth mentioning that a large number of tasks (232 for no-overflow, 51 for termination, but only 1 for unreach-call) were solved by sanity checks *before* a verifier algorithm was started. In these instances safety was evident, as in the control flow graph, we do not have any paths from the initial location to the error locations.

Unfortunately, the portfolio implementation contained an oversight, making IMC-based analyses become K-Induction-based analyses by mistake. This is reflected in our results: we wanted to run IMC with abstraction and with reversal, but ended up running K-Induction in these instances, and these represent our least efficient configurations (see Figure 2). We will move to a more robust portfolio-engine in the future so that oversights like this can be more easily avoided. However, both abstraction and reversal solved tasks that the non-reversed and not-abstracted configurations did not, therefore, we are confident in their advantages overall.

The new algorithm, GSAT, produced really promising results this year. It proved safety in 152 tasks, and found a bug in 33 instances. However, it could only output correctness witnesses this year, and no violation witnesses, so the 33 successful verification tasks are correct, but unconfirmable.

EmergenTheta produced a few wrong results. In the unreach-call category it produced 2, in termination 20, and in no-overflow 15 false positive results. It produced no false negative (false safe) results. The cause of the wrong results are partly due to a mishandling of floating point NaN values, and due to the preprocessing step mapping the new properties to reachability.

4 Tool Setup and Configuration

EmergenTheta is widely configurable, and choosing a successful configuration for a verification task at hand can be complicated. If using the competition archive [4] for software verification, we recommend using the following input options, starting our EMERGENT portfolio:

```
theta-start.sh <input> --portfolio EMERGENT --svcomp
```

To minimize the output verbosity, the option `--loglevel RESULT` can be added to the arguments. We also used these options at SV-COMP 2025.

5 Software Project and Data-Availability Statement

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 under an Apache 2.0 license[2], and the binary release of the competition version (`6.8.6`) is published on Zenodo [4].

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. pp. 195–208. Springer International Publishing, Cham (2021)

[1] https://ftsrg.mit.bme.hu/en/
[2] https://github.com/ftsrg/theta/releases/tag/svcomp25

2. Bajczi, L., Ádám, Z., Molnár, V.: C for Yourself: Comparison of Front-End Techniques for Formal Verification. In: 10th FormaliSEICSE 2022. pp. 1–11. ACM (2022). https://doi.org/10.1145/3524482.3527646
3. 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: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 371–375. Springer Nature Switzerland, Cham (2024)
4. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Mondok, M., Molnár, V.: EmergenTheta - SV-COMP'25 Verifier Archive (Nov 2024). https://doi.org/10.5281/zenodo.14194484
5. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. pp. 415–442. Springer International Publishing, Cham (2022)
6. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
7. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Xia, T., Zheng, X.: A modular program-transformation framework for reducing specifications to reachability (2025), https://arxiv.org/abs/2501.16310
8. 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
9. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state—space generation. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 328–342. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
10. 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
11. Dobos-Kovács, M., Vörös, A.: Evaluation of SMT solvers in abstraction-based software model checking. In: Proceedings of the 11th Latin-American Symposium on Dependable Computing. p. 109–116. LADC '22, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3569902.3570187
12. 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
13. 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
14. Molnár, V., Majzik, I.: Saturation Enhanced with Conditional Locality: Application to Petri Nets. In: Donatelli, S., Haar, S. (eds.) Application and Theory of Petri Nets and Concurrency. pp. 342–361. Springer International Publishing, Cham (2019)
15. Mondok, M., Molnár, V.: Efficient Manipulation of Logical Formulas as Decision Diagrams. In: Renczes, B. (ed.) Proceedings of the 31st PhD Mini-Symposium. pp. 61–65. Budapest University of Technology and Economics, Department of Measurement and Information Systems (2024). https://doi.org/10.3311/MINISY2024-012
16. 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

17. 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
18. Tonetta, S.: Abstract Model Checking without Computing the Abstraction. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009: Formal Methods. pp. 89–105. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

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

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

ESBMC v7.7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction

(Competition Contribution)

Tong Wu[1(✉)], Xianzhiyu Li[1], Edoardo Manino[1], Rafael Sá Menezes[1,2], Mikhail R. Gadelha[3], Shale Xiong[5], Norbert Tihanyi[4], Pavlos Petoumenos[1], and Lucas C. Cordeiro[1,2]

[1] The University of Manchester, Manchester, UK
tong.wu-11@postgrad.manchester.ac.uk
[2] Federal University of Amazonas, Manaus, Brazil
[3] Igalia, A Coruña, Spain
[4] Eötvös Loránd University, Budapest, Hungary
[5] Arm®, Cambridge, UK

Abstract. ESBMC v7.7 improves the verification of concurrent C programs by incorporating techniques such as dynamic thread scheduling, incremental SMT solving, and partial order reduction (POR). These improvements enhance the tool's performance, particularly in exploring complex multi-threaded executions. The new scheduler prioritizes higher-thread identifiers during context switches, which helps explore deeper program states. The use of incremental SMT solving and a refined POR algorithm reduces the exploration of unreachable interleavings and redundant states. These updates enable ESBMC to detect bugs faster, making it a more effective tool for ensuring the safety of multi-threaded applications.

1 Software Architecture

The Efficient SMT-based context-Bounded Model Checker (ESBMC) [6,7,5] is a context-bounded model checker for the verification of single- and multi-threaded software. It uses a Clang-based [9] front-end to transform the input C program into an intermediate representation in the GOTO language [4]. Then, it employs symbolic execution to generate SMT formulae and pass them to a selection of SMT solvers. For the verification of multi-threaded software, ESBMC constructs the reachability tree by depth first search under Sequential Consistency [1]. This way, it can symbolically and explicitly explore all possible sequential executions up to a (bounded) number of context switches. As a result, ESBMC can automatically identify many concurrency-related issues such as race conditions and deadlocks.

T. Wu–Jury member

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 223–228, 2025.
https://doi.org/10.1007/978-3-031-90660-2_16

2 Verification Approach

Reverse Priority Scheduling. ESBMC v7.7 introduces an advanced thread scheduling algorithm that dynamically prioritizes thread selection when a context switch occurs. In earlier versions, the scheduler would always search for eligible threads in ascending order of thread identifier t_i,[6] starting with the main thread id $t_i = 0$ [5]. The new approach modifies this behavior as follows:

① It first attempts to identify newer thread, which have identifier t_i higher than the current thread t_c. Among these threads, the scheduler selects the one with the smallest identifier that is eligible for scheduling.
② If no newer threads are available, the scheduler reverses the search direction and looks for older eligible threads, which have identifier $t_i \leq t_c$. Note that the scheduler can still choose t_c, which indicates it will continue executing the following steps in the current thread. In this case, it selects the eligible thread with the largest identifier.
③ If no eligible threads are found in either direction, there is no further interleaving possible in the current execution state. Thus, the scheduler reverts to exploring unexplored states by popping the current state from the reachability tree and backtracking.

The formal policy σ for our reverse priority scheduling can be expressed as:

$$\sigma(t_c, S) = \begin{cases} \min\{t_i | t_i \in S \wedge t_i > t_c\} & \text{if } \max S > t_c \quad ① \\ \max S & \text{if } \max S \leq t_c \quad ② \\ \emptyset & \text{if } S = \emptyset \quad ③ \end{cases}$$

where t_c is the current thread and S represents the set of all threads eligible for scheduling. In general, this strategy prioritizes interleavings with newly-created threads, enabling ESBMC to explore new execution paths earlier, which allows ESBMC to find bugs six times faster (see Section 3).

Incremental SMT Solving. ESBMC leverages incremental SMT solving [12] in an attempt to reduce the exploration of unreachable interleavings. Specifically, non-incremental mode only calls the SMT solver once when reaching the end of an interleaving. As such, it has no early mechanism to determine which boolean conditions hold (e.g., assumptions), thus producing a formula that may still contain some unreachable states. In contrast, incremental mode removes all unreachable states by checking goto guards (if and loop conditions), thread guards (interleaving conditions) and assertions immediately [10]. This is achieved through the push/pop interface offered by many state-of-the-art solvers [3,11,12,2]. Figure 1 illustrates the different behaviour after the thread guard `pthread_join`. In ESBMC v7.7, we add incremental checking of assumptions with the `--smt-symex-assume` option.

[6] New threads are created with higher identifiers in ESBMC.

```
1  #include <pthread.h>
2  int x = 0;
3  void *thread1() {
4    x = 1;
5    return 0;
6  }
7  int main() {
8    pthread_t t1;
9    pthread_create(&t1,
         0, thread1, 0);
10   pthread_join(t1, 0);
11   x = 2;
12 }
```

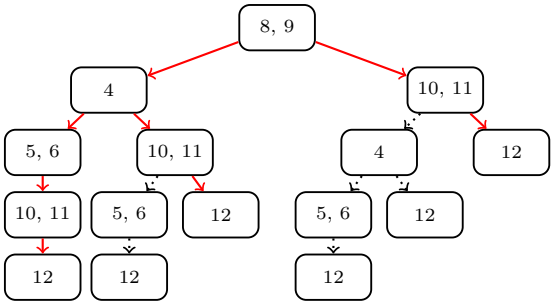


Fig. 1. Reachability tree (right) of a concurrent program (left). Boxes are execution states (by line number), red arrows are reachable paths, black dotted arrows are unreachable paths that can be cut by incremental SMT.

Partial Order Reduction. ESBMC employs an optimal partial order reduction algorithm based on the normal form in [8] to eliminate redundant equivalent interleavings during model checking. In ESBMC v7.7, we refine our implementation by performing a more accurate analysis of shared read/write variables that are accessed by global and local pointers. Specifically, memory leak checks are deferred until the main thread has terminated. This strategy avoids false positives, since in SV-COMP the check should happen after program termination.

Data Races. ESBMC v7.7 offers extended support for data race checking. The base method introduces a boolean flag b_x for each variable x involved in an assignment. When x is updated, b_x is set to true before the assignment and reset to false immediately after. To identify data races, an assertion ensures that b_x is false whenever x is accessed. Previous versions relied on unique flag identifiers generated from variable names. As such, the same memory address could be protected by multiple flags, especially for arrays with non-constant indices and members of compound types. The new version eliminates the dependency on variable names. Memory is represented as an infinite-size array, where variables are encoded as addresses and used as indices of the array. Additionally, we force an extra context switch when setting each flag to increase the probability of triggering a data race earlier.

Pthread Operational Models. ESBMC v7.7 features improved operational models for the pthread library, which reduces the number of unnecessary context switches. Specifically, we restrict context switches to occur exclusively on user program variables which are shared by at least two threads.

3 Strengths and Weaknesses

Table 1 reports the relative performance of each individual technique in Section 2, compared to the previous version of ESBMC. The last row reports their cumulative effect in ESBMC v7.7, which has Incremental SMT disabled and POR

partially enabled by default (more information can be found in zenodo dataset). All results are computed on the SV-COMP'25 Concurrency Safety benchmarks.

Table 1. Experimental Results for ESBMC Improvements

Individual Technique	Correct True	Correct False	Incorrect True	Incorrect False
Reverse Priority Scheduling	+66	+18	+8	+7
Incremental SMT Solving	-3	+1	+3	+6
Partial Order Reduction	+16	-20	+15	+0
Data Races	-5	+16	-9	-14
Pthread Operational Models	+31	+3	+5	+3
ESBMC v7.7	**+97**	**+21**	**+13**	**+10**

In addition, our experiments show that Reverse Priority Scheduling accelerates bug detection by approximately 600%, and the first interleaving is enough to reach a bug in 59 additional instances. Furthermore, Incremental SMT achieves an average of 53% fewer interleavings on the 248 instances verified by both approaches and produces 18 unique correct results that would otherwise timeout. However, the repeated solver calls increase verification time, with fewer successful verifications than the non-incremental mode. Finally, Partial Order Reductions reduce the verification time required to prove correctness by 40%.

At the same time, the improvements expose 23 new incorrect results that were previously timeouts. Additionally, the Partial Order Reduction improvements turn 20 correct results into 15 incorrect results and 5 timeouts. These are mainly due to the low context-switch limit of 3 we set and the absence of a mechanism to detect shared memory access between local variables and pointers. We plan to address these issues in future work.

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 Apache License 2.0. The participated version in SV-COMP 2025 is available at https://doi.org/10.5281/zenodo.13867976. A refined version and all of our experiments data are available at https://doi.org/10.5281/zenodo.14534503.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer **29**(12), 66–76 (1996). https://doi.org/10.1109/2.546611
2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., et al.: cvc5: A versatile and industrial-strength SMT solver. In: TACAS 2022, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: TACAS 2009, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. LNCS, vol. 5505, pp. 174–177. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_16
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004. pp. 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
5. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using smt-based context-bounded model checking. In: ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. pp. 331–340. ACM (2011). https://doi.org/10.1145/1985793.1985839
6. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering **38**(4), 957–974 (2012). https://doi.org/10.1109/TSE.2011.59
7. Gadelha, M.Y.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: an industrial-strength C model checker. In: ASE. pp. 888–891. ACM (2018). https://doi.org/10.1145/3238147.3240481
8. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. LNCS, vol. 5643, pp. 398–413. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_31
9. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: CGO. pp. 75–88. San Jose, CA, USA (Mar 2004). https://doi.org/10.1109/CGO.2004.1281665
10. Morse, J.: Expressive and efficient bounded model checking of concurrent software. Ph.D. thesis, University of Southampton, UK (2015), https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.658818
11. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621

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

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

Mopsa-C with Trace Partitioning and Autosuggestions (Competition Contribution)

Raphaël Monat[1](✉), Abdelraouf Ouadjaout[2], and Antoine Miné[2]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, 59000 Lille, France
raphael.monat@inria.fr
[2] LIP6, Sorbonne Université, 75005, Paris, France

Abstract. We present advances we brought to Mopsa for SV-Comp 2025. Most notably, Mopsa now supports bounded trace partitioning, constant widening with thresholds, and can check that all memory has been correctly deallocated. Further, Mopsa now integrates a sound support of bitfields. While Mopsa at SV-Comp previously relied on a fixed, homogeneous set of configurations to verify tasks, it can now automatically leverage semantic information from a previous analysis to trigger heuristic precision improvements in further analyses. With these improvements, Mopsa wins a silver medal in the *SoftwareSystems* category and ranks fifth in the *NoOverflows* category.

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 [9]. The implementation of Mopsa aims at exploring new perspectives for the design of static analyzers. Journault et al. [13] describe the core of Mopsa principles, and Monat [22, 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 [27]; it proceeds by induction on the syntax, is fully context- and flow-sensitive, and committed to be sound. This is the third time Mopsa participates in SV-Comp [24, 23]. We have brought several enhancements, including major precision improvements, described below.

Trace partitioning. Mopsa now supports a variant of trace partitioning [16]. Trace partitioning keeps some abstract states separate (depending on the analysis trace) to improve precision. In our implementation, we keep a small, bounded abstract trace to separate abstract states while maintaining full analysis coverage. The abstract trace consists in the k latest trace markers. Currently, trace markers correspond to control conditions (`if`, `switch`, different `return` locations), and case disjunctions when handling C stubs [27].

R. Monat—Jury member

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 229–235, 2025.
https://doi.org/10.1007/978-3-031-90660-2_17

Widening with constant thresholds. Mopsa relies on the traditional abstract interpretation use of widening [9] to enforce finite convergence when analyzing loops. However these widening operators may generalize some constraints too quickly, which can be difficult to recover from. To address this issue, Mopsa now supports the standard widening with thresholds [4]. It is implemented as a plug-in, which observes the analyzer and performs some constant propagation to decide relevant thresholds for each variable. This approach greatly improves precision when analyzing loops guarded by (in)equalities.

Memory deallocation check. Mopsa can prove that programs have successfully deallocated all memory. During the end of the analysis of a program, it queries the recency abstraction [2] to ensure that all memory dynamically allocated through `malloc` and other glibc functions has been properly deallocated. This allows us to support the *MemCleanup* property of SV-Comp (especially in the corresponding `uthash` subcategory of *SoftwareSystems*).

Sound bitfield support. The low-level C memory representation of cells [20] we use works at the byte level, making bitfield reasoning hard. We have thus implemented a domain translating bitfield operations into equivalent byte-level operations with bitmasks. This approach is currently sound, although imprecise. The precision could be improved through new numeric abstractions in the future.

Other improvements. Mopsa leverages relational numeric domain through Apron's interface [12], relying on static packing [4] to remain scalable. In order to improve precision for small and intricate programs, the last configuration used in our SV-Comp driver (Section 2) does not rely on packing (i.e., all variables are used in the same polyhedron). Noticing performance improvements in this case, we decided to rely on the PPLite polyhedra implementation [3]. Mopsa does not support the analysis of recursive functions. We added support to unroll recursive functions of bounded depth.

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 sequentially tries 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 control of the precision-efficiency ratio. A breakdown of the results is shown in Fig. 1. Similarly to last year [23], we use five configurations. Confs. 1 and 3 are unchanged from last year. Conf. 2 now only unrolls the first 2 iterations of loops (down from 10 last year – but the precision suggestion hook mentioned below can override this parameter). Conf. 4 adds the bounded trace partitioning, where up to 7 trace markers can be kept. Conf. 5 also enables bounded trace partitioning (with up to 10 trace markers), and relies on PPLite without static packing for best precision.

Max. Conf.	Tasks proved correct		Tasks reaching 900s timeout	
1	7002		389	
2	7743	(+741)	970	(+581)
3	8489	(+746)	3377	(+2407)
4	8660	(+171)	5378	(+2001)
5	8933	(+273)	8440	(+3062)

Fig. 1. Results of our sequential portfolio at SV-Comp 2025. Max. Conf. i represents the sequence of increasingly precise analyses from Conf. 1 up to Conf. i. Max. Conf. 2 is able to prove 741 tasks correct in addition to the 7002 proved by Conf. 1, although 970 tasks reach the resource limits when analyzed by Conf. 1 and 2 (581 more than by Conf. 1 alone). There are 33592 tasks in total, including 22356 correctness tasks. Mopsa can only prove program correctness for now; it yields "unknown" when unable to prove a program correct.

Heuristic Autosuggestions. An important improvement is that the set of configurations is not fixed and uniform for all programs anymore. Each analysis can suggest enabling options that will improve the precision of further analyses. These options are decided by a plug-in observing the analysis of Mopsa. It currently supports three semantic heuristics:

Bounded recursion unrolling. Upon detection of a call to a recursive function `f` having parameter `n`, if `n` lies in interval $[0, \texttt{hi}]$, and $\texttt{hi} < 1000$, further analyses will inline recursive functions up to bound `hi`.[3]

Loop unrolling for precise allocations. If the program contains a loop, for which we can semantically infer that (i) there are less than 30 iterations and that (ii) iterations perform allocations, further analyses will unroll this loop to keep the allocated memory blocks distinct and enhance precision.

Single loop unrolling. If the program semantically reaches a single loop, for which we can infer an upper bound on the number of iterations, further analyses will fully unroll this loop.

3 Strengths and Weaknesses

Mopsa participated in the following categories, targeting C programs: *ReachSafety, MemSafety, NoOverflows* and *SoftwareSystems*. An overview of results can be found in the competition report [5]. Figure 2 highlights the progress made by Mopsa in specific subcategories, thanks to the improvements brought for this year's edition. Note that in two subcategories, the score of Mopsa decreased due to our now-sound bitfield encoding.

Strengths. Mopsa is quite scalable: our cheapest configuration is able to analyze a given program within the allocated resource budget in 98.8% of the cases. Thanks to this scalability, Mopsa is particularly competitive in the *SoftwareSystems* track, focusing on verifying real software systems. This year, Mopsa ranked

[3] In other cases, Mopsa relies on the function's prototype to return the top value, and assumes the recursive function has no side-effects. To keep this approach sound, our SV-Comp driver thus returns `unknown` whenever a recursive function has been encountered during the analysis.

Category	Prop.	|tasks|	Mopsa'24	Mopsa'25	Best score, verifier (2025)	
Hardness	R	4012	432	518	7426	SVF-SVC [17]
Heap	R	240	190	226	314	PredatorHP [29]
Loops	R	774	298	376	1031	AISE [34, 14]
Recursive	R	160	12	60	150	UTaipan [10]
Heap	M	247	40	154	331	PredatorHP [29]
Juliet	M	3271	2224	2530	4709	CPAchecker [1]
LinkedLists	M	134	58	96	220	PredatorHP [29]
Main	N	1989	1920	2138	2756	UAutomizer [11]
AWS	R	341	36	76	326	Bubaak [6, 8]
DDL	R	2420	3476	3602	3602	Mopsa
uthash	M	192	96	108	246	Bubaak* [6, 8, 7]
uthash	N	162	204	300	300	Mopsa

Fig. 2. Mopsa's improvements for selected subcategories of the *ReachSafety, MemSafety, NoOverflows* and *SoftwareSystems* tracks, comparing the scores reached at SV-Comp 2024 and 2025. 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.

second with 2164 points, closely trailing CPAchecker [1] with 2238 points. It is the best verifier in the *DeviceDriversLinux-ReachSafety* and the *uthash-NoOverflows* subcategories. In *uthash-NoOverflows* there are only two verifiers able to score points; the second is UAutomizer [11], with 6 points. The second strength of Mopsa lies in the *NoOverflows* track, where it ranked fifth, with results near those of Goblint [30], which is the first abstract-interpretation based verifier to enter the competition.

Weaknesses. Mopsa can only prove programs correct for now, and is currently unable to provide counterexamples otherwise. We plan to leverage the recent works of Milanese and Miné [18, 19] to address this issue. Mopsa does not support the termination property, and cannot precisely analyze concurrency-related verification tasks, but we could leverage previous abstract interpretation work targeting those properties [32, 31, 21, 33]. Our SV-Comp driver currently tries a sequence of increasingly precise configurations: this approach is not efficient, we are planning to develop techniques deciding what would be the best configuration to analyze a given program, following the works of Oh et al. [26], Mansur et al. [15], Wang et al. [35].

4 Software Project and Contributors

Mopsa is available on Gitlab [28], 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 2025 are the authors of this paper.

Data-Availability Statement. The exact version of Mopsa and the driver that participated in SV-Comp 2025 are available as a Zenodo archive [25].

Bibliography

[1] Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpachecker 2.3 with strategy selection - (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 359–364, Springer (2024)

[2] 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)

[3] Becchi, A., Zaffanella, E.: Pplite: Zero-overhead encoding of NNC polyhedra. Inf. Comput. **275**, 104620 (2020)

[4] Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. Foundations and Trends in Programming Languages pp. 71–190 (2015)

[5] Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS, LNCS, Springer (2025)

[6] Chalupa, M., Henzinger, T.A.: Bubaak: Runtime monitoring of program verifiers - (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 535–540, Springer (2023)

[7] Chalupa, M., Richter, C.: Bubaak-split: Split what you cannot verify (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 353–358, Springer (2024)

[8] Chalupa, M., Richter, C.: Bubaak: Dynamic cooperative verification (competition contribution). In: Proc. TACAS, LNCS, Springer (2025)

[9] 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)

[10] Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate taipan and race detection in ultimate - (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 582–587, Springer (2023)

[11] Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: Ultimate automizer and the commuhash normal form - (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 13994, pp. 577–581, Springer (2023)

[12] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV, pp. 661–667, Springer (2009)

[13] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, pp. 1–18 (2019)

[14] Lin, Y., Chen, Z., Wang, J.: AISE v2.0: Combining loop transformations (competition contribution). In: Proc. TACAS, LNCS, Springer (2025)

[15] Mansur, M.N., Mariano, B., Christakis, M., Navas, J.A., Wüstholz, V.: Automatically tailoring abstract interpretation to custom usage scenarios.

In: CAV (2), Lecture Notes in Computer Science, vol. 12760, pp. 777–800, Springer (2021)
[16] Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: ESOP, Lecture Notes in Computer Science, vol. 3444, pp. 5–20, Springer (2005)
[17] McGowan, C., Richards, M., Sui, Y.: SVF-SVC: Software verification using SVF (competition contribution). In: Proc. TACAS, LNCS, Springer (2025)
[18] Milanese, M., Miné, A.: Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In: VMCAI, Springer (2024)
[19] Milanese, M., Miné, A.: Under-approximating memory abstractions. In: Proc. of the 31th International Static Analysis Symposium (SAS'24), Lecture Notes in Computer Science (LNCS), vol. 14995, Springer (Oct 2024), http://www-apr.lip6.fr/~mine/publi/article-milanese-al-sas24.pdf
[20] Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES (2006)
[21] Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: VMCAI, Lecture Notes in Computer Science, vol. 8318, pp. 39–58, Springer (2014)
[22] Monat, R.: Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries. Ph.D. thesis, Sorbonne Université, France (2021)
[23] 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: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 387–392, Springer (2024)
[24] 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)
[25] Monat, R., Ouadjaout, A., Miné, A.: Mopsa at sv-comp 2025 (Nov 2024), https://doi.org/10.5281/zenodo.14208644
[26] Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. In: PLDI, pp. 475–484, ACM (2014)
[27] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, pp. 223–247 (2020)
[28] Ouadjaout, A., Monat, R., Miné, A., Journault, M.: Mopsa (2022), URL https://gitlab.com/mopsa/mopsa-analyzer
[29] Peringer, P., Soková, V., Vojnar, T.: Predatorhp revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: TACAS (2), Lecture Notes in Computer Science, vol. 12079, pp. 408–412, Springer (2020)
[30] 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)

[31] Urban, C.: Function: An abstract domain functor for termination - (competition contribution). In: TACAS, Lecture Notes in Computer Science, vol. 9035, pp. 464–466, Springer (2015)
[32] Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: SAS, Lecture Notes in Computer Science, vol. 8723, pp. 302–318, Springer (2014)
[33] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblint approach. In: ASE, pp. 391–402, ACM (2016)
[34] Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 347–352, Springer (2024)
[35] Wang, Z., Yang, L., Chen, M., Bu, Y., Li, Z., Wang, Q., Qin, S., Yi, X., Yin, J.: Parf: Adaptive parameter refining for abstract interpretation. In: ASE, pp. 1082–1093 (2024)

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

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


Nacpa: Native Checking with Parallel-Portfolio Analyses
(Competition Contribution)

Thomas Lemberger(✉) and Henrik Wachowitz

LMU Munich, Munich, Germany
thomas.lemberger@sosy.ifi.lmu.de

Abstract. We present Nacpa, a meta-verifier based on parallel portfolio and native compilation of backend verifiers. Nacpa does not implement any software analyses itself, but uses the Java-based CPAchecker as off-the-shelf verification backend in different configurations; each called as a separate, external process. To avoid the overhead of starting the Java Virtual Machine multiple times and to improve the run time on fast-to-solve tasks, we created a natively compiled version of CPAchecker for Nacpa. Nacpa is a conceptually simple framework, yet proves to be competitive in SV-COMP 2025.

1 Verification Approach

Nacpa is a meta-verifier for C that implements an efficient parallel portfolio on the basis of an external verifier—currently CPAchecker [1, 2]—with two goals: (1) create a technologically simple parallel portfolio of different verifier configurations, and (2) achieve faster startup times than CPAchecker can achieve in its default distribution that is based on the Java Virtual Machine (JVM). Nacpa does not implement any new program analysis but delegates all analysis tasks to the external verifier.

Parallel Portfolio. SV-COMP [3] gives verifiers a time limit of 900 s *CPU time* per task. This provides verifiers with a strong incentive not to run analyses concurrently, but sequentially, so that CPU time is only used when beneficial. But verifiers employing a sequential portfolio of analyses—like CPAchecker before version 4.0—have a practical problem: The analyses are executed in a fixed sequence and the analysis that can successfully solve a verification task may be scheduled late in that sequence. This means that it can take a significant amount of time to solve an otherwise fast-to-solve verification task. While this approach is valid for SV-COMP, where it—in essence—only matters that a result is produced within the *CPU* time limit, it is not a good day-to-day experience for users and makes debugging difficult.

Nacpa addresses this issue by executing all analyses in parallel. Figure 1 shows the workflow of Nacpa. After receiving a program P and a specification φ, Nacpa first extracts features from the program under verification by calling CPAchecker

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 236–241, 2025.
https://doi.org/10.1007/978-3-031-90660-2_18

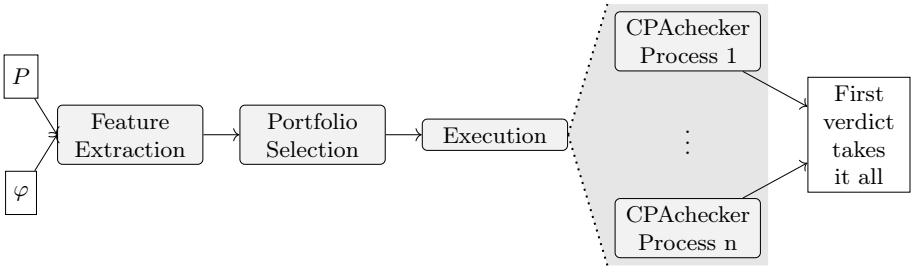


Fig. 1: Workflow of Nacpa

with a configuration [2,4] that extracts and outputs these features. Based on the features, Nacpa selects one of multiple pre-defined analysis portfolios (each analysis portfolio is a set of configurations together with a wall-time limit). For each analysis in the selected portfolio, Nacpa launches a separate verifier process. The first process that finishes with a verdict of `TRUE` or `FALSE` wins, and Nacpa reports this verdict. If all processes finish with `UNKNOWN`, Nacpa returns `UNKNOWN`.

While Nacpa currently only calls CPAchecker and no other verifiers, we use CPAchecker as-is and call it as an external process. Because of this, we are certain that this external parallelization generalizes well to other verifiers.

Faster Startup Time. Many successful verifiers are implemented in Java [2,5–8]. But setting up the JVM, loading all the classes and starting the verifier can take several seconds [9]. Multiplying this overhead by the number of analyses in the portfolio can lead to a significant overhead. To improve on this, we compile CPAchecker to a native executable with Oracle GraalVM. This not only reduces the overhead of the parallel portfolio but also significantly improves the run time for fast-to-solve tasks, compared to the traditional CPAchecker.

2 Software Architecture

Nacpa 1.0 leverages CPAchecker (revision b24a0863) as verification backend. We use Oracle GraalVM 22.0.1 to compile this revision of CPAchecker to a native executable. The native executable is bundled with the necessary third-party Libraries, e.g., MathSAT 5, and the necessary configurations for Nacpa.

Nacpa itself is written in Go because it has parallelization directives built into the language and compiles to a statically linked executable with low overhead.

Portfolio Selection. Nacpa uses seven hard-coded portfolios. The configurations of these portfolios are taken from CPAchecker and are semantically equivalent to the configurations used by CPAchecker in SV-COMP 2025, with two exceptions: (1) we discard almost all time limits (see below) and (2) if the program contains recursive function calls or concurrency features (`pthread_create`), we add configurations for recursive and concurrent programs to the selected portfolio. This is different from CPAchecker's SV-COMP submission, which starts with analyses that do not support recursion or concurrency, and only switches to supporting

configurations when the original analyses fail because one of these features is encountered. Compared to CPAchecker, Nacpa's approach to recursion and concurrency is simpler, but solves the same number of tasks.

Managing the Portfolio. Nacpa starts a separate process for each analysis. Nacpa then simultaneously waits on all processes to finish. Whenever a process finishes, Nacpa parses the produced console output for the reported verdict. If the run did not crash and the verdict is `TRUE` or `FALSE`, Nacpa terminates all remaining processes and reports this verdict to the user. Otherwise, Nacpa continues to wait on the remaining processes.

Resource Limits. To enforce time limits on the individual analysis runs, we rely on Go's internal process management. Nacpa runs most analyses without any time limit. There are two exceptions: data-flow analysis and symbolic execution are limited to only a few seconds of runtime each (5 s and 10 s, respectively), because these analyses only help on fast-to-solve tasks. Nacpa does not enforce any memory limits on the individual analysis runs. As soon as one analysis runs into the SV-COMP memory limit, Nacpa dies.

Native Compilation. Compiling a large project like CPAchecker to a native binary poses several challenges [9]. The biggest challenge for Nacpa is the extensive use of Java reflection in CPAchecker: When GraalVM builds the native binary, it only includes classes that are reachable from the program entry. It is not able to derive classes that are reached through code reflection, and these will miss during run time. To avoid this we need to tell GraalVM about them when starting the compilation process. We contribute a build script that collects this reflection information from exemplary CPAchecker runs.

3 Strengths and Weaknesses

Regarding its analysis, Nacpa fully depends on CPAchecker [2] and shares all strengths and weaknesses. The parallel portfolio and native compilation introduce some additional strengths and weaknesses.

Strengths. The parallel portfolio of Nacpa is conceptually simple, with the implementation consisting of less than 800 lines of source code. Nacpa is also conceptually independent of CPAchecker. It implements a parallel portfolio that is independent of CPAchecker's internal parallel portfolio. Other verifiers can be used in the backend by adding the command-line to call to Nacpa and adjusting the verdict parsing for the new output.

Specific to CPAchecker, Nacpa significantly speeds up the analysis for fast-to-solve verification tasks (similar to the speed up CPA-Daemon provides with its native backend [9]).

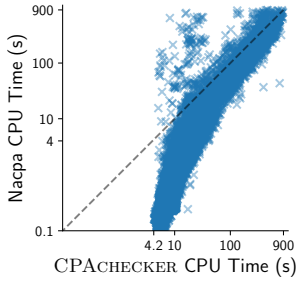


Fig. 2: Comparison of the CPU-time seconds spent by CPAchecker and Nacpa for each solved task

Figure 2 shows the CPU time seconds that CPAchecker (x-axis) and Nacpa (y-axis) require for each verification task in SV-COMP 2025 [3]. The plot only shows data for tasks that both CPAchecker and Nacpa solved correctly. Each data point below the diagonal represents a task for that Nacpa is faster than CPAchecker, and each data point above the diagonal represents a task for that CPAchecker is faster than Nacpa. The plot shows that Nacpa is significantly faster than CPAchecker for a large number of tasks. Nacpa's fastest correct verification run (0.11 s CPU time) is a magnitude faster than CPAchecker's (4.2 s CPU time).

On a small number of tasks, Nacpa's different configuration of parallel-portfolio strategies leads to some better verification results than CPAchecker: Nacpa solves 120 tasks in SV-COMP 2025 that CPAchecker can not solve. This accounts for 0.5 % of all tasks that Nacpa solved correctly.

Weaknesses. Because Nacpa splits its strategies into separate processes, the exchange of information is more difficult than in a multi-threaded approach like CPAchecker's internal parallel portfolio. For example, each analysis run that Nacpa starts parses the program on its own, while CPAchecker's internal parallel portfolio only parses the program once for all analyses. But we observe that this redundant program parsing has no significant negative effect in SV-COMP.

In contrast, the native compilation of CPAchecker sometimes has strong disadvantages: While it starts significantly faster than traditional CPAchecker, it does not provide a just-in-time compiler. For some tasks, this leads to worse performance than the JVM provides (cf. our experiments with CPA-Daemon [9]). We can see outliers above the diagonal in Fig. 2. For these tasks, Nacpa is significantly slower than CPAchecker. This, together with the different configuration of parallel-portfolio strategies, leads to some worse verification results [3] than CPAchecker: CPAchecker solves 446 tasks in SV-COMP 2025 that Nacpa can not solve. This accounts for 2 % of all tasks that CPAchecker solved correctly.

4 Setup and Configuration

Nacpa 1.0, the version used for SV-COMP 2025, is shipped as a statically linked binary and with a CPAchecker version natively compiled for Ubuntu 24.04 on x86.

Installation. The Nacpa 1.0 distribution is available on Zenodo [10]. The easiest way to install Nacpa is through fm-weck [11]. The following command installs Nacpa into a new directory called `nacpa/`:

```
pipx run fm-weck install -d nacpa/ nacpa:1.0
```

Use. To run Nacpa on program `prog.c` with program property `spec.prp` and data-model `ILP32`, execute from the directory where Nacpa is installed:

```
./bin/nacpa --spec spec.prp --data-model ILP32 prog.c
```

Nacpa requires the program-under-verification to be preprocessed. Nacpa supports the data-models `ILP32` and `LP64` and all SV-COMP properties.

Project Information. Nacpa participates in all categories of SV-COMP. It is maintained by Henrik Wachowitz and Thomas Lemberger at LMU Munich.

Data-Availability Statement. The source code of Nacpa is available at https://gitlab.com/sosy-lab/software/nacpa and the version used in SV-COMP 2025 is archived at Zenodo [10]. Nacpa is licensed under Apache-2.0.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

References

1. 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
2. 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 (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
3. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
4. 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, https://www.sosy-lab.org/research/pub/2018-ISoLA.Strategy_Selection_for_Software_Verification_Based_on_Boolean_Features.pdf
5. 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 (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
6. 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
7. 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
8. 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
9. Beyer, D., Lemberger, T., Wachowitz, H.: CPA-Daemon: Mitigating tool restarts for Java-based verifiers. In: Proc. ATVA. Springer (2024)
10. Lemberger, T., Wachowitz, H.: Nacpa release 1.0. Zenodo (2024). https://doi.org/10.5281/zenodo.14203473
11. Beyer, D., Wachowitz, H.: FM-Weck: Containerized execution of formal-methods tools. In: Proc. FM. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_3

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

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

PROTON 2.1: Synthesizing Ranking Functions via fine-tuned locally hosted LLM (Competition Contribution)

Diganta Mukhopadhyay[1], Ravindra Metta[1,2](✉), Hrishikesh Karmarkar[1], and Kumar Madhukar[3]

[1] TCS Research, Tata Consultancy Services, Pune, India
{diganta.m,r.metta,hrishi.karmarkar}@tcs.com
[2] School of CIT, Technical University of Munich, Munich, Germany
[3] Department of Computer Science, IIT Delhi, New Delhi, India
madhukar@cse.iitd.ac.in

Abstract. PROTON 2.1 presents (1) a new termination checking technique that uses a fine-tuned local LLM to synthesize ranking functions, and (2) support for multiple SAT solvers for non-termination checking.

1 PROTON 2.1: Ranking function synthesis via LLMs

The termination check in PROTON 1.1.2 [17] is based on loop unwinding assertions via CBMC [13]. In SV-COMP 2024 [11], this check failed on 123 benchmarks (e.g. Fig. 1), mainly as a large number of unwindings ($\geq 10^5$) are needed to satisfy the unwinding assertions in these benchmarks. To effectively check termination of such kinds of programs, we synthesize ranking functions by integrating a *small, locally hosted* LLM based on the Llama-3.2-1B-Instruct model (henceforth referred to as Llama-1B) [1] into the latest version of PROTON (2.1). This approach, henceforth referred to as LLM-T-Check, is similar to [16,15]. As out-of-the box Llama-1B failed to synthesize valid ranking functions even for trivial programs, we fine-tuned it to unlock its ability to guess candidate ranking functions (henceforth referred to as CRFs) and designed a check to validate the correctness of these guessed CRFs, as described below.

Generating Fine-Tuning Data: For each program P_i of the 1536 terminating benchmark programs in SV-COMP, and for each loop L_j in P_i, we assembled a prompt π_{ij} containing the source code of P_i and the body of loop L_j following a template similar to the one in [16]. Then, an instance of the Llama-3-70B-Instruct model [3] (henceforth referred to as Llama-70B) hosted on HuggingFace was queried once with each of these π_{ij}. The response r_{ij} was parsed to extract the loop variant v_{ij} and supporting invariant u_{ij} guessed by Llama-70B. Each P_i was then annotated with ACSL [9] specifications corresponding to u_{ij}, v_{ij} and was passed to the *wp* [8] plugin of Frama-C [14]. If Frama-C was able to validate

R. Metta—Jury member

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 242–247, 2025.
https://doi.org/10.1007/978-3-031-90660-2_19

u_{ij}, v_{ij} a data point d_{ij} was added to the dataset D consisting of P_i, L_j, and v_{ij}. This gave us a high-quality fine-tuning dataset D with 321 programs.

Fine-Tuning Llama-1B: The 321 benchmarks in D include only 22 of the 123 benchmarks that are hard for PROTON 1.1.2. To offset this under-representation, while simultaneously ensuring that the model is able to solve programs from outside the class of programs that are hard for PROTON 1.1.2, we separated these 22 programs into a set F. Then, during training, we ensured that the model sees data points from F with an increased probability (0.7) than those in $D \setminus F$ (0.3). Using D and F, Llama-1B was fine-tuned with a loss that when given a prompt with P_i and L_j, rewards responses containing u_{ij}, v_{ij}, and penalizes other responses. The fine-tuning dataset and the scripts are available in [6].

Guessing a CRF: The llama.cpp [2] model inferencing library via the llama-cpp-python [4] bindings provides an API to efficiently run a GGUF model locally, and is optimized to consume minimal CPU and memory resources. We use a custom prompt template based on the ones used by [16] to run our fine tuned LLM via llama.cpp and generate a text response, comprising of the CRF and supporting invariant. We parse this response using a Python script.

Validating the CRF: We then instrument the source program with assertions that characterize the validity of the CRF guessed above as follows. First, we add a new variable `oldVal` initialized to the max value of the type of `oldVal` (at line 2 in Fig. 1) that tracks the value of the ranking function in the previous iteration (line 6). We then add two assertions at the head of the loop (line 4 and 5) to check that the value of the ranking function is less than `oldVal` and remains non-negative. We run randomized tests on this program to quickly check if the CRF can be violated (restricted to max 100 tests in SV-COMP 2025). If this CRF is not violated by the randomized testing, we run CBMC with a small number of unwinds to check for any violations of the CRF missed by randomized testing. If CBMC does not report any violations, we assume that the CRF represents a valid ranking function, and report termination. Thus, this is an unsound validation check, and hence not ideal. However, this strategy has been surprisingly effective in catching invalid ranking functions and has not produced any false positives for the SV-COMP benchmarks.

```
1  int i = 0, k = 0;
2  int oldVal = INT_MAX;
3  while(i < 100000) {
4    assert(100000 - i < oldVal);
5    assert(100000 - i >= 0);
6    oldVal = 100000 - i;
7    int j = nondet();
8    if (!(1 <= j && j < 100000))
9      return 0;
10   i = i + j; k ++; }
```

Fig. 1: count-by-nondet.c

Table 1: Evaluation Results

	Our Evaluation			SV-COMP Run		
Dataset	F	D	T	$F \cap R$	$D \cap R$	$T \cap R$
Successful	12	44	21	6	11	3
Total	22	321	1215	8	24	57

Evaluation: The remaining 1215 terminating benchmarks in SV-COMP outside D were never seen during fine-tuning, and therefore may be safely considered as a test set. Let us call these T. With this training-testing split, we evaluated the fine-tuned LLM, and the results are given in the first three columns of

Table 1. Also, during the SV-COMP competition run, LLM-T-Check was run on 81 terminating benchmarks, since PROTON 1.1.2 returned UNKNOWN or ERROR on these (see Sect. 2). Let R denote the set of these 81 benchmarks. The last three columns of Table 1 shows how many of these programs were in F, D and T, and the performance of the LLM on each of these tests.

Does the Model Generalize: We found that the fine-tuned LLM used in LLM-T-Check is able to guess good CRFs for 21 out of 1215 programs that were not in D. This is supported by the data in Table 1, which shows several instances where the fine-tuned LLM was able to guess potentially correct CRFs for programs outside D, both in our own evaluation and in the SV-COMP runs. For instance, the `aaron2-1.c` benchmark was not in D because Llama-70B guesses an incorrect CRF for this (`tx`), but the fine-tuned LLM was able to guess a correct one (`x-y`). Further, we found that our model was able to make good guesses for CRFs even in modified versions of the SV-COMP benchmarks that were not even present in the test set. Thus, the fine-tuned model displays evidence of generalization.

1.1 Support for multiple SAT solvers in non-termination checking

PROTON 1.1.2 encodes an assertion in each loop to check if a program-state visited during some arbitrary iteration of the loop recurs in a subsequent iteration, using Z3 as the backend solver. In PROTON 2.1, we added support for Glucose [7] and Kissat [12] SAT solvers too. We architected the PROTON 1.1.2 scripts such that adding support for a new SAT or SMT solver just takes a 4-line API to invoke the solver. This enables PROTON 1.1.2 to be invoked with a solver of users' choice, or even with multiple solvers one after another.

2 Software Architecture

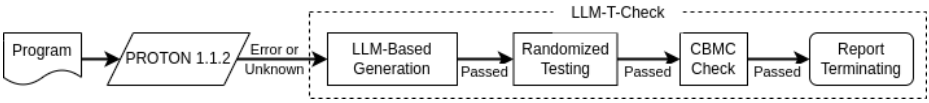


Fig. 2: Tool flow of LLM-T-Check

Figure 2 shows the flow of LLM-T-Check implemented in PROTON 2.1. It starts with calling PROTON 1.1.2 [17]. LLM-T-Check is called only if PROTON 1.1.2 returns unknown or error. LLM-T-Check starts by performing LLM Inference using our fine-tuned LLM via llama.cpp, and obtains a text response, with a limit of 64 new tokens and a context length of 2048 tokens. If the token limit is exceeded by the prompt itself, or if the response does not contain any grammatically valid CRF, we declare the generation as failed and return unknown. If the generation succeeds, we first perform randomized testing to test the validity of the CRF. If the tests fail, we again return unknown, else we move on to running CBMC (version 5.95.0) with small number of unwinds (pre-configured to max 3 unwinds for SV-COMP 2025) to further validate the CRF. If CBMC does not

invalidate the CRF, we report the program as terminating. The ranking function generation uses llama.cpp via the llama-cpp-python bindings. Other components of PROTON 2.1 are the same as PROTON 1.1.2; we refer the interested readers to [17] for further details.

3 Strengths and Weaknesses

In SV-COMP 2025, LLM-T-Check ran on 206 benchmarks that PROTON 1.1.2 could not solve. Of these, 23 are recursive, which our instrumenter does not support. Of the remaining 183, 102 were non-terminating and 81 were terminating.
Strengths: For the non-terminating benchmarks, incorrect ranking functions guessed by the LLM were rejected by the validation phase. Thus, even though our current validation check is unsound in theory, it is unlikely to produce false positives in practice. From the 81 terminating benchmarks, many require upwards of 100000 loop-unwinds for termination, and hence the termination check of PROTON 1.1.2 did not scale on these. LLM-T-Check is able to solve 14 of these 81. It is interesting to note that benchmark `array-examples/standard_sentinel-2` is solved only by LLM-T-Check with $(10^5 - \texttt{i})$ as the ranking function, while none of the other tools in SV-COMP 2025 are able to solve this. This is a promising first step in integrating locally-hosted LLMs, which in addition to their privacy and cost benefits, allows us to build offline modules that can be bundled into software verification tools.
Weaknesses: LLM-T-Check was unable to find ranking functions for 67 of the 81 programs. Of these, 45 are complex programs with multiple loops from categories such as `uthash` and `seq-mthreaded`. These programs exceed the token limit for the online hosted teacher Llama-70B, and hence will require more complex solutions to be able to generate CRFs. For the remaining 22, the guessed CRFs were invalidated during the checking phase.
Future work: Currently we are limited by the token limit of the online LLM when generating D. To overcome this, we are planning to use a fine-tuning technique that is independent of online hosted models, such as using feedback from solvers, circumventing the token limit.

4 Tool Configuration, Setup, and Contributors

PROTON 2.1 comes with an MIT license. Instructions to install and run, including a sample run command, are in the `README.txt` in PROTON 2.1's Zenodo Archive [5]. PROTON 2.1 opted to participate only in the Termination category in SV-COMP 2025 [10].

Software Project and Contributors: PROTON 2.1 is developed and maintained by the authors at TCS Research, IIT Delhi and IIT Bombay. We thank everyone who has contributed to the development of PROTON 2.1, Clang and LLVM Infrastructure, CBMC, Glucose Syrup, Kissat, Z3, HuggingFace Transformers, PyTorch, Meta-Llama-3-70B-Instruct, Llama-3.2-1B-Instruct, llama.cpp, and llama-cpp-python.

5 Data Availability Statement

PROTON 2.1 is publicly available at [5]. For reproducibility of the fine-tuning process, training scripts and datasets are publicly available at [6]. For any queries, please contact us.

References

1. Llama-3.2-1B-Instruct. https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct, accessed: 2024-11-07
2. llama.cpp. https://github.com/ggerganov/llama.cpp, accessed: 2024-11-07
3. Meta-Llama-3-70B-Instruct. https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct, accessed: 2024-08-14
4. Python Bindings for llama.cpp. https://github.com/abetlen/llama-cpp-python, accessed: 2024-11-07
5. PROTON SV-COMP 2025 Competition Contribution (Nov 2024). https://doi.org/10.5281/zenodo.14209458
6. Training Scripts and Data set of LLM-PROTON (Feb 2025). https://doi.org/10.5281/zenodo.14854742
7. 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
8. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual, http://frama-c.com/download/frama-c-wp-manual.pdf
9. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/download/acsl.pdf
10. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS (2025)
11. Beyer, D.: State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In: TACAS. pp. 299–329 (2024). https://doi.org/10.1007/978-3-031-57256-2_15
12. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In: Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. pp. 51–53 (2020)
13. C Bounded Model Checker. https://github.com/diffblue/cbmc
14. Correnson, L., Cuoq, P., Kirchner, F., Maroneze, A., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual, http://frama-c.com/download/frama-c-user-manual.pdf
15. Kamath, A., Mohammed, N., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S.K., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Leveraging LLMs for Program Verification. In: FMCAD. pp. 107–118 (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_16
16. Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S.K., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Finding Inductive Loop Invariants using Large Language Models. CoRR **abs/2311.07948** (2023). https://doi.org/10.48550/ARXIV.2311.07948
17. Metta, R., Karmarkar, H., Madhukar, K., Venkatesh, R., Chakraborty, S.: PROTON: PRObes for Termination Or Not (Competition Contribution). In: TACAS. pp. 393–398 (2024). https://doi.org/10.1007/978-3-031-57256-2_27

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

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


RacerF: Data Race Detection with Frama-C

(Competition Contribution)

Tomáš Dacík[1](✉) and Tomáš Vojnar[1,2]

[1] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
idacik@fit.vut.cz

[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract. RacerF is a static analyser for detection of data races in multithreaded C programs implemented as a plugin of the Frama-C platform. The approach behind RacerF is mostly heuristic and relies on analysis of the sequential behaviour of particular threads whose results are generalised using a combination of under- and over-approximating techniques to allow analysis of the multithreading behaviour. In particular, in SV-COMP'25, RacerF relies on the Frama-C's abstract interpreter EVA to perform the analysis of the sequential behaviour. Although RacerF does not provide any formal guarantees, it ranked second in the *NoDataRace-Main* sub-category, providing the largest number of correct results (when excluding metaverifiers) and just 4 false positives.

1 Verification Approach

RacerF is a static analyser whose primary goal is to provide fast and scalable data race detection, without sacrificing too much precision. It relies on using a backend static analyser that summarises (in a way specific for the chosen analyser) the sequential behaviour of particular threads – or, more precisely, classes of threads since RacerF distinguishes threads according to their entry point function only. The existing threads are discovered incrementally since a newly discovered thread may create further threads. The results of the analysis of the sequential behaviour are generalised for the multi-threaded setting using a combination of an under-approximating and an over-approximating strategy.

RacerF comes with several analysis backends differing in their scalability and precision. In SV-COMP'25, we use the most precise of them – Frama-C's value analysis plugin EVA based on abstract interpretation combining several abstract domains to characterize sets of integers and addresses that may get assigned to the program variables [3]. Over the backend's results, RacerF runs several analyses to perform data race detection as a final step. We briefly describe these analyses in the rest of this section. More details can be found in [7].

1.1 Analysing Multithreaded Programs Using Sequential Behaviour

The main idea behind RacerF is to discover all program threads, distinguished statically according to their entry functions, and analyse them as sequential programs, starting with initial states that are computed using a set of equations over

T. Dacík—Jury member.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 248–253, 2025.
https://doi.org/10.1007/978-3-031-90660-2_20

currently discovered threads. We have two strategies to construct those equations, based on under-approximating or over-approximating thread behaviours. The approximations computed by the different analysis backends are then available to subsequent analysis phases performed by RacerF, which may but need not respect the guaranteed under-/over-approximation.

Under-approximating strategy. The idea behind this strategy is to ignore all interleavings. Each thread is analysed with the initial state corresponding to the join of states discovered as reachable at its create statements, and its writes to global variables are not propagated to other threads. When there is no cyclic dependency in thread creation, it is enough to analyse each thread just once.

Over-approximating strategy. This strategy over-approximates considered interleavings (including also those ruled out by some synchronization method). In this setting, each thread is analysed with the initial state given as the join of all states encountered by analyses of all discovered threads. This strategy may require several re-analyses of individual threads, but we aggressively use widening to reduce this number. This strategy needs to be accompanied by a program transformation that ensures that the computed over-approximation of the initial state is not lost during the sequential analysis of threads. Intuitively, the exchange of information between thread analyses happens at the initial states only, and so we need to preserve the information obtained from analyses of other threads throughout the re-analysis of a given thread and not destroy it by some local assignments before it gets propagated to a code location where it actually matters (see [7] for more details). This transformation was implemented after our submission to SV-COMP'25, but its absence does not manifest in the results (it would manifest only on programs that our memory access analysis cannot handle precisely at that time.)

Combined strategy. For SV-COMP'25, we provide a wrapper that combines both strategies in a natural way. The under-approximation is only allowed to report races, and the over-approximation is only allowed to claim a program race-free. We start with the over-approximation, and if it is inconclusive, we run the under-approximation. If both strategies are inconclusive, we report `unknown`. The situation in which running the combination is beneficial within SV-COMP is, however, quite rare as the result of using purely under- or over-approximation would lead to results different from using their combination in 11 out of 1029 data race benchmarks only.

1.2 Analysis Pipeline

We call an *analysis context* a triple consisting of a thread, a sequence of a fixed number of the latest calls (each of them formed by a call site and the function called, with the number being a parameter of the analysis – for SV-COMP'25, set to two), and a program statement. We run the following pipeline of analyses to detect data races.

Lockset analysis. This analysis computes the set of sets of locks held for each program context (using an algorithm inspired by [8]) and provides must- and may-queries to check whether two contexts are guarded by a common lock. Trylocks and read-write locks are supported.

Table 1: Results of RacerF in *NoDataRace-Main* sub-category.

	Correct		Wrong		Unconfirmed			
Analyser	True	False	True	False	True	False	Score	Time [s]
RacerF	674	68	0	4	0	30	1352	1300

Active-threads analysis. The goal of this analysis is to provide information which threads may- and must-run in parallel in particular contexts. This information is then used to not report data races on memory accesses where one happens before the thread of the second is created, or after it is surely joined.

Memory access analysis. EVA represents memory addresses as pairs consisting of a *base* and an *offset*. Each variable (global, local, or formal) has its associated base. Dynamic allocations have so-called *dynamic bases*, either strong or weak (representing multiple allocations). The memory access analysis tracks accesses to memory bases and distinguishes their states in a way inspired by [11]. Those states are used to select bases for which data race detection should be performed.

Data race detection. Finally, over the discovered memory accesses, a may and a must data race detection is performed. Intuitively, the may date race detection checks whether at least one of the accesses is a write access, the accesses may come from distinct threads and may happen in parallel, the concerned locksets may have empty intersection, and the concerned memory blocks may overlap (with the must data race detection modified accordingly and accompanied by several more checks to eliminate common sources of false positives).

Reducing Incorrect Verdicts. As already mentioned, to reduce the number of incorrect verdicts, we can leverage the distinction of the detected races to may and must races, essentially reporting a data race only when a must data race is detected, and reporting no data race only if no may data race is found. Otherwise, an unknown result is produced.

Further, since the SV-COMP'25 benchmark also contains programs implementing lock-free algorithms, which we cannot analyse, we have implemented a simple detection of the code pattern of *active waiting* (essentially detecting empty control loops), which is the common denominator of such programs, and report unknown if we encounter such active waiting.

2 Strengths and Weaknesses

Despite its heuristic nature, RacerF can provide very precise results as Table 1 shows [2]. RacerF does not provide a true/false verdict for 253 programs, of which 17 cannot be parsed by Frama-C, 60 contain unsupported features (3 cases of semaphores, 57 cases of custom atomic functions), and in 176 cases, our analyser returns unknown (in 20 cases, active waiting is detected; and in 156 cases, our may/must data race classification is inconclusive).

Strengths. RacerF is quite fast. In just 8 cases, it needs more than 5 seconds to compute a result, and it also never runs out of time or memory. This is in stark contrast with model checkers. The only competitive SV-COMP'25 participant from this point of view is Goblint [10, 12], which, however, also

timed out in a few cases. What we are especially proud of is that RacerF is the only SV-COMP'25 participant (excluding tools with a negative score) which provides correct results for all 5 programs derived from the Linux kernel in the `ldv-linux-3.14-races` benchmark. Besides it, just Goblint is able to provide the correct result for one of those tasks. All the other tools either run out of resources, fail, or return an `unknown` verdict. RacerF needs just 2 minutes for one of them and about 30 seconds for the others.

Weaknesses. RacerF does not provide any formal guarantees for its results. There are 4 cases in which our heuristic approach reports a false alarm. All of them rule out a data race by some intricate condition that is out of the scope of our analyses. Two of them (`pthread-ext/09_fmaxsym-zero.c` and `pthread-ext/11_fmaxsymopt-zero.c`) are variants of the same program that relies on the fact that an array is zero-initialized and a certain condition cannot hold more than once throughout the run of the program. The other two programs `goblint-regression/13-privatized_40-traces-ex-6_true.c` and `pthread-atomic/time_var_mutex.c` use synchronisation patterns that are beyond the scope of our lockset analysis.

Another weakness of our analyser is a lack of path-sensitivity. In some cases, especially on the `ldv-races` benchmark, RacerF returns `unknown` because it considers some trivially infeasible path that prevents it from classifying the encountered issue as a must data race. In future versions, this can be improved by adding preprocessing to eliminate trivial branching conditions and dead code.

Finally, RacerF can only generate trivial witnesses and thus almost one third of its `false` answers is unconfirmed. In fact, RacerF provides traces for detected races, but we have not yet implemented their conversion to witnesses.

3 Software Project and Tool Setup

RacerF is implemented in OCaml as a plugin of the Frama-C platform [1,5]. It relies on the Frama-C's fork of CIL [9] as its frontend and uses the abstract interpretation plugin EVA [3] to perform sequential analyses of individual threads. The library OCamlgraph [4] is used to represent several graph structures and solve systems of equations created during thread analysis. In SV-COMP'25, RacerF is run via a Python script `racerf-sv.py` which performs the following:

Preprocessing. Since Frama-C discards the `_Atomic` attributes during parsing, we encode them as type aliases, e.g., the declaration `_Atomic int x` is transformed to `atomic_int x`. We also further preprocess several aspects of input that are problematic for Frama-C (e.g. we replace variable-length arrays in non-final fields of structures with arrays of a constant size). The preprocessing is implemented in the script `preprocess.py`.

Combination of under- and over-approximation. The script runs two configurations of RacerF and reports the verdict as discussed in Section 1.

Witness postprocessing. During the competition, it turned out that RacerF produces syntactically incorrect witnesses. The script `fix_witness.py` fixes this by filling in the correct metadata. The witness graph itself is not modified.

Usage. A binary of RacerF is available at Zenodo [6] (the only runtime dependencies are GCC and Python version 3.10 or newer). The wrapper script can be run as (see the attached file `README.md` for more information):

```
./racerf-sv.py -machdep=[gcc_x86_32|gcc_x86_64] input.c
```

Software project. RacerF is available under the MIT license as a part of the authors' project of concurrency analysers for C, maintained by Tomáš Dacík.

Participation. RacerF participated in the *NoDataRace-Main* sub-category only.

Data-Availability Statement. RacerF is available under the MIT license at https://github.com/TDacik/Deadlock_Racer. The version participating in SV-COMP'25 is available under the tag svcomp25 and archived at Zenodo [6].

Acknowledgements. We appreciate discussions with Julien Signoles from the Frama-C team at CEA, France and with Adam Rogalewicz from FIT BUT, Czechia. The research was supported by the Czech Science Foundation project 23-06506S and the FIT BUT internal project FIT-S-23-8151. Tomáš Dacík was supported by the Brno PhD Talent Scholarship of the Brno City Municipality. The collaboration with Julien Signoles was supported by the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

References

1. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. Commun. ACM **64**(8), 56–68 (2021), https://doi.org/10.1145/3470569
2. Beyer, D., Strejček, J.: Improvements in Software Verification and Witness Validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
3. Blazy, S., Bühler, D., Yakobowski, B.: Structuring Abstract Interpreters Through State and Value Abstractions. In: Proc. VMCAI. pp. 112–130 (2017). https://doi.org/10.1007/978-3-319-52234-0_7
4. Conchon, S., Filliâtre, J.C., Signoles, J.: Designing a Generic Graph Library using ML Functors. In: Symposium on Trends in Functional Programming (2007)
5. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Software Analysis Perspective. In: Proc. SEFM. p. 233–247. Springer-Verlag (2012). https://doi.org/10.1007/978-3-642-33826-7_16
6. Dacík, T., Vojnar, T.: RacerF (SV-COMP 25). Zenodo (2024), https://doi.org/10.5281/zenodo.14507645
7. Dacík, T., Vojnar, T.: RacerF: Lightweight Static Data Race Detection for C Code (2025), https://arxiv.org/abs/2502.04905
8. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. SIGOPS Oper. Syst. Rev. **37**(5), 237–252 (2003). https://doi.org/10.1145/1165389.945468

9. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: Compiler Construction. pp. 213–228 (2002). https://doi.org/10.1007/3-540-45937-5_16
10. 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 (3). pp. 381–386. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_25
11. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems (TOCS) **15**(4), 391–411 (1997). https://doi.org/10.1145/265924.265927
12. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static Race Detection for Device Drivers: The Goblint Approach. In: Proc. ASE. p. 391–402. Association for Computing Machinery (2016), https://doi.org/10.1145/2970276.2970337

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

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

SVF-SVC: Software Verification Using SVF (Competition Contribution)

Cameron McGowan(✉), Matthew Richards, and Yulei Sui

University of New South Wales, Sydney, Australia
{cameron.mcgowan,matthew.richards1,y.sui}@unsw.edu.au

Abstract. The Static Value-Flow Analysis Framework (SVF) is a tool that enables interprocedural static value-flow analysis for LLVM-based languages by leveraging sparse and on-demand analysis. This work, SVF-SVC, presents an adaptation of SVF for its debut in SV-COMP 2025. We detail our development which uses SVF as a library to correctly parse SV-COMP program specifications and produce witnesses statements for C programs in the ReachSafety, MemSafety and SoftwareSystems categories. We evaluate SVF-SVC's performance in SV-COMP 2025 and pave the way for its participation in future editions.

Keywords: Static Analysis · SVF · Abstract Execution · Competition on Software Verification · SV-COMP.

1 Verification Approach

The Static Value-Flow Analysis Framework (SVF) [10] is a tool that enables scalable and precise interprocedural static analysis for LLVM-based languages [6] by leveraging sparse [13] and on-demand analysis. SVF is capable of performing pointer alias analysis [4], memory SSA form construction [11], value-flow tracking [9] and abstract execution for program understanding and memory error checking. SVF performs its analysis by reasoning about properties on a set of graph representations of code. In this value-flow analysis, SVF uses the points-to information generated by Andersen's pointer analysis [1] to construct an interprocedural memory SSA form, in which the defuse chains of both top-level and address-taken variables are captured. Sparse static analysis on this offers a more scalable solution compared to its non-sparse counterpart. SVF achieves this by first conducting fast pointer analysis that over-approximates value-flows and propagates data-flow facts sparsely along only the pre-computed value-flows instead of all control flow points. SVF also incorporates cross-domain sparse abstract execution [3] that interweaves correlations between values across multiple abstract domains (e.g., memory address and interval domains) by establishing implications of values from one domain to another. This allows SVF to refine spurious alias relations using interval domain information to enhance the precision of interval analysis with refined alias results.

These value-flow results can then be exploited to evaluate the reachability of code sections in the ReachSafety category. Furthermore, memory leak detection

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 254–259, 2025.
https://doi.org/10.1007/978-3-031-90660-2_21

can be formulated as a source-sink problem of whether memory allocation will reach a free site across every execution path. By identifying memory allocations as sources and frees as sinks SVF can detect memory leaks in the MemSafety category using its reachability capabilities. Similarly, buffer-overflow errors can be detected by combining reasoning about the reachability of untrusted user input (the source) to array accesses (the sink) and evaluation of the array accesses at this point via abstract execution. This work represents the first attempt to incorporate SVF into the verification competition, despite numerous challenges related to formatting and input/output that needed to be addressed. Fig. 1 contains a high level overview of how SVF operates.

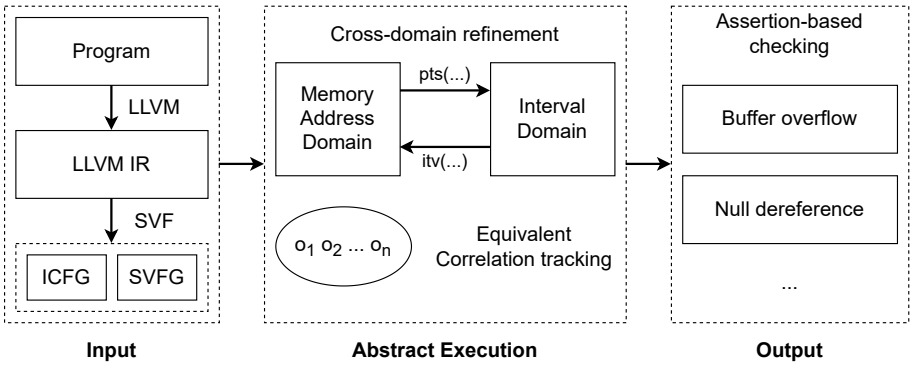


Fig. 1. High level overview of SVF [3].

2 SVF-SVC Architecture and SV-COMP Adapter

Before our work, the program input formats accepted by SVF were incompatible with the specifications defined in SV-COMP. SVF also did not provide any witness formats which could be validated by an approved validator in SV-COMP. SVF-SVC performs the role of an adapter which correctly translates SV-COMP programs and category properties into an SVF compatible program with appropriate arguments. It then takes the output from SVF and generates format version 1.0 witness statements for SV-COMP. The whole process is shown in Fig. 2.

SVF-SVC performs the adaptation by inspecting input files for non-determinate function declarations and inserting appropriate translations for each function at the start of the C file. After compilation to LLVM-IR using Clang [5], SVF then analyses the LLVM-IR file and produces an SVF verdict to be interpreted by SVF-SVC to generate a valid witness.

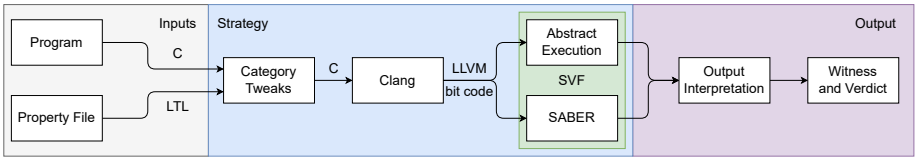


Fig. 2. Program flow of SVF-SVC.

2.1 Reach Safety

Reach safety analysis is performed by wrapping the main function with an assert ensuring a global variable is in its initial state. The reach_error function is overwritten to put the global variable into the error state when reached so that the SVF assert will correctly capture the error. The wrapping technique is summarised in Fig. 3.

```
int svf_svc_reach_test = 0;
void reach_error() {
  svf_svc_reach_test = 1; }

Original C File

void svf_main() {
  main();
  svf_assert(svf_svc_reach_test == 0); }
```

Fig. 3. Wrapper used for reach safety.

2.2 Memory Checks

Memory leaks and double-free checking are done using the SABER [12] component of SVF and by passing the "-leak" and "-dfree" options respectively. SVF-SVC can also test for buffer overflows with abstract execution by passing the "-overflow" flag into SVF. SVF-SVC does not currently support other memory checks in this competition submission.

2.3 Software Systems

This category is simply real-world applications of the other categories. As such, there are no special handlers needed for this category beyond those previously mentioned besides ensuring that SVF-SVC ignores unsupported properties. The aforementioned formatting challenges prevented SVF-SVC from scoring any points in this category.

3 Evaluation

Table 1. SVF-SVC results in SV-COMP 25. [2]

Result	Count
Confirmed Correct True	6315
Unconfirmed Correct True	725
Confirmed Correct False	72
Unconfirmed Correct False	1827
Incorrect True	1416
Incorrect False	1503
Unknown	15124
Error	2684

Whilst SVF-SVC was not the strongest competitor in its debut, a reasonable initial performance was achieved in the verification of true properties. In contrast, SVF-SVC performed weakly in asserting falsities with less than 60% of its claims being correct. The reason behind this is likely due to the conservative over-approximations performed by SVF under the hood. The over-estimation of values produced by abstract execution in theory ensures that if a property is still not violated by the conservative upperbound provided, then it will not be violated by real-world conditions either. However, there could be cases where the over-estimation violates the required properties even though the actual bounds are smaller and non-violating.

Another issue faced was that majority of our correct false claims were unconfirmed due to SVF-SVC not specifying exactly which property was violated, leading to errors with the validators. In a significant portion of test cases SVF-SVC reported UNKNOWN, to some extent due to limitations with the current functionality of SVF but largely due to compatibility issues with SVF-SVC's wrapper for certain test cases in SV-COMP. In theory SVF-SVC's approach should be accurate when reporting false statements, however SVF does not currently have the capability to correctly reason about loops with complex conditions as the widening and narrowing rules used to effectively analyse them only work on monotonic conditions. This leads to many incorrect results in the reachability and memory safety categories. Beyond implementing strategies in SVF to handle such cases, accurately identifying these cases for separate handling would reduce the number of erroneous claims made by SVF-SVC (Table 1).

The largest areas for improvement in future editions would be furthering SVF-SVC's compatibility in existing categories to support the full functionality of SVF which should reduce the number of UNKNOWN responses of SVF-SVC. Utilising internal representations in SVF to output more extensive witnesses would assist in ensuring the confirmation of correct outputs by validators. A tuning of the efficiency precision tradeoff in SVF for greater precision at the cost of speed would improve results by reducing the numerous incorrect false

claims SVF-SVC makes and minimising the high penalties associated with them. Improvements to SVF's internal implementation to more accurately detect cases where a precise answer is UNKNOWN rather than incorrectly claiming that properties are true or false would also significantly improve SVF-SVC's score in future editions of SV-COMP.

In the process, it would be hoped that options for greater precision in SVF would allow for stronger real-world performance in cases where users want a low number of false positives in vulnerability detection at the cost of reduced speed and efficiency. In our initial attempt at this year's SV-COMP, we did not modify the internal implementation of SVF for precision tuning (due to time limitations); however, we plan to enhance SVF for future competitions.

4 Software Project and Contributors

SVF-SVC is available on GitHub [7], and is released under the GNU Affero General Public License version 3. The exact version (1.4) used in the competition is available as a Zenodo archive [8] and can run on Ubuntu 24.04 LTS. The execution of SVF-SVC is as follows:

```
./svf_svc.py --bits {32,64} --prop FILE --witness FILE c_file
```

Where `c_file` is the program to be tested in the format applicable to SV-COMP. The contributors who adapted SVF for SV-COMP 2025 are the authors of this paper.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Andersen, L.O., Lee, P.: Program analysis and specialization for the c programming language (2005), https://api.semanticscholar.org/CorpusID:20876553
2. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
3. Cheng, X., Wang, J., Sui, Y.: Precise sparse abstract execution via cross-domain interaction. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639220
4. Chow, F., Chan, S., Liu, S.M., Lo, R., Streich, M.: Effective representation of aliases and indirect memory operations in ssa form (01 2000). https://doi.org/10.1007/3-540-61053-7_66
5. Lattner, C.: Llvm and clang: Next generation compiler technology. In: The BSD conference. vol. 5, pp. 1–20 (2008)
6. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. p. 75. CGO '04, IEEE Computer Society, USA (2004)

7. Richards, M., McGowan, C.: Lasagnenator/svf-svc-comp (2024), https://github.com/Lasagnenator/svf-svc-comp
8. Richards, M., McGowan, C.: svf-svc-comp: Release v1.4 (binaries) (Nov 2024). https://doi.org/10.5281/zenodo.14208597
9. Steffen, B., Knoop, J., Rüthing, O.: The value flow graph: A program representation for optimal program transformations. vol. 432, pp. 389–405 (01 2006). https://doi.org/10.1007/3-540-52592-0_76
10. Sui, Y., Xue, J.: Svf: interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th International Conference on Compiler Construction. p. 265–266. CC '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2892208.2892235
11. Sui, Y., Yan, H., Zheng, Z., Zhang, Y., Xue, J.: Parallel construction of interprocedural memory ssa form. Journal of Systems and Software **146**, 186–195 (2018). https://doi.org/10.1016/j.jss.2018.09.038
12. Sui, Y., Ye, D., Xue, J.: Detecting memory leaks statically with full-sparse value-flow analysis. IEEE Transactions on Software Engineering **40**(2), 107–122 (2014). https://doi.org/10.1109/TSE.2014.2302311
13. Sui, Y., Ye, S., Xue, J., Yew, P.C.: Spas: scalable path-sensitive pointer analysis on full-sparse ssa. In: Proceedings of the 9th Asian Conference on Programming Languages and Systems. p. 155–171. APLAS'11, Springer-Verlag, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25318-8_14

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

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

Theta: Various Approaches for Concurrent Program Verification (Competition Contribution)

Csanád Telbisz, Levente Bajczi(✉), Dániel Szekeres, and András Vörös

Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics, Budapest, Hungary
csanadtelbisz@edu.bme.hu, {bajczi,szekeres,vori}@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 2025, we complement our existing approach (abstraction-aware partial order reduction) for multi-threaded programs with a happens before propagator-based BMC check, expecting a significant increase in performance. We again utilize our portfolio with dynamic algorithm selection from last year, with improvements regarding solver choice and configuration ordering. In this paper, we detail our algorithmic improvements in Theta regarding the verification of concurrent software.

1 Verification Approach

Theta [17,11] has been a participant in SV-COMP as a standalone tool since 2022. Earlier versions of Theta exclusively applied abstraction-refinement-based model checking algorithms [4,1] mainly focusing on multi-threaded tasks. The focus remained on concurrency; however, different verification approaches have also been implemented in Theta.

The main contribution for this year's SV-COMP version of Theta is a symbolic *bounded model checking* algorithm for the verification of concurrent programs. Compared to abstraction-based analyses, this BMC approach has the advantage of being faster, while it has the disadvantage of providing a bounded correctness proof in many cases. Since Theta is not branded as a bounded model checker, only complete safe results are accepted. If the BMC algorithm produces an incomplete safe result, the tool falls back on an abstraction-based analysis (see more details on our algorithm selection portfolio in Section 2).

The applied BMC algorithm is based on reasoning about the happens-before relation of concurrent program instructions. Our algorithm builds on the concepts of several partial order-based verification algorithms [2,15,18]. The program and the property are symbolically encoded into a Satisfiability Modulo Theories (SMT) formula along with some scheduling constraints based on possible

L. Bajczi—Jury member representing Theta at SV-COMP 2025.

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 260–265, 2025.
https://doi.org/10.1007/978-3-031-90660-2_22

happens-before relations. The (partial) models provided by the SMT solver are used to analyze possible partial orders of concurrent instructions and to prevent scheduling inconsistencies that may arise. An inconsistency corresponds to a cycle in the happens-before relation, since a valid execution of concurrent threads must have a linearization of instructions. A conflict clause is generated and given to the SMT solver when such a happens-before cycle is found to exclude the invalid program execution. The cycle detection and conflict generation algorithm is integrated into Z3 as a custom user propagator [9] via JavaSMT [3].

The first stage of the algorithm is based on the happens-before propagator algorithm of *Sun et al.* [15]. However, their algorithm is insufficient for verification under the sequential consistency memory model [6]: the axioms of [15] can be directly mapped into the rules that define the *weak sequential consistency* memory model [18] that are shown to be insufficient for sequential consistency. Therefore, we extend our algorithm with a second stage when the first stage finds a candidate program execution (i.e., a valid program execution under weak sequential consistency) that violates the safety property. To check whether the happens-before relation representing the program execution is also valid under sequential consistency as well, we explicitly encode the total store order (order of *write* instructions) to guarantee that every write instruction is ordered (a requirement of sequential consistency [18]).

We also implemented a novel optimization for the happens-before propagator [16]. The search space of the SMT solver is reduced by extending the encoding formula. We achieve this by analyzing the program structure and possible partial orders of program instructions, and collect possible scheduling inconsistencies (cycles that may arise in the happens-before relation during verification) before starting the verification decision procedure. Our algorithm searches for potential happens-before cycles of bounded size. Conflict clauses formulated from these potential cycles are appended to the encoding formula. This enhancement greatly reduces the solver search space and improves the verification performance.

The mainline verification method of Theta for proving (unbounded) safety is still a configurable *Counterexample-Guided Abstraction Refinement* (CEGAR) algorithm. For the verification of multi-threaded programs, our approach uses an abstraction-based partial order reduction algorithm and a cone-of-influence reduction technique tailored specifically to concurrent programs. For recursive tasks, an interprocedural analysis is applied with call stack abstraction that can handle infinitely recursive programs. However, this year, only minor improvements have been implemented for our abstraction-based analyses. Detailed descriptions of these analyses in Theta have been presented previously in [4,11,17].

2 Software Architecture

We use portfolio-based algorithm selection as in previous years of SV-COMP [4]. Each configuration is executed in a separate process. A generic interface allows the easy development of complex portfolios defined by finite-state machines. Dynamic algorithm selection is used to select a performant configuration for each

input task, with several ways of recovering when a selected algorithm takes too long or encounters an exception. For reachability properties in concurrent tasks, the introduced BMC algorithm is applied first to discover unsafe tasks and prove safety for loop-free programs (or programs where complete loop unrolling can be performed). For programs where complete loop unrolling is not possible (because the number of loop iterations cannot be determined), each loop is unrolled twice for the BMC algorithm. However, in this case, only unsafe results are accepted: we switch to an abstraction-based analysis otherwise. Therefore, Theta always provides complete proofs of safety. The architecture of Theta has not changed considerably since last year's SV-COMP: Theta parses and transforms the input program into a CFA (supporting multi-threading), then, based on the configuration in the portfolio, spawns one or more worker Theta processes that perform the verification. We refer the interested reader to our previous tool paper for a more detailed description of the architecture of Theta [4].

Theta is implemented in Java and Kotlin, and uses Z3 [13] versions 4.12.2, 4.13.0 and 4.5.0 (the latter two versions are integrated natively via the Java API, while the former one is used via SMT-LIB), MathSAT5 [10] version 5.6.10, CVC5 [7] version 1.0.8 and Princess [14] version 2023-06-19 as SMT solvers under the hood. There were major C-frontend updates in Theta, this year, mainly concerning memory handling language elements, and new C language features introduced in benchmarks since last year's SV-COMP.

3 Strengths and Weaknesses of the Approach

The main scope of development for Theta has been reachability properties and data-race freedom. In these categories, Theta only gives 2 wrong verdicts, which is a notable precision among other verifiers at SV-COMP (actually, the wrong verdicts are the results of the hurried development to adapt to last-minute changes in the language features used in the SV-COMP benchmarks, e.g., the use of atomic types). Theta produces several wrong verdicts for other properties (such as memory safety or termination), however, these properties are only experimental in Theta. We plan to properly support these verification properties in future versions.

In Figure 1, we include a comparison of the results with the performance of Theta in SV-COMP, last year. The figure includes both confirmed and unconfirmed correct results (since result validation and the possibility of an unconfirmed verdict status have only been introduced this year for data race tasks). The figure shows the categories affected by recent development: reachability tasks with memory operations or multi-threading, and data-race detection tasks. The figure highlights the performance increase achieved by the new version of Theta.

We also performed an internal evaluation of our algorithms separately (not in a portfolio). The BMC method is able to provide a (bounded) verdict for 520 tasks for the reachability property in the concurrency category out of the 544 programs whose language features are supported by this analysis. Without the optimization described in Section 1, Theta could solve only 514 tasks. The

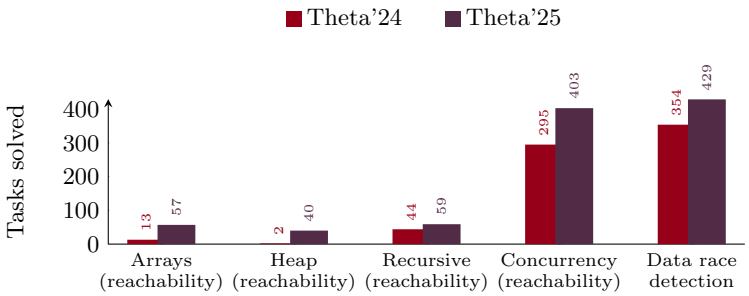


Fig. 1: Comparison of successful tasks for Theta on common tasks

optimization achieves a more significant improvement in CPU time: the verification time is reduced by more than 30% on average. While many of these results are only obtained by applying a bounded loop unrolling (and thus these verdicts are not reported by Theta as final results), other BMC tools among SV-COMP competitors apply a similar strategy. Therefore, we executed `Deagle` (a BMC tool with a bounded loop unrolling strategy [12], winner of SV-COMP concurrency category [8]) on our infrastructure for a fair comparison. `Deagle` achieved the same result: 520 solved tasks on the same set of programs. While Theta is disadvantaged on the full benchmark set due to its frontend limitations, this comparison on a major portion of tasks clearly shows the potential of our algorithm. It also underlines the need for frontend improvements in Theta to support even more language features.

Our internal experiments also reveal that our CEGAR analysis is capable of verifying 365 programs for reachability in the concurrency category. Unfortunately, we slightly misconfigured the algorithm-selection portfolio for concurrent tasks for the competition. Therefore, the best-performing CEGAR configuration using predicate abstraction was not selected for concurrent programs. We calculated that around 2% more tasks could be solved by a proper configuration.

4 Tool Setup and Configuration

Theta is highly configurable [11], and choosing a suitable configuration for a verification task can be challenging. For software verification, we recommend using our complex portfolio in the competition archive [5]: `./theta-start.sh <input> --svcomp --portfolio STABLE`. To minimize the output verbosity, `--loglevel RESULT` can be added. We used these options at SV-COMP 2025.

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 (`6.8.6`) used in the competition is available at [5]. Theta participated in the reachability, memory-safety, concurrency, overflow detection, and termination categories of SV-COMP 2025.

Funding. This research was partially funded by the EKÖP-24-{2,3} New National Excellence Program under project numbers EKÖP-24-2-BME-118, EKÖP-24-3-BME-213 and EKÖP-24-3-BME-159, and the Doctoral Excellence Fellowship Programme under project numbers 400434/2023 and 400443/2023; funded by the NRDI Fund of Hungary.

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 2022. Lecture Notes in Computer Science, vol. 13244, pp. 474–478. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_34
2. Alglave, J., Kroening, D., Tautschnig, M.: Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. Lecture Notes in Computer Science, vol. 8044, pp. 141–157. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_9
3. Baier, D., Beyer, D., Friedberger, K.: Javasmt 3: Interacting with SMT solvers in java. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. Lecture Notes in Computer Science, vol. 12760, pp. 195–208. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_9
4. 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: TACAS 2024. Lecture Notes in Computer Science, vol. 14572, pp. 412–417. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_30, https://doi.org/10.1007/978-3-031-57256-2_30
5. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Molnár, V.: Theta - SV-COMP'25 Verifier Archive (Nov 2024). https://doi.org/10.5281/zenodo.14194483
6. Bajczi, L., Telbisz, C., Szekeres, D., Vörös, A.: On Stability in a Happens-Before Propagator for Concurrent Programs (Reproducibility Study). In: TACAS 2025. LNCS , Springer (2025), https://ftsrg.mit.bme.hu/paper-tacas25-ocfix/paper.pdf
7. 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
8. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. LNCS, Springer (2025)
9. Bjørner, N.S., Eisenhofer, C., Kovács, L.: Satisfiability Modulo Custom Theories in Z3. In: Dragoi, C., Emmi, M., Wang, J. (eds.) VMCAI 2023. Lecture Notes in Computer Science, vol. 13881, pp. 91–105. Springer (2023). https://doi.org/10.1007/978-3-031-24950-1_5

[1] https://github.com/ftsrg/theta/releases/tag/svcomp25

10. 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
11. 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
12. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) TACAS 2022. Lecture Notes in Computer Science, vol. 13244, pp. 424–428. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25
13. 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
14. 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
15. Sun, Z., Fan, H., He, F.: Consistency-preserving propagation for SMT solving of concurrent program verification. Proc. ACM Program. Lang. **6**(OOPSLA2), 929–956 (2022). https://doi.org/10.1145/3563321
16. Telbisz, C.: Efficient automatic verification of concurrent programs. Master's thesis, Budapest University of Technology and Economics (2024), https://theta.mit.bme.hu/publications/telbiszcsMsc2024.pdf
17. 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
18. Zennou, R., Atig, M.F., Biswas, R., Bouajjani, A., Enea, C., Erradi, M.: Boosting Sequential Consistency Checking Using Saturation. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. Lecture Notes in Computer Science, vol. 12302, pp. 360–376. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6_20

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

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

Author Index

A
Abate, Alessandro III-63
Ádám, Zsófia III-187
Aniva, Leni I-104

B
Baarir, Souheib II-45
Bajczi, Levente I-3, III-187, III-217, III-260
Baranowski, Mark S. II-239
Barrett, Clark I-104
Basin, David III-43
Baumeister, Jan I-60
Beckert, Bernhard II-257
Berg, Jeremias II-108
Berthomieu, Jérémy I-355
Beyer, Dirk III-151, III-192
Blanchette, Jasmin I-85
Blicha, Martin II-67
Blondin, Michael III-23
Bogaerts, Bart II-3, II-108
Budde, Carlos E. I-167

C
Cadilhac, Michaël II-323, III-23
Casares, Antonio II-323
Chalupa, Marek I-303, III-212
Chatterjee, Krishnendu II-130, II-217, II-279
Chen, Tian-Fu III-129
Chen, Yu-Fang III-87
Chen, Zhenbang III-199
Chin, Wei-Ngan I-20
Chocholatý, David II-23
Chung, Kai-Min III-87
Cohen, Liron I-336
Cordeiro, Lucas C. III-223
Cui, Xin-Yi III-23
Czerner, Philipp III-23

D
Dacík, Tomáš III-248

E
Egolf, Derek II-155
Esparza, Javier III-23
Eugster, Patrick II-67

F
Favorito, Marco I-295
Feinstein, Yoav II-303
Finkbeiner, Bernd I-60, II-177

G
Gadelha, Mikhail R. III-223
Ghasemirad, Shabnam III-43
Giesl, Jürgen III-205
Gonçalves, Rafael II-343
Gopalakrishnan, Ganesh II-239
Gouveia, Filipe II-343

H
Hartmanns, Arnd I-167
Havlena, Vojtěch II-23
Heemstra, Jan III-139
Henzinger, Thomas A. I-303
Holík, Lukáš II-23
Hranička, Jan II-23
Hsieh, Cho-Jui I-315
Hsieh, Min-Hsiu III-87
Huang, Wei-Jia III-87
Huber, Nikolaus III-3
Hym, Samuel III-3

J
Jabs, Christoph II-108
JafariRaviz, Mahdi II-217
Jana, Suman I-315

© The Editor(s) (if applicable) and The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 267–269, 2025.
https://doi.org/10.1007/978-3-031-90660-2

Janzen, Marvin II-257
Järvisalo, Matti II-108
Jensen, Nicolaj Ø. I-211
Ji, Chenxi II-364
Jiang, Jie-Hong R. III-129
Jin, Qirui I-315
Jongmans, Sung-Shik I-145

K
Kafshdar Goharshady, Ehsan II-279
Karmarkar, Hrishikesh III-242
Kaufmann, Daniela I-355
Keiren, Jeroen J. A. I-191
Kern, Philipp II-257
Khouri, Basel II-88
Kofroň, Jan II-67
Kolter, Zico I-315
Kondylidou, Lydia I-85
Kosaian, Katherine I-254
Koyejo, Sanmi I-104
Křetínský, Jan I-233
Kupferman, Orna II-198, II-303

L
Laarman, Alfons III-109
Larsen, Kim G. I-211
Laveaux, Maurice I-191
Le, Quang Loc I-20
Lejemble, Thibault II-45
Lemberger, Thomas III-236
Lengál, Ondřej II-23, III-87
Leshkowitz, Ofer II-198
Li, Wenhua I-20
Li, Xianzhiyu III-223
Lin, Jyun-Ao III-87
Lin, Yao III-199
Lingsch-Rosenfeld, Marian III-192
Liu, Si III-43
Lommen, Nils III-205
Lotan, Raz I-375
Lynce, Inês II-343

M
Madhukar, Kumar III-242
Manino, Edoardo III-223
Maoz, Shahar I-40
Mathew, Prince I-276
Mazzocchi, Nicolas I-303
McGowan, Cameron III-254
Meggendorfer, Tobias I-167, I-233
Menezes, Rafael Sá III-223
Metta, Ravindra III-242
Metzger, Niklas II-177
Micskei, Zoltán III-187
Middeldorp, Aart I-124
Midtgaard, Jan III-3
Miné, Antoine III-229
Miranda, Brando I-104
Mitra, Sayan II-364
Molnár, Vince III-217
Monat, Raphaël III-229
Mondok, Milán III-217
Mukhopadhyay, Diganta III-242
Multazzu, Luca III-43

N
Nayak, Satya Prakash II-177
Novotný, Petr II-279

O
Ohlmann, Pierre II-323
Osama, Muhammad III-109
Osborne, Nicolas III-3
Otoni, Rodrigo II-67
Ouadjaout, Abdelraouf III-229

P
Parker, David III-63
Penelle, Vincent I-276
Petoumenos, Pavlos III-223
Prokop, Maximilian I-233

Q
Quatmann, Tim II-130

R
Rakamarić, Zvonimir II-239
Reynolds, Andrew I-85
Richards, Matthew III-254
Richter, Cedric III-212
Rivera, Matias Barandiaran II-67
Rowe, Reuben N. S. I-336
Rozier, Kristin Yvonne I-254

S
Santos, José Fragoso II-343
Saona, Raimundo II-217
Saoudi, Mazigh II-45

Saraç, N. Ege I-303
Schäffeler, Maximilian II-130
Scheerer, Frederik I-60
Schmuck, Anne-Kathrin II-177
Schnitzer, Yannik III-63
Schöpf, Jonas I-124
Schulz, Jakob III-23
Shaked, Matan I-336
Sharygina, Natasha II-67
Shenwald, Noam II-303
Shevrin, Ilia I-40
Shi, Zhouxing I-315
Shoham, Sharon I-375
Siber, Julian I-60
Síč, Juraj II-23
Song, Yahui I-20
Sopena, Julien II-45
Spargo, Naomi III-3
Sprenger, Christoph III-43
Srba, Jiří I-211
Sreejith, A. V. I-276
Stramaglia, Anna I-191
Strejček, Jan III-151
Sui, Yulei III-254
Sun, Chuyue I-104
Svoboda, Jakub II-217
Szekeres, Dániel I-3, III-217, III-260

T
Telbisz, Csanád I-3, III-260
Teuber, Samuel II-257
Thanos, Dimitrios III-109
Tihanyi, Norbert III-223
Tripakis, Stavros II-155
Tsai, Wei-Lun III-87

V
Van Caudenberg, Daimy II-3
Vendramin, Leandro II-3
Vizel, Yakir II-88
Vojnar, Tomáš III-248
Vörös, András I-3, III-260

W
Wachowitz, Henrik III-236
Wagenpfeil, Tobias I-60
Wang, Ji III-199
Wang, Zili I-254
Weininger, Maximilian I-167, II-130
Wienhöft, Patrick I-167
Wijs, Anton III-139
Willemse, Tim A. C. I-191
Winkler, Tobias II-130
Wu, Tong III-223

X
Xiong, Shale III-223

Y
Yatskan, Roy I-40

Z
Zarkhah, Ashkan I-233
Zhang, Huan I-315, II-364
Zhu, Shufang I-295
Žikelić, Đorđe II-279
Zilken, Daniel II-130